Learn Programming in Go (golang): A Rich Guide for Beginners

A comprehensive introduction to Google's
Go programming language (golang)
for beginners & intermediate programmers

Jens Schendel

Version 1.0.3, June 2022

Table of Contents

Section 1 – Introduction

Lecture 1 – Welcome greeting and an invitation to learn

> *I don't know where I am going, but I am on my way..*
> *Voltaire*

Understand the importance of doing the exercises and that making mistakes is part of the learning.

Lecture 2 – Why Go of all things? Brief history, classification, and typification of Go

> *Make a living by doing what you enjoy, and you never have to work a day in your life..*
> *Mark Twain*

Who invented it?

- Google (Go on Wikipedia)
- Rob Pike, Unix, UTF-8
- Robert Griesemer studied with the inventor of Pascal, worked on Google's V8 JavaScript engine
- Ken Thompson led the implementation of U**x and invented the B programming language and thus the predecessor of C and participated in bringing C to life.

Why now exactly Go again?

- Highly efficient compiling
- Go creates compiled programs
- There is a "garbage collector" (GC)
- There is no virtual machine to run code in, no emulator and no interpreter
- Fast execution times
- Ease of use, "Ease of programming"

In summary, three main features that make Go so successful:

1. compiles easily and very quickly – even large projects compile in seconds and minutes, not hours
2. efficient execution with very high execution speed
3. Ease of Programming – programming should be done with ease, not pain in the brain

Brad Fitzpatrick put it together in some slides in 2014 and these slides are available on go.dev: https://go.dev/talks/2014/gocon-tokyo.slide#31

What Go can be used for

- Everything "that Google does" and all internet services that need to meet the highest standards and be highly scalable.
- networking
- http/https, tcp, udp
- concurrency / parallel programming
- conditional system programming
- automation, command-line tools
- crypto
- image processing

Creation principles

- meaningful, understandable, sophisticated
- clean, clear, easy to read

Companies using Go

> Google, YouTube, Netflix, Google Confidential, Docker, Kubernetes, among others, InfluxDB, Twitter, Apple, Cloudflare, DropBox, and others, more examples in detail

Trivia:

> The inventor of Node.js has abandoned Node in favor of Go instead
>
> Go programmers are currently the highest paid programmers in the US – 5th in the world (June 2022)

Lecture 3 – Learning notes on this course

> *Being a student is easy. Learning requires actual work.*
> *William Crawford*

As you can hear I am not a native English speaker. I hope you can deal with and sorry, if my pronunciation is sometimes not the best.

Some terms like fmt I will pronounce as "f.m.t." or "format", not like you often hear as "fumpt". I just don’t like that and "format" make much more clear, what is going on than "fumpt".

The course overview and the PDF is part of the the course. Use it!

Follow the course at your own pace. Don’t skip lectures if you’re not sure you know the content.

Typing, not copying, is the path to success!

Very important: Do the practice. Do all the exercises! Here is where the Karate, the Kung-Fu, the Voodoo, Mojo and the Magic take place. That is learning!

So, ready to learn Go now? Go for it!

Section 2 – The course – an overview

Lecture 4 – Sections and contents overview

> *The best time to plant a tree was twenty years ago. The second best time is now.*
>
> *Chinese proverb*

Lecture 5 – Accompanying course outline as PDF (also on github)

An option to download exactly this document in front of you.

Lecture 6 – Sources of information on the web about Go from the makers/developers and others

Links to important generally accessible sources for the programming language Go (golang)

go.dev and go.dev/play

pkg.go.dev and especially pkg.go.dev/std

blog.go.dev Oldies but goldies.

go.dev/play Let's go to the playground!

go.dev/doc/effective_go Examples and typical applications of programming concepts in Go.

go.dev/doc/faq Questions asked again and again - and the corresponding answers.

gobyexample.com You rarely learn faster than from a good example.

Brad Fitzpatrick about Go Presentation 2014 in Tokyo.

The Go Proverbs speak for themselves.

forum.golangbridge.org is a forum to ask Go specific questions.

Github repo exclusively for this course with some examples which didn't work out to be presented on Go playground

Section 3 – Development environment (and an IDE, if you insist on)

Lecture 7 – Terminals/consoles/shell/bash/command prompt

> *There is no reason for any individual to have a computer at his home.*
> *Ken Olson*

Brief history of computing:

The beginning: ENIAC – programming by wiring circuits: https://en.wikipedia.org/wiki/ENIAC

Later computer had to be programmed by making use of specific features of the hardware with so-called assembly language: https://en.wikipedia.org/wiki/Assembly_language

Later there came finally an abstraction layer of the operating system: https://en.wikipedia.org/wiki/Operating_system

And eventually to the widely use of terminals: https://en.wikipedia.org/wiki/Computer_terminal

And early platform independent Operating systems like DOS: https://en.wikipedia.org/wiki/DOS were more or less nothing else, but first personal computers which integrated a terminal. And there were many of that. But more and more important became that this OS had to be IBM PC compatible: https://en.wikipedia.org/wiki/IBM_PC_compatible

Communication with your OS you did with the shell. https://en.wikipedia.org/wiki/Shell_(computing) like the shell of an egg this layer provides access to the kernel, the core, of the computer. And so it is until today. IT doesn't matter much which OS or computer we're talking about, a Mac, a IBM compatible PC, a tablet, your mobile or your MS windows based machine. It doesn't even matter if you have a graphical user Interface https://en.wikipedia.org/wiki/Graphical_user_interface at all, basically you can boil it down to working on the shell, aitting at a console, connected with a terminal or a terminal emulation, using for example a bash (bourne again shell) or a command prompt. I know it's not entirely correct, but for simplicity's sake I'll use the terms synonymously throughout this course. So when I talk about the terminal on the Mac, I mean a bash on Linux, as well as a command prompt on Microsoft windows. I hope I remember to point out different syntax where appropriate, and in the following Lectures I give you a very limited help on different operating systems.

See also:
https://itnext.io/unix-shells-and-terminals-6012fe713e4f and
https://www.geeksforgeeks.org/difference-between-terminal-console-shell-and-command-line/

Lecture 8 – Bash for Windows

> *Linux is not a threat to Windows.*
> *Bill Gates, 1999*

Under MS Windows, you won't have to use a U**X-oide prompt necessarily, but you can do so: https://www.cygwin.com/ or use Windows' Subsystem for Linux (WSL): https://docs.microsoft.com/en-us/windows/wsl/install

Lecture 9 – Brief introduction to the bash

> *Where there is shell, there is way.*
> *U**X/Linux community saying*

For beginners: https://tldp.org/LDP/Bash-Beginners-Guide/html/index.html

and if you need basic knowledge about Linux, I recommend this free Video Tutorial from Shawn Powers – in more than 5 hours the instructor will teach you the Linux basics in an entertaining and easy to understand way.

Lecture 10 – Brief introduction to the command prompt (cmd.exe)

> *I was gratified to be able to answer promptly, and I did.*
> *I said I didn't know.*
> *Mark Twain*

https://en.wikipedia.org/wiki/Cmd.exe

https://www.makeuseof.com/tag/a-beginners-guide-to-the-windows-command-line/

To create an empty file on MS Windows you can use the GGI or on the command line:

```
copy con file.txt
```

```
Press CTRL+Z
```

Or you can directly use:

```
copy /b NUL file.txt
```

or even

```
copy nul file.txt > nul
```

It's your choice.

Lecture 11 – Installation of Go on macOS, MS Windows and Linux

> *Easy peasy lemon squeezy.*
> *Dish soapTV-commercial for Sqezy*

Short introduction to the installation of Go – all pretty straight forward.

Download the appropriate installer or package for the target system, extract it if needed, install it or have it installed by a package manager.

Please pay attention to the correct architecture (386, AMD64, ARM in 32-bit, 64 bit). If you want you can also compare the check sums with the ones on the website to avoid manipulations. Possibly add the path to Go itself to the default path of the user (or all users).

Link on the web: https://go.dev/dl/

Lecture 12 – Environment variables (especially paths)

> *Nintendo's philosophy is never to go the easy path; it's always to challenge ourselves and try to do something new.*
>
> *Shigeru Miyamoto*

Good advice: https://go.dev/blog/orgranizing-go-code

Lecture 13 – Modern triple jump: writing, compilation, execution of Go code

> *Learn from the mistakes of others. You can't live long enough to make them all yourself.*
>
> *Eleanor Roosevelt*

This is the essence to which you can boil everything down, if you can do it without fancy source code, IDE use, syntax checking, automated suggestion, package management, code display, etc.:

1. Write source code.
2. Compile it.
3. Run it.

Lecture 14 – Native Go commands

> *The beginning is always today.*
>
> *Mary Shelley*

`go env` – shows us the environment variables set by Go

`go fmt code.go` – formats the Go code in the file `code.go` if there are no gross syntactical errors and checks in advance if the code meets the requirements to be delivered to the compiler

`go run code.go` – compiles the source code `code.go` and brings the result to execution.

Lecture 14 – An IDE (Integrated Development Environment) for macOS, MS Windows und Linux

> *Man is a tool-using animal. Without tools he is nothing, with tools he is all.*
> *Thomas Carlyle*

Installation and setup of Visual Studio Code on macOS, MS Windows und Linux.

Lecture 16 – Some off-topic but helpful: Brief overview over github and how to use it

> *Inside every large program, there is a small program trying to get out.*
> *C. A. R. Hoare*

What is git? https://en.wikipedia.org/wiki/Git

Install git: https://git-scm.com/downloads

Use github.com (gitlab.com): https://www.github.com

Suggestion TL;DR;

Install git on your local machine.

Create a free account and repository on github (with `.gitignore` file for Go).

Create an access token (with scope ("repo")) or get familiar with SSH authentication.

On your local machine, change on the command prompt, shell, terminal to a directory where you want to manage your projects. There you enter:

```
git clone https://github.com/accountname/repositoryname.git
```

For SSH certification at github.com the input is:

```
ls
```

You will be asked for account name and access token (as password).

You have a local copy of the repository.

If you change something in your repository locally, you can upload the changes (if you are in the directory) with :

```
git add .
git commit -m "a meaningful message"
git push origin main
```

If you want to update already (elsewhere or by others) executed changes from github locally you can do that with:

```
git pull origin main
```

overlapping changes require special attention, see github.com for how to proceed in individual cases.

Section 4 – Variables, values and types

Lecture 17 – Let's go to the playground!

> *The true object of all human life is play. Earth is a task garden, heaven is a playground.*
>
> Gilbert K. Chestert

Playground's buttons

run

Runs Go code in the browser by transmitting the code to the playground's web server, compiling it, executing it, and transmitting the output back.

fmt

Portability and teamwork on the same code base is made a lot easier if the same rules are followed. "Format", often pronounced "fumpt", ensures that these rules are followed.

- "Idioms" themselves are speech patterns common in both spoken language and programming languages

Example: The phrase "it's raining cats and dogs." would be in German more like "Es schüttet wie aus Kübeln." (means approx. "pouring like from buckets."). Different idioms in different languages.

- When talking about "idiomatic Go" it means to write Go code in the way it is common in the Go community and intended by Google.
- The function "Format" in the Playground, as well as a `go fmt gofile.go` on the terminal/shell ensure that Go code meets that requirements (which probably also ensures that the compiler can process the code quickly).

share

This is a great way to ensure that code can be easily shared on forums like forum.golangbridge.org and that everyone adheres to the same formatting right away.

Ideal for this course to share small examples and make them available in this course outline.

Example: https://go.dev/play/p/MAohLsrz7JQ

Lecture 18 – Hello world, hello control flow

> *Through your ideas, you open the window of your mind and say a hello to the world.*
>
> *Mehmet Murat Ildan*

Control flow (not flow control)

https://en.wikipedia.org/wiki/Control_flow

Sequence

Go code is read, interpreted and executed sequentially "from left to right" within a line and the lines from "top to bottom".

Loops

There are no `while`- or `do-while`-loops in Go – at least not as keywords. All loops are implemented as `for` loops.

Example:

`:=` declares and initializes a variable with a start value.

```
for identifier := startValue; ConditionUpToWhichTheLoopShouldRun; variableChangeStatement {codeBlockForRepeatedExecution}
```

Conditionals

Conditions (conditionals) check whether one or more conditions apply and, if necessary, execute code blocks.

Example:

```
if condition { code }
```

Example control structures: https://go.dev/play/p/Br14DY1_e0x

Lecture 19 – Spoiler of packages and acquaintance with the variadic parameter

> *I am functioning within normal parameters.*
> *Lieutenant Commander Data, Star Trek: The Next Generation*

Introduction to packages and variadic parameter

Notation for using exports from imported packages in Go

`package.Identifier`

Example:
`fmt.Println()`

- translates more to something like "call the function `Println()` from package `fmt`".
- The identifier of a variable, constant or function serves as a name.

Packages

- Packages contain prescribed code that can be imported and used

- similar to includes of header files in C

Variadic parameters

- The notation `...<some type>` is necessary to specify variable parameters
- the type `interface{}` is the so called empty interface, every value of any type is also assigned to the type `interface{}`.
- This means that `...interface{}`allows you to pass any number of values and arguments of any type.

Ignore/discard return values

- Use underscore `_` to discard return values.

Why can't you have unused variables in Go?

- Code pollution
- The compiler does not allow

Example packages, variable parameters: https://go.dev/play/p/5XT6UP1xi3E

Lecture 20 – Some Terminology and the Short Declaration Operator

> *Gophers, ya great git! Not golfers! The little brown furry rodents!*
> Sandy to Carl Spackler (Bill Murray) in Caddyshack

Terminology

https://go.dev/ref/spec show us that in Go we distinguish:

Keywords

All sequences of characters that are already predefined for use in Go.

- sometimes keywords are called "reserved"
- a keyword can be used exclusively for its purpose determined by the creators of Go

Operators

- in `2 + 2`the "+" is the operator
- An operator is a character (or sequence of characters) that represents an operation, like the "+" represents an arithmetic operator for creating a sum.

Operands

- In `2 + 2` the "2"s are the operands

Statements

In programming, a statement is the small unit that can contain an instruction for a program to perform an action. A program is created by stringing together statements that are executed as a sequence.

Expressions

In programming, an expression is a sequence of values, constants, variables, operators, and functions that are interpreted and finally executed by the programming language to get a value from it.

Thus, `2 + 3`is an expression that results in the value "5".

Introduction of the Short Declaration Operator

The statements

```
var identifier int
```

```
identifier = 5
```

can be abbreviated as

`identifier := 5` (this implies the type integer in this case)

Example: https://go.dev/play/p/4QIPR6YRpu3

Lecture 21 – The keyword var comes with a little secret

> *Circumstances and outcome are always variable.*
> *Steven Redhead, Life Is A Cocktail*

Short recap

```
var y = 23
```

Declaration of the variable with the identifier "y"

Value assignment: Value is "23"

Implied type assignment and thus initialization.

Declaration AND value assignment = initialization

```
var z int
```

Declaration of the variable with the identifier "z"

Type assignment: Type is `int`

Implied value assignment and initialization by automatic assignment of a "zero" value (zero value, in some cases nil) that means for example `false` for boolean numbers, `0` for integers, `0.0` for floats, `""` for strings and `nil` for pointers, functions, interfaces, slices, channels and maps.

Example: https://go.dev/play/p/y8GcFghTvdF

Lecture 22 – Types and Typing – It's all about types!

> *You need a lot of different types of people to make the world better.*
> *Joe Louis*

There's the saying "Go suffers no fools." and that derives from a phrase refusing to tolerate stupidity. The expression comes from the New Testament (II Corinthians 11:19), where Paul sarcastically says, "For ye suffer fools gladly, seeing ye yourselves are wise." [c. 1600]

In Go it means "to suffer no fools" and stay robust and reliable as it adheres to strict typing.

If you declare a variable to be of a certain type, this variable can only hold values of a certain type. And if the compiler says "the same type" than the compiler means "exact the same type" and not something similar or something "which could also be some other type" under certain circumstances or "can expressed in a value of another type".

Example: https://go.dev/play/p/clyuLTvww7k

So with

```
var z int = 23
```

declared outside a function z has "package scope" and that means package-wide scope.

Primitive types

In computer science we talk about a primitive/elementary data type in one of the following types:

- a base type is a data type provided by a programming language as a basic building block. Most languages allow more complicated composite types to be constructed on the basis of basic types.
- a built-in type is a data type for which the programming language provides built-in support. In most programming languages all basic data types are built-in. (int, float, char, string, etc.)

In addition, many languages also have a number of composite data types. Opinions differ as to whether a built-in type that is not basic should also be considered "primitive".

The page https://en.wikipedia.org/wiki/Primitive_data_type serves some more information in detail.

Composite data types

In computer science, a composite data type or aggregate data type is any data type that can be constructed in a program from the primitive data types of a programming language and/or from other composite data types.

It is sometimes also referred to as a structural or aggregate data type, although the latter term can also refer to arrays, lists, etc. The process of constructing a composite type is often referred to in English as "composition".

See also on Wikipedia: https://en.wikipedia.org/wiki/Composite_data_type
In some definitions also Strings and Arrays are referred to composite data types.

Example: https://go.dev/play/p/nTdTbz3OE9C

Lecture 23 – There is a value in every type: The zero value.

> *All of us start from zero. We take the right decision and become a hero.*
> *Godiva*

Need for a zero value

The point of a strongly typing language like Go is to assign a value to a variable even when declaring it, so that you don't - as in C, for example - suddenly find yourself with a fantasy value after you have declared a variable but not yet explicitly assigned a value to it.

Go helps you with this. Ease of programming – Programming in Go is supposed to be easy.

Different types get different zero values2

- `false` for booleans
- `0` for integer numbers
- `0.0` for floating point numbers
- `""` for strings
- `nil` for
  Pointers
  Functions
  Interfaces
  Slices
  Channels
  Maps

https://en.wikipedia.org/wiki/Null#Computing

A generally recommended "best practice" is to use the Short Declaration Operator `:=` as often as possible to keep scope of variables as close as possible to the code blocks they’re used in, but to use `var` for

- Zero Values
- Package scope

Example: https://go.dev/play/p/j-QevkgqcfX

Lecture 24 – The package fmt brings our code in good shape

> *To play a soccer game, you have to be in very good shape.*
> *Ronaldo*

The package `fmt` is kind of taking care of the format of code as it serves basic functions for input and output in Go.

Overview: https://pkg.go.dev/fmt#pkg-overview

Example for "verbs" like `%v` within a format string:https://go.dev/play/p/qzgj4LA0yNi

Examples for "rune literals" respectively "escaped" characters like `\n` or `\t` :https://go.dev/ref/spec#Rune_literals

Formatted output

Differences of the various input and output oriented functions:

https://pkg.go.dev/fmt#Print

https://pkg.go.dev/fmt#SPrint

https://pkg.go.dev/fmt#Fprint

Example: https://go.dev/play/p/QNsKk0F7HN5

Group 1: General output on `stdout` (standard output)

- `func Print(a ...interface{}) (n int, err error)`
- `func Printf(format string, a ...interface{}) (n int, err error)`
- `func Println(a ...interface{}) (n int, err error)`

Group 2: General output to a string (string, therefore prefixed "s"), which in our case can also be a variable of type `string`!

- `func Sprint(a ...interface{}) string`
- `func Sprintf(format string, a ...interface{}) string`
- `func Sprintln(a ...interface{}) string`

Group 3: General output to a file (file, therefore prefixed "f"), which in our case can also be a response from a server or similar!

- `func Fprint(w io.Writer, a ...interface{}) (n int, err error)`
- `func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)`
- `func Fprintln(w io.Writer, a ...interface{}) (n int, err error)`

Lecture 25 – DIY – Provide your own type in Go!

> *If you can't have a seat at the table, build your own table.*
> *Anounymous*

In Go, of course, we can create our own (even composite) types.

Example: https://go.dev/play/p/TOJllU_6kzK

Lecture 26 – Type change in Go is not only about appearance, it's conversion (not casting)

> *These sudden conversions do not please me.*
> *Oscar Wilde*

Go has its own language to talk about itself. Old terminology has been thrown overboard because they are loaded with linguistic legacies. In Go, programming has been reinvented, and therefore new terms are used to talk about some concepts to respect their characteristics in Go.

For example, in Go we no longer talk about objects, but about creating types and values of a certain type, like "value of type this-or-that". Of course, much of object-oriented programming (OOP) is also reflected in Go, but the terms should be avoided, because in case of doubt, concepts and application differ in detail from other programming languages!

For this reason, we do not speak of "casting" in Go. We are not evil wizards either, at least most of us are not. So we talk in case of changing a values type about, "conversion".
Ladies and Gents, "conversion is the term of the day!

Example: https://go.dev/play/p/I0Zvw5Rtvkz

Section 5 – Level 1 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecturr0e 27 – To the Keyboards! Get ready ... Fire!

> *It does not matter how slowly you go as long as you do not stop.*
> *Confucius*

Just a few reminders to motivate you!

Takeaway: Do it!

Lecture 28 – Practice 1

1. Create the variables with the identifiers "x", "y" and "z" using the Short Declaration Operator and assign the following values:

   a) `23`

   b) `"Papa Smurf"`

   c) `true`

2. Output the values of the variables with

   a) several single `fmt.Println()` statements for each identifier.

   b) a single `fmt.Printf()` statement for all identifiers at once.

Lecture 29 – Practice 1 – an example solution

https://go.dev/play/p/0J1wFTHe6rY

Lecture 30 – Practice 2

1. Create the variables with the identifiers "x", "y" and "z" using the keyword `var` globally and declare their types as:

   a) `int`

   b) `string`

   c) `bool`

2. Output the values of the variables in the main() function.

Additional question: What do you call these values assigned by the compiler? (Put in a comment in the code.)

Lecture 31 – Practice 2 – an example solution

https://go.dev/play/p/QveQuN_MEKC

Lecture 32 – Practice 3

Based on your code example from the previous practice exercise ...

1. … assign to the three variables at the scope level of the whole package the values

   a) `23`

   b) `"Smurfette"`

   c) `true`

and in function `main()`

2. use `fmt.Sprintf()`

   a) to assign all three values to a separate variable with the identifier "s", which you create using the Short Declaration Operator,

   b) and output the value stored in "s".

Lecture 33 – Practice 3 – an example solution

https://go.dev/play/p/Gqk-uyD10ue

Lecture 34 – Practice 4

FYI: Simple explanation of the terminology "underlying type"

https://go.dev/ref/spec#Underlying_types

For this practice exercise ...

1. Create your own variable type based on the type `int`.

2. Create the variable "x" with the type you create using `var`

3. In der Funktion `main()`

   a) Output the value of "x".

   b) Output the type of "x".

   c) Assign the value 23 to "x" with the simple assignment operator.

   d) Output the value of "x".

Lecture 35 – Practice 4 – an example solution

https://go.dev/play/p/VKHZ36K3_uY

Lecture 36 – Practice 5

Based on your code example from the previous exercise ...

1. create with var a variable with the identifier "y" at the scope level of the whole package and assign it the "underlying type" of the type you just created (so an `int`).

2. In function `main()`

Assign the value of "x" to "y" and use "conversion", so the conversion of the value of the variable "x" into the underlying type `int`.

3. Output the value of "y".

4. Output the type of "y"

Lecture 37 – Practice 5 – an example solution

https://go.dev/play/p/04M-2FwU4U3

Quiz 1 – Hooray, a quiz!

Lecture 38 – Practice 6 Quiz common solution

Section 6 – Fundamentals: The basics

Lecture 39 – The bool type: to be or not to be

> *To be or not to be: that is the question.*
> *Hamlet*

https://en.wikipedia.org/wiki/George_Boole

https://en.wikipedia.org/wiki/Boolean_algebra

https://go.dev/ref/spec#Boolean_types

Examples:

https://go.dev/play/p/AHr7iSOcnhK

https://go.dev/play/p/znA51euWOSk

Lecture 40 – Brief inview how computers do what they do

> *I do not fear computers. I fear lack of them.*
> *Isaac Asimov*

Computers internally calculate only with binary numbers, meaning in the number system with base two.

Zeros and ones are considered like light switches in the binary system. Each of these switches is called a "bit" (a compound term of "binary digit"). These bits require interpretation by us (done within the computer usually). Generally eight such bits are called a byte.

10101010 is an example for a byte. A byte can represent all values from 0 to 255 meaning a total of 256 values.

Values for each digit (from right to left) are described by 0 and 1 and each higher digit represents one to the power of 2 raised by 1.

Example:

23 in the decimal system (number system with base 10) corresponds to the binary number:

0 0 0 1 0 1 1 1 (16 + 4 + 2 + 1 = 23)

128 64 32 16 8 4 2 1

https://en.wikipedia.org/wiki/Binary_code

EA further interpretation can already take place in the computer, but must be standardized according to a consistent, previously agreed system. An example of this is the ASCII system, which assigns letters and characters to numerical values.

https://en.wikipedia.org/wiki/ASCII

ASCII uses 7 bits to describe the character set. UTF-8 uses 32 bits and can thus represent over a million different characters, theoretically even more than 2 million. 1

https://en.wikipedia.org/wiki/UTF-8

The basic functioning of computers is based on the ability to perform arithmetic operations with binary numbers very quic2 then in integrated circuits and finally on universally programmable CPUs (Central Processing Unit), which we know today. The number of circuits in the form of transistors and thus the computing power seems to have doubled at regular intervals over the last decades.

See also:https://en.wikipedia.org/wiki/Moore%27s_law

Lecture 41 – Numeric types: Draw a number!

> *Numbers don't lie. Women lie, men lie, but numbers don't lie.*
> *Max Holloway*

Numeric types describe numbers

https://go.dev/ref/spec#Numeric_types

The types int, float and complex represent the set of integer, floating point and complex numbers, respectively. They are collectively referred to as numeric types.

Integers describe integers.

Floats describe numbers with decimal places.

Complex describes complex numbers (neglected here).

The size of a type can be specified independently of the architecture. The different "subtypes" are not compatible with each other. An int32 and an int are therefore not interchangeable, even if their size is the same on many architectures. Cue here: (strict) static typing!

Integers differ again in signed and unsigned. The left bit is used as an indicator for a sign, which "halves" the possible representable size of "unsigned int" for signed integers.

byte an alias for`uint8`

rune an alias for`int32`

Rule of thumb: Just use int and float64. What is good enough for the compiler is good enough for us. https://go.dev/play/p/feXfquemxaz But we can specify the type exactly if we want: https://go.dev/play/p/Y8sag2jIedM

So if you want (or need) to save memory, you can just do that. An example would be functions or massively parallel running Goroutines whose simple counters are always in the lower three-digit

range anyway, or packages that run massive parallel and otherwise occupy too much never-used memory.

Package `runtime` offers `GOOS` and `GOARCH`

https://go.dev/play/p/nJCcrYrxfDK

Lecture 42 – Realize that: String is a type!

> *Recreate life's strings to weave your own path.*
> *Diana Matoso*

Realize that: strings are a type.

TL;DR;

1. Strings are a distinct data type in Go.
2. The values in strings are immutable (read-only).
3. String values are "slices of byte (`uint8`)".
4. Strings can be empty.

Example from the video: https://go.dev/play/p/uJTjvIDBeOE

Things to know about strings' underlying type

The "slices of byte" are based on a data structure "pointer to the beginning of the slice" and "length of the slice in byte" means the values also require a subsequent interpretation. This already has a lot of similarity to well-known concepts like arrays of characters, but is less restrictive.

Explanation how characters are output at all, if in Go only data of the type byte are arranged together: https://golangbyexample.com/character-in-go/

Explanation and further information by Rob Pike himself, which helps to understand UFT-8 in Go and to see data from the string not only as a string of characters: https://go.dev/blog/strings

If you like you can pre-learn what a slice is and why it is not an array, but a structure in Go, which has freed itself from the restrictions of an array and offers many more features: https://go.dev/blog/slices

Lecture 43 – Numerical systems: 2, 8, 10, 16 – binary, octal, decimal or hexadecimal

> *We are greater than, and greater for, the sum of us.*
> *Heather McGhee*

We get to know number system (again).

Decimal system (base 10)

0 1 2 3 4 5 6 7 8 9 = ten digits

10K	1000s	100s	10ths	Single		
10^4	10^3	10^2	10^1	10^0 (1)		
1	2	3	4	5	=	12345

Binary system (base2)

0 and 1 = two digits

16ths	8ths	4ths	2ths	Single		
2^4	2^3	2^2	2^1	2^0		
1	0	1	1	1	=	23

Binary number 32 bit with all digits written out: 00000000000000000000000000010111 equals 23 in the decimal system.

Hexadecimal (base16)

0 1 2 3 4 5 6 7 8 9 A B C D E F = sixteen digits

65536s	4096s	256s	16ths	Single		
16^4	16^3	16^2	16^1	16^0		
0	0	0	1	7	=	23
0	0	0	7	B	=	123

Example output in Go to here:https://go.dev/play/p/s9dhT-2JRsp

Octal (base8)

0 1 2 3 4 5 6 7 = eight digits

4096s	512s	64s	8th	Single
8^4	8^3	8^2	8^1	8^0

Simple conversion table online: https://www.rapidtables.com/convert/number/hex-dec-bin-converter.html

Numeral system: https://en.wikipedia.org/wiki/Numeral_system

Last sample output in Go: https://go.dev/play/p/VqPMaj4HCXd

Lecture 44 – Constants – the constants in life and in Go

> *The only constant in life is change.*
> *Heraclitus*

Constants in Go come typed and untyped.

Constants, examples about typing: https://go.dev/play/p/aiEMqUimivfr

Constants, examples of declaration: https://go.dev/play/p/3gM12a21s26

Lecture 45 – Iota

> *The present is the only things that has no end.*
> *Erwin Schrödinger*

Iota is a predefined identifier that can be used during the declaration of constants to be able to use an integer incremented by 1 at each assignment.

Iota examples: https://go.dev/play/p/VgNTaj-U4tj

Lecture 46 – Bit shifting: The shifting station for all that little information trains!

> *Life is not a problem to be solved, but a reality to be experienced.*
> *Søren Kierkegaard*

Single Bits can be moved in Go by simple means to manipulate values of variables and constants!

Simple example of bit shifting: https://go.dev/play/p/xMvj6ako5HV

More complex example of bit shifting (constants and Iota): https://go.dev/play/p/rJWLoZpp3r6

Example of other bit-manipulating operations: https://go.dev/play/p/fhPRztZQHu0

Article on Medium to Bit Operators:
https://medium.com/learning-the-go-programming-language/bit-hacking-with-go-e0acee258827

Section 7 – Level 2 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 47 – Further notes on exercises

> *A bruise is a lesson… and each lesson makes us better.*
> *George R.R. Martin, A Game of Thrones*

You're doing well, my friend.

Takeaway: Practice now!

Lecture 48 – Practice 1

Write a short program that assigns the type `uint32` to a variable. Assign this variable a value from its valid value range.

Lecture 49 – Practice 1 – an example solution

https://go.dev/play/p/uBTJFYwDDqB

Lecture 50 – Practice 2

Use the following operators and create one expression with each of them. Assign the values (evaluation) of each expression to a variable (using the Short Declaration Operator).

1. == in variable a
2. <= in variable b
3. >= in variable c
4. != in variable d
5. < in variable e
6. > in variable f

Output the values of the variables a to f with only one statement in separated lines.

Lecture 51 – Practice 2 – an example solution

https://go.dev/play/p/uqqeb3vFR9t

Lecture 52 – Practice 3

Create both "typed" and "untyped" constants.

Output the assigned values and assigned/assumed types in a statement.

Lecture 53 – Practice 3 – an example solution

https://go.dev/play/p/F0RXUC_W6sO

Lecture 54 – Practice 4

Write a program that

- assigns the type int and the value 23232 to a variable.
- Output this value side by side as binary, decimal, and hexadecimal.
- Assign the bit pattern of this value shifted 1 to the left to a new variable.
- Output the value of this variable side by side as binary, decimal, and hexadecimal.

Tip: "Verb" for the output in binary notation is `%b` and width of the expression widened to 32 digits and padded (to the left) with zeros: `%032b`

Lecture 55 – Practice 4 – an example solution

https://go.dev/play/p/XQN7t6hgG_i

Lecture 56 – Practice 5

Create a variable of type `string` with the Short Declaration Operator and assign it as value using a "raw string literal". The value should contain a newline **without** using an escaped character like `\n` .

Lecture 57 – Practice 5 – an example solution

https://go.dev/play/p/P_HdKpJbE21

Lecture 58 – Practice 6

Create four constants of the next years starting with the current year using the expression `iota` in all value assignments. Output the constants side by side separated by spaces.

Lecture 59 – Practice 6 – an example solution

https://go.dev/play/p/s6ZUSKxfUhU

Quiz 2 – Yuppie, another quiz!

Lecture 60 – Practice 7 Quiz 2 common solution

Section 8 – Control Flow – let it flow!

Lecture 61 – Control flow - let it flow, man!

> *The river is everywhere.*
>
> *Herman Hesse, Siddhartha*

In computer science, control structures specify the order in which the steps of an algorithm are processed. In imperative programming languages they are implemented by (control) statements (control structures). With control structures, programs can react to different states by executing program parts only conditionally (conditional statement) or repeatedly (loop).

Flow control (english): https://en.wikipedia.org/wiki/Control_flow

Lecture 62 – Loops – init, cond, post

> *Now... We are going in a loop.*
> *Ramakrishna, Springs of Indian Wisdom*

A `for` loop is introduced by the keyword `for`. It allows a block of code enclosed by `{}` to be executed repeatedly. `for` indicates a period of time. So the meaning is rather "as long as". So simplified in some kind of pseudo code: `as long as this is (still) true, do that`.

With three literals the "this", which can apply, can be described.

1. **Init**ialization: A count variable is declared for the duration of the loop execution and initialized with a start value.

2. **Cond**ition: A termination condition for further iterations is set.

And

3. **Post**-increment or -decrement means that the value of the count variable is changed **after** each iteration

Simple Example: https://go.dev/play/p/EzVL5BAuRLz

Simple explanation: https://gobyexample.com/for

An example, how pre-decrement must be implemented in Go: https://go.dev/play/p/STbMiE5IoAg

For advanced users again here: https://yourbasic.org/golang/gotcha-increment-decrement-statement/

Lecture 63 – Loops – they come nested

> *There are a world of answers, outside the loop.*
> *Anthony Liccione*

Example for a simple nested loop with two `for` statements:
https://go.dev/play/p/60S-W0WL-lZ

Lecture 64 – Loops – understanding the for-statement/documentation

> *Life is follow up loop with distraction.*
> *Deyth Banger, The Diary 2*

Examples and definition of `for` statements in the specifications:
https://go.dev/ref/spec#For_statements

Extended Backus-Naur form to represent syntax rules in programming languages:
https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

From the documentation of the programmers/producers of Go (formerly "Effective Go") with simple examples: https://go.dev/doc/effective_go#for

Lecture 65 – Loops – break and continue

> *Take a break can lead to breakthroughs.*
> *Russell Eric Dobda*

Examples for a `break` statement to get out of an "infinite" loop:
https://go.dev/play/p/XCL_YM_49kn

Example, for the possible usage of `break` and `continue`: https://go.dev/play/p/llOC6Ug_grj
and again with a different condition (bit operator):https://go.dev/play/p/llOC6Ug_grj

Lecture 66 – Loops – let's output ASCII

> *Chaos is merely order waiting to be deciphered.*
> *José Saramago, The Double*

https://en.wikipedia.org/wiki/ASCII

Example: https://go.dev/play/p/iBU2NI3k1MB

Lecture 67 – Conditionals: if – the conditional jump

> *Jump!*
> *Van Halen*

Example: https://go.dev/play/p/qjCzx5T9JIs

Lecture 68 – Conditionals: if, else if, else – if this, then that, otherwise what?

> *I'm handsome, no ands, buts or ifs!*
> *Colin Mochrie*

Example: https://go.dev/play/p/EiBkYlcDkkX

Lecture 69 – Loops, conditionals and the modulo

> *Every man should measure himself by his own standard.*
> *Lat., Metiri se quemque suo modulo ac pede verum est.]*
> *Horace*

Example: https://go.dev/play/p/oNiJ5e5FRgq

Lecture 70 – Conditionals: switch – a brief look in the documentation

> *I get a thrill meeting kids who are into alternative music.*
> *Kurt Cobain*

The switch statement

`switch / case / default`

Brief look in the Specs: https://go.dev/ref/spec#Switch_statementsr

Lecture 71 – Conditionals: the switch statement in action

> *You have as many options as you give yourself.*
> *Kasie West*

The switch statement

`switch / case / default`

- fall-through is not a default, which means that a `break` is not necessary!
- fall-through possible though
- several cases one after the other can be evaluated in one `case` statement
- The cases can also be testable expressions as well ("cases run if `true`")

Examples:

`switch` with `bool` values: https://go.dev/play/p/MDbGAhPMPD6

Fall-through is not a default: https://go.dev/play/p/3rD3sP-Us8F

Fall-through possible though: https://go.dev/play/p/ukNYdl_STc2

Use of `default`: https://go.dev/play/p/J8GeuI7uT52

`switch` for one value: https://go.dev/play/p/797_OcWssv2

`switch` for evaluation of more than one value in one line: https://go.dev/play/p/3KlWR2n_Oq4

Lecture 72 – Conditionals: Logical operators ahead!

> *Logic will get you from A to B. Imagination will take you everywhere.*
> *Albert Einstein*

https://go.dev/ref/spec#Logical_operators

```
fmt.Println(true && true)
fmt.Println(true && false)
fmt.Println(true || true)
fmt.Println(true || false)
fmt.Println(!true)
```

https://go.dev/play/p/LdJlmtdodwC

https://go.dev/play/p/9I0RdyiJSrF

Lecture 73 – browsh – a sneak peek at a Go programming example

> *Knowledge is power.*
> *Sir Francis Bacon*

Website and download: https://www.brow.sh und https://www.brow.sh/downloads/

Source code (in brackets large parts of the Go code)

https://github.com/browsh-org/browsh
(https://github.com/browsh-org/browsh/tree/master/interfacer/src/browsh)

Section 9 – Level 3 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 74 – Practice 1

Output the numbers 1 to 10000.

Lecture 75 – Practice 1 – an example solution

https://go.dev/play/p/8AJQXsRhKgV

Alternative (not so elegant): https://go.dev/play/p/3QrB_9XpVps

Lecture 76 – Practice 2

Output each "rune code point" of the capital letters of the alphabet three times. This should look something like this:

```
65
U+0041 'A'
U+0041 'A'
U+0041 'A'
66r
U+0042 'B'
U+0042 'B'
U+0042 'B'
```

… to the character "Z", meaning A to Z

Lecture 77 – Practice 2 – an example solution

https://go.dev/play/p/Gmqrisyyjx5

Lecture 78 – Practice 3

Create a loop with

```
for condition { }
```

and use it to output all the years you already live (and have lived in).r

Lecture 79 – Practice 3 – an example solution

https://go.dev/play/p/9SnkJMv-7FM

Lecture 80 – Practice 4

Create a loop with

```
for init; ; post { }
```

and use it to output all the years you already live (and have lived). But the condition you put in the code block and leave the loop with a break.

Lecture 81 – Practice 4 – an example solution

https://go.dev/play/p/mMXhnKGhJGN

Lecture 82 – Practice 5

Give the remainder (modulo) that results when the numbers between 10 and 100 (both inclusive) are divided by 4.

Lecture 83 – Practice 5 – an example solution

https://go.dev/play/p/lVIvpyZeg0t

Lecture 84 – Practice 6

Create a program that uses an `if` statement.

Lecture 85 – Practice 6 – an example solution

https://go.dev/play/p/BKFdRkT5qgW

Lecture 86 – Practice 7

Extend the program from practice 6 that it now also uses `else if` and `else` .

Lecture 87 – Practice 7 – an example solution

https://go.dev/play/p/AqF2PPj3MAX

Lecture 88 – Practice 8

Create a program that uses a `switch` statement without explicitly specifying an expression.

Lecture 89 – Practice 8 – an example solution

https://go.dev/play/p/LggCkSQX4W4

Lecture 90 – Practice 9

Create a program that uses a switch statement and queries an expression of type `string` named "favSport". Specify three sports in the `case` distinction and a `default` case the output "I don’t care about sports.".

Lecture 91 – Practice 9 – an example solution

https://go.dev/play/p/bEGk9J_mKwO

Lecture 92 – Practice 10

Output these logical comparisons and their outcomes:

```
fmt.Println(true && true)
fmt.Println(true && false)
fmt.Println(true || true)
fmt.Println(true || false)
fmt.Println(!true)
```

Lecture 92 – Practice 10 – an example solution

https://go.dev/play/p/ALIStHmRy1p

Quiz 3 – all good things come in threes

Lecture 94 – Practice 11 Quiz 3 common solution

Section 10 – Grouping data

Lecture 95 – Arrays are only the beginning

> *We face a wide array of threats, which means we have to have a wide array of capabilities.*
>
> *Mac Thornberry*

General information about arrays: https://en.wikipedia.org/wiki/Array_data_structure

and about arrays in Go in specific: https://go.dev/ref/spec#Array_types and https://go.dev/doc/effective_go#arrays

Example of a declaration of an array with keyword var: `var x [5]int`

Example for a declaration and value assignment with Short Declaration Operator (and "composite literal", syntax with "the underlying type" preceded by length in square brackets and trailing value list in braces separated by commas): `x := [5]int{1,2,3,4,5}`

Arrays in Go

- Arrays are a data structure to put similar values (of the same type) in an order and make them accessible by an index.
- Arrays have values, i.e. you copy all values of one array into another, not only a reference to the first value. (“Call-by-Value”)
- The length of an array (number of elements) is part of its specific type, that means arrays of different length are considered to be of a different type.

Example: https://go.dev/play/p/W1XHLX4WOTP

Lecture 96 – Slices – meet the composite literal

> *No matter how thin you slice it, it's still baloney.*
>
> *Al Smith*

Example of a composite literal used for a slice during initialization using the Short Declaration Operator: https://go.dev/play/p/gkEqV0CSMXB

Lecture 97 – Slices are the better arrays

> *A lot of movies are about life, mine are like a slice of cake.*
> *Alfred Hitchcock*

Lecture 98 – Slices and range like team up together

> *I am not interested in slice of life, what I want is a slice of the imagination.*
> *Carlos Fuentes*

Examples of using `range`: https://go.dev/play/p/n6vHSYmc1uM

Lecture 99 – Slicing a slice – best idea since sliced bread

> *Being an actor is really, really hard, no matter how you slice it.*
> *Amanda Peet*

Create and outputting slices

You can iterate over slices with for loops using `range`

Output a slice with colon inside square brackets `[:]` (operator here is the colon ":") from (position enclosed) to below (means the position after the colon is not enclosed).

Evaluate slices from/to with expressions

Original slice remains untouched, but you can assign the results to a new slice.

Example of slicing a slice: https://go.dev/play/p/k340QgOsmDK

Lecture 100 – Append() – how to add something to a slice

> *In the pie chart of my brain growing up, there's a huge slice for 'Ghostbusters'.*
> *Evan Goldberg 2*

Specification and description of `append()` with interesting usage examples: https://go.dev/ref/spec#Appending_and_copying_slices

More examples for different use of `append()`: https://go.dev/play/p/ccjUNtl2L3r

Lecture 101 – Append-Paradox – deleting something from a slice

> *I still love pizza, but instead of eating half, I eat a slice.*
> *Bill Engvall*

There is no built-in "delete from slice" function in Go, instead, you should use `append()` and compose the new slice from the slice up to the element to be deleted and the elements after the element to be deleted:

Examples: https://go.dev/play/p/VI0z1wuNSsW

Lecture 102 – How to make a slice? With make(), of course!

> *On the weekends, some people garden; I slice salmon.*
> *Jerry Della Femina*

Expanding a slice means overhead/additional workload during execution. Remember, that each slice is of its own type and expansion means preparation of a new slice of bigger size, copying the content of the old one and eventually destruction of the old slice.

Example: https://go.dev/play/p/RU5f3avV4dD

If the required size and capacity at compile time are known (or at least well estimable), slices can and should be created with the `make()` function.

`make()` on effective Go: https://go.dev/doc/effective_go#allocation_make

Syntax: `make(type, length, capacity)`

returns a slices (!) with the requested properties whose values specified in length are already filled with zero values (zero values of the underlying type of the slice).

Example of using `make()`: https://go.dev/play/p/pG9_4VIZdAW

Lecture 103 – Multidimensional slices – they come from an outer dimension?

> *I never want to be just one thing - I want to be multidimensional.*
> *Katy Perry*

Slices can have multiple dimensions.

Example of slices of two dimensions deriving from one-dimensional slices (of `string`): https://go.dev/play/p/ow3NB26XEU6

Lecture 104 – Map – an introduction, and the comma okay idiom

> *You can't use an old map to explore a new world.*
> *Albert Einstein*

A `map` is a distinct data type in Go that allows to search unordered lists of values of a type (element type) by values of a (possibly different) specific key (key type).

For those with some more experience in programming, maps are the implementation of hash tables in the Go programming language. Searching maps is very effective and fast even with large amounts of data and within extreme big sets of data. https://medium.com/kalamsilicon/hash-tables-implementation-in-go-48c165c54553

If the searched key value is not present in the map, `nil` (the zero value of the element type) is returned. But queries to maps also return value of type bool that confirms or denies the presence of the key value.

This is made possible by the so-called "comma okay" idiom, which allows a distinction between `nil` and "not present" when querying key values.

Example map as data type and the expression called "comma okay idiom": https://go.dev/play/p/64QGjbFVhbR

Lecture 105 – Map – how to add elements to a map and iterate over one with range

> *There's no map to human behaviour.*
> *Bjork*

Example: https://go.dev/play/p/wmBTK_ost_y

Lecture 106 – Map – how to delete elements from a map with delete()

> *The time you want the map... is before you enter the woods.*
> *Brendon Burchard*

The `delete(map, KeyValue)` function can be used to remove an entry from a map.

Example: https://go.dev/play/p/6ERPT6nCrvL

Section 11 – Level 4 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 107 – Practice 1

Use a "composite literal" to...

- create an array with five elements of type `int`
- manually assign a value to each index position
- use a `for` loop with `range` to output the array and its index
- use a formatted output
- and then output the type of the array

Lecture 108 – Practice 1 – an example solution

Example: https://go.dev/play/p/bsae0ydR1Ff

Lecture 109 – Practice 2

Use a "composite literal" to...

- create a slice from values of type `int`
- assign 10 values
- use a `for` loop with `range` to output the slice and its index
- use formatted output
- and then output the type of the slice

Lecture 110 – Practice 2 – an example solution

Example: https://go.dev/play/p/JR4b8E3yqC_X

Lecture 111 – Practice 3

Create the following slice with values of type `int`:

```
[42 43 44 45 46 47 48 49 50 51]
```

Use "Slicing" to achieve the following outputs (without changing the slice)

```
[42 43 44 45 46]
[47 48 49 50 51]
```

```
[44 45 46 47 48]
[43 44 45 46 47 78 49 50]
```

Lecture 112 – Practice 3 – an example solution

Example:https://go.dev/play/p/0g1Gxe-AiB_g

Lecture 113 – Practice 4

Perform the following steps.

Start with the slice:

```
x := []int{42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
```

- Append the value 51 with `append()`
- Output the slice
- Attach in only one statement the values 52, 53 and 54
- Output the slice
- Attach in only one statement the following slice

```
y := []int{56, 57, 58, 59, 60}
```

Output the slice.

Lecture 114 – Practice 4 – an example solution

Example:https://go.dev/play/p/tlJjoczViim

Lecture 115 – Practice 5

Perform the following steps.

Start with the slice:

```
x := []int{42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
```

Use `append()` and slicing to assign the following slice to the new slice `y` to be created:

```
[42, 43, 44, 48, 49, 50, 51]
```

Lecture 116 – Practice 5 – an example solution

Example: https://go.dev/play/p/Pg1d724mtUG

Lecture 117 – Practice 6

Create a slice to store the names of all German states. Use make() and append() to do this.

Goal: The array that the slice is based on should not be created more than once.

Consider the length of your slice. What is the capacity?

Output all values along with with their index position, without using `range`.

Note that you don't have to care about the UTF-8 encoded character in "Thüringen" and "Baden-Württemberg" on the playground.

Here are the states:

```
`Bayern`, `Baden-Württemberg`, `Berlin`, `Brandenburg`, `Bremen`,
`Hamburg`, `Hessen`, `Mecklenburg-Vorpommern`, `Niedersachsen`,
`Nordrhein-Westfalen`, `Rheinland-Pfalz`, `Saarland`, `Sachsen`,
`Sachsen-Anhalt`, `Schleswig-Holstein`, `Thüringen`
```

Lecture 118 – Practice 6 – an example solution

Example: https://go.dev/play/p/1BHPpsYRNdo(from video) or https://go.dev/play/p/TMKXkElrsEL (also okay)

A clue what does not work in this lecture: https://go.dev/play/p/0DkjJnMcQnv

Lecture 119 – Practice 7

Create a slice of slice of `string` (`[][]string`). Save the following values:

```
"James", "Bond", "Bond, James Bond"
"Papa", "Smurf", "Smurf, Papa Smurf"
"Rick", "Sanchez", "Smartest Man in the Universe"
"Morty", "Smith", "ProfessionalSidekick"
```

Use two nested `for` loops with `range` to output the number (index) of the slice and below each all values of the corresponding slice with indented position within the slice.

Something like:

Slice Number: 0

Position 0: James

Position 1: Bond

Position 2: Bond, James Bond

aso ...

Lecture 120 – Practice 7 – an example solution

Example:https://go.dev/play/p/Z2CP3o0vR-E

Lecture 121 – Practice 8

Create a map with a key of type `string` corresponding to a person's "first name last name" and a value of type []string that stores their favorite things. Store seven records in your map. Output all values, along with their key value in the map and their index position in the slice.

```
`Stan Smith`, `America`, `Jesus`, `Family`
`Francine Smith`, `Lipstick`, `Pink dress`, `Crying under the shower`
`Hayley Smith`, `Headband`, `Tank Top`, `Flip-flops`
`Steve Smith`, `Computer`, `Girls`, `Friends`
`Roger Smith`, `TV`, `Alcohol`, `Drugs`
`Klaus Heissler`, `Ski-jumping`, `Swimming`, `Rap & Hip Hop`
`Jeff Fischer`, `Smoking weed`, `Fish (the Band)`, `his hat`
```

Lecture 122 – Practice 8 – an example solution

Example:https://go.dev/play/p/I9QSwf24EfB

Lecture 123 – Practice 9

Building on the code from the previous practice, add an entry for itself to the list.

Lecture 124 – Practice 9 – an example solution

Example: https://go.dev/play/p/-XNz5pWtRr9

Lecture 125 – Practice 10

Remove Klaus' entry from the map, making sure that the entry is only deleted if it exists!

Output the entire values with a `for` loop and with the use of `range`.

Lecture 126 – Practice 10 – an example solution

Example:https://go.dev/play/p/FP4_XWS3yTd

Quiz 4 – this time there is no mercy!

Lecture 127 – Practice 11 Quiz 4 common solution

Section 12 – Strucs: How to give data a structure

Lecture 128 – Structs – they bring structure to life!

> *The human mind is a dramatic structure in itself and our society is absolutely saturated with drama.*
>
> *Edward Bond*

Structs offer the possibility to build composite structures from different data types and to assign variables as values. This has already much of objects and classes from other programming languages, nevertheless we speak in Go of "values of type", meaning values of a certain type.

Usually these structs are assigned to their own data type with the keyword type and are initialized at the beginning when a value assignment is to be made.

After definition of a type as struct as in

```
type  struct TypeIdentifier struct {}
```

value assignment takes place in the form

```
x := struct TypeIdentifier{}
```

or with

```
var x struct TypeIdentifier
```

the zero value is assigned to all elements of the struct. This is also true for value assignments omitted in an initialization.

Details in an example: https://go.dev/play/p/xtMciUkfdd2

Lecture 129 – Embedded structs – when structs contain structs

> *What we observe as material bodies and forces are nothing but shapes and variations in the structure of space.*
>
> *Erwin Schrödinger*

Structs can contain other structs as elements. A type specification of the inner struct is not necessary for the declaration, but it is necessary for the initialization. Elements of structs can be accessed with

the dot operator ("."). It is not necessary to name elements of embedded structs in the hierarchy of the expression, but is is good practice.

A call like `outerStruct.elementInnerStruct` is usually sufficient.

However, in order to avoid name collisions it is possible to replace them with `outerStruct.innerStruct.elementInnerStruct` to be make the element explicitly called.

Example: https://go.dev/play/p/K9cX21NaLLG

Lecture 130 – The necessary look into the manual

> *We might be the holographic image of a two-dimensional structure.*
> *Brian Greene*

Structs: https://go.dev/ref/spec#Struct_types

Brief introduction of Go's structs: https://www.geeksforgeeks.org/structures-in-golang/

Lecture 131 – Anonymous structs – strucs without names

> *I don't think that scheduling is uncreative. I think that structure is required for creativity.*
> *Twyla Tharp*

Anonymous structs: https://go.dev/play/p/FiGxAvrmJav

Lecture 132 – Aftermath: After dinner let's clean the dishes

> *The universe is built on a plan the profound symmetry of which is somehow present in the inner structure of our intellect.*
> *Paul Valery*

Ease of Programming

Most statements follow the same pattern:

```
var identifier type
type identifier struct{}
keyword identifier type // more generic
```

Stay accurate and write code that is as readable as possible. Go allows "shortcuts", but good coding includes understandable and human-readable code.

Is Go an object-oriented programming language?

https://en.wikipedia.org/wiki/Object-oriented_programming

https://go.dev/doc/faq#Is_Go_an_object-oriented_language

The strict typing of Go requires always being aware of types of your variables. https://go.dev/play/p/3Yl_vwMBuX92

Browse the documentation and the many offerings from and about Go: https://go.dev/and https://go.dev/doc/

By this point, you've already gained enough knowledge to seek out and benefit from learning opportunities on your own, like simple tutorials.

Section 13 – Level 5 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 133 – Practice 1

Create your own data type `person` which has an underlying type `struct` so that can store the following data:

- Given Name
- Family Name
- Age
- several favorite ice cream flavors

Create two values of type person. Output the values using range that are specified in an element of type `[]string` (slice if string) for the favorite ice cream flavors.

Lecture 134 – Practice 1 – an example solution

Example:https://go.dev/play/p/x-tzDo6IQlL

Lecture 135 – Practice 2

Take the code from the previous practice and store the values of type `person` in a map with the key of the type containing the last name. Access each value in the map and also output the values contained in the slice.

Lecture 136 – Practice 2 – an example solution

Example: https://go.dev/play/p/VCWnfnPzgjI

Lecture 137 – Practice 3

Create a new type: `vehicle`

- The underlying type is a `struct`.
- The fields:
  `numberOfDoors`
  `color`
- Create two new types:`truck` and `limo`
- The underlying types of each of these new types are a `struct`.
- Embed the type `vehicle` in `truck and limo`.
- Give the `truck` the field "fourWheel", set to type `bool`.
- Give the `limo` the field "luxury", set to type `bool`.

Use of the `vehicle, truck` and `limo structs`:

- Create a value of type `truck` with the identifier `bigCar`. Use a composite literal and assign values to all fields
- Create a value of type `limo` with the identifier `smallCar`. Use a composite literal and assign values to all fields.
- Output these two "values of type".
- Output at least one of the values of the embedded fields.

Lecture 138 – Practice 3 – an example solution

Example:https://go.dev/play/p/LS4I7tLlRi8

Lecture 139 – Practice 4

Create an anonymous struct, meaning a struct without an identifier.

Lecture 140 – Practice 4 – an example solution

Example:https://go.dev/play/p/kuachSzII1l

Quiz 5 – last but not least: A short struct quiz on the fly

Lecture 141 – Practice 5Quiz 5 common solution

Section 14 – Functions – where programming really starts

Lecture 142 – Functions and their syntax – where all the fun begins

> *It's the repetition of affirmations that leads to belief. And once that belief becomes a deep conviction, things begin to happen.*
>
> *Muhammad Ali2*

What's functions and what are they good for?
https://en.wikipedia.org/wiki/Procedural_programming

Function in Go are a kind of type, better they are a type! https://go.dev/ref/spec#Function_types

Simplified syntax of functions in Go in general (in my own words):

```
func (r receiver) identifier(parameterList parameterType(s))
(return(s)Types){ code }
```

Example simple function without return values: https://go.dev/play/p/-WKV0HitEVC

Example of a function and passing one argument: https://go.dev/play/p/r28rNrkfcrC

Example of a function and passing one argument and returning one value:
https://go.dev/play/p/2iy4Kpqf3rz

Example of a function and passing two arguments and returning two values:
https://go.dev/play/p/1fM-HamDUQI

Lecture 143 – Variadic parameters – a second look

> *If you have a procedure with 10 parameters, you probably missed some.*
>
> *Alan Perlis*

Example for passing arguments of the same type as variadic parameters to a function:
https://go.dev/play/p/ufplT7uRpKP and https://go.dev/play/p/K1iopHq23gO and
https://go.dev/play/p/2O7izJ4tLF7

Example of passing parameters of different types followed by a variadic parameter:
https://go.dev/play/p/DZlIMDHgv9k

Main takeaway:

- Variadic parameters can be used in a function signature to pass a list of any number of values of the same type to a function, which is available within the function as a slice of these values.
- Variadic parameters are specified with the ... operator and can be specified only **once** and as the **last parameter** in the signature of a function.

Example for using variable parameters to pass arguments and a function to sum up any number of summands within a function: https://go.dev/play/p/61a2gr5rN0l

Lecture 144 – Slices – let's unfurl them

> *He was frightened by this glimpse of what was in her and wouldn't watch it unfurl.*
>
> *Lauren Groff*

Specs: https://go.dev/ref/spec#Passing_arguments_to_..._parameters

Example "unfurl" or "unfold" the values in slice and pass them as a series of values to a function: https://go.dev/play/p/9NQQRA0o0yU

Lecture 145 – Defer – we start with a delay tactic

> *To defer to someone else's definition of a life well-lived is a Faustian bargain.*
>
> *Rachel Simmons*

The keyword `defer` is used to delay the execution of code (in functions) until the time when the enclosing code block is terminated or "crashes" or is about to crash ("panicking").

Example: https://go.dev/play/p/yJluFvksOdr

Lecture 146 – Methods – Functions come with method (if you allow)

> *The true method of knowledge is experiment.*
>
> *William Blake*

```
func (r receiver) identifier(parameterList parameterType(s))
(return(s)Types){ code }
```

Simple example of a method: https://go.dev/play/p/kMbeR19zsnj

Can functions/methods in Go have multiple receivers? https://www.iops.tech/blog/method-receiver-types-in-go/(and other interesting things about methods)

Simple example of methods and example for the use of pointers to structs in methods (as receivers in functions): https://gobyexample.com/methods

The example inspired by the previous link but simplified: https://go.dev/play/p/yluUG7GvnnB

Lecture 147 – Methods – once again with feeling

> *Art and science have their meeting point in method.*
> *Edward G. Bulwer-Lytton*

A more realistic example for a method in Go: https://go.dev/play/p/WhVHnImT092

Lecture 148 – Methods – a few words about "call by value" vs "call by reference"

> *There is no method but to be very intelligent.*
> *T. S. Eliot*

Go does not like to speak of "Call by Value" and "Call by Reference", but means exclusively in any case "Call by Value". Please don't get confused if in other programming languages the concepts "Call by Value" and "Call by Reference" are distinguished.

The makers of Go also consider a pointer as "Call by Value", because a pointer is also a value in the end, just of the type pointer. This concept simplifies thing reasonably and avoids inaccuracies and misunderstandings. For the rest of the course, we will take this approach and consider all passed arguments as "Call by Value".

Example method "Call by Value": https://go.dev/play/p/0UDVng9waxy

Example method NOT-"Call by Reference": https://go.dev/play/p/AXgeg0jbvvv It's just a pointer value!

Lecture 149 – Insertion: Stay tuned!

> *Man needs difficulties; they are necessary for health.*
> *Carl Jung*

Lecture 150 – Interfaces and Polymorphism I

> *A picture is worth a thousand words. An interface is worth a thousand pictures.*
> *Ben Shneiderman*

Interfaces serve as an abstraction layer between types and methods.w1r

Basically, a variable, or better any entity can be of more than one type. And you can select the type by the kinds of methods a type implements.

Initial example ("Cleaning" was necessary): https://go.dev/play/p/HwSuG8vb6NZ

Example for the use of interfaces: https://go.dev/play/p/nSW34o0RmS9

Example of the use of `switch` depending on type: https://go.dev/play/p/MGHpZBFo91R

Lecture 151 – Interfaces and Polymorphism II

> *I think the major good idea in Unix was its clean and simple interface: open, close, read, and write.*
>
> *Ken Thompson*

Example extended: https://go.dev/play/p/tYPGsMjAFp9

Bill Kennedy's blog about "Composition with Go": https://www.ardanlabs.com/blog/2015/09/composition-with-go.html

Lecture 152 – Interfaces reloaded

> *I don't want to use my creative energy on somebody else's user interface.*
>
> *Jeff Bezos*

Example: https://go.dev/play/p/K-tCfDjXa7k

Inspired by a GoByExample example: https://gobyexample.com/interfacesr

Lecture 153 – Interfaces revolutions

> *As far as the customer is concerned, the interface is the product.*
>
> *Jef Raskin*

Our example: https://go.dev/play/p/yGTd4MtgD5

Inspired by Jordan Orelli example: https://jordanorelli.com/post/32665860244/how-to-use-interfaces-in-go

Lecture 154 – Anonymous functions – they don't need names to do their jobs

> *Coincidence is God's way of remaining anonymous.*

Albert Einstein

Example for the use of anonymous functions: https://go.dev/play/p/GKRUs69ERmC

Lecture 155 – Func expressions – we are at the entrance of the rabbit hole

> *Funk, I don't think I have anything to do with funk. I've never considered myself funky.*
>
> *David Bowie*

Example for the use of function expressions: https://go.dev/play/p/Xg2oaVmnO8k

Difference of declaring a function to using function expressions

Here is one unique property to each:

- A function declaration binds an identifier, the *function name*, to a function; so the function name will be an identifier which you can refer to.
- A function literals represents an anonymous function. Function literals are *closures*, they capture the surrounding environment: they may refer to variables defined in a surrounding function. Those variables are then shared between the surrounding function and the function literal, and they survive as long as they are accessible.

Source: https://stackoverflow.com/questions/46323067/when-to-use-function-expression-rather-than-function-declaration-in-go

Lecture 156 – A function can be a return value – believe it!

> *No one is a friend to his friend who does not love in return.*
>
> *Plato*

Repetition functions: https://go.dev/play/p/4tRAtHQA5t-

Repetition of function expressions/anonymous functions: https://go.dev/play/p/0aQHGRpLnFO

New: Functions can also serve as type for return values: https://go.dev/play/p/OnKmQiAVQum

Lecture 157 – Callbacks – pass functions as arguments to other functions

> *If you don't do it someone else will.*
>
> *T.J. Masters*

In computer science, a callback function refers to a function that is passed to another function as a parameter and is called by the latter under defined conditions and arguments.

Example of simple callback: https://go.dev/play/p/nkfxqul4cVp

Lecture 158 – Closure – put it in a capsule and see by time what you put it

> *You can't ever move on without the proper closure.*
> *Erica Mena*

A closure is an anonymous function that is returned by another function and gets access to a value during its creation (context). Outside of the closure function this value is not accessible.

Simple example of closure: https://go.dev/play/p/XCEevDFD0BG

Please remember Lecture 155 – function expressions.

Lecture 159 – Recursions – welcome to the Matrix!

> *Life can only be understood backwards; but it must be lived forwards.*
> *Søren Kierkegaard*

Recursion is the term used to describe a function that calls a copy of itself while it is running.

Example factorial: https://go.dev/play/p/d7GITyM7IN1

Section 15 – Level 6 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 160 – Short recap (and a tip against procrastination)

> *No, I don't ever give up. I'd have to be dead or completely incapacitated.*
> *Elon Musk*

Topics that should now be conceptually understood:

- Functions
- Purpose of functions is

  code abstraction
  reuse of existing code blocks
- func, receiver, identifier, parameters, return value(s) (types)
- parameters vs arguments

- Variational functions
    - Multiple "variadic" parameters
    - Multiple "variadic" arguments
- Return values
    - Multiple returns
    - "Named" returns
- Function expressions
    - assign a function to a variable and make it of type function!
- callbacks
    - pass a function to another function as argument
- closure
    - store the scope of a variable in another variable, making it visible only in the inner scope.
- recursion
    - functions that call themselves, example: factorial
- interfaces and "empty interface" `(interface{})`

> *Do what is important rather than what is urgent.*
> *Stephen R. Covey*

and follow this advices: https://www.mindtools.com/pages/article/newHTE_96.htm

Lecture 161 – Practice 1

In this exercise:

- Create a function with the identifier `foo` that returns a value of type `int`.
- Create a function with the identifier `bar` that returns a value of type int and a value of type `string`.
- Create valid function calls for both functions and assign the return values to newly created variables.
- Output the values of that variables.

Lecture 162 – Practice 1 – an example solution

Example: hhttps://go.dev/play/p/9_uxLAS96ye

Lecture 163 – Practice 2

Create a function with the identifier `foo`, which

- takes a variadic parameter of type `int`
- pass a value of type `[]int` to the function in an appropriate way
- return a sum of all passed input values as return value (type `int`)

Create a function with the identifier `bar`, which

- takes values of type `[]int` as parameters
- return a sum of all passed input values as return value (type `int`)

Lecture 164 – Practice 2 – an example solution

Example: https://go.dev/play/p/2OAYS2qkJAz

Lecture 165 – Practice 3

Create two functions and call them. Use `defer` to delay the first call until after the second call.

Lecture 166 – Practice 3 – an example solution

Example: https://go.dev/play/p/pgJdCfzi7kZ

Lecture 167 – Practice 4

Create a type with underlying struct with the identifier `person`.

Choose appropriate types for the elements:

Given name

Family name

Age

- assign a method to the type `person` which has the identifier `says`.

  The method accesses the struct defined in `person` and outputs a string with full name and age.

- Create a value of type `person` .
- Call the method `says` for the value.

Lecture 168 – Practice 4 – an example solution

Example: https://go.dev/play/p/k6mcxkTXixZ

Lecture 169 – Practice 5

Create a type `square` and a type `circle` based on structs.

Create a method `area` that returns a value of type `float64` and assign the method to both types.

> Area of a circle equals π * r * 2
>
> Area of a square = side length * side length

Create a type `shape` that defines an interface defined by the implementation of the method `area .`

Create a function with the identifier `info` that takes the type `shape` and outputs the area.

Create/output using `info` :

Value of type `square` .

Value of type `circle` .

Call the function `info` for both values!

Lecture 170 – Practice 5 – an example solution

Example: https://go.dev/play/p/Z_eXTHRLBCs

Lecture 171 – Practice 6

Create and use an anonymous function.

Lecture 172 – Practice 6 – an example solution

Example: https://go.dev/play/p/sfzhS6Wi_2j

Lecture 173 – Practice 7

- Assign a function (that does something) to a variable and call that function.
- Extend the example so that the function also takes a value as a parameter and outputs it.

Lecture 174 – Practice 7 – an example solution

Example: https://go.dev/play/p/QMyJ75fRW-T and https://go.dev/play/p/xoHgMLmdKsS

Lecture 175 – Practice 8

- Create a function that returns a function as a return value.
- Assign the returned function to a variable.
- Call the function through using the variable.

Lecture 176 – Practice 8 – an example solution

Example: https://go.dev/play/p/Pkku9pf4z6-

Lecture 177 – Practice 9

Create a callback, that means create a function that takes a function (and a function value) as a parameter. Then pass a function (which e.g. outputs a string) and a value (for example "You shall not pass!") and make it execute.

Lecture 178 – Practice 9 – an example solution

Example: https://go.dev/play/p/d-mlKw_Ztgc

Lecture 179 – Practice 10

Closures "encapsulate" the scope of a variable in a block of code. Create a function with the identifier `andOneMoreIce` that increments a variable `soManyIces` within a returned anonymous function.

Create a variable with the identifier `spongbob` and a variable with the identifier `patrick`, to each of which you assign the function `andOneMoreIce` . Call the `spongbob` function three times and the `patrick` function twenty-three times.

Lecture 180 – Practice 10 – an example solution

Example: https://go.dev/play/p/pNYWVMEkynK

Quiz 6 – honeycomb of questions

Lecture 181 – Practice 11 Quiz 6 common solution

Section 16 – Pointers – they point at

Lecture 182 – Concept memory simplified …

> *The life of the dead is placed in the memory of the living.*
>
> *Marcus Tullius Cicero*

When it comes to pointers, new students tend to curl up under the table in a prefetal position sucking their thumbs or run constantly against walls in unguided overzeal. Good news: There is no need to worry about pointers in Go! In other programming languages pointer are like the Facebook status saying “It’s complicated.”, in Go it is not. Pointers are our friends and that is because Go abandons any pointer arithmetic. Well, so I can spare you that at least, but to understand what pointers are good for we should have a brief look at our RAM, so at the memory in our computer.

See also:
https://en.wikipedia.org/wiki/Pointer_(computer_programming)
https://en.wikipedia.org/wiki/Pointer_(computer_programming)#Go

Lecture 183 – Pointer – the unknown being

> *When the sage points at the moon, all that the idiot sees is the finger.*
>
> *Anthony de Mello*

Example of use of pointer, declaration, (de)referencing, type assignment:
https://go.dev/play/p/W37yF4jdCCq

Lecture 184 – When and how to use pointers

> *[…] any random function may write random values without having a clue where they point, has not been debunked as the sheer idiocy it really is.*
>
> *Erik Naggum*

Simple example without pointer: https://go.dev/play/p/cYJHFNdqJMi

Simple example with pointer: https://go.dev/play/p/x1KlPBLBz9t

Example with pointer and composite data type (struct): https://go.dev/play/p/Em-bfrTqNaR

Mutation / to mutate = change a value (by means of a pointer pointing to the location in memory where the value is located).

Lecture 185 – Method Sets – methods come in whole sets at once

> *A set is a Many that allows itself to be thought of as a One.*
> *Georg Cantor*

The method set of a type determines the methods that can be applied to an operand of that type. Each type has at least one (possibly empty) method set associated with it:

- The method set of a defined type T consists of all methods declared with the receiver type T.
- The method set of a pointer to a defined type T (where T is neither a pointer nor an interface) is the set of all methods declared with receiver *T or T .
- The method set of an interface type is the intersection of the method sets of each type in the type set of the interface (the resulting method set is usually just the set of declared methods in the interface).

Further rules apply for structs (and pointers to structs) that contain embedded fields, as described in the section on struct types. Every other type has an empty method set.

In a method set, each method must have a unique method name that does not contain spaces.

In my own words:

Method sets specify which methods are associated with a type. Thus, they are exactly what their name implies: the set of all methods that a type implements. What is the set of methods of a given type? It is its method set.

This results in four (+1) cases that we should have a look at:

We start with: https://go.dev/play/p/RE_aHvu4paB

Receiver and value are not pointers: https://go.dev/play/p/xHebbfODG_b

Receiver no pointer, but value passed as pointer (an address to that value): https://go.dev/play/p/6eJxQmL2qUT

Receiver and value are both pointers: https://go.dev/play/p/lsvuFsdBrQm

Receiver is pointer, but value is none: https://go.dev/play/p/owxiOWwEI5s **(Compiling fails!)**

> But this code works, because it is a direct call of a method: https://go.dev/play/p/qGB78AiPInu

Section 17 – Level 7 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 186 – Practice 1

Create a value and assign it to a variable.

Output the address where the value is stored.

Lecture 187 – Practice 1 – an example solution

Example: https://go.dev/play/p/WxcetDHpSDY

Lecture 188 – Practice 2

Create a type (identifier `person`) in the form of a struct with the elements

- givenname string
- familyname string
- age int
- address string

Create a function `change` with `*person` (pointer to type `person`) as parameter type.

In the function, change the value stored under `*person` in the address element.

Important: To dereference the struct.element use: `(*value).field`

`p1.address` and `(*p1).address` should be equivalent because:

> *"As an exception, if the type of x is a named pointer type and (*x).f is a valid selector expression denoting a field (but not a method), x.f is shorthand for (*x).f."*
> *https://go.dev/ref/spec#Selectors*

In Function `main()`

Create a valid value of type `person` .

Output the value.

Call `change` with passing the correct parameter to change the value.

Output the value.

Lecture 189 – Practice 2 – an example solution

Example: https://go.dev/play/p/sXFK2HFlW7q

Quiz 7 – this time it's all about pointers

Lecture 190 – Practice 3 Quiz 7 common solution

Section 18 – Application and the standard library – let's make something useful

Lecture 191 – JSON package documentation – read once saves a lot of debugging

> *6 hours of debugging can save you 5 minutes of reading documentation.*
> *Jakob @jcsrb (Twitter)*

Information about packages that belong to the so called "Standard Library" in Go can be found on: https://pkg.go.dev/std

Here we find in different categories (where necessary) packages that are "shipped" with Go, that means implemented in Go itself and available for import.

For example the package JSON from the category "encoding" we find under https://pkg.go.dev/encoding/json

Index: https://pkg.go.dev/encoding/json#pkg-index

Example: https://pkg.go.dev/encoding/json#pkg-examples

Functions: https://pkg.go.dev/encoding/json#pkg-functions

Types: https://pkg.go.dev/encoding/json#pkg-types and even the complete source code: https://pkg.go.dev/encoding/json#section-sourcefiles down to the most fundamental functions in different parts of Go (here only exemplary): https://cs.opensource.google/go/go/+/go1.18.1:src/encoding/json/encode.go documented down to the smallest detail and provided with profound comments.

Further general and Go-specific sources of information on JSON:

- https://en.wikipedia.org/wiki/JSON
- https://yourbasic.org/golang/json-example/
- https://medium.com/go-walkthrough/go-walkthrough-encoding-json-package-9681d1d37a8f
- https://golang.org/pkg/encoding/json/#Marshal

Lecture 192 – JSON marshal

> *Fortitude is the marshal of thought, the armor of the will, and the fort of reason.*
>
> *Francis Bacon*

https://pkg.go.dev/encoding/json#Marshal

Example: https://go.dev/play/p/vRNd7lpE0fa

Lecture 193 – JSON unmarshal

> *Everything you seek is inside of you.*
>
> *Joey Klein*

https://pkg.go.dev/encoding/json#Unmarshal

How to get a proper formatted struct in Go if you have JSON only: https://mholt.github.io/json-to-go/

Example: https://go.dev/play/p/OPe695t0Ib2

Lecture 194 – The Writer and the Reader interfaces – the names say it all

> *I love deadlines. I love the whooshing noise they make as they go by.*
>
> *Douglas Adams*

The whole section here is about examples how to make use of the standard library. We want to start to make use of it and become familiar to its inner structure. I started at somewhere looking for how JSON encode and JSON decode work and was looking for some examples and stumbled over our an important construct in the Go programming language: the Writer (respectively Reader) Interface.

So in this lecture I would like to let you participate and follow my footsteps to take a look at this by exploring some packages of the standard library and discover the magic of the concept and the power of the writer interface.

Brief recall what an interface is in Go and “in real life” by the example of different power plugs.

I started looking around for encode/decode. Before we had a look at JSON marshal/unmarshal on https://pkg.go.dev/encoding/json#pkg-functions

But also two types here called Type Encoder and Decoder:
https://pkg.go.dev/encoding/json#Encoder Here the encoder

That type has a function, better to say it implements a methods: The function Encode:
https://pkg.go.dev/encoding/json#Encoder.Encode

And I like to think about Encode and Decode as of a function is hardwired in it. Like you have some logic how to deal with JSON to struct and struct to JSON in a cable or an adapter.

So when marshal/unmarshal of some functions making use of that ability to bring some data into my program or leading some data out, so are Encoder and Decoder the direct versions the cabling with no values I have to assign marshal/unmarshal to just the blank wire.

And to do so you need a new Encoder and you can have one with the function NewEncoder. And see: https://pkg.go.dev/encoding/json#NewEncoder it needs a writer. So next ask yourself, what is a writer? A writer can be many things, but it does not write. But you can write to it. It can be a file, it can be a stream, it can be a web connection all that kind of things one can write to, or at, or into. And with a Reader, of course, it is the other way round . Stuff is coming in and your program can read from it: A Keyboard, a file, a Camera, whatever.

So the Encoder type is doing the writing out and marshal data to JSON, while the Decoder type is reading JSON and putting it in our program, well unmarshaling. We can really make also directly use of that. And probably you will do that in our exercise. Now I show you about the Writer (and maybe a little of the Reader) interface. And if you understand that and how to use them you can dive deep in the Go language design then you will be able to solve that practice as it will be similar.

We have a look in a very basic package. Package `io`: https://pkg.go.dev/io and pretty far below we have type Writer: https://pkg.go.dev/io#Writer It is not a typewriter, but a type called Writer. And as you can see that is an interface.

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

What we learned about interfaces? This interface says: Everything which has a method Write with `[]byte` as parameter type and returning in `int` n (number bytes written) and err an possible error that is also of type Writer. Reader of course the other way round but following the same logic.

But imaging how “pretty” that basically is: If you have the method write then you are a writer. No matter what else you can do. If I can write to you, you’re a writer. May also be you’re of whatever type, but this Interface says: You are a writer then, if you can write (as requested in the interface).

And remember that is exactly what NewEncoder needs to encode to: https://pkg.go.dev/encoding/json#NewEncoder an io.Writer. That means that everything which implements the type Write, is a Writer, thus we can encode JSON to, or? Yes, that is right!

Awesome it is!

But that’s not the end here. There is a rat tail of implications here – good ones.

Here, let’s look at package os: https://pkg.go.dev/os#pkg-index and here we see a type file. And it implements dozens of methods and one of them is Write: https://pkg.go.dev/os#File.Write

Look at the signature:

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

Exactly how the Writer interface expects that. What does that mean?

Anything which is of type file is also of type Writer. Now, remember what we learned about interfaces and pointers and receivers and how we can apply this knowledge with here in the standard library:

I can take a pointer to a file, and due to the fact it is also of type Writer because it implements the Writer interface we can place the pointer wherever a function asks for a type Writer.

Even here: https://pkg.go.dev/encoding/json#NewEncoder It ask for an io.Writer. If we have something of type file it is also of type Writer, because it implements the writer interface, we can take a pointer to a file and encode to it. That is the beauty of the standard library enfolding in front of you.

Interfaces are really really mighty tools and here you can also understand why this is called an interface. Remember our power sockets and plugs. It connects different types with each other to let data flow. And the methods are like that little plugs which you need to be able to connect them.

Same with reader and decoder:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

Everything with the method Read is also of type Reader because it implements the Reader interface.

Is that powerful? Interfaces are the Swiss army knifes with a leather man combined and attached to a well equipped toolbox.

Even in simple appearing functions like `fmt.Println()` you can you can find Writers realized.

Look at our little hello world program: https://go.dev/play/p/GaGKGQWiLxY

We could look in the standard lib and find https://pkg.go.dev/fmt#Println

If we like we can have a look at the source code:
https://cs.opensource.google/go/go/+/go1.18.4:src/fmt/print.go;l=273

`fmt.Println()` is only one line of code: `return Fprintln(os.Stdout, a…)`

All arguments are handled over to `fmt.Fprintln()` only and that is asking for something from called `os.Stdout`.

Let us look that up: https://pkg.go.dev/fmt#Fprintln and you see it takes a Writer. For me the F means file and from that perspective any Writer can go there. But you don't have to rely on my intuition. And also what that `os.Stdout` is: https://pkg.go.dev/os#pkg-variables

It's a variable. Crazy things going on here and we may not understand this, but see: it has NewFile. And what does NewFile do? What does it give us? https://pkg.go.dev/os#NewFile It gives us a pointer to a file. We don't need to understand what insane stuff if going on here in `os.Stdout`. But whatever it does is assigns a pointer to a file to `os.Stdout` and that is what? Right, it is a Writer.

`os.Stdout` implements the Writer interface. Because is is of type `*file` and that means it is also of type Writer.

It is totally legit to take that: `Fprintln(os.Stdout, a…)`

and use it on the playground like this:

```
fmt.Fprintln(os.Stdout, "Hello, 世界", 23, "and something else.")
```

That is totally okay to do so and `fmt.Println()` is doing the exact same thing.

Just sneaking around in io/ioutils you can find: https://pkg.go.dev/io/ioutil#WriteFile nice to have, or? This package io is full of writers https://pkg.go.dev/io and look at this one here: https://pkg.go.dev/io#WriteString

A Function taking a Writer, whatever writer, and allow us to write a string to it.

```
func WriteString(w Writer, s string) (n int, err error)
```

Let's test that: `io.WriteString(os.Stdout, "Hello, I'm a string only, mostly harmless.")`

Wow! That impressive, isn't it? (Try on your own here!)ww

Keep in Mind. We don't need our standard out for that, right? Any writer we can make write. We can write to files, we can write answers to requests on our web server, we can write to all Writers.

This is where the Alpha meets the Omega, beginning comes to an end and where the dog bites its tail. Remember where we came from. We checked out encode/decode from package encoding/JSON and got lost in the depth of the standard library. But as we looked around and saw all the wonders that the different packages have to offer and they're all kind of connected through the Writer and the Reader interfaces. That whole thing we're walking through here is a very detailed and complex figment, built and handled with care and well maintained. And it follows understandable principles which makes it easy to feel comforted here.

Did you realize that type interfaces are named after the activity they request to implement as a method? If something has the method Write, it's a Writer and if it has a method Read, it's a Reader. If it has both it is a ReadWriter and so on. And such things run through the entire design of the Go language. Remember that and with every visit of the standard library new miracles will be revealed before your eyes! Looking around in the standard library can be a revelation, it can be an epiphany.

Final example: https://go.dev/play/p/zLjYQAChGOp

Lecture 195 – Sort – simply sort

> *Gryffindor, where dwell the brave at heart!*
>
> *J.K. Rowling, Harry Potter and the Sorcerer's Stone*

Search on your own by exploring the standard lib: https://pkg.go.dev/std

Code to start with: https://go.dev/play/p/Y9pfq2SKZln

Code for sorting https://go.dev/play/p/-xs5jS4vKmd

Lecture 196 – Sorting – this time adapted to your own needs

> *Nothing like a little disaster for sorting things out.*
> *David Hemmings*

Code to start with: https://go.dev/play/p/L5mo9lAiddN

Look at the examples for package sort: https://pkg.go.dev/sort@go1.18.4#example-package

Example, sorting a composite type (struct)by age and by name:
https://go.dev/play/p/YdYQmDO9l_D

Lecture 197 – Bcrypt – let's take a look at some encryption (and decryption)

> Encryption works. Properly implemented strong crypto systems are one of the few things that you can rely on.
> Edward Snowden

More about bcrypt: https://en.wikipedia.org/wiki/Bcrypt

Necessary only if you want to run the example in your own development environment:

```
go get golang.org/x/crypto/bcrypt
go get -u golang.org/x/crypto/bcrypt
go env -w GO111MODULE=off
(go env -w GO111MODULE=auto) (for setting back)
```

Complete example with password conversion to a hash value using bcrypt and subsequent comparison of a password with the hash value (from Go Playground):
https://go.dev/play/p/tNtgXo4-_E9

Section 19 – Level 8 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 198 – Practice 1

Take the following code as a basis: https://go.dev/play/p/Rjnj9IXyJZZ

Create the values in the slice `[]user` in JSON and output that.

Help: Remember what you have to do to make a variable available outside its package!

Lecture 199 – Practice 1 – an example solution

Example: https://go.dev/play/p/WS_v3WxW5W3

Lecture 200 – Practice 2

Imagine: You get the following feedback in JSON after the function call to disable an antivirus program:

```
{"action":"stop","beta":false,"error":
{"code":0},"finished":true,"language":"enu","last_stage":"stopped"
,"package":"AntiVirus","pid":28386,"scripts":
[{"code":0,"message":"","type":"stop"}],"stage":"stopped","status"
:"stop","status_description":"translate from systemd
status","success":true,"username":"","version":"1.5.3-3077"}
```

Transform the contained data into a suitable structure in Go and then output the Boolean value that indicates success.

Help: https://mholt.github.io/json-to-go/

Lecture 201 – Practice 2 – an example solution

Example: https://go.dev/play/p/dYGebWnM-of

Lecture 202 – Practice 3

Take the following code as a basis: https://go.dev/play/p/d-Uwc9GGAju

"Encode" the value of type `[]user` into JSON and send the result to `Stdout`.

Help: Use `json.NewEncoder(os.Stdout).Encode(v interface{})`

Lecture 203 – Practice 3 – an example solution

Example: https://go.dev/play/p/97GggjLcV-I

Lecture 204 – Practice 4

Take the following code as a basis: https://go.dev/play/p/VZ_40FX_pkr

Sort`[]int` and`[]string`.

Lecture 205 – Practice 4 – an example solution

Example: https://go.dev/play/p/NBRo37C7x14

Lecture 206 – Practice 5

Take the following code as a basis: https://go.dev/play/p/Rxl3OsT38g6

Sort `[]user` by

- Name
- Age

Sort each `[]string` "sayings" of each user alphabetically.

Output everything clearly. For example, similar to:

Name, Age

- Saying1
- Saying2
- Saying3

etc...

Lecture 207 – Practice 5 – an example solution

Example: https://go.dev/play/p/LYF_8EfL1U1

Section 20 – Concurrency – feels like Go was made for

Lecture 208 – Concurrency versus Parallel Processing

> It is far easier to design a class to be thread-safe than to retrofit it for thread safety later.
>
> Brian Goetz, Software Developer (about Java)

Go was the first programming language to be developed after the widespread introduction of multiprocessor systems as a programming language with special parallel processing capabilities.

https://en.wikipedia.org/wiki/Parallel_computing

https://en.wikipedia.org/wiki/Multiprocessing

https://www.techspot.com/article/2363-multi-core-cpu/

https://en.wikipedia.org/wiki/Go_(programming_language)#History

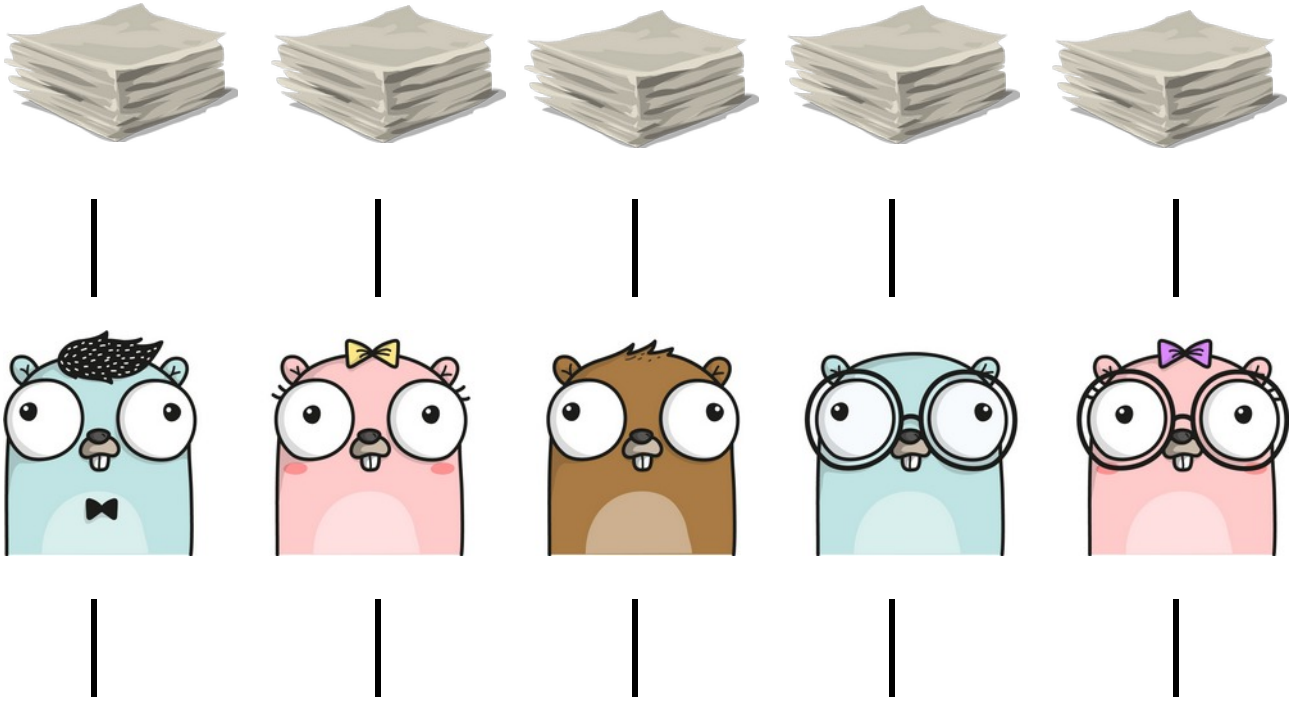
https://en.wikipedia.org/wiki/rGo_(programming_language)#Concurrency:_goroutines_and_channels

30 minutes, Rob Pike on concurrency and parallel processing in Go (worth watching):

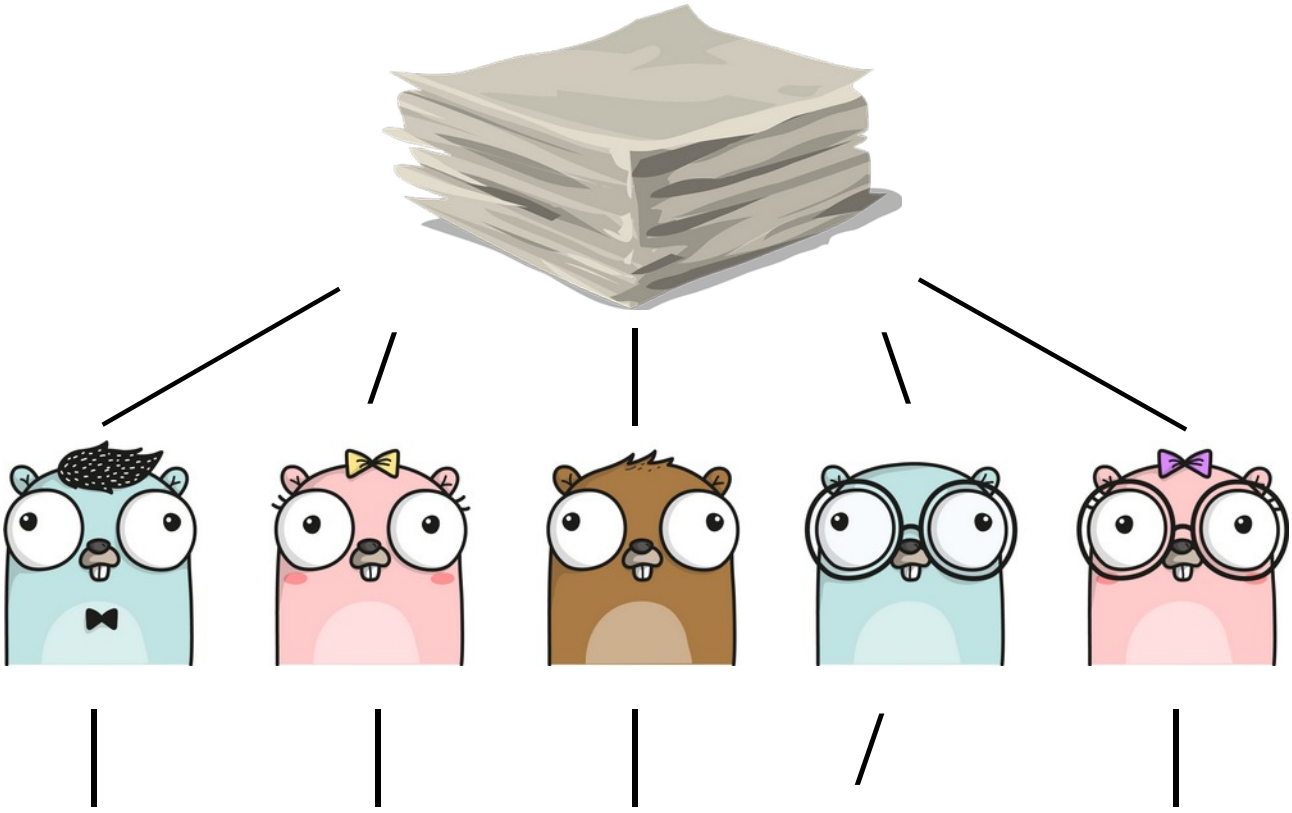
https://www.youtube.com/watch?v=oV9rvDllKEg

- Parallelism means multiple routines work off multiple job stacks
- Concurrency design pattern means multiple routines going (not necessarily on multiple CPUs) taking jobs from one stack

Parallelism



Concurrency



The scheduler of the operating system takes care of the distribution of the corresponding tasks to different Goroutines to (available) resources (like CPUs). The scheduler itself is not a Goroutine though

Lecture 209 – WaitGroup – Let's wait until they're done there

> *Patience is not simply the ability to wait - it's how we behave while we're waiting.*
>
> *Joyce Meyer*

A `WaitGroup` waits for a number of Goroutines to complete. The main Goroutines calls the `add()` method to set the number of Goroutines to wait for. Then each of the Goroutines is executed and calls `done()` when it finishes. At the same time, `wait()` can be used to block the program flow until all the Goroutines have finished. Writing concurrent code becomes super easy this way: We just need to put a "go" in front of a function or method call to make it its designated single running Goroutine.

From the package runtime we use in our example in the video:

`runtime.GOOS`

`runtime.GOARCH`

`runtime.NumCPU()`

`runtime.NumGoroutine()`

and from package `sync`:

`sync.WaitGroup (als Dateityp)`

with the methods:

```
func (wg *WaitGroup) Add(delta int)
```

```
func (wg *WaitGroup) Done()
```

```
func (wg *WaitGroup) Wait()
```

Initial code: https://go.dev/play/p/nTen45PX5rJ

and our completed WaitGroup: https://go.dev/play/p/idkJ8t-MdgY

Go func literal

A Goroutine started with the so-called “Go func literal” (an anonymous function trailing the keyword go) is something like create a task and put it on the task stack without creating a name. This is not even necessary, because such functions have no return value anyway, but organize the communication with each other, or with the higher-level Goroutine via channels (which we will learn about in a later section). Within these anonymous functions though we can of course also call functions which get return values. Our anonymous Goroutine can meanwhile serve as wrapper for the call of necessary sub-functions. Example is here an anonymous function, which is started by

prefixed keyword go as a Goroutine, and only in a sub-function opens a file and receives (and processes) an error message as return value.

Wait groups are elements of the control flow

In each job, several workers can work in parallel, but it must be ensured that certain intermediate goals are achieved. A foreman is needed.

It's like building a house. All workers can build on the basement, but before putting on the roof, you should wait for the first floor to be in place.

This requires orchestration to prevent different workers from waiting for each other to finish or to release resources. If that happens it is called a deadlock. We don't like deadlocks, so we work with wait groups, into which we put all the tasks we want to wait for to be completed before proceeding. Each worker shouts "Done" loudly when they are done and the foreman, crosses those off the list and **doesn't** continue until all the workers report their job as done.

Lecture 210 – Method Sets reloaded – this time you want to know

> Self-image sets the boundaries of individual accomplishment.
>
> Maxwell Maltz

The method set of a type determines the interfaces that the type implements and the methods that can be called by the receiver of the type.

Receiver is pointer, but value is none: https://go.dev/play/p/owxiOWwEI5s **(Compiling fails!)**

You need call it by a pointer value: https://go.dev/play/p/lsvuFsdBrQm

But this code works, note the difference by using `c.area()`!
https://go.dev/play/p/NMpgTOmeM5P

Lecture 211 – Concurrency – A look at the documentation

> I'm a hoarder. For me, documentation has always been key, and I've kept everything from my past.
>
> Diane Keaton

Effektive Go: https://go.dev/doc/effective_go#concurrency

go.dev: https://go.dev/ref/spec#Go_statements

Illustration from e-book: https://livebook.manning.com/book/go-in-action/chapter-6/56

About the term "multiplexing": https://en.wikipedia.org/wiki/Multiplexer ww

Lecture 212 – DIY Race Condition – If you don't have work, you create work for yourself

> Man is the only kind of varmint sets his own trap, baits it, then steps in it.
>
> John Steinbeck

Our own race condition: https://go.dev/play/p/u925EZYoipI

In the text editor one can also create a `race-condition.go` locally and bring it to the execution by `go run race-condition.go`.

The additional specification of a "build command" -race shows us race conditions found by the compiler: `go run -race race-condition.go`

Lecture 213 – Mutex – Let's just put a pad lock in front of it

> All of us are like locks. No matter how strong the bolt, there's always a key out there that opens it.
>
> Jonathan Kellerman

What is a mutex (more than mutual exclusion)? https://en.wikipedia.org/wiki/Mutual_exclusion

Wo finden sich Mutex und deren Methoden in Go? https://pkg.go.dev/sync@go1.18.2#Mutex

Our race condition removed by application of a mutex: https://go.dev/play/p/R_upKpgVHhI

Locally, `go run -race race-condition.go` also no longer finds race conditions.

Lecture 214 – The Package Atomic - is it going nuclear now?

> The way to win an atomic war is to make certain it never starts.
>
> Omar N. Bradley

The Atomic package can be found in the directory `sync` (which is a sub package): https://pkg.go.dev/sync/atomic#pkg-overview

Atomic brings its own methods for secure manipulation and reading of context for Goroutines:https://pkg.go.dev/sync/atomic#pkg-functions

Our race removed condition by applying two methods from the sync/atomic package: https://go.dev/play/p/YAGVrA9n9LI

Please note that the output of the amount of running Goroutines and the counter has been adjusted again. It is interesting to see that by using the scheduler of the operating system, the counter is not necessarily (!) counted up in the correct order. Note the "C:" values in the row at the output.

Section 21 – Level 9 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 215 – Practice 1

Start two additional Goroutines in addition to the main Goroutine.

Each additional Goroutine should output something.

Use `WaitGroup` to ensure that each Goroutine can be terminated as long as the program exists.

Lecture 216 – Practice 1 – an example solution

Example: https://go.dev/play/p/GWzJy1-Grlr

Lecture 217 – Practice 2

This practice is designed to deepen your understanding of method sets:

- create a struct of type `Person`
- with the help of a pointer receiver assign a method `speak` of this type `Person`: `*Person`
- Create an interface of type `Human` and require in it that a `Human` must implement the method `speak` to be considered of type `Human` .
- Create the function with the identifier `saySomething` that takes a value of type `Human` as a parameter.
- The function should call the `speak` method.

Represent in code:

- You can pass a value of type `*Human` to `saySomething`
- You cannot pass a value of type `Human` to `saySomething`
- You can call `valueOfTypePerson.speak()` without any problem!

In case you need help: https://go.dev/play/p/FMIL_L3TkaU

Lecture 218 – Practice 2 – an example solution

Example: https://go.dev/play/p/FMIL_L3TkaU

Lecture 219 – Practice 3

Using Goroutines, create a program that contains

- contains a variable that contains the value of the counter

and

- starts a series of Goroutines

Each Goroutine should read the counter, store it in a new variable, end the process request with `runtime.Gosched()`, increment the new variable, and write back to the counter variable. Use `WaitGroup` to wait for all your Goroutines to finish.

Create so a race condition and prove it by compiling the code with the "build flag" `-race` and run it on your local system.

In case you need help: https://go.dev/play/p/HBb29hoVefo

Lecture 220 – Practice 3 – an example solution

Example: https://go.dev/play/p/HBb29hoVefo

Lecture 221 – Practice 4

Use methods provided in package `sync` for the `Mutex` type to bypass the race condition from the code in practice 3.

It makes sense to remove `runtime.Gosched()`. Why is that?

Lecture 222 – Practice 4 – an example solution

Example: https://go.dev/play/p/BoMcTN_olQs

Lecture 223 – Practice 5

Use methods provided in Package `sync/Atomic` to bypass the race condition from the code in Practice 3.

Lecture 224 – Practice 5 – an example solution

Example: https://go.dev/play/p/Sy8BngwZY8Y

Lecture 225 – Practice 6

Create a small program that prints your current OS and CPU architecture to the console.

Transfer it locally to your computer and run it with:

```
go run
```

```
go build
```

Lecture 226 – Practice 6 – an example solution

Example: https://go.dev/play/p/6DGZUkaPPOj

Section 22 – Channels – no, it's not TV!

Lecture 227 – Introduction and explanation of channels

> We are not cisterns made for hoarding, we are channels made for sharing.
>
> Billy Graham

https://go-proverbs.github.io/

Don't communicate by sharing memory, share memory by communicating.(Link leads to video where Rob Pike explains the proverbs - worth watching)

https://en.wikipedia.org/wiki/Communicating_sequential_processes

Channels are similar to the channels of a walkie-talkie. There, you also agree to communicate on a channel and have to press and hold the talk button and confirm receipt of the message with copy/roger/over or whatever. Channels in Go are even simpler. You have to make sure that the sender and receiver exchange their information at the exact moment of communication, otherwise the program stalls because channels block. Channels block execution and that is a simple mechanism to ensure coordinated information transfer between Goroutines, basically between tasks, threads, or concurrently executing modules or independently running blocks of code.

https://go.dev/doc/effective_go#channels

https://pkg.go.dev/go/types#Chan

https://go.dev/ref/spec#Channel_types

Lecture 228 – Channels TL;DR; Channels block (they are just stubborn constructs!)

> Smiles form the channels of a future tear.
>
> Lord Byron

Channels block!

At least they do, when they’re full.

Simple example showing that an (unbuffered) channel blocks further program execution: https://go.dev/play/p/XX4anBZk4CI

Another example, that shows that an (unbuffered) channel blocks only the one concurrently executed Goroutine but not the further program execution: https://go.dev/play/p/oxrg0BhsoPr

And another example, that shows that a (buffered) channel starts blocking only the one concurrently executed Goroutine if its buffer ("capacity") is exceeded: https://go.dev/play/p/nPIflHQPDyz

Lecture 229 – Directional channels – give a direction to your channels' lives

> I don't make it in regular channels, and that's okay for me.
> Jim Carrey

Code to start with: https://go.dev/play/p/0-WQkafUUpZ

Receive-only channel, i.e. from this channel you can **receive** only values. https://go.dev/play/p/CoqjwiftcQy

Send-Only channel, to this channel we can **only send** values: https://go.dev/play/p/T_r_2qWyL8X

Examples of attempts to assign values to unidirectional channel from another unidirectional channel: https://go.dev/play/p/QbOg_8yv1V5 or https://go.dev/play/p/x9gEyARBap8 (**both fail!**) an to assign to a bidirectional channel, values from a unidirectional channel: https://go.dev/play/p/tEx4dh9xf1C (**also fails!**)

The right idea, but wrong syntax for conversion: https://go.dev/play/p/XLrEyaHBoJg (**wrong!**)

And finally, the **correct conversion of values from the bidirectional channel to values of the unidirectional channel type**: https://go.dev/play/p/M5cdLpGsQHD

Lecture 230 – Using channels – a kind of application example

> No one wants to have 300 channels on your wireless device.
> Lowell McAdam

Example of using a channel in a Goroutine that takes a directional channel as the type for its parameter, and runs concurrently as a "sender" (data input to the channel) to another Goroutine: https://go.dev/play/p/ZU8YqjYHftg or to the main program: https://go.dev/play/p/kZ9cP4wUM59

And here between two Goroutines, but with a `WaitGroup` ensuring the communication is visible: https://go.dev/play/p/9F3UX0TEXYF

Lecture 231 – Range & Close – get done and then close that

> The more channels putting money into quality programming the better.
>
> Kayvan Novak

Here, instead of using a "send-only channel" construct in a function, the data is given immediately into a Goroutine with an anonymous function and the channel is closed after sending. The data itself is retrieved from the channel as above with `range`: https://go.dev/play/p/PgXEj7RNruN

Lecture 232 – Select – Choose your favorite communication channel

> When we grew up, we had three channels on television and only one day of cartoons and if you missed it, you missed it.
>
> Butch Hartman

How does the `select` statement work? Similar to `switch`, but different from it, different communication situations for channels are distinguished in the cases.

Specs: https://go.dev/ref/spec#Select_statements

An example for using `select`: https://go.dev/play/p/ux7l2nA4_7a or https://go.dev/play/p/xJvb7sW-8MP

Lecture 233 – , ok – Hey, that's not comma okay!

> I was working on the proof of one of my poems all the morning, and took out a comma. In the afternoon I put it back again.
>
> Oscar Wilde

Insertion in advance starting from the last state of our example: https://go.dev/play/p/xJvb7sW-8MP

Experiment about closing channels: https://go.dev/play/p/Gjjbe9lA-3b

Corrected and with channel `end` type "chan bool" and slices in which we collect values from the channels: https://go.dev/play/p/4xlap_jbAgT This illustrates what happens when channels are not opened and closed "together", and so explains where the zero values come from. Play with the arrangement of the `close()` statements and the amount of data written to the channels.

Working version: https://go.dev/play/p/rBUFliJC78Q

Lecture 234 – Fan in – Channels built to a funnel

> I've always believed that you can funnel good things toward yourself by thinking positively.
>
> Jim Carrey

Code to start with from last lecture: https://go.dev/play/p/rBUFliJC78Q

Fan In: Data from different channels supplied by different Goroutines are merged into one channel.

Example 1: https://go.dev/play/p/P_3x85nvTMb

Example 2 (Rob Pike): https://go.dev/play/p/BvtrPEIH_fP

Source:

https://go.dev/blog/io2013-talk-concurrency (edited and improved by Andrew Gerrand), origin slides: https://go.dev/talks/2012/concurrency.slide#25 (Slide 25)

Lecture 235 – Fan out – Fly, my pretties, fly, fly!

> Fly, my pretties, fly, fly!
>
> Wicked Witch of the West, The Wizard of Oz (1939),
> actuality a misquote

Fan Out: A recurring task, is distributed amo2ng several concurrent Goroutines.

Example of a task distribution of similar tasks to concurrent processes: https://go.dev/play/p/rsRfpshgpCn (for example such a task could be to decode all videos in a folder)

Example of a task distribution of the same tasks to a limited number of concurrent processes (limitation of the "throughput"): https://go.dev/play/p/yU1ZJ0OocVg

Lecture 236 – Package Context – We give Goroutines a context

> For me context is the key - from that comes the understanding of everything.
>
> Kenneth Noland

(This has nothing to do with closures where context is encapsulated at the time of calling a function.)

The "context" package defines the Context type, which transfers deadlines, termination signals, and other request-specific values across API boundaries and between processes.

- `context.Background`: https://go.dev/play/p/cByXyrxXUf
- `context.WithCancel`

Discard`CancelFunc`: https://go.dev/play/p/XOknf0aSpx

Using `CancelFunc` :https://go.dev/play/p/UzQxxhn_fm

- Example: https://go.dev/play/p/Lmbyn7bO7e

  func WithCancel(parent Context) (ctx Context, cancel CancelFunc):
  https://go.dev/play/p/wvGmvMzIMWw

  cancelling Goroutinesby deadline
  func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
  https://go.dev/play/p/Q6mVdQqYTt

  by timeout:
  func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
  https://go.dev/play/p/OuES9sP_yX

  by value:
  func WithValue(parent Context, key, val interface{}) Context
  https://go.dev/play/p/8JDCGk1K4P

Other sources:
https://pkg.go.dev/context
https://go.dev/blog/context
https://medium.com/@matryer/context-has-arrived-per-request-state-in-go-1-7-4d095be83bd8
https://peter.bourgon.org/blog/2016/07/11/context.html

Section 23 – Level 10 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 237 – Practice 1

Get this code snippet working: https://go.dev/play/p/-DpZPo8o5JQ

a) with the help of a "go func literal" meaning an "anonymous" function directly called as Go routine

or alternatively (!):

b) by using a buffer in the channel.

Lecture 238 – Practice 1 – an example solution

Examples:

a) https://go.dev/play/p/SHr3lpX4so

b) https://go.dev/play/p/Y0Hx6IZc3U

Lecture 239 – Practice 2

Get these code snippets working:

a) https://go.dev/play/p/_DBRueImEq

b) https://go.dev/play/p/oB-p3KMiH6

Lecture 240 – Practice 2 – an example solution

Examples:

a) https://go.dev/play/p/BhhgKXOYAgA

b) https://go.dev/play/p/QHxrG8UEiuq

Lecture 241 – Practice 3

Start with: https://go.dev/play/p/LOuSO5msenD and get the values from the channel using a loop using "range" (including outputting them).

Lecture 242 – Practice 3 – an example solution

Example: https://go.dev/play/p/gtrDPXhmSpn

Lecture 243 – Practice 4

Start with: https://go.dev/play/p/AdSZFm_LQYl and get the values from the channel using a with `select` statement (including outputting them).

Lecture 244 – Practice 4 – an example solution

Example: https://go.dev/play/p/GF3mwS8qhaS

Lecture 245 – Practice 5

Start with: https://go.dev/play/p/WKQQwMY4B2- and use a ", ok" (comma okay) statement twice (before and after the `close()`) to show that the channel is empty and no more values come from the channel.

Lecture 246 – Practice 5 – an example solution

Example: https://go.dev/play/p/vec88muf3Y6

Lecture 247 – Practice 6

Write a program that writes (sends) 100 values to a channel and then receives and outputs all values from that channel.

Lecture 248 – Practice 6 – an example solution

Example: https://go.dev/play/p/DBvNT2Ixyhp

Lecture 249 – Practice 7

Write a program that started 10 Goroutines and have each Goroutine write 10 numbers to a channel. Receive all 100 values from the channel.

Lecture 250 – Practice 7 – an example solution

Example:

a) https://go.dev/play/p/0t7rIve7bud

or

b) https://go.dev/play/p/wlfiiJarmF-

or

c) https://go.dev/play/p/WqYnBC_CiKn

Section 24 – Error handling – if an issue occurs, handle it

Lecture 251 – Overview: Understanding the need for error handling

> Knowledge rests not upon truth alone, but upon error also.
> Carl Jung

Errors? Error Handling?

We're aiming at writing code without errors. So after debugging our code we're fine. We just write code without any errors, or?

Wrong! This is based on the assumptions that we're perfect and that there is only one kind of possible errors, but that is wrong.

Errors are underestimated. There is more information in error-prone code than in error-free code. Errors have value.

Syntax errors are caught and eliminated as far as possible. Logic errors though, which are however syntactically permitted, are for example a value to pass, where a Pointer would be necessary or vice versa. Often byzantine errors that cannot be recognized immediately.
https://en.wikipedia.org/wiki/Byzantine_fault

Runtime errors like "devision by zero" and other exceptional errors like a file does not exist when we want to write to are quite different. A special class of this runtime errors are exceptions. Errors that should not happen, but where the compiler could not determine a state at runtime that they might occur, nor evaluate whether they are intentional or not. Access to unallocated memory for example is such a case. How would the compiler know if an access is intended or not? But if there is a statement that breaks our program, it is an exception. But that can also be very simple one like "the printer is out of paper". This is an error message which comes from our program and we as the programmers have to take care of the cases which might occur – at least as many as possible.

https://news.ycombinator.com/item?id=6233968

https://go.dev/play/p/aCraGRTaFi-

https://go.dev/play/p/0Z-2VXFvZ-l

https://go.dev/play/p/HFzgX2VWx5t

Go does not know exceptions. Here is why:

The Go FAQ say: https://go.dev/doc/faq#exceptions

Quora says: https://www.quora.com/Why-does-Go-not-have-exceptions?q=why%20does%20go%20not%20have%20exce

Errors are values and have value: https://blog.golang.org/errors-are-values

Handle errors where they potentially occur. And not recognize, write a note and collect that notes at a central place to take care later. If my children are playing with Lego and I don't want to step on one in the dark, I should also handle that situation in time and give my kids a hint while they are playing there. So they can cleaning up. I will and not write down the danger, wait until the evening and circumnavigate the potential dangers.

Description:
https://go.dev/ref/spec#Errors

https://pkg.go.dev/errors

https://go.dev/doc/effective_go#errors

Proverbs: https://go-proverbs.github.io/

> *Errors are values.*
>
> *Don't just check errors, handle them gracefully.*
>
> *Golang proverbs*

Lecture 252 – Checking for errors means check and also handle

> Any man can make mistakes, but only an idiot persists in his error.
> Marcus Tullius Cicero

Wherever possible, you should check for errors unless you are going on some sort of "infinite loop". The final decision on how deep your error checking and handling should go is up to you, but with functions or methods like `fmt.Println()`, are assumed that they will not produce an error while outputting an error. Otherwise you would have to check for errors again when outputting the error, etc... But you can certainly do that in Go.

Example 1: https://go.dev/play/p/7CWqDNg-EUP

Example 2 (from package `fmt scan`, does not “run” on playground, but shows error handling): https://go.dev/play/p/z1UETDUmTi3

Example 3 (write file, does not run on Playground): https://go.dev/play/p/Kd-2dtSJnp6

Example 4 (read file, does not run on Playground): https://go.dev/play/p/nRVgx_vQdNy

Lecture 253 – Error output and write to log files

> An error doesn't become a mistake until you refuse to correct it.
> Orlando Aloysius Battista

For handling error messages we have a small selection of options for direct output, writing to log files and error messages.

`fmt.Println()`

Output during program execution on console `(stdout)`

Example 5: https://go.dev/play/p/aA_RDoZJxt3

`log.Println()`

Output during program execution on console (default: `stdout`) or to a file. A timestamp is prepended.

Examples 6 and 7:

https://go.dev/play/p/4IbHLaWz4F7 (default stdout)

https://go.dev/play/p/z_6rXow39Vr (redirected to a file)

`log.Fatalln()`

In case of a fatal error, it is written to the log file, but the parent code block is exited directly. `os.Exit(1)` means program termination! No execution of deferred functions!

Example 8: https://go.dev/play/p/x0QJjAOj4Em

`log.Panicln()`

deferred functions are still running.

- `panic()`
- "recover" can be called (see next lecture)

Example 9: https://go.dev/play/p/3MTDBJYq1PO

All examples as they have been used in this lecture in github:
https://github.com/jagottsicher/myGoError-Handling

As preparation for the next lecture:

https://go.dev/ref/spec#Run_time_panics

https://go.dev/ref/spec#Handling_panics

Lecture 254 – Recovering – recovering from errors

> Better to trust the man who is frequently in error than the one who is never in doubt.
>
> Eric Sevareid

Basics in the specs:

https://go.dev/ref/spec#Run_time_panics and https://go.dev/ref/spec#Handling_panics

Recover works: https://go.dev/blog/defer-panic-and-recover

Example defer: https://go.dev/play/p/_JGy1gxEb8u

Example for dealing with `defer()` and `ownError` https://go.dev/play/p/Ujf1dRatTMb

Lecture 255 – Errors can come with greetings

> Be precise. A lack of precision is dangerous when the margin of error is small.
>
> Donald Rumsfeld

We can add additional information to our errors.

- `errors.New()` und `fmt.Errorf()`
- `builtin.error`

> Error values in Go aren't special, they are just values like any other, and so you have the entire language at your disposal.
>
> Rob Pike

Examples:

`errors.New()`: https://go.dev/play/p/hhA1_o6VeUl or error in one variable: https://go.dev/play/p/ZH4iDeY3gnV

`fmt.Errorf()`: https://go.dev/play/p/9h-WwonU6V7

`fmt.Sprintf`, in a variable to generate an error message and a struct containing an error: https://go.dev/play/p/r3Vkhc7Ope0

Section 25 – Level 11 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 256 – Practice 1

Start with this code snippet: https://go.dev/play/p/fg678yRAfJz

Instead of using the underline identifier `_` to discard the error information, check for errors in the return value and handle the error appropriately.

Lecture 257 – Practice 1 – an example solution

Example: https://go.dev/play/p/PiYKM8hoymH

Lecture 258 – Practice 2

Start with this code snippet: https://go.dev/play/p/BzKj25Kt0f-

Create an error message and use `fmt.Errorf()`

Lecture 259 – Practice 2 – an example solution

Example: https://go.dev/play/p/fvGiqNDqpgR

Lecture 260 – Practice 2 – more solutions

More example solutions:

https://go.dev/play/p/7Eu2FEfFgpS

https://go.dev/play/p/PfQWW1aNVMa

Lecture 261 – Practice 3

Create a type `ownError` with underlying struct that implements the `builtin.error` interface. Create a function `foo()` that takes a value of type `error` as parameter. Then create a value of type `ownError` and pass it to `foo()`.

Lecture 262 – Practice 3 – an example solution

Example: https://go.dev/play/p/KRJhYSRHV7c

Also legit: https://go.dev/play/p/U9PD9gTItYw and little reminder what conversion is: https://go.dev/play/p/XE6zjL4DhDM

Lecture 263 – Practice 4

Start with this code snippet: https://go.dev/play/p/KxtVZlxsDCq

Use the `rootAcheError` struct as a value of type `error`.

If you like, use latitude "31.224361 N" and longitude "121.469170 W" as other values.

Lecture 264 – Practice 4 – an example solution

Example: https://go.dev/play/p/53Rojm_H9bG

Section 26 – Writing documentation – think about others!

Lecture 265 – Introduction and overview

> Lack of documentation is becoming a problem for acceptance.
>
> Wietse Venema

Software is read much more often than it is written or re-written. Incomplete, poor or missing documentation leads to the following grievance in software development.

A developer creates a function and changes it for various reasons. There is an urgent need for the implementation or pressure from the team (especially in agile software development) is so big that the documentation of one's own work is often put on the back burner or even completely disregarded. Later, this takes revenge when a successor is confronted with the same or similar task and has to read and understand the code. The time it takes to understand undocumented code often exceeds the time to try to rewrite a small module - again without documentation, of course.

This leads to a seemingly endless chain of undocumented code that no one can track as an individual. But the “one programmer working as a lone wolf” practically doesn't exist anymore

anyway, instead programmers are more likely to be found at system houses, gaming industry, hardware manufacturers, the big software forges or even enterprises like Google, Microsoft, Meta, Amazon, Netflix etc... There, the value and importance of complete and simple to achieve documentation was early recognized. The habit of equipping one's code here and there with good comments quickly became a virtue.

And Go makes an art out of this virtue. In Go the functionality is practically built in already to extract useful documentation from well maintained comments. For this, there is only a minimum of effort and a few requirements to fulfill and simple agreements to keep. The reward is a level of documentation that makes it possible to understand code down to the last little function and make that information available to everyone.

You have to differ 3 similar named entities of difference use.

1. godoc.org was once a place on the web where the standard library **and** third-party packages were managed and documented. And there was **golang.org** where the standard library alone was documented and maintained. Go is relatively young and you may come across this domains. But they are merged into **go.dev**, respectively, **pkg.go.dev**. (Also the playground used to be on play.golang.org and is now on go.dev/play) and links can be used that way.

2. `go doc` which is a command to read the documentation on the console.

3. `gordoc` was once used to read the documentation on the console and can furthermore make the documentation available locally in the browser but nicely (as HTML) via a self-started web server. Today the latter only it is used as far as I know.

In this section I want to lead you through both reading and writing good documentation.

Lecture 266 – Go doc – everything you need on a terminal

> I think it's important to have some documentation of the past.
>
> Henry Rollins

go doc returns the documentation from for package, constants , functions2, types, handling variables and methods.

go doc takes no, one or two arguments.

No argument: Prints the documentation for the package in the current directory

One argument as a Go-syntax-like representative of the element whose documentation you want to see.

Examples (<sym> means here "identifier")

- `go doc <pkg>`
- `go doc <sym>[.<method>]`
- `go doc [<pkg>.]<sym>[.<method>]`
- `go doc [<pkg>.][<sym>.]<method>`

The following applies: The first successfully found element in this list is displayed. If there is a <sym> but no package, the package in the current directory is displayed. However, if the argument starts with an uppercase letter, it is always assumed to be a <sym> in the current directory.

Two arguments

First argument must be a complete path to a package.

Example: `go doc <pkg> <sym>[.<method>]`

Lecture 267 – Godoc – documentation worth looking at

> Incorrect documentation is often worse than no documentation.
> Bertrand Meyer

Godoc extracts and generates documentation from and for Go programs. It used to support two modes:

- **with -http flag**

  Starts a webserver and presents the documentation on a website

  Example (simple): `godoc or godoc -http=:8080`

  Starts a web server accessible on http://localhost:6060 respectively http://127.0.0.1:8080

  Example (with searchable index) `godoc -http=:8080 -index`

  Example for use with activated playground: `godoc -http=:8080 -play`
  and test with playground support for the code examples (examples):
  http://localhost:8080/pkg/encoding/json/#example_Unmarshal

(Outdated AIK, here only for historical reasons and not explained in detail.)

- **without -http flag**:

  command line oriented mode
  prints text documentation and exits
  -src flag: godoc prints the exported interface of a package in go-source form or the implementation of a particular exported language

Lecture 268 – pkg.go.dev – the package documentation formerly known as godoc.org

> Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.
> Dick Brandon

pkg.go.dev (formerly godoc.org)

An example for a simple Go code that we can publish in a repository on github:

`meaning`

```
├── deepThought
│   └── deepThought.go https://go.dev/play/p/u5UHOmBY02o
├── LICENCE
├── main.go https://go.dev/play/p/BQaOzDrGTBg
└── README.md
```

Either you create the complete file structure as a project in your Go path in the src subfolder or you create a symbolic link (symlink) from there in your project folder (in MS Windows environment: https://www.howtogeek.com/howto/16226/complete-guide-to-symbolic-links-symlinks-on-windows-or-linux/), to make the data available for Go (and thus for godoc).

Syntax for Symlinks under Linux: `ln -s /destinationFile/orDirectory /Reference/orDirectory`

More about symbolic links: https://en.wikipedia.org/wiki/Symbolic_link

This project can now also be made available on github.com in its own repository.

If you copy the externally accessible URL (without leading http://, respectively, https:// but possibly from our repository on github.com) to your Go-Code behind the URL https://pkg.go.dev/ the documentation appears after a refresh in the results of the search. Sharing is caring!

Lecture 269 – Writing documentation – Spoiler: it's easy peasy!

> Good code is its own best documentation.
>
> Steve McConnell

Documentation is an important part of making software accessible and maintainable. It should be well written and accurate, but it must also be easy to write and maintain. Ideally, it should be tied to the code itself, so that the documentation evolves along with the code. The easier it is for programmers to create good documentation, the better for everyone.

> https://blog.golang.org/godoc-documenting-go-code

godoc analyzes the Go source code - including comments - and creates the documentation as HTML or plain text. The end result is documentation that is closely linked to the documented code. Using godoc's web interface, for example, you can navigate from the documentation of a function to its implementation with one click.

Comments are good comments when you would want to read even if godoc did not exist.

To document the following:

- packages
- constants
- functions
- types
- variables

- methods

You write a comment directly before the declaration without leaving a blank line.

You always start with the name of the element you want to comment.

To packages applies:

- the first line appears separately in the package list
- If you have a large amount of text, it is better to create the documentation in a doc.go file (example is the package fmt).

The best thing about godoc's minimalist approach is how easy it is to use. By far the majority of official sources, including the entire standard library, as well as commonly available third-party code, follow these conventions.

Section 27 – Level 12 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 270 – Practice 1

Create a package `dog`. The package should have an exported function `Years()` which converts dog years to human years (1 human year = 7 dog years). Document the code with comments.
Use this code in your function `main()` in `main.go` to import the package:
https://go.dev/play/p/QX4vZSoWxGF

a) Run your program and make sure it works

b) Run a local server with `godoc` and look at your documentation.

Lecture 270 – Practice 2

Publish the code on github. Get your documentation on pkg.go.dev and take a screenshot of just enjoy it. Delete your code from github (if you like). Check on pkg.go.dev if your code is no longer available there. Stop pollution - https://go.dev/about#removing-a-package

Lecture 271 – Practice 1 & 2 – an example solution

Example: https://go.dev/play/p/34pupI6uxz9

Lecture 272 – Practice 3

Use the command `go doc` on the command line to read the documentation for for:

- fmt
- fmt print

- strings
- strconv

Lecture 273 – Practice 3 – – an example solution

Section 28 – Tests and benchmarks

Lecture 274 – Introduction and overview of tests and benchmarks in Go

> Good tests kill flawed theories; we remain alive to guess again.
>
> Karl Popper

A brief introduction in tests in Go.

Tests require

- a file whose filename ends with `_test.go`
- in the same folder/package as the element/code to be tested and **must** belong to the same package.
- the test runs in a function with a signature like `func TestAbc(*testing.T)`

Execute a test with

```
go test
```

To handle the result of the test we use `t.Error` to signal the error.

Usually in the form, "Expected x, got y." or "Got this, expected that."

Example in video: https://go.dev/play/p/I3e05Cc-gF-

Lecture 275 – Table tests – testing as if on an assembly line

> Life tests our willingness, in ways large and small, to tell the truth.
>
> Jon Lovett

Example in video: https://go.dev/play/p/8oSlRFGoxdk

Lecture 276 – Examples allow the combination of documentation and tests

> You can't do a fine thing without having seen fine examples.

> William Morris Hunt

Tests can be found in the documentation as examples.

Locally you can have a look at after start of: `godoc -http=:8080`

Further information: https://go.dev/blog/examples

Example in video: https://go.dev/play/p/IX8BhOgLVQC

Package with testfile and example on pkg.go.dev:
https://pkg.go.dev/github.com/jagottsicher/myGoMeaning

Lecture 277 – Staticcheck: More beautiful and easier (to lint code is so from 2015)

> *Programming is usually taught by examples*.
> Niklaus Emil Wirth, Inventor of Pascal

When code is "cleaned up" according to certain criteria and virtually freed from bad style, this is called "linting". Often this is a thankless task, but fortunately there are so-called linters out there. Golint used to be one option, but is not developed further:
https://pkg.go.dev/golang.org/x/lint#section-readme.

golint differed from `go fmt` in the respect that it goes beyond pure formatting like indentation, giving advice and hints on how to improve the code. But today, tools like https://staticcheck.io/ (on github: https://github.com/dominikh/go-tools) should be used instead, and many IDEs (including VS Code) bring many features for linting already.

`go vet` goes one step further and points out suspicious rlooking constructs and structures, and helps to simplify code further.

Lecture 278 – Benchmarks/BET: We set a bad example

> I'm not trying to be sexy. It's just my way of expressing myself when I move around.
> Elvis Presley

Our example, from which we want to create a benchmark:

```
myGoBenchmarking
├── LICENSE
├── main.go https://go.dev/play/p/6nQ01KGcwZb
├── myConcat
│    ├── myConcat.go https://go.dev/play/p/Zl8VZf9lHNr
│    └── myConcat_test.go https://go.dev/play/p/6UGdL9m92Gp
└── README.md
```

Lecture 279 – Benchmarks/BET: Let the games begin!

> The benchmark of quality I go for is pretty high.
>
> Jimmy Page

Comparison of our simple `Concat` implementation with the `strings.Join` function and subsequent performance comparison.

```
myGoBenchmarking
├── LICENSE
├── main.go https://go.dev/play/p/ERuh6akrOqS
├── myConcat
│   ├── myConcat.go https://go.dev/play/p/wWFv5PZXlhM
│   └── myConcat_test.go https://go.dev/play/p/GLIBGkwDRRK
└── README.md
```

Lecture 280 – About the coverage of Go code in tests

> Everything has to be well thought out - what do you really need, when can you do with less coverage.
>
> Debbie Allen

Coverage in programming means how much of our code is covered by tests.

With the `-cover` flag a test coverage analysis is started.

With `-coverprofile <anyName>` an analysis is written to a file.

Examples:

```
go test -cover .
go test -coverprofile analysis.txt
go tool cover -html=analysis.txt
```

More information: `go tool cover -h`

Edited myConcat_test.go: https://go.dev/play/p/TtzVdBbiYBf

Lecture 281 – BET summary

> I am unable to think of any critical, complex human activity that could be safely reduced to a simple summary equation.
>
> Jerome Powell

Think about BET:

- Benchmark

- Example
- Test

Create test files ending in `_test.go` and import package `testing`:

`func BenchmarkYourIdentifier(b *testing.B)`

Useful:

`b.ResetTimer()` ((re-)set the timer for the benchmarks)

`b.N` (number of benchmark runs, for a statistically relevant average)

`func ExampleYourIdentifier()`

```
// Output:
// Copy expected output here!
```

`func TestYourIdentifiert(t *testing.T)`

```
t.Error("Expected:", x, "got:", y)
```

or `t.Errorf("%v", x)`

Commands:

```
godoc -http=:8080
go test .
go test -bench .
go test -cover .
go test -coverprofile testname.txt
go tool cover -html=testname.txt
```

Section 29 – Level 13 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 282 – Practice 1

Given is a Go project with the following structure:

```
myGoBETpractice
├── main.go https://go.dev/play/p/ikrP1kvLoHw
└── mathUtils
    └── mathUtils.go https://go.dev/play/p/eexQ-DkRuNf
```

You can get it from github: https://github.com/jagottsicher/myGoBETpractice

Make sure that the package `mathUtils` is in your path.

a)
Run `main.go` and add a suitable test file in and for the package `mathUtils`.

b)
Add a benchmark for the `TheSumOf` function in the package `mathUtils`.

In the folder of the package `mathUtils`, run:

```
go test .
go test -bench .
```

c)
Implement the following code in Package `mathUtils`:

```
func addTwoIntegers(a, b int) int {
     return a + b
}

func SumUpIntegers(inputValues ...int) int {
     var sum int
     sum = inputValues[0]
     for _, v := range inputValues[1:] {
          sum = addTwoIntegers(sum, v)
     }
     return sum
}
```

and remove the comment on line 13 in `main.go` in the parent folder.

Execute `main.go`.

d)
In the folder of the package `mathUtils` run:

```
go test .
go test -bench .
```

Add another benchmark, but this time for the `SumUpIntegers` function in the `mathUtils` package.

In the folder of the package `mathUtils` run:

```
go test .
```

```
go test -bench .
```

e)
Provide file `mathUtils.go` with documentation in the form of comments for

- the package `mathUtils`

- The function `addTwoIntegers`

- The function `SumUpIntegers`

Start a server with

```
godoc -http=:8080
```

and go on the site http://localhost:8080 in your browser. Find the package `mathUtils.`

Stop the "godoc-webserver"

f)
Add an example for each of the calls of the functions `addTwoIntegers` and `SumUpIntegers`. The values 1, 2 and 3 should be passed as arguments and the output should be "Sum 6".

Execute `go test` and make sure that the examples and the tests pass.

Start

```
godoc -http=:8080
```

and call http://localhost:8080 in the browser and find the two examples in the package `mathUtils`.

Enjoy the examples in the documentation and stop the "godoc-webserver".

g)
Add a test for each of the calls of the functions `addTwoIntegers` and `SumUpIntegers`.

Test the values 1,2 and 3 as arguments against the result 6 and provide for a proper output in case of "FAIL" like "Expected: x, got y" or similar.

Execute `go test` and ensure that the tests pass.

h)
In the folder of the package `mathUtils` run:

```
go test -cover .
go test -coverprofile filename
go tool cover -html=filename
```

Add one more test for each of the calls of the functions `addTwoIntegers (CHANGED TO TheSumOf)` and `SumUpIntegers`.

This time test the values according to the following table:

```
Arguments                 Result
-----------------------------------------
1, 2, 3, 4, 5, 6          21
1, 2, 4, 8, 16            31
1, -2, 3, -4, 5           3
-5, 1, 1, 1, 1, 1         0
6, 5, 4, 3, 2, 1          21
```

Ensure proper output in case of "FAIL" like "Expected: x, got y" or similar.

Run `go test` and make sure that all tests pass.

In the folder of the package `mathUtils` run:

```
go test .
go test -bench .
go test -cover .
go test -coverprofile filename
go tool cover -html=filename
```

i)
Be happy about a successful solution for BET (**B**enchmark, **E**xample, **T**est) in an adequate way

Lecture 283 – 291 – Practice 1 a) – i) – an example solution

You can find the complete code of my solution on github:
https://github.com/jagottsicher/myGoBETpracticeSolution

Section 30 – Package Management & Go Modules

Lecture 292 – Package Manager and the thing with the dependencies

> To me, DIY means minimizing dependency on what others make for me.

> Fumio Sasaki

https://research.swtch.com/deps

PDF is available here: https://research.swtch.com/deps.pdf

https://en.wikipedia.org/wiki/Package_manager

Lecture 293 – How to use Go modules – general instructions

> [...] Google's internal source code system, which treats software dependencies as a first-class concept,...
> https://research.swtch.com/deps

https://en.wikipedia.org/wiki/First-class_citizen

Source which we shimmy along: https://go.dev/blog/using-go-modules and more detailed information in the following parts of this series of blog posts. The series provides a comprehensive introduction to dependency management with Go Modules.

Lecture 294 – Create a Go module yourself

> It takes half your life before you discover life is a do-it-yourself project.
> Napoleon Hill

https://go.dev/blog/using-go-modules:

`go mod init` creates a new module and initializes the `go.mod` file that describes the module.

We end up with this Code: https://github.com/jagottsicher/myGoModules-01-Start

Lecture 295 – Add dependencies to a Go module

> The beginning is the most important part of the work.
> Plato

https://go.dev/blog/using-go-modules:

`go get` changes the required version of a dependency and/or adds a dependency.

`go build, go test` adds new dependencies to the `go.mod` as needed.

`go list -m all` outputs all current dependencies of the current module.

Lecture 296 – Dependencies upgrade/fulfill/downgrade

> Roads do not upgrade or maintain themselves. Bridges do not repair themselves or rebuild themselves.
>
> Martin O'Malley

https://go.dev/blog/using-go-modules:

`go get domain/folder`

`go get domain/folder@versiontag` modifies the required version of a dependency and/or adds a dependency.

`go list -m -versions domain/folder` outputs a list of all available versions of a module.

`go mod tidy` compares the code of our Go module with the dependencies required up to that point from `go.mod` and removes dependencies and modules that are no longer needed, adds new ones if necessary.

Domain not necessarily means an internet domain. It can also be a package name or project folder.

Actual stae of our example for working with Go modules:
https://github.com/jagottsicher/myGoModules-02-Practice

Section 31 – Level 14 – Karate, Kung-Fu, Voodoo, Mojo, Magic, the Force & Skill

Lecture 297 – Practice 1

They can find the current code from the previous lessons at
https://github.com/jagottsicher/myGoModules-02-Practice

With `git clone git@github.com:jagottsicher/myGoModules-02-Practice.git` you can download it to your computer, but be aware that it should be place outside of `~/name/go/src` , respectively, outside of your `GOPATH`.

a)
Perform an upgrade in the project after a major version jump, as described in https://go.dev/blog/using-go-modules#upgrading-a-dependency-to-a-new-major-version.

b)
Remove dependencies that are no longer used as described in https://go.dev/blog/using-go-modules#removing-unused-dependencies .

Use `go mod tidy`.

Lecture 298 – Practice 1 a), b) – an example solution

Example afterwards: https://github.com/jagottsicher/myGoModules-03-Practice-Solution

Section 32 – Goodbye and Farewell – live long and prosper!

Lecture 299 – You did it - now celebrate!

> The more you praise and celebrate your life, the more there is in life to celebrate.
>
> Oprah Winfrey

Lecture 300 – Beyond the horizon it may already be waiting for you…

> We have always held to the hope, the belief, the conviction that there is a better life, a better world, beyond the horizon.
>
> Franklin D. Roosevelt