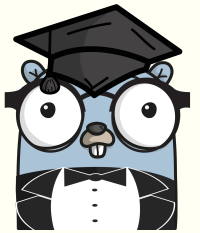
INTRODUCTION TO PROGRAMMING IN GO

Pascal van Dam

May 6, 2022



Introduction

Trainer & Student Introduction

ABOUT THE TRAINER

ABOUT THE STUDENT

- Introduce yourself shortly
- Do you have any experience with
  - Go
  - Other programming languages
  - Kubernetes
  - Linux

ABOUT THIS COURSE

This course:

- Is developed for professionals that are already familiar with a programming language and would like to get up & running with Go.
- It will introduce you to the essentials of the Go programming language
- Enables you to write programs in an idiomatic way in Go
- Introduces you into the rites and habits of a Gopher

CERTIFICATION

Currently there is no official certification program for Go

If you want to standout in the crowd, solve a non trivial problem (OpenSource) and publish it on github or gitlab

Agenda day 1/3 part 1

- Introduction to the course
- Introduction to Go
- Philosophy behind Go
- Install and configure your Go environment
- Choose and install your tools
- Go packages

Agenda day 1/3 part 2

- My first Go program
- Basic buildingblocks
- Reserved words and names
- Go's built-in library
- Declaring variables and constants
- Lifetime and scope
- Pointers in Go

Agenda day 1/3 part 3

- Basic Data Types in Go
- Numbers, strings and booleans
- Go and it’s foundation in UTF-8
- String handling
- Iota alias the Constant generator

Agenda day 1/3 part 4

- Complex datatypes
- Arrays
- Slices
- Maps

Agenda day 1/3 part 5

- Structs
- Literal structs
- Handling structs
- Embedding structs
- Anonymous fields in structs

Agenda day 1/3 part 6

- Functions
- Declaring and calling
- Variadic functions
- Recursive functions
- Function objects
- Anonymous functions
- Deferred functions

Agenda Day 2/3 Part 1

- Packages in Go
- Kinds of Packages

Agenda day 2/3 part 2

- Modules in Go
- History of modules
- Semantic Versioning
- How create your own module

Agenda day 2/3 part 3

- Error handling
- Panic and recover
- Patterns and best practices

Agenda Day 2/3 Part 4

- Object-Oriented programming in Go
- What is Object Oriented Programming
- What is missing in Go?
- Using structs for
- Using pointer receiver based methods
- Data hiding and encapsulation

Agenda day 3/3 part 1

- Object-Oriented programming in Go
- Create and use interfaces to enforce a contract
- Interface satisfaction
- Interface type assertions and switches

AGENDA DAY 3/3 PART 2

- I/O and networking in Go
- File operations
- JSON serialization and marshalling
- Building an http server
- Building restful API in Go

Agenda Day 3/3 Part 3

» Templating in Go

Agenda day 3/3 part 3

- Go and concurrency
- What is concurrency
- Goroutines, waitgroups and channels
- Buffered and unbuffered channels
- Waiting on multiple channels
- Critical section; mutexes
- Critical section; atomic operations
- Patterns and best practives

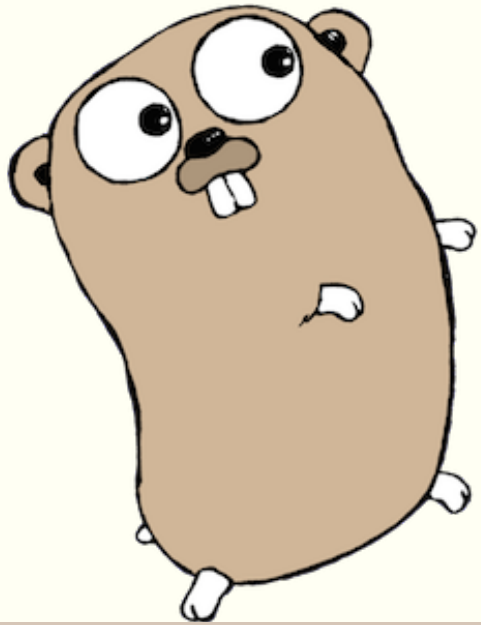
Introduction to Go

Introduction to Go 1/2

- Pronounciation: Go or Golang?
- Developed by Google in 2007
- Robert Griesemer, Rob Pike and Ken Thompson
- OpenSourced in 2009
- In high demand for 2020, 2021 and 2022
- Latest release: 1.17

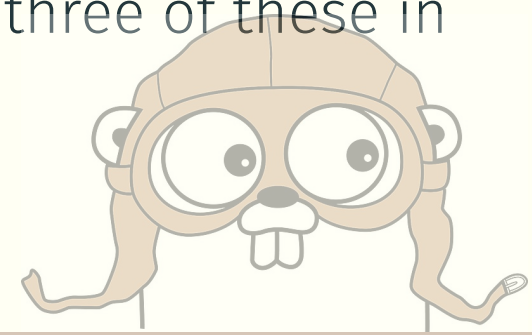
Introduction to Go 2/2

- Primay website: http://golang.org
- Tiobe 13th place over 2021, steadily rising
- Go Nuts (http://groups.google.com/group/golang-nuts/)
- Mascot is a Gopher



WHY DEVELOP A BRAND NEW LANGUAGE?

- No new systems language emerged for over a decade
- C and C++ did not evolve with the new world and it's needs
  - Massive multi-core processing
  - Massive multi-threading
  - Massive multi-networking
- Computing power grows faster and more failsafe every day
- Software development does not
- One had to chose between one of the three:
  - Fast execution and slow inefficient building (like C++)
  - Effiction compilation but slow execution (.NET or Java)
  - Ease of programming but slow executuon (dynamic languages like Python)
- We need a language that facilitates all three of these in one. Go is designed to do this

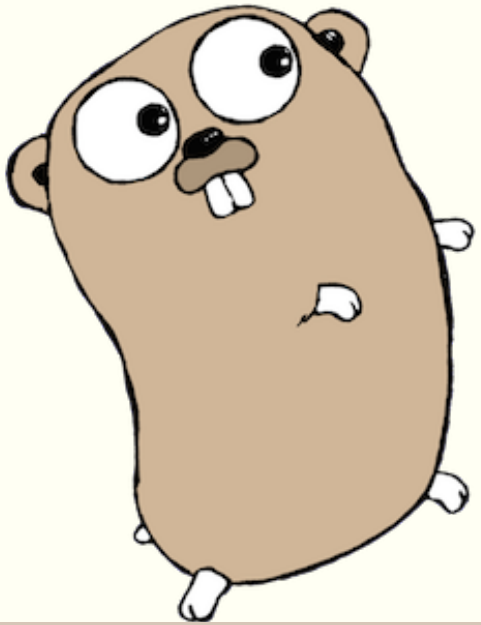


GO(ALS)

- Improve programming productivity in modern environments
  - Static typing and runtime-efficiency like C
  - Readability and usability like Python and JavaScript
  - High-performance in networking and multiprocessing env
  - Fast compilation

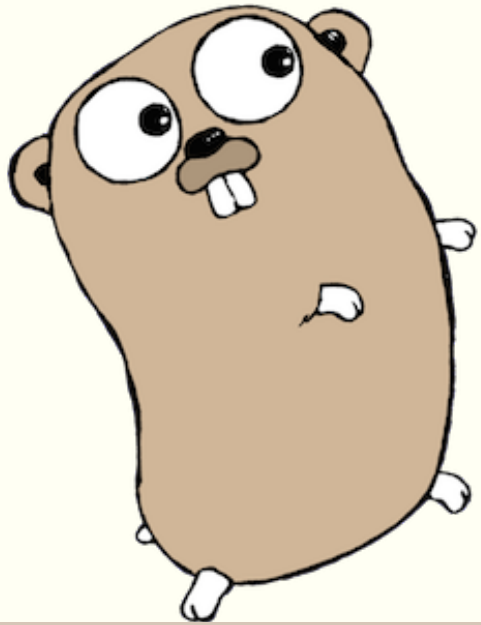
GO(ALS)

- Improve programming productivity in modern environments
  - Static typing and runtime-efficiency like C
  - Readability and usability like Python and JavaScript
  - High-performance in networking and multiprocessing env
  - Fast compilation
- And important, have some fun!



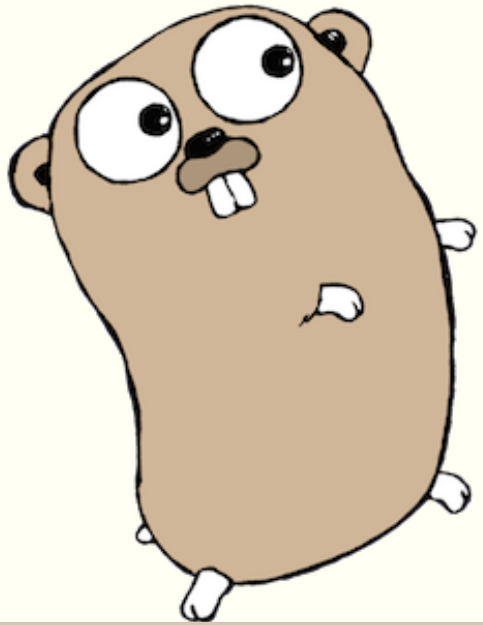
INFLUENCES ON GO DEVELOPMENT

- The C family (C, C++, C# and Java) for the main syntax
- The Niklaus Wirth family (Pascal, Modula, Oberon) for Declarations and Packages
- Limbo and Newsqueak for the concurrency mechanism used



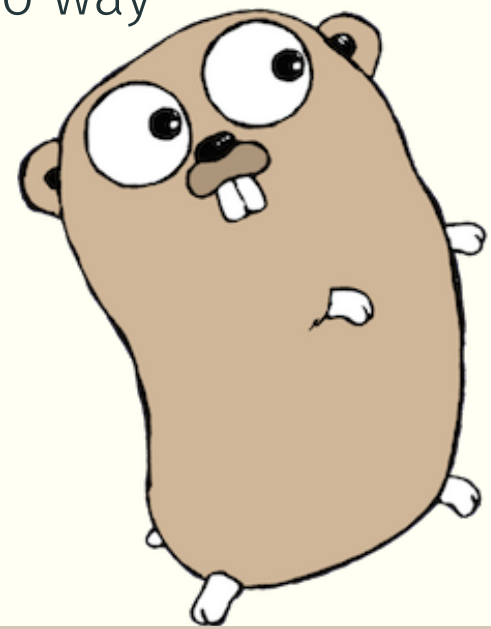
PROJECTS REALIZED IN GO

- Docker
- Kubernetes (K8S)
- Helm
- Ethereum
- Terraform
- Etc



CAN I DO ... IN GO?

- Procedural programming? -> Yes
- Object Oriented Programming> -> Yes, in a Go way
- Funcional Programming? -> Yes, in a Go way
- Concurrent Programming? -> Yes, in a Go way



WHAT FEATURES DOES GO LACK?

What features does Go lack?

- Quite some...

What features does Go lack?

- Quite some...
- But let's not focus on what is IMPOSSIBLE...

What features does Go lack?

- Quite some...
- But let's not focus on what is IMPOSSIBLE...
- but on what IS POSSIBLE..

What features does Go lack?

- Quite some...
- But let's not focus on what is IMPOSSIBLE...
- but on what IS POSSIBLE..
- and how these help us solve problems the Go way!

What features does Go lack?

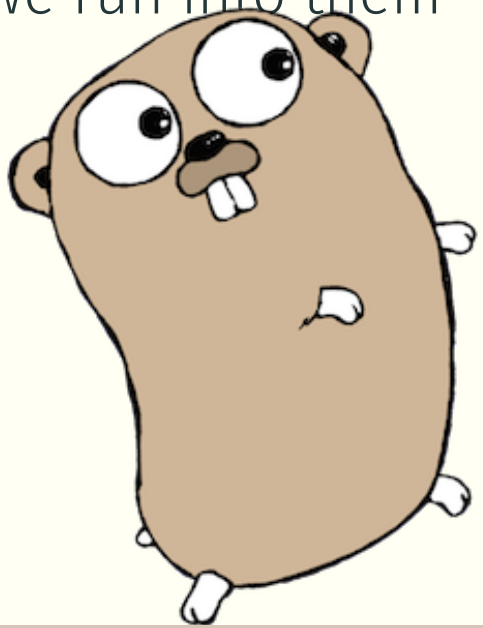
- Quite some...
- But let's not focus on what is IMPOSSIBLE...
- but on what IS POSSIBLE..
- and how these help us solve problems the Go way!
- We will discover and name them when we run into them

What features does Go lack?

- Quite some...
- But let's not focus on what is IMPOSSIBLE...
- but on what IS POSSIBLE..
- and how these help us solve problems the Go way!
- We will discover and name them when we run into them
- and provide a Go wise solution.

WHAT FEATURES DOES GO LACK?

- Quite some...
- But let's not focus on what is IMPOSSIBLE...
- but on what IS POSSIBLE..
- and how these help us solve problems the Go way!
- We will discover and name them when we run into them
- and provide a Go wise solution.



GO MINDSET

- Go is Go
- Go is stubborn
- Go goes its own way
- Repect Go
- Let Go of attachments to other programming languages, they only hold you back
- Find Go in your search for solutions
- When programming in Go, do like the Gophers do

GO PROVERBS

1. Don’t communicate by sharing memory, share memory by communicating.
2. Concurrency is not parallelism.
3. Channels orchestrate; mutexes serialize.
4. The bigger the interface, the weaker the abstraction.
5. Make the zero value useful.
6. interface says nothing.
7. Gofmt’s style is no one’s favorite, yet gofmt is everyone’s favorite.
8. A little copying is better than a little dependency.
9. Syscall must always be guarded with build tags.
10. Cgo must always be guarded with build tags.
11. Cgo is not Go.
12. With the unsafe package there are no guarantees.
13. Clear is better than clever.
14. Reflection is never clear.
15. Errors are values.
16. Don’t just check errors, handle them gracefully.
17. Design the architecture, name the components, document the details.
18. Documentation is for users.
19. Don’t panic.

Simple helloworld in Go

- package declaration
- import(s)
- func main()
- No semi-colon

code/helloworld.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          fmt.Println("Hello World")
8  }
```

GO SHOW - HTTP-CLIENT EXAMPLE

```
1  package main
2
3  import (
4          "bufio"
5          "fmt"
6          "net/http"
7  )
8
9  func main() {
10
11         resp, err := http.Get("http://st99node01.itgildelab.net")
12         if err != nil {
13                 panic(err)
14         }
15         defer resp.Body.Close()
16
17         fmt.Println("Response status:", resp.Status)
18
19         scanner := bufio.NewScanner(resp.Body)
20         for i := 0; scanner.Scan() && i < 5; i++ {
21                 fmt.Println(scanner.Text())
22         }
23
24         if err := scanner.Err(); err != nil {
25                 panic(err)
26         }
27 }
```

GO SHOW - HTTP-SERVER EXAMPLE

```
1  package main
2
3  import (
4          "fmt"
5          "net/http"
6  )
7
8  func hello(w http.ResponseWriter, req *http.Request) {
9
10         fmt.Fprintf(w, "hello\n")
11 }
12
13 func headers(w http.ResponseWriter, req *http.Request) {
14
15         for name, headers := range req.Header {
16                 for _, h := range headers {
17                         fmt.Fprintf(w, "%v: %v\n", name, h)
18                 }
19         }
20 }
21
22 func main() {
23
24         http.HandleFunc("/hello", hello)
25         http.HandleFunc("/headers", headers)
26
27         http.ListenAndServe(":8090", nil)
28 }
```

GO SHOW - CONCURRENCY EXAMPLE

```go
package main

import (
	"fmt"
	"sync"
	"time"
)

func worker(id int) {
	fmt.Printf("Worker %d starting\n", id)

	time.Sleep(time.Second)
	fmt.Printf("Worker %d done\n", id)
}

func main() {

	var wg sync.WaitGroup

	for i := 1; i <= 5; i++ {
		wg.Add(1)
		i := i
		go func() {
			defer wg.Done()
			worker(i)
		}()
	}
	wg.Wait()
}
```

Install and configure your Go environment

Installation Options

- Install from distro repos (apt,yum, dnf)
- Download latest from http://go.dev

code/install-linux.sh

```
# Download latest Go 1.17 version

ARCH=amd64
curl -LO https://go.dev/dl/go1.17.6.linux-${ARCH}.tar.gz

# Remove old Go release

sudo rm -rf /usr/local/go

# Unpack Go in /usr/local

sudo tar -C /usr/local -xzf go1.17.6.linux-amd64.tar.gz

# Add Go to user's path:

echo 'export PATH=$PATH:/usr/local/go/bin' > ~/.profile
```

Installation on MacOS

- On Intel Apple devices: download latest https://go.dev/dl/go1.17.6.darwin-amd64.pkg
- On Apple Silicon devices: download latest https://go.dev/dl/go1.17.6.darwin-arm64.pkg
- Use installer to install in /usr/local/go
- Access Go using the terminal

Installation on Windows

- Download latest https://go.dev/dl/go1.17.6.windows-amd64.msi
- Use installer to install in /usr/local/go
- Access go using 'cmd'

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version Shows the currento Go version

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version Shows the currento Go version

go env Shows the Go environment variables

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version Shows the currento Go version

go env Shows the Go environment variables

go run Runs the specified Go source file

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version	Shows the currento Go version
go env	Shows the Go environment variables
go run	Runs the specified Go source file
go build	Compiles the specified Go source file

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version	Shows the currento Go version
go env	Shows the Go environment variables
go run	Runs the specified Go source file
go build	Compiles the specified Go source file
go help	Provides help to the Go (sub-)commands

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version	Shows the currento Go version
go env	Shows the Go environment variables
go run	Runs the specified Go source file
go build	Compiles the specified Go source file
go help	Provides help to the Go (sub-)commands
go fmt	Formats / pretty prints Go source

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version	Shows the currento Go version
go env	Shows the Go environment variables
go run	Runs the specified Go source file
go build	Compiles the specified Go source file
go help	Provides help to the Go (sub-)commands
go fmt	Formats / pretty prints Go source
go doc	Shows documentation of a package

GO COMMANDS

The `go` command gives us access to various functionality of the Go SDK

go version	Shows the currento Go version
go env	Shows the Go environment variables
go run	Runs the specified Go source file
go build	Compiles the specified Go source file
go help	Provides help to the Go (sub-)commands
go fmt	Formats / pretty prints Go source
go doc	Shows documentation of a package
go vet	Reports likely mistakes in your source

GO ENVIRONMENT VARIABLES

- Check GO enviroment variables with `go env`
- `GOROOT` defines the location of you Go SDK (/usr/local/go)
- `GOPATH` defines the location of your Go source codes (/go)
  - `src` path to your .go files
  - `bin` path to 'installed' builts.
  - `pkg` path to your packages
- When using modules `src` is not needed anymore.
- Change `GOROOT` if you have (multiple) version(s) of Go installed in different paths

IDE AND EDITORS FOR GO

- VScode: (or short `code` with Go extension
- GoLand: specific IDE for Go
- Atom: with Go-Plus Atom package
- Vim/Neovim: with vim-go plugin

My first Go program

FIRST STEPS IN GO 1/2

Simple helloworld in Go

- The entrypoint of every Go program is the `main()` function
- A `return` from main will exit the program
- Packages needed are `imported`.
- No spillage in Go
- You declare it, you use it!
- Reasoning: concious programming

code/helloworld.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          fmt.Println("Hello World")
8  }
```

FIRST STEPS IN GO 2/2

Using a custom package

code/demo2.go

```
1  package main
2
3  import (
4          "demo2/itgilde"
5          "fmt"
6  )
7
8  var msg = itgilde.GetQuote()
9
10 func main() {
```

code/quote.go

```
1  package itgilde
2
3  func GetQuote() string {
4
5          msg := "Together, stronger, better..."
6          return msg
7  }
```

FIRST STEPS IN GO 2/2

Using a custom package

code/demo2.go

```
1  package main
2
3  import (
4          "demo2/itgilde"
5          "fmt"
6  )
7
8  var msg = itgilde.GetQuote()
9
10 func main() {
```

code/quote.go

```
1  package itgilde
2
3  func GetQuote() string {
4
5          msg := "Together, stronger, better..."
6          return msg
7  }
```

» Go 1.11 and later; use `go mod init demo2`

First steps in Go 2/2

Using a custom package

code/demo2.go

```
1  package main
2
3  import (
4          "demo2/itgilde"
5          "fmt"
6  )
7
8  var msg = itgilde.GetQuote()
9
10 func main() {
```

code/quote.go

```
1  package itgilde
2
3  func GetQuote() string {
4
5          msg := "Together, stronger, better..."
6          return msg
7  }
```

- Go 1.11 and later; use `go mod init demo2`
- Or set `go env -w GO111MODULE=off` (not recommended!)

FIRST STEPS IN GO 2/2

Using a custom package

code/demo2.go

```
1  package main
2
3  import (
4          "demo2/itgilde"
5          "fmt"
6  )
7
8  var msg = itgilde.GetQuote()
9
10 func main() {
```

code/quote.go

```
1  package itgilde
2
3  func GetQuote() string {
4
5          msg := "Together, stronger, better..."
6          return msg
7  }
```

- Go 1.11 and later; use `go mod init demo2`
- Or set `go env -w GO111MODULE=off` (not recommended!)

GO GENERAL SYNTACTICS

- The language is case sensitive
- `main()` is a reserved function
- `init()` is a reserved function (in packages)
- Blocks are enclosed in curly braces `{}`
- Optional use of semicolons
- Remember Go proverb: documentation is for users

RUNNING A GO PROGRAM

- To run, use `go run <go source file>`
- and check the output in a terminal

code/rungo.out

```
1  package itgilde
2
3  func GetQuote() string {
4
5          msg := "Together, stronger, better..."
6          return msg
7  }
```

COMPILING/BUILDING A GO PROGRAM

- To build a go program, use `go build <go source file>`
- The resulting binary is static
- Use Go ENV to configure the build proces
  - To build for a different architecture (GOARCH)
  - To build for a different operating system (GOOS)
  - No multiarch compiler & tool-chains needed!

code/buildgo.out

```
1  package itgilde
2
3  func GetQuote() string {
4
5          msg := "Together, stronger, better..."
6          return msg
7  }
```

COMMENTS IN GO

» Comments in Go serve 2 purposes:
  - Improving readability & clarity of source code
  - As input for Go doc as integrated documentation

code/comment-single.go

```
1  // Example of a single line comment
2  // Another one
3  package main
4
5  import "fmt"
6
7  func main() {
8          fmt.Println("Hello World")
9  }
```

code/comment-multi.go

```
1  package main
2
3  import "fmt"
4
5  /*
6   * Multi line comment example
7   *
8   */
9
10 func main() {
11         fmt.Println("Hello World")
12 }
```

COMMENTS USED AS INPUT FOR GODOC 1/2

code/quote2.go

```
1  // Package itgilde shares common knowledge and mastery of the Guild
2  // This is the initial version of the package.
3  // Contact the Guildmaster in case of findings, participations etc.
4  package itgilde
5
6  // GetQuote cites a Quote from the Guild
7  func GetQuote() string {
8
9          msg := "Together, stronger, better and smarter"
10         return msg
11 }
```

- Comments for a package or function are directly attached (no newline) to the item they describe
- The comment just above the package will describe the package for `go-doc`
- The comment just above the function will describe the function when queried for using `go-doc`

COMMENTS USED AS INPUT FOR GODOC 2/2

code/godoc-quote2.out

```
1  // Package itgilde shares common knowledge and mastery of the Guild
2  // This is the initial version of the package.
3  // Contact the Guildmaster in case of findings, participations etc.
4  package itgilde
5
6  // GetQuote cites a Quote from the Guild
7  func GetQuote() string {
8
9          msg := "Together, stronger, better and smarter"
10         return msg
11 }
```

VARIABLES IN GO

- Go is a statically typed language
- Variable declarations reserves memory for a specific type and value
- The memory location is identified by the name of the variable

DECLARING VARIABLES IN GO

- The keyword `var` declares a new global or local var and initializes it
- If no value is explicitly specified by default the `zero value` for that variable type will be assigned
- Remember, go dictates; 'you declare it, you use it'
- Unused vars result in compile time errors

```
1  package main
2
3  var my_global_var = "I am global"
4
5  func main() {
6          var msg string = "my string" // type with init. value
7          var f1 float64               // assigned default zero value for float
8          var f2 = 3.1415              // by type inference -> float
9          var v4, v5 float32 = 5, 6
10
11         var (
12                 hex       = 0xdeadbeaf
13                 str       = "another string"
14                 unix_perm = 0644
15         )
16 }
```

DEFAULT ZERO VALUES FOR UNASSIGNED VARS

- Strings will be assigned the empty string value, e.g. `""`
- Booleans will get assigned a default value of `false`
- The 0 value for numeric types likes `ints, floats etc`

SHORT VARIABELE DECLARTION IN FUNCTIONS

In functions Go allows the use of a shortened variable declaration using `Pascal/Modula` style assignments:

```
1  package main
2
3  func main() {
4
5          my_str := "A string value"
6          pct := 21
7
8  }
```

IDENTIFIERS IN GO

- Identifiers name the vars, constants, functions etc in Golang
- Identifiers are case-sensitive
- Identifiers can use any UTF-8 character(!)
- Identifiers can be a combination of letters, numbers and select special symbols
- Identifiers should always start with a letter
- Special case is the underscore (_) which is called the `Blank Identifier`
- The `Blank Identifier` is used to discard/ignore return values.

SCOPING OF IDENTIFIERS IN GO

- Identifiers starting with a `Capital` letter will be considered public and/or exported.
- Identifiers starting with a `lower case` letter will be considered private

CONSTANTS IN GO

- Constants in Go are considered readonly identifiers
- The keyword `const` is used to define and initialize the value
- One cannot declare constants using the `:=` syntax

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          const userName = "Gophers"
7          updateContent(userName)
8          fmt.Println("In main(): Hello " + userName)
9  }
10
11 func updateContent(userName string) {
12         userName = "Pythonistas"
13         fmt.Println("In updateContent: Hello " + userName)
14 }
```

ENUMERATED CONSTANTS IN GO

Using constant grouping one can get functionality like enum in C/C++

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  const (
8          red   = 0xff0000
9          blue  = 0x0000ff
10         green = 0x00ff00
11         black = 0x0
12         white = 0xffffff
13 )
14
15 func main() {
16         fmt.Printf("The html code for red is: %6x\n", red)
17 }
```

IOTA, THE CONSTANT GENERATOR

The keyword `iota` can be used to generate constants

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  // An example of the use of iota, or the constant generator
8
9  const (
10         Monday  = iota // Monday = 0
11         Tuesday        // Tuesday = 1
12         Wednesday
13         Thursday
14         Friday
15         Saturday
16         Sunday // Sunday = 6
17 )
18
19 func main() {
20         fmt.Printf("Sunday is day: %d\n", Sunday)
21 }
```

IOTA, THE CONSTANT GENERATOR

You can even use expressions with `iota`

```
1  package main
2
3  import "fmt"
4
5  const (
6          _   = 1 << (iota * 10) // Skip 0
7          KiB                    // 1024 bytes
8          MiB                    // 1048576 bytes
9          GiB                    // 1073741824 bytes
10 )
11
12 func main() {
13         fmt.Printf("A MiB is %d bytes\n", MiB)
14 }
```

POINTERS IN GO

- Go supports the pointer type
- A variable is a unit of storage containing a value
- The value of a pointer is the address to this value (reference)
- A pointer is the actually the address where the variable is stored
- The `&` operator gets the address of the variable (reference)
- The `*` operator is it's inverse and gets the value of the stored variable (de-reference)
- Pointer arithmic & magic (like *p+1) is not allowed in Go

POINTERS IN GO - ANOTHER EXAMPLE

```
1  package main
2
3  import "fmt"
4
5  func swap_int(x *int, y *int) {
6          *x, *y = *y, *x
7  }
8
9  func main() {
10         a := 3
11         b := 5
12         fmt.Printf("a == %d and b == %d\n", a, b)
13         swap_int(&a, &b)
14         fmt.Printf("a == %d and b == %d\n", a, b)
15
16 }
```

POINTERS IN GO - THE NIL OR ZERO VALUE

- In Go the default zero value for pointers is the `nil` value
- Compare the `nil` value to `null` in C and C++
- New variables can also be instantiated by the `new` function
- If the pointer return value of `new` function is not nil then the variable can be safe accessed.

```
1  package main
2
3  func main() {
4          p * int
5          p = new(int)
6
7          if p != nil {
8                  // safe to derefence now
9                  *p = 14
10         }
11
12 }
```

Go Primitive Types

LITERALS

There are five types of literals in Go

- Integer literals
- Floating point literals
- Rune literals
- String literals
- Imaginary literals

INTEGER LITERALS

- Sequence of numbers
- By default base10
  - `0b` for binary
  - `0o` r `00` for octal
  - `0x` for hexadecimal
- To make big integer literals more readable use underscores `(_)`
- E.g. `100_000_012` for 100000012
- E.g. `0xFACE_B00C` for base16 (hex) 0xFACEBOOC

FLOATING POINT LITERALS

- Have decimal points to describe the fractional portion
- Underscores can also be used to make floating point literals more readable

RUNE LITERALS

- Used to represent single characters
- Enclosed in single quotes
- Rune literals can be written as
  - Single Unicode characters (’A’)
  - 8-bit numbers, 16-bit numbers in octal or hexadecimal notation
  - 32-bit Unicode numbers `('\U00000061')`
- Escaped `rune` literals `\n, \t, \'', \"", \\`
- Underscores can also be used to make floating point literals more readable

STRING LITERALS

- Used to represent strings
- Enclosed within double-quotes (`\"`)
- Mind that single- and double quotes are not interchangeable in Go
- One can escape characters like LF, doublequotes, tabs etc.
- One can also use a raw literal by enclosing the string in backticks (`` ` ``)

IMAGINARY LITERALS

- Used to represent imaginary part of a complex number
- Has the same properties as a floating point number

BOOLEAN TYPE

- Variables of type bool have one of the two values:
  - `true` or
  - `false`
- Default value is `false`

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          var flag bool // defaults to false
8          var isSet = true
9
10         fmt.Println ("flag  == ",flag)
11         fmt.Println ("isSet == ",isSet)
12 }
```

NUMERIC TYPES

Go knows 12 different types plus some aliases, grouped in 3 categories

- Integer types
- Floating point types
- Complex types

INTEGER NUMERIC TYPES

» Default values is 0 for all numeric types

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          // signed integers
8
9          var n1 int8   // -128 -> 127
10         var n2 int16  // -32,768 -> 32,767
11         var n3 int32  // -2,147,483,648 -> 2,147,483,647
12         var n4 int64  // -9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807
13
14         // unsigned integers
15
16         var u1 uint8  // 0 -> 255 aliased as byte
17         var u2 uint16 // 0 -> 65,535
18         var u3 uint32 // 0 -> 4,294,967,295
19         var u4 uint64 // 0 -> 18,446,744,073,709,551,615
20
21         fmt.Println(n1,n2,n3,n4,u1,u2,u3,u4)
22 }
```

INTEGER NUMERIC ALIASES

- A `byte` is an alias for `uint8`
- A `int` is an alias for a `int32` or `int64`
- A `rune` is an alias for a `int32`
- A `uintptr` is an alias for a `int32` or `int64`

THE IDIOMATIC CHOICE FOR AN INTEGER TYPE

Which integer type to choose?

1. For binary fileformats or network protocols -> use specic integer type
2. When writing library functions use two functions for int64 and uint64
3. In all other cases; use int

INTEGER OPERATORS

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var x int = 20
7          var y1,y2,y3,y4 int
8
9          y1 = x + 2
10         y2 = x - 2
11         y3 = x * 2
12         y4 = x / 2
13
14         fmt.Println(y1,y2,y3,y4)
15
16         fmt.Println(x)
17         x++
18         x += 2
19         fmt.Println(x)
20         x *= 2
21         fmt.Println(x)
22 }
```

FLOATING POINT NUMERIC TYPES

There are 2 floating point types in Go

1. `float32` covering
   1.401298464324817070923729583289916131280e-45 ->
   3.402823466385288598117041834845169254 40e+38
2. `float64` covering
   4.940656458412465441765687928682213723651e-324 ->
   1.797693134862315708145274237317043567981e+308
3. Default value: 0

FLOATING POINT NUMERIC TYPES - GOTCHA'S

The equal to (==) and it's inverse (!=) comparisson operators are available but are of doubtful use with floating point type variables

```
1  package main
2
3  import "fmt"
4  import "math"
5
6  func main() {
7
8          var f1 float64
9
10         f1 = math.Sqrt(2)
11         f2 := f1
12
13          f4 := f1 * f2
14
15         if (f4 != 2) {
16                 fmt.Println ("Bad idea! f4 != 2 -> f4 == ",f4)
17         }
18 }
```

COMPLEX NUMERIC TYPES

Go has first-class support for complex numbers

- `complex64` 32 bit real and 32 bit imaginary part
- `complex128` 64 bit real and 64 bit imaginary part
- Default value: 0 for real part and 0 for imaginary part
- Functions for complex numbers can be found in the `math/cmplx` package

COMPLEX NUMERIC TYPES

```go
package main

import (
        "fmt"
        "math"
        "math/cmplx"
)

func main() {

        x := complex(2.5, 3.1)
        // y := complex(10.2, 2)
        fmt.Println(real(x))
        fmt.Println(imag(x))
        fmt.Println(cmplx.Abs(x))

        i := complex(0, 1)
        j := i * i

        fmt.Println(real(j))

        e := math.Exp(1)

        fmt.Println(e)

        s1 := math.Pi * i
        fmt.Println(s1)

        euler := cmplx.Exp(s1) + 1

```

Strings in Go

- Can be compared with: ==, !=, >, >=, < and <=
- Can be concatenated with the + operator
- Default value is the empty string

EXPLICIT TYPE CONVERSIONS

Go doesn't have automatic type promotion (implicit type casting)

- When variable types do not match use explicit type conversion
- Even different sized variables for the same type need to be converted
- Implication: there is not truthiness in Go
- Go prefers clarity over concisness

EXPLICIT TYPE CONVERSIONS - EXAMPLES

```
1  package main
2
3  import "fmt"
4
5
6  func main() {
7    var n int = 20
8    var f float64 = 37.1
9
10   var z float64 = float64(n) + f
11   var d int = n + int(f)
12
13   fmt.Println (n,f,z,d)
14 }
```

Go conditionals

CONDITIONALS IN GO - IF

- Go knows the `if`, `else` and `elseif` constructs
- Go has the short-circuit evaluation implemented
- Logical operatos:

  AND The `&&` operator

  OR The `||` operator

  NOT The `!` operator

```
1  package main
2
3  func main() {
4          P := true
5          Q := false
6
7          if P && Q {
8                  // Execute if at least P is true
9          }
10 }
```

CONDITIONALS IN GO - SCOPED VARIABLES IN IF

- Special to Go is the ability to scope vars in the `if` statement
- The instance of this var only exists while in the if clause

```
1  package main
2
3  import (
4          "fmt"
5          "math"
6  )
7
8  func pow(x, n, lim float64) float64 {
9          if v := math.Pow(x, n); v < lim {
10                 return v
11         }
12         return lim
13 }
14
15 func main() {
16         fmt.Println(
17                 pow(3, 2, 10),
18                 pow(3, 3, 20),
19         )
20 }
```

CONDITIONALS IN GO - IF EXAMPLES

```
1  package main
2
3  import (
4          "fmt"
5          "math/rand"
6  )
7
8  func main() {
9
10         var myval = 0
11
12         if n := rand.Intn(10); n == 0 {
13                 fmt.Println("That's too low")
14         } else if n > 5 {
15                 fmt.Println("That's too big:", n)
16         } else {
17                 fmt.Println("That's a good number:", n)
18                 myval = n
19         }
20
21         fmt.Println(myval)
22 }
```

Conditionals in Go - switch

- Most programmers don't like switch statements in other languages
- They don't like the limitations on the test cases
- They don't like the default fallthrough behaviour
- Go switches are different

CONDITIONALS IN GO - BASIC OR REGULAR SWITCH

- Executes the first case equal to the condition expression
- Cases are evaluated top to bottom
- Stopping when a case succeeds
- No automatic fallthrough unless specified
- If none of the cases matches, default case will be executed

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          i := 4
8          fmt.Print("Write ", i, " as ")
9          switch i {
10         case 1:
11                 fmt.Println("one")
12         case 2:
13                 fmt.Println("two")
14         case 3:
15                 fmt.Println("three")
16         default:
17                 fmt.Println("Should not happen")
18         }
19 }
```

CONDITIONALS IN GO - BLANK SWITCH

Regular switches are limited in their tests. Blank switches have no expression in the switch part but can have individual expressions on each case statement. This allows for greater flexibility

```
1  package main
2
3  import (
4          "fmt"
5          "time"
6  )
7
8  func main() {
9
10         t := time.Now()
11
12         // Blank switch
13
14         switch {
15         case t.Hour() < 12:
16                 fmt.Println("AM")
17         default:
18                 fmt.Println("PM")
19         }
20 }
```

CONDITIONALS IN GO - SWITCHES

While blank switches are very flexible, idomatic Go advises you not do so, if all your case statements are actually comparissons against the same variable, just use the regular/expression statement switch

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          // This is a wrongly applied blank switch, change into expression switch
8
9          switch {
10         case a == 2:
11                 fmt.Println("a == 2")
12         case a == 3:
13                 fmt.Println("a == 3")
14         case a == 4:
15                 fmt.Println("a == 4")
16         default:
17                 fmt.Println("a == 5")
18         }
19 }
```

CONDITIONALS IN GO - THE FALLTHROUGH CASE

- The statement `falltrough` prevents leaving the case block after the first 'match'
- Be carefull with `fallthrough` as it might not do what you expect

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          for i := 0; i <= 15; i++ {
7                  fmt.Printf("%d is:\n", i)
8                  switch {
9                  case i%2 == 0:
10                         fmt.Printf("- is a factor of 2\n")
11                         fallthrough
12                 case i%3 == 0:
13                         fmt.Printf("- is a factor of 3\n")
14                         fallthrough
15                 case i%5 == 0:
16                         fmt.Printf("- is a factor of 5\n")
17                 }
18         }
19 }
```

Conditionals in Go - breaking out

- Altough the default behaviour is not to fallthrough, Go does have `break`
- The `break` statement can be used to:
  - Break early
  - Break conditionally
  - Break out something else then case block (e.g. a labeled loop)

CONDITIONALS IN GO - BREAKING OUT EXAMPLE

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7  testLoop:
8          for val := 1; val < 7; val++ {
9                  fmt.Printf("%d", val)
10                 switch {
11                 case val == 1:
12                         fmt.Println("->Start")
13                 case val == 5:
14                         fmt.Println("->Break")
15                         break testLoop
16                 case val > 2:
17                         fmt.Println("->Running")
18                         break // superfluous
19                 default:
20                         fmt.Println("->Progress")
21                 }
22         }
23         fmt.Println("Out of loop")
24 }
```

Go Loops

LOOPS IN GO

There are 4 types of loops in Go.

- A complete C-style for loop
- A condition only for loop
- An infinite for loop
- The for-range loop

The complete C-style for loop

- Similar to the for loop in C, Java or JavaScript
- No parentheses
- Three parts separated by semicolons
  - Part 1: Initialization
  - Part 2: Comparission
  - Part 3: Increment

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          for i := 0; i < 10; i++ {
7                  fmt.Println(i)
8          }
9  }
```

THE CONDITION ONLY FOR LOOP

» This give similar functionality like the while loop in C, C++ and Java

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          i := 1
7          for i < 100 {
8                  fmt.Println(i)
9                  i = i * 2
10         }
11 }
```

THE INFINITE FOR LOOP

- This will give an infinite loop, handy for event loops, waiting to things to happen
- To break out of the loop we can use `break` and `continue` statements

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          for {
8                  fmt.Print("Hello ")
9          }
10 }
```

THE INFINITE FOR LOOP - BREAKING OUT

- Go does not have a `do {} while` loop solution
- To convert such a loop to Go, we can use the following idiomatic solution

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          var input string
8
9          fmt.Print("Type exit to quit: ")
10
11         for {
12                 fmt.Scanln(&input)
13                 if input == "exit" {
14                         fmt.Println("Good bye!")
15                         break
16                 }
17                 fmt.Printf("You typed: %s\n", input)
18         }
19 }
```

BREAKING OUT NESTED LOOPS

- A `break` only affectes the current iner loop it's in
- Mind this in nested loops

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          for outer := 0; outer < 5; outer++ {
7                  if outer == 3 {
8                          fmt.Println("Breaking out of outer loop")
9                          break // break from outerloop
10                 }
11                 fmt.Println("The value of outer is", outer)
12                 for inner := 0; inner < 5; inner++ {
13                         if inner == 2 {
14                                 fmt.Println("Breaking out of inner loop")
15                                 break // break from innerloop
16                         }
17                         fmt.Println("The value of inner is", inner)
18                 }
19         }
20         fmt.Println("Exiting program")
21 }
```

CONTROLLED BREAKING OUT NESTED LOOPS WITH LABELS

- A `break` only can break out of the current loop
- Using labels we can control which loop to break

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7  outerloop:
8          for outer := 0; outer < 5; outer++ {
9                  if outer == 3 {
10                         fmt.Println("Breaking out of outer loop")
11                         break // break from outerloop
12                 }
13                 fmt.Println("The value of outer is", outer)
14                 for inner := 0; inner < 5; inner++ {
15                         if inner == 2 {
16                                 fmt.Println("ALso breaking out of outer loop")
17                                 break outerloop // break also from outer loop
18                         }
19                         fmt.Println("The value of inner is", inner)
20                 }
21         }
22         fmt.Println("Exiting program")
23 }
```

BREAKING OUT BUT CONTINUEING

- Where `break` exits the current loop completely, `continue` will break out the current loop but resume the loop in the next iteration
- In the example the nr 5 will be skipped in the output

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          for i := 0; i < 10; i++ {
7                  if i == 5 {
8                          fmt.Println("Continuing loop")
9                          continue // break here but resume loop
10                 }
11                 fmt.Println("The value of i is", i)
12         }
13         fmt.Println("Exiting program")
14 }
```

CONTROLLED BREAKING OUT AND CONTINUEING WITH LABELS

» Also `continue` has a variant with a label that can specify which loop to break and continue

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7  outerloop:
8          for outer := 0; outer < 5; outer++ {
9                  if outer == 3 {
10                         fmt.Println("Breaking out and resuming of outer loop")
11                         continue // next iteration of outer loop
12                 }
13                 fmt.Println("The value of outer is", outer)
14                 for inner := 0; inner < 5; inner++ {
15                         if inner == 2 {
16                                 fmt.Println("Also breaking and continueing outer loop")
17                                 continue outerloop // next iteration of outer loop
18                         }
19                         fmt.Println("The value of inner is", inner)
20                 }
21         }
22         fmt.Println("Exiting program")
23 }
```

THE FOR RANGE LOOP

This type of loop allows us to iterate over (for each)

- Bytes or runes in a string
- Arrays
- Maps
- Slices
- Channels

Their in-depth discussion will be postponed until we cover these complex datatypes in their own module

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          a := []string{"Gophers", "Rustaceans", "Pythonistas"}
7          for _, s := range a {
8                  fmt.Println(s)
9          }
10 }
```

Go Strings, Runes and Bytes

Strings in Go

- Are implemented in Go using a sequence of bytes
- No partitcular character coding enforced
- Several Go library functions as well as the for-range loop assume UTF-8 encoding

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var s1 = "a string"
7          var s2 string = "another string"
8
9          s3 := s1 + " and " + s2
10
11         fmt.Printf("s1: %s type %T\n", s1, s1)
12         fmt.Printf("s2: %v type %T\n", s2, s2)
13         fmt.Printf("s3: %v type %T\n", s3, s3)
14 }
```

OPERATIONS ON STRINGS

- Index expressions can be used to specificy substrings from strings
- Like `s[4-10]`
- String indexes start counting from 0
- Strings are immutable
- The Built-in `len()` function returns the length of the string
- Strings can be joined using the + operator

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          var s = "Hello Gophers"
8          s1 := s[:5]
9          s2 := s[6:]
10
11         fmt.Println(s2, " - ", s1)
12         fmt.Println(len(s))
13 }
```

STRING CONVERSIONS

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var s string = "Hello, Gophers"
7          var a rune = 'p'
8          var s1 = string(a)
9          var b byte = 'q'
10         var s2 = string(b)
11         var x int = 66
12         var s3 = string(x)
13
14         s4 := "unicode check "
15         s5 := s4 + "□"
16         var bs []byte = []byte(s)
17         var rs []rune = []rune(s)
18         fmt.Println(s1, s2, s3, bs, rs)
19         fmt.Println(s4, len(s4))
20         fmt.Println(s5, len(s5))
21 }
```

ITERATING OVER STRINGS

The `for-range` loop can be used to iterate over strings

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          str := "Hello  Gophers"
7
8          for i, c := range str {
9                  fmt.Printf("%2d %c (%3d)\n", i, c, c)
10         }
11
12 }
```

Go Arrays

The Go language does have arrays

- Arrays are very strict and so have a limited use
- Provide an up-beat to slices

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var x [5]int                  // Creates an array of 5 integers
7          var y = [3]int{100, 200, 300}  // Creates an initialized array of 3 ints
8          var z = [5]int{1, 3: 1, 4: 20} // [ 1, 0, 0, 1, 20 ]
9
10         var qs = [...]string{"aap", "noot", "mies"}
11
12         fmt.Println(x, y, z, qs)
13 }
```

GO ARRAYS - OPERATIONS

```go
package main

import "fmt"

func main() {

        var a = [3]int{1, 2, 3}
        var b = [3]int{1, 2, 3}
        var cs = [...]string{"aap", "noot", "mies"}
        var ds = [...]string{"aap", "noot", "mies"}

        var tictactoe = [3][3]int{[3]int{1, 0, 0},
                [3]int{0, 1, 0},
                [3]int{0, 0, 1}}

        var tictactoe2 = [3][3]int{[3]int{1, 0, 0},
                [3]int{0, 1, 0},
                [3]int{0, 0, 1}}


        fmt.Println("Array t1 and t2 are equal:", (tictactoe == tictactoe2))

        fmt.Println(len(a), len(b))

        fmt.Println("Array a and b are equal:", (a == b))
        fmt.Println("Array c and d are equal:", (cs == ds))

        fmt.Println("\n---------")

        for i := 0; i < len(tictactoe); i++ {
```

Go Slices

GO SLICES

Slices are a more flexible alternative to arrays in Go

- We don't specify a size when declaring a slice
- We cannot compare slices

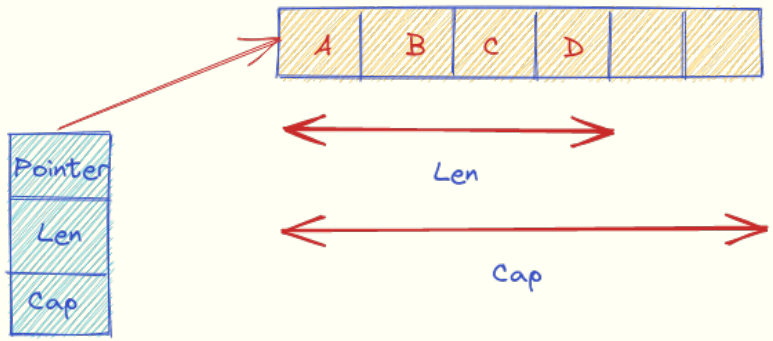
```go
package main

import "fmt"

func main() {

        var x = [...]int{10, 20, 30} // This is an array
        var y = []int{10, 20, 30}    // This is a slice

        var z = []int{1, 5: 4, 6, 10: 100, 15}

        var tictactoe [][]int // A slice of slices

        fmt.Println(x, y, z, tictactoe)

        if tictactoe == nil {
                fmt.Println("Empty slice")
        }
}
```

GO SLICES - OPERATIONS

- Go slices cannot be compared
- You can append to slices
- Mind the call by value used, this is a Go thing
- The `len()` function can be used to count the items in a slice

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var primes = []int{1, 3, 5, 7, 11}
7          var s = []string{"one", "two", "three"}
8
9          fmt.Println(s)
10         s = append(s, "four", "five")
11         fmt.Println(s)
12
13         fmt.Println(len(primes))
14         primes = append(primes, 13, 17, 19)
15         fmt.Println(len(primes))
16
17         new_primes := []int{23, 29, 31}
18         primes = append(primes, new_primes...)
19         fmt.Println(len(primes))
```

GO SLICES - IMPLEMENTATION




```
1  type SliceHeader struct {
2          Pointer uintptr
3          Len     int
4          Cap     int
5  }
```

- `len` returns the nr of items in the slice
- `cap` returns the capacity of the slice
- Slides grow automatically when the nr of items exceeds the capacity of the slice
- When the size exceeds capacity a new slice will be created and the old contents will be copied
- New capacity will be doubled while < 1024 and raised with 25pct above that threshold
- The old slice will be cleaned-up by the Garbage Collector

GO SLICES - RIGHT SIZING 1/3

- We can declare an initially sized growing slice using slice literal or the default nil value
- This creates slices that grow starting with cap == 1, len == 1 -> grow actions are costly
- We can rightsize slices using the `make` function
- With `make` we can specify type, length and optional capacity of the slices to be created

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var x = make([]int, 5, 1024)
7
8          fmt.Println(x, len(x), cap(x))
9  }
```

GO SLICES - RIGHT SIZING 2/3

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          primes := make([]int, 5, 10)
8          fmt.Println(primes, len(primes), cap(primes))
9
10         primes = append(primes, 1)
11         fmt.Println(primes, len(primes), cap(primes))
12
13         more_primes := make([]int, 0, 10)
14         fmt.Println(more_primes, len(more_primes), cap(more_primes))
15
16         more_primes = append(more_primes, 1, 3, 5, 7, 11)
17         fmt.Println(more_primes, len(more_primes), cap(more_primes))
18 }
```

GO SLICES - RIGHT SIZING 3/3

The question; which slice declaration to use?
Goal is keep the nr of time the slice needs to grow to a minimum (performance)

- If there's a chance that the slice doesn't need growth => `var` to create a nil slice
- If there's a need to initialize the slice => `var` with literal values
- If the values don't change => `var` with literal values
- If slice is being used as a buffer => specify non-zero length
- If the exact nr of items is known you can set length and use index to set values
- In all other cases => use `make` with zero length and specified capacity

Slicing slices

- To create a slice from a slice use a `slice expression`
- Bracket notation with starting_offset and ending_offset separated by a colon
- No starting_offset will default to 0
- No ending_offset will default to the end of the slice
- Please note: memory is not copied but shared

GO SLICES - SLICING EXAMPLES

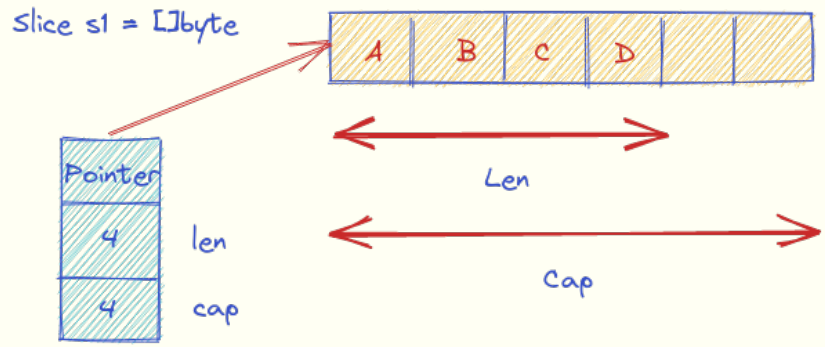
```go
package main

import "fmt"

func main() {
        x := []int{1, 2, 3, 4}
        y := x[:2]
        z := x[1:]
        d := x[1:3]
        e := x[:]

        fmt.Println("x:", x)
        fmt.Println("y:", y)
        fmt.Println("z:", z)
        fmt.Println("d:", d)
        fmt.Println("e:", e)
}
```

Memory from sliced slices is shared not copied:

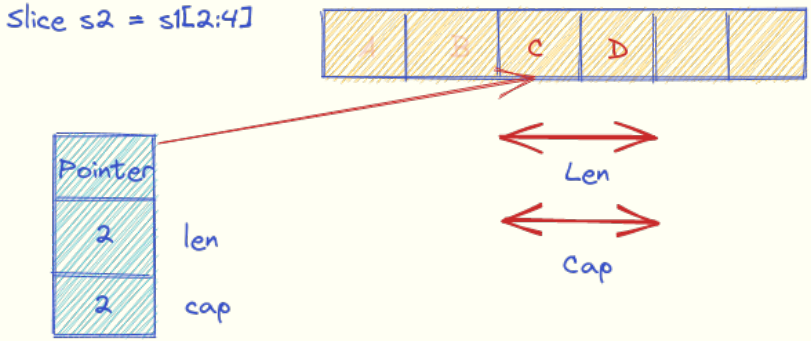
```
1  package main
2
3  import "fmt"
4
5  func main() {
6          x := []int{1, 2, 3, 4}
7          y := x[:2]
8          z := x[1:]
9          x[1] = 20
10         y[0] = 10
11         z[1] = 30
12         fmt.Println("x:", x)
13         fmt.Println("y:", y)
14         fmt.Println("z:", z)
15 }
```

GO SLICES - SLICING EXPLANATION

Parent slice



Sliced slice



Appending to sliced slices is leading to even more confusion

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          x := []int{1, 2, 3, 4}
7          y := x[:2]
8          fmt.Println(y)
9          fmt.Println(cap(x), cap(y))
10         y = append(y, 30)
11         fmt.Println("x:", x)
12         fmt.Println("y:", y)
13 }
```

GO SLICES - THE GO WAY OF APPENDING TO SLICED SLICES

The idiomatic Go way is

- Just don't append to sliced slices
- Use full slice expression (three-part slice expression) which specifies the the last pos in the parent's slice that is available for the subslice; e.g. `[2:4:4]`

GO SLICES - APPENDING TO SLICED SLICES

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          x := make([]int, 0, 5)
7          x = append(x, 1, 2, 3, 4)
8          y := x[:2:2]
9          z := x[2:4:4]
10         fmt.Println("x:", x)
11         fmt.Println("y:", y)
12         fmt.Println("z:", z)
13         fmt.Println(cap(x), cap(y), cap(z))
14         y = append(y, 30, 40, 50)
15         x = append(x, 60)
16         z = append(z, 70)
17         fmt.Println("x:", x)
18         fmt.Println("y:", y)
19         fmt.Println("z:", z)
20 }
```

GO SLICES - CONVERTING ARRAYS TO SLICES

Arrays can be easily converted to slices, but beware of the same memory sharing properties!

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          ar := [4]int{1, 3, 5, 7}
7          y := ar[:2]
8          z := ar[2:]
9          ar[0] = 11
10         fmt.Println("ar:", ar)
11         fmt.Println("y :", y)
12         fmt.Println("z :", z)
13 }
```

GO SLICES - COPYING SLICES 1/2

To produce a copy of a slice that is independent of it's source use `copy`

- `num = copy (dst, src)`
- The destination slice is described by the first argument
- The source slice is described by the second argument
- The copy function returns the number of elements copied

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          x := []int{1, 2, 3, 4}
7          y := make([]int, 4)
8          num := copy(y, x)
9          fmt.Println(x)
10         fmt.Println(y, num)
11
12         x[0] = 5
13         fmt.Println(x)
14         fmt.Println(y)
15 }
```

GO SLICES - COPYING SLICES 2/2

- Copy copies as many values as possible from source to destination slice
- Limitation is the size of the destination slice
- Using sub-slices you can also copy out of the middle of a slice
- You can also copy arrays to slices

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          s := make([]int, 0, 6)
8          s = append(s, 1, 2, 3, 4, 5, 6)
9          d1 := make([]int, 4)
10         d2 := make([]int, 8)
11         num := copy(d1, s)
12         fmt.Println(d1, num)
13         num = copy(d2, s)
14         fmt.Println(d2, num)
15 }
```

GO SLICES - ITERATE OVER SLICES

The `for range` loop can be used to iterate over slices

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var primes = []int{1, 3, 5, 7, 11, 13, 17, 19, 23}
7
8          for _, num := range primes {
9                  fmt.Printf("%d\n", num)
10         }
11
12 }
```

Go Maps

For sequential data you use slices, for data in key-value form one uses maps. There are a couple of ways to declare a map in Go

- To create a map variable and set it to it's default value (nil -> nilMap) use: `var nilMap map[string]int`
- To create an empty map variable use:

```
mtMap = map[string]int{}
```

- To create a non-empty map variabele use:

```
myMap = map[string]int{} {"Go": 2011}, }
```

- To create a presized map variable use:

```
ages := make(map[int][]string, 10)
```

Note that you can't directly write to nilMaps (they are nil) but you can directly write to empty maps.

GO MAPS - EXAMPLES

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          var nilMap map[string]int
8          days := map[string]int{}
9
10         days["january"] = 31
11         days["february"] = 28
12
13         nilMap = make(map[string]int, 12)
14         nilMap["october"] = 31
15
16         fmt.Println("nilMap == ", nilMap["test"])
17         fmt.Printf("January has %2d days\n", days["january"])
18         fmt.Printf("May has %2d days\n", days["may"])
19         fmt.Printf("October has %2d days\n", nilMap["october"])
20 }
```

GO MAPS - RULES AND FACTS

- The key for a map may be anything that is comparable
- So funcs, slices and maps are not allowed
- Maps automatically grow when adding elements
- The zero value for a map is nil
- The zero value for a mapped value is the type's zero value
- Maps are not comparable

GO MAPS - READING AND WRITING

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          const leap_year bool = true
8          month := "february"
9
10         days := make(map[string]int, 12)
11         days["january"] = 31
12         days["february"] = 28
13
14         if leap_year == true {
15                 days[month]++
16         }
17
18         fmt.Printf("%s has %2d days\n", month, days[month])
19 }
```

GO MAPS - ADDING TO AND DELETING FROM MAPS

```
1  package main
2
3  func main() {
4          days := map[string]int{"jan": 31, "feb": 28, "mar": 31, "apr": 30}
5
6          days["may"] = 31
7          days["nuts"] = 30
8          days["june"] = 30
9
10         delete(days, "nuts")
11 }
```

GO MAPS - THE COMMA OK IDIOM

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          days := map[string]int{"jan": 31, "feb": 28, "mar": 31, "apr": 30, "may": 0}
7
8          fmt.Println(days["jan"])
9          fmt.Println(days["may"])
10         fmt.Println(days["nuts"])
11
12         v, ok := days["jan"]
13
14         v, ok = days["may"]
15         fmt.Println(v, ok)
16         v, ok = days["nuts"]
17         fmt.Println(v, ok)
18
19 }
```

GO MAPS - ITERATING OVER MAPS

```go
package main

import "fmt"

func main() {
        days := map[string]int{"jan": 31, "feb": 28, "mar": 31, "apr": 30, "may": 30,
                "june": 30, "july": 31, "aug": 31, "sep": 30, "oct": 31, "nov": 30, "dec": 31}

        // Mind that the order is unspecified by design

        for key, value := range days {
                fmt.Println(key, value)
        }
}
```

Go Structs

GO STRUCTS

Structs are used for storing structurized information

- Structs can contain data of different types
- Structs are ideal for passing information in functions
- In Go structs are a so called user-defined type

GO STRUCTS - EXAMPLE

```go
package main

import "fmt"

func main() {
        type programmer struct {
                name     string
                language string
                mascot   string
        }

        var kjell, jarmo, sill programmer

        kjell = programmer{"gopher", "golang", "gopher"}
        jarmo = programmer{"rustacean", "rust", "crab"}
        niilas := programmer{language: "java", mascot: "duke"}

        sill.name = "pythonista"
        sill.language = "python"
        sill.mascot = "snake"

        fmt.Println(kjell, jarmo, sill, niilas)
        fmt.Printf("Mascot: %s\n", sill.mascot)
}
```

- Anonymous structs are structs without a name associated to them
- Use cases
  - When translating ext data in a struct or v.v. ((un)marshalling)
  - When writing tests for Go

GO STRUCTS - ANONYMOUS STRUCTS EXAMPLES

```go
package main

import "fmt"

func main() {
        var person struct {
                name string
                age  int
                pet  string
        }
        person.name = "kjell"
        person.age = 19
        person.pet = "chinchilla"
        pet := struct {
                name string
                kind string
        }{
                name: "Roderick",
                kind: "chinchilla",
        }
        fmt.Println(person, pet)
}
```

GO STRUCTS - ANONYMOUS FIELD IN STRUCTS

A struct can also have so called anonymous fields

- Anonymous fields are fields that have a type and not an associated name
- Only one anonymous field per type is allowed

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          type employee struct {
7                  string
8                  age    int
9                  salary int
10         }
11
12         emp := employee{}
13
14         emp.string = "Sander"
15         emp.age = 51
16         emp.salary = 7000
17
18         fmt.Println(emp.string)
19 }
```

GO STRUCTS - NESTED STRUCTS

```go
package main

import "fmt"

type Address struct {
        city    string
        country string
}

type User struct {
        name    string
        age     int
        address Address
}

func main() {
        p := User{
                name: "Greta Gopher",
                age:  19,
                address: Address{
                        city:    "Prinsenbeek",
                        country: "NL",
                },
        }
        fmt.Println("Name:", p.name)
        fmt.Println("City:", p.address.city)
}
```

GO STRUCTS - COMPARING AND CONVERTING

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          type firstPerson struct {
8                  name string
9                  age  int
10         }
11
12         type secondPerson struct {
13                 name string
14                 age  int
15         }
16
17         g := secondPerson{}
18
19         f := firstPerson{
20                 name: "Sander",
21                 age:  52,
22         }
23
24         g = secondPerson(f)
25
26         fmt.Println(g.name)
27         fmt.Println(f == g)
28 }
```

GO STRUCTS - COMPARING AND CONVERTING ANONYMOUS STRUCTS

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7          type firstPerson struct {
8                  name string
9                  age  int
10         }
11         f := firstPerson{
12                 name: "Sander",
13                 age:  52,
14         }
15         var g struct {
16                 name string
17                 age  int
18         }
19         g = f
20
21         g.age = 12
22
23         fmt.Println(f == g)
24 }
```

Go Funcs

GO FUNCS

Funcs are the functions in Go

- Functions are first class citizens in Go
- Functions can have 0 or more arguments
- Functions can return 0 or more results/values

GO FUNCS - EXAMPLES

```
1  package main
2
3  import "fmt"
4
5  func add(a int, b int) int {
6          return a + b
7  }
8
9  func div(numerator int, denominator int) int {
10         if denominator == 0 {
11                 return 0
12         }
13         return numerator / denominator
14 }
15
16 func main() {
17         fmt.Println(add(2, 8))
18         fmt.Println(div(8, 4))
19         fmt.Println(div(4, 8))
20         fmt.Println(div(4, 0))
21 }
```

GO FUNCS - MULTIPLE RETURN VALUES

Multiple return values are a common practive in Go

- Multiple return values are specified by their types encloded in `()`
- The preferred way to communicate errors to the caller
- By convention the `error` value is the last or only value returned
- We discuss these in more detail when we discuss error handling in Go

GO FUNCS - MULTIPLE RETURN VALUES EXAMPLES

```go
package main

import (
        "errors"
        "fmt"
        "os"
)

func div(numerator int, denominator int) (int, error) {
        if denominator == 0 {
                return 0, errors.New("Can´t divide by zero")
        }
        return numerator / denominator, nil
}

func main() {

        n := 5
        for n >= 0 {
                res, err := div(15, n)
                if err != nil {
                        fmt.Println(err)
                        os.Exit(1)
                }

                fmt.Println(res)
                n--
        }
}
```

GO FUNCS - IGNORING RETURN VALUES

- One can ignore return values using the blank variable `(_)`
- One can silently have all return values dropped (`Println`)
- Except for lib funcs like `Println` always make it explicit that return values are ignored

GO FUNCS - IGNORING RETURN VALUES - EXAMPLE

```
1  package main
2
3  import (
4          "errors"
5          "fmt"
6  )
7
8  func div(numerator int, denominator int) (int, error) {
9          if denominator == 0 {
10                 return 0, errors.New("Can´t divide by zero")
11         }
12         return numerator / denominator, nil
13 }
14
15 func main() {
16
17         n := 5
18         for n > 0 {
19                 res, _ := div(15, n)
20                 fmt.Println(res)
21                 n--
22         }
23 }
```

GO FUNCS - NAMING RETURN VALUES

- Go allows you to name your return values in the func declaration
- This means they are directly available in the function body, initialized and ready without further ado

GO FUNCS - NAMING RETURN VALUES - EXAMPLES

```
1  package main
2
3  import (
4          "errors"
5          "fmt"
6          "math"
7          "os"
8  )
9
10 func solvqeq(a, b, c float64) (solution1, solution2 float64, err error) {
11
12         discriminant := b*b - 4*c
13         if discriminant < 0 {
14                 err = errors.New("Does not have any real solutions")
15                 return solution1, solution2, err
16         }
17         d := math.Sqrt(discriminant)
18         solution1 = ((-b + d) / 2)
19         solution2 = ((-b - d) / 2)
20         return solution1, solution2, err
21 }
22
23 func main() {
24         // s1, s2, err := solvqeq(3, 2, 5)
25         s1, s2, err := solvqeq(1, -6, 5)
26         if err != nil {
27                 fmt.Println(err)
28                 os.Exit(1)
29         }
30         fmt.Printf("Solutions calculated: x = %f or x = %f\n", s1, s2)
```

GO FUNCS - NAMING RETURN VALUES CAVEATS

- Beware, named return parameters declare an intent (documents)
- What's in the `return` statement is leading
- Be careful not to shadow your named return value

Saving keystrokes while sacrificing code read- and understandability

- Named paramaters allow you to forego on specifying an explict return statement (e.g. with return values)
- It saves keystrokes but requires your code's user to scan back to what the current value is what is returned
- Even if software is hard to write, it should be easy to read

GO FUNCS - BLANK RETURNS - EXAMPLES

```go
package main

import (
        "errors"
        "fmt"
        "math"
        "os"
)

func solvqeq(a, b, c float64) (solution1, solution2 float64, err error) {

        discriminant := b*b - 4*c
        if discriminant < 0 {
                err = errors.New("Does not have any real solutions")
                return // Blank return
        }
        d := math.Sqrt(discriminant)
        solution1 = ((-b + d) / 2)
        solution2 = ((-b - d) / 2)
        return // Blank return
}

func main() {
        s1, s2, err := solvqeq(3, 2, 5)
        // s1, s2, err := solvqeq(1, -6, 5)
        if err != nil {
                fmt.Println(err)
                os.Exit(1)
        }
        fmt.Printf("Solutions calculated: x = %f or x = %f\n", s1, s2)
```

GO FUNCS - VARIADIC PARAMATERS AND SLICES

Go supports functions with a variadic number of parameters

- Only the last parameter can be variadic
- You can also feed the variadic argument a slice

GO FUNCS - VARIADIC PARAMATERS AND SLICES - EXAMPLES

```
1  package main
2
3  import "fmt"
4
5  func Avg(vals ...float64) float64 {
6
7          var sum float64 = 0.0
8          var nr int = 0
9
10         for _, v := range vals {
11                 sum = sum + v
12                 nr++
13         }
14         return sum / float64(nr)
15 }
16
17 func main() {
18
19         s := []float64{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
20         fmt.Println(Avg(3, 4))
21         fmt.Println(Avg(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
22         fmt.Println(Avg(s...))
23 }
```

GO FUNCS - FUNCTIONS ARE VALUES

In Go functions are 1st class citizens

- We can assign functions to variables
- We can create functions on the fly

```
1  package main
2
3  import (
4          "fmt"
5          "math"
6  )
7
8  func main() {
9
10         // declare a function variable
11         getSquareRoot := func(x float64) float64 {
12                 return math.Sqrt(x)
13         }
14
15         // Use the function
16         fmt.Println(getSquareRoot(144))
17 }
```

GO FUNCS - ANONYMOUS FUNCTIONS

Functions defined within the scope of a function are called `anonymous` functions

- An anonymous function has no name
- An anonymous function only exist within the scope it's created
- It's also known as a literal- or lambda- function

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          fmt.Printf("99 (°F) = %.2f (°C)\n",
7                  func(f float64) float64 {
8                          return (f - 32.0) * (5.0 / 9.0)
9                  }(99),
10         )
11 }
```

GO FUNCS - CLOSURES

Functions declared in functions are called `closures`

- Closures are specialized form of anyonymous functions
- Closure => functions declared in functions are able to access and modify variables declared in the outer function
- Closures allow to limit a function's scope (hiding/encapsulation)

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          n := 0
7          counter := func() int {
8                  n += 1
9                  return n
10         }
11         fmt.Println(counter())
12         fmt.Println(counter())
13         fmt.Println(counter())
14         fmt.Println(counter())
15         fmt.Println(counter())
16 }
```

GO FUNCS - FUNCTIONS AS PARAMETERS

Using functions as parameters in functions can be very powerful

- Commonly used pattern like in the sort package
- Allows customization of compare method

GO FUNCS - FUNCTIONS AS PARAMETERS - EXAMPLES

```
1  package main
2
3  import (
4          "fmt"
5          "sort"
6  )
7
8  func main() {
9          type City struct {
10                 Name       string
11                 Province   string
12                 Population int
13         }
14
15         cities := []City{
16                 {"Bergen", "Limburg", 5105},
17                 {"Nijmegen", "Gelderland", 85_023},
18                 {"Maasstricht", "Limburg", 76_820},
19         }
20
21         fmt.Println(cities)
22         sort.Slice(cities, func(i int, j int) bool {
23                 return cities[i].Population < cities[j].Population
24         })
25
26         fmt.Println(cities)
27 }
```

Functions in Go can also return functions

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          areaF := getAreaFunc()
7          res := areaF(2, 4)
8          fmt.Println(res)
9  }
10
11 func getAreaFunc() func(int, int) int {
12         return func(x, y int) int {
13                 return x * y
14         }
15 }
```

Defer delays the execution of functions

- Defer is used to cleanup after you leave the function in any way
- When a function has many exits (returns) you don't want to check at every junction
- Defer can take care of this and will be invoked at every exit
- Beware: the whole functions is evaluated directory (including arguments), only the execution is delayed

GO FUNCS - DEFERING FUNC INVOCATION - EXAMPLES

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          i := 1
7          defer logNum(i) // deferred function call: logNum(1)
8          fmt.Println("main()")
9          i++
10         defer logNum(i)     // deferred function call: logNum(2)
11         defer logNum(i * i) // deferred function call: logNum(4)
12         return              // explicit return
13 }
14
15 func logNum(i int) {
16         fmt.Printf("Num %d\n", i)
17 }
```

Recursive functions in Go

```
1  package main
2
3  import "fmt"
4
5  func fac(n float64) float64 {
6          if n == 0 {
7                  return 1
8          }
9          return n * fac(n-1)
10 }
11
12 func main() {
13         fmt.Println(fac(6))
14         fmt.Println(fac(7))
15 }
```

GO FUNCS - CALL BY VALUE

Go is a so called by value language

- All types are passed by value
- Slices and maps are also passed by value, but there the value is a pointer

GO FUNCS - CALL BY VALUE - EXAMPLES

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var s []int
7          for i := 1; i <= 3; i++ {
8                  s = append(s, i)
9          }
10
11         fmt.Println(s)
12         reverse(s)
13         fmt.Println(s)
14 }
15
16 func reverse(s []int) {
17         for i, j := 0, len(s)-1; i < j; i++ {
18                 j = len(s) - (i + 1)
19                 s[i], s[j] = s[j], s[i]
20         }
21 }
```

Go error handling

Error handling in Go

- Go doesn't support exceptions
- Reasons:
  - Exceptions add a new code path
  - Having errors as returned values allows Go to force them to use them
  - The happy and not so happy flow are clearly recognizable in the code

GO ERROR HANDLING - EXAMPLE

```
1  package main
2
3  import (
4          "errors"
5          "fmt"
6          "math"
7          "os"
8  )
9
10 func solvqeq(a, b, c float64) (solution1, solution2 float64, err error) {
11
12         discriminant := b*b - 4*c
13         if discriminant < 0 {
14                 err = errors.New("Does not have any real solutions")
15                 return solution1, solution2, err
16         }
17         d := math.Sqrt(discriminant)
18         solution1 = ((-b + d) / 2)
19         solution2 = ((-b - d) / 2)
20         return solution1, solution2, err
21 }
22
23 func main() {
24         // s1, s2, err := solvqeq(3, 2, 5)
25         s1, s2, err := solvqeq(1, -6, 5)
26         if err != nil {
27                 fmt.Println(err)
28                 os.Exit(1)
29         }
30         fmt.Printf("Solutions calculated: x = %f or x = %f\n", s1, s2)
```

There a 2 ways to create an Error from a string in Go

- The `errors.New` method
- The `fmt.Errorf` function

```
1  func SquareRoot(f float64) (float64, error) {
2          if f < 0 {
3                  return 0.0, errors.New("error argument is < 0")
4          }
5          return math.Sqrt(f), nil
6  }
7
8  func SquareRoot2(f float64) (float64, error) {
9          if f < 0 {
10                 return 0.0, fmt.Errorf("error f == %f should be >= 0", f)
11         }
12         return math.Sqrt(f), nil
13 }
```

GO ERROR HANDLING - SENTINEL ERRORS

Sentinel errors are used to signal that further processing cannot start or continue

- By convention there names start with `Err` e.g. `ErrFormat`
- Examples: crypto/rsa and archive/zip package

```
1  package main
2
3  import (
4          "archive/zip"
5          "bytes"
6          "fmt"
7  )
8
9  func main() {
10         data := []byte("This is not a zip file")
11         notAZipFile := bytes.NewReader(data)
12         _, err := zip.NewReader(notAZipFile, int64(len(data)))
13         if err == zip.ErrFormat {
14                 fmt.Println("Told you so")
15         }
16 }
```

GO ERROR HANDLING - PANIC

A `panic` can be caused by

- A programming error
- An environmental problem
- Invoked by the program using `panic()`

```
1  package main
2
3  import "os"
4
5  func doPanic(msg string) {
6          panic(msg)
7  }
8
9  func main() {
10         doPanic(os.Args[0])
11 }
```

GO ERROR HANDLING - PANIC AND RECOVERY

We can check if a panic occured in a deferred function and recover if needed/possible
Mind that panic/recovery is not execption handling and should not be used as such!

```
1  package main
2
3  import "fmt"
4
5  func div2(i int) {
6          defer func() {
7                  if v := recover(); v != nil {
8                          fmt.Println(v)
9                  }
10                 fmt.Println("Invoked defer")
11
12         }()
13         fmt.Println(2 / i)
14 }
15
16 func main() {
17         for _, val := range []int{1, 2, 0, 6} {
18                 div2(val)
19         }
20 }
```

Go packages - standard library

The Go distribution contains more than 250 standard built-in packages for common functionality:

- This is a standard library written in Go itself
- The API is the same for all systems (Windows, Linux, Mac OSX)
- Only package call is OS/system specific and non-portable by design
- Information about the built-in packages can be found on:

  `http://golang.org/pkg/`
- You can also use the `go doc` online documentation

Package `fmt` has functionality for formatted output

```
1  package main
2
3  import (
4          "fmt"
5          "os"
6  )
7
8  func main() {
9          var i int = 23
10
11         fmt.Printf("%v\n", i)
12         fmt.Printf("%d\n", i)
13
14         l, err := fmt.Fprintf(os.Stderr, "This message is printed to stderr\n")
15         if err != nil {
16                 fmt.Fprintf(os.Stderr, "Fprintf: %v\n", err)
17         }
18
19         fmt.Printf("%d bytes were written\n", l)
20
21         s := fmt.Sprintf("Hello Gophers")
22         fmt.Printf("String 's' is of length %d and contains: %s\n", len(s), s)
23 }
```

Package `fmt` has functionality for formatted output and input

```
1  package main
2
3  import (
4          "fmt"
5          "os"
6  )
7
8  func main() {
9          var color string
10         q := "What is you favorite color?"
11         fmt.Printf("%s ", q)
12         _, err := fmt.Fscanf(os.Stdin, "%s", &color)
13         if err != nil {
14                 fmt.Fprintf(os.Stderr, "Fscanf: %v\n", err)
15                 os.Exit(1)
16         }
17
18         fmt.Printf("I agree, %s, is a beautiful color\n", color)
19 }
```

Package `fmt` has functionality for formatted output and input

```
1  package main
2
3  import (
4          "fmt"
5          "os"
6  )
7
8  func main() {
9          var color string
10         q := "What is you favorite color?"
11         fmt.Printf("%s ", q)
12         _, err := fmt.Fscanf(os.Stdin, "%s", &color)
13         if err != nil {
14                 fmt.Fprintf(os.Stderr, "Fscanf: %v\n", err)
15                 os.Exit(1)
16         }
17
18         fmt.Printf("I agree, %s, is a beautiful color\n", color)
19 }
```

Package `flags` has functions to parse commandline arguments

```
1  package main
2
3  import (
4          "flag"
5          "fmt"
6          "time"
7  )
8
9  func main() {
10         var label string
11         flag.StringVar(&label, "label", "lift-off", "label to displaced at t = 0")
12         num := flag.Int("n", 10, "Countdown from n to t = 0 ")
13         flag.Parse()
14
15         n := *num
16         i := n
17
18         for i > 0 {
19                 fmt.Printf("t - %3d\n", i)
20                 i--
21                 time.Sleep(time.Second)
22         }
23
24         fmt.Printf("t reached => %s....\n", label)
25 }
```

Package os has functions to query the OS in a portable way

```
1  package main
2
3  import (
4          "fmt"
5          "os"
6          "strings"
7  )
8
9  func main() {
10         var env []string = os.Environ()
11
12         fmt.Println("os.Args: ", os.Args)
13
14         my_hostname, err := os.Hostname()
15         if err != nil {
16                 fmt.Fprintf(os.Stderr, "os.Hostname() -> %v\n", err)
17         }
18
19         fmt.Printf("This go programming runs on: %s\n", my_hostname)
20
21         for i, v := range env {
22                 my_env := strings.Split(v, "=")
23                 fmt.Fprintf(os.Stdout, "%3d: %s is set to %s\n", i, my_env[0], my_env[1])
24         }
25
26         os.Exit(0)
27 }
```

Package `strconv` converts basic data types to and froms strings

```
1  package main
2
3  import (
4          "fmt"
5          "strconv"
6  )
7
8  func main() {
9          i, err := strconv.Atoi("-42")
10         fmt.Printf("%v %T -> %v\n", i, i, err)
11         s := strconv.Itoa(-42)
12         fmt.Printf("%v %T\n", s, s)
13
14         u, err := strconv.ParseUint("42", 10, 64)
15         fmt.Printf("%v %T %v\n", u, u, err)
16
17         s = strconv.FormatFloat(3.1415, 'E', -1, 64)
18         fmt.Printf("%s\n", s)
19 }
```

Package `strings` has all kind of functions to manipulate strings

```
1  package main
2
3  import (
4          "fmt"
5          s "strings"
6  )
7
8  var p = fmt.Println
9
10 func main() {
11         p("Contains:  ", s.Contains("test", "es"))
12         p("Count:     ", s.Count("test", "t"))
13         p("HasPrefix: ", s.HasPrefix("test", "te"))
14         p("HasSuffix: ", s.HasSuffix("test", "st"))
15         p("Index:     ", s.Index("test", "e"))
16         p("Join:      ", s.Join([]string{"a", "b"}, "-"))
17         p("Repeat:    ", s.Repeat("a", 5))
18         p("Replace:   ", s.Replace("foo", "o", "0", -1))
19         p("Replace:   ", s.Replace("foo", "o", "0", 1))
20         p("Split:     ", s.Split("a-b-c-d-e", "-"))
21         p("ToLower:   ", s.ToLower("TEST"))
22         p("ToUpper:   ", s.ToUpper("test"))
23 }
```

Package `regexp` provides functions for processing regexp

```
1  package main
2
3  import (
4          "fmt"
5          "log"
6          "regexp"
7  )
8
9  func main() {
10
11         words := [...]string{"Seven", "even", "Maven", "Amen", "Never", "eleven"}
12
13         for _, word := range words {
14
15                 found, err := regexp.MatchString(".even", word)
16
17                 if err != nil {
18                         log.Fatal(err)
19                 }
20
21                 if found {
22
23                         fmt.Printf("%s matches\n", word)
24                 } else {
25
26                         fmt.Printf("%s does not match\n", word)
27                 }
```

Package `regexp` provides functions for processing regexp

```
1  package main
2
3  import (
4          "fmt"
5          "log"
6          "regexp"
7  )
8
9  func main() {
10
11         words := [...]string{"Seven", "even", "Maven", "Amen", "eleven"}
12
13         re, err := regexp.Compile(".even")
14
15         if err != nil {
16                 log.Fatal(err)
17         }
18
19         for _, word := range words {
20
21                 found := re.MatchString(word)
22
23                 if found {
24
25                         fmt.Printf("%s matches\n", word)
26                 } else {
27
```

Package `regexp` provides functions for processing regexp

```
1  package main
2
3  import (
4          "fmt"
5          "regexp"
6  )
7
8  func main() {
9
10         words := [...]string{"Seven", "even", "Maven", "Amen", "eleven"}
11
12         re := regexp.MustCompile(".even")
13
14         for _, word := range words {
15
16                 found := re.MatchString(word)
17
18                 if found {
19
20                         fmt.Printf("%s matches\n", word)
21                 } else {
22
23                         fmt.Printf("%s does not match\n", word)
24                 }
25         }
26 }
```

Go packages - external packages

GO PACKAGES - EXTERNAL PACKAGES

The Go ecosystem stretches far beyond the standard package library

- `GoMySQL` for Mysql
- `go-pgsql` for PostgreSQL
- `gomongo` for MongoDB
- `redis.go` for Redis
- `GoAuth` for OAuth
- `goprotobuf` for Google Protocol Buffers
- `fsnotify` to get triggered on file operations
- Etc

GO PACKAGES - HOW TO INSTALL

The package `nanoid` provides functionality to create unique ID-strings. It can be found on: `github.com/aidarkhanov/nanoid`

```
1  package main
2
3  import "fmt"
4  import "github.com/aidarkhanov/nanoid"
5
6
7
8  func main() {
9          id := nanoid.New()
10         fmt.Println(id)
11 }
```

GO PACKAGES - HOW TO INSTALL

```
cd projects
go mod init nano-test
go mod tidy
```

GO PACKAGES - MANAGING

```
# Get list of versions of external package

go list -m -versions github.com/aidarkhanov/nanoid

# Get/download package

go get github.com/aidarkhanov/nanoid

# Upgrade to latest version (minor/patch)

go get -u github.com/aidarkhanov/nanoid

# Update go.mod / remove obsoletes etc

go mod tidy

# Request a specific version

go get github.com/aidarkhanov/nanoid@v1.0.5
```

Go packages - Creating your own

GO PACKAGES - CREATING YOUR OWN

This module teaches you how to create you own Go package for self-use and distribution. But what actually is a Go `package`?

- A Go package is a kind of a workspace in your project
- A directory in your project

```
mkdir projects
cd projects
git clone http://thegitcave.org/pascal/pubmod.git
Cloning into 'pubmod'...
warning: redirecting to https://thegitcave.org/pascal/pubmod.git/
warning: You appear to have cloned an empty repository.

cd pubmod
go mod init thegitcave.org/pascal/pubmod
go: creating new go.mod: module thegitcave.org/pascal/pubmod
vi pubmod.go
git add .
gitt commit -m 'Initial commit'
[master (root-commit) f4f4228] Initial commit
 2 files changed, 10 insertions(+)
 create mode 100644 go.mod
 create mode 100644 pubmod.go
git push
warning: redirecting to https://thegitcave.org/pascal/pubmod.git/
Enumerating objects: 4, done.
```

Prior to Go 1.11 or with `GO111MODULE=off`

- Create a directory structure under `$GOPATH/src`
- `recursive/recursive.go`
- Put the main program for example in
- `src/prg/main.go`

```
cd ~/go/src
mkdir rec-funcs
mkdir recursive


.
 — rec-funcs
|   └— main.go
├— recursive
|   └— recursive.go
```

Prior to Go 1.11 or with

`GO0111MODULE=off -> ~/go/src/recursive/recursive.go`

```
1  // Package recursive contains recursive functions
2  package recursive
3
4  // Calculates factorial of n >= 0
5  func Factorial(n float64) float64 {
6          if n == 0 {
7                  return 1
8          }
9          return n * Factorial(n-1)
10 }
```

Prior to Go 1.11 or with

`GO0111MODULE=off -> ~/go/src/recursive/recursive.go`

```
1  package main
2
3  import (
4          "fmt"
5          "recursive"
6  )
7
8  func main() {
9
10         fmt.Println(recursive.Factorial(8))
11 }
```

Post Go 1.11 - Why modules

- Work outside `$GOPATH` workspace
- Specify dependencies and use the most compatible version
- Manage dependencies using the native Go tooling

Recipe for creating a module

1. Create a git repo, e.g. `pubmod`
2. Clone the empty git repo `git clone ...`
3. Initialize go.mod file `go mod init ...`
4. Edit/create your package `vi pubmod.go`
5. Publish the module

   ```
   git add . ; git commit ... ; git push
   ```

GO MODULES - PUBLISH YOUR OWN MODULES

```
mkdir projects
cd projects
git clone http://thegitcave.org/pascal/pubmod.git
Cloning into 'pubmod'...
warning: redirecting to https://thegitcave.org/pascal/pubmod.git/
warning: You appear to have cloned an empty repository.

cd pubmod
go mod init thegitcave.org/pascal/pubmod
go: creating new go.mod: module thegitcave.org/pascal/pubmod
vi pubmod.go
git add .
gitt commit -m 'Initial commit'
[master (root-commit) f4f4228] Initial commit
 2 files changed, 10 insertions(+)
 create mode 100644 go.mod
 create mode 100644 pubmod.go
git push
warning: redirecting to https://thegitcave.org/pascal/pubmod.git/
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 48 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 374 bytes | 374.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To http://thegitcave.org/pascal/pubmod.git
 * [new branch]      master -> master
```

GO MODULES - PUBLISH YOUR OWN MODULES - PUBMOD.GO

```
1
2
3
4
5
```

```go
package pubmod

func Hello() string {
        return "Hello from pubmod!"
}
```

GO MODULES - PUBLISH YOUR OWN MODULES - USING

```
1  package main
2
3  import "fmt"
4  import "thegitcave.org/pascal/pubmod"
5
6  func main() {
7          fmt.Println (pubmod.Hello())
8  }
```

‘

```
1  go mod init main.go
2  go: creating new go.mod: module main.go
3  go: to add module requirements and sums:
4          go mod tidy
5  go mod tidy
6  go: finding module for package thegitcave.org/pascal/pubmod
7  go: downloading thegitcave.org/pascal/pubmod v0.0.0-20220209175630-f4f42284e450
8  go: found thegitcave.org/pascal/pubmod in thegitcave.org/pascal/pubmod v0.0.0-20220209175630-f4f42284e450
```

Versioning scheme: <major>.<minor>.<patch>

- `<major>` == 0, development version
- `<major>` version changes do not guarantee downward API compatability
- `<minor>` version changes do guruantee downward API compatability
- `<patch>` patch release/bug fix

GO MODULES - PUBLISH YOUR OWN MODULES - UPDATING

```
vi pubmod.go
git add pubmod.go
git commit -m 'Added ByeBye function'
[master 35264d9] Added ByeBye function
 1 file changed, 3 insertions(+), 1 deletion(-)
git push
warning: redirecting to https://thegitcave.org/pascal/pubmod.git/
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 48 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 378 bytes | 378.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To http://thegitcave.org/pascal/pubmod.git
   f4f4228..35264d9  master -> master
git tag v0.1.0
git push origin v0.1.0
warning: redirecting to https://thegitcave.org/pascal/pubmod.git/
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To http://thegitcave.org/pascal/pubmod.git
 * [new tag]         v0.1.0 -> v0.1.0
```

```
1
2
3
4
5
6
7
8
9
```

```go
package pubmod

func Hello() string {
        return "Hello from pubmod!"
}

func ByeBye() string {
        return "Byebye Gophers"
}
```

GO MODULES - PUBLISH YOUR OWN MODULES - USING

```
1
2
3  import "fmt"
4  import "thegitcave.org/pascal/pubmod"
5
6  func main() {
7          fmt.Println (pubmod.Hello())
8          fmt.Println (pubmod.ByeBye())
9  }
```

GO MODULES - PUBLISH YOUR OWN MODULES - UPDATE GO.MOD

```
module main.go

go 1.16

require thegitcave.org/pascal/pubmod v0.1.0
```

```
go mod tidy
go: downloading thegitcave.org/pascal/pubmod v0.1.0
```

a

Go Testing

Go provides an integrated way to test your code

- Built-in `testing` package
- Invoke testing with `go test`
- Covers any go file that ends with `_test.go`
- Any function names start start with `Test`
- To prevent golint from complaining, start the function names with `T`

Use cases:

- Running ad-hoc tests from the commandline
- Running automated tests
- Integrate `go test` in your CI/CD pipeline(s)

GO TESTING - SIMPLE PROGRAM

```
1  package main
2
3  import "fmt"
4
5  func sum(x int, y int) int {
6          // This bug is intentionally introduced
7          return x + y + 1
8  }
9  func main() {
10         x := 5
11         y := 5
12         fmt.Printf("%d + %d = %d\n", x, y, sum(x, y))
13 }
```

GO TESTING - SIMPLE TEST CASE

```
1  package main
2
3  import "testing"
4
5  func Test_sum(t *testing.T) {
6          x := 5
7          y := 5
8          want := 10
9          got := sum(x, y)
10         if want != got {
11                 t.Logf("Testing a sum of: %d %d", x, y)
12                 t.Errorf("Sum was incorrect want: %d, but got: %d", want, got)
13         }
14 }
```

GO TESTING - SIMPLE TEST CASE

- The only parameter is: t *testing.T
- The Test method's name t begins with the word Test followed by a word/phrase
- The t variable is used for signalling a failure
- `t.Log` can be used to provide debug information for any test
- The test code must be saved in a file named _test.go such as: seq_test.go

GO TESTING - TABLE DRIVEN TESTING

```
1  package main
2
3  import "fmt"
4
5  func sumseq(n int) int {
6          sum := 0
7          for i := 1; i <= n; i++ {
8                  sum = sum + i
9          }
10
11         return (sum)
12 }
13
14 func main() {
15         fmt.Println(sumseq(5))
16 }
```

GO TESTING - TABLE DRIVEN TESTING

```
1  package main
2
3  import "testing"
4
5  func Testsumseq(t *testing.T) {
6          tables := []struct {
7                  x int
8                  n int
9          }{
10                 {0, 0},
11                 {1, 1},
12                 {2, 3},
13                 {3, 6},
14                 {4, 10},
15                 {5, 15},
16         }
17
18         for _, table := range tables {
19                 total := sumseq(table.x)
20                 if total != table.n {
21                         t.Errorf("sumseq of (%d) was incorrect, got: %d, want: %d.", table.x, total, table.n)
22                 }
23         }
24 }
```

GO TESTING - BENCHMARKING

```
1  package main
2
3  import (
4          "math/big"
5          "testing"
6  )
7
8  func TestFib1(t *testing.T) {
9          result := Fibonacci1(30)
10         expected_result := big.NewInt(832040)
11         if result.Cmp(expected_result) != 0 {
12                 t.Error("result should be 832040, got", result)
13         }
14 }
15
16 func TestFib2(t *testing.T) {
17         result := Fibonacci2(30)
18         expected_result := big.NewInt(832040)
19         if result.Cmp(expected_result) != 0 {
20                 t.Error("result should be 832040, got", result)
21         }
22 }
23
24 func BenchmarkFib1(b *testing.B) {
25         for n := 0; n < b.N; n++ {
26                 _ = Fibonacci1(30)
27         }
28 }
29
30 func BenchmarkFib2(b *testing.B) {
```

Go debugging

The Go runtime can provide stack traces using the GOTRACEBACK variabele

- `GOTRACEBACK=none` delivers no stack trace.
- `GOTRACEBACK=single` default, stacktrace of the current go-routine
- `GOTRACEBACK=all` delivers stacktrace of all user go-routines
- `GOTRACEBACK=system` delivers stacktrace of ALL go-routines (incl. system)
- `GOTRACEBACK=crash` like system, but includes coredumping
- Etc

```
GOTRACEBACK=all go run my-goroutines.go

GOTRACEBACK=system go run helloworld.go

# Ubuntu
sudo systemctl stop apport
sudo systemctl status apport
```

GO DEBUGGING - CORE DUMPING

```
1  package main
2
3  import "math/rand"
4
5  func main() {
6          for {
7                  sum := 0
8                  n := rand.Intn(1e6)
9                  sum += n
10                 if sum%42 == 0 {
11                         panic(":(")
12                 }
13         }
14 }
```

```
1  GOTRACEBACK=crash go run crash-me-debug.go
```

The resulting coredump/core file can be analyzed with `delve`

```
root@odroid-h2p-n02:/home/pascal/go/src# dlv core crash-me-debug core.1627987
Type 'help' for list of commands.
(dlv) bt
 0  0x00000000004587c1 in runtime.raise
    at /usr/lib/go-1.17/src/runtime/sys_linux_amd64.s:165
..
 9  0x000000000042e366 in runtime.gopanic
    at /usr/lib/go-1.17/src/runtime/panic.go:1147
10  0x000000000045b6a8 in main.main
    at ./crash-me-debug.go:11
11  0x0000000000430d33 in runtime.main
    at /usr/lib/go-1.17/src/runtime/proc.go:255
12  0x0000000000456f61 in runtime.goexit
    at /usr/lib/go-1.17/src/runtime/asm_amd64.s:1581
(dlv)
```

Delve can also do live debugging on an executable

```
dlv exec ./crash-me-debug
Type 'help' for list of commands.
(dlv) break main.main
Breakpoint 1 (enabled) set at 0x45b62a for main.main() ./crash-me-debug.go:5
(dlv) continue
> main.main() ./crash-me-debug.go:5 (hits goroutine(1):1 total:1) (PC: 0x45b62a)
     1:        package main
     2:
     3:        import "math/rand"
     4:
=>   5:        func main() {
     6:                for {
     7:                        sum := 0
     8:                        n := rand.Intn(1e6)
     9:                        sum += n
    10:                        if sum%42 == 0 {
```

Go Types

Go `types` allows one to create userdefined types.

```
1  package main
2
3  import "fmt"
4
5  type Liters float64
6  type Gallons float64
7  type Area float64
8  type Volume float64
9
10 type ProgrammingLanguage struct {
11         Name     string
12         Callname string
13         Mascot   string
14         Year     int
15 }
16
17 func main() {
18         var kerosine Gallons = 20.8
19
20         fmt.Printf("%T %f", kerosine, kerosine)
21 }
```

Go Methods

GO METHODS

Go supports methods on user-defined types

- Methods for a type must be defined at the package block level
- Method declarations look just like function declarations
- But with the so called `receiver` addition
- The `receiver` is actually a special argument to the function
- The `receiver` is enclosed in parenthesis
- By convention we use a single char variable for the method's type

GO METHODS - EXAMPLE RECEIVER FUNCTION

```go
package main

import "fmt"

type Lang struct {
        Name     string
        CallName string
        Mascot   string
        Year     int
}

func (l Lang) String() string {
        return fmt.Sprintf("%s %s %s, Published %d", l.Name, l.CallName, l.Mascot, l.Year)
}

func main() {
        l := Lang{Name: "Go", CallName: "Gopher", Mascot: "Gopher", Year: 2009}
        fmt.Println(l.String())

}
```

Go methods - pointer- and value-receivers

When to use which?

- Use a pointer receiver if
  - your method modifies the receiver, use a pointer receiver
  - your method needs to handle nil instances, use a pointer receiver
- Use a value receiver if
  - your method does not modify the receiver

GO METHODS - POINTER- AND VALUE-RECEIVERS

```go
package main

import (
        "fmt"
        "time"
)

type Counter struct {
        total       int
        lastUpdated time.Time
}

func (c *Counter) Increment() {
        c.total++
        c.lastUpdated = time.Now()
}

func (c Counter) String() string {
        return fmt.Sprintf("total: %d, last updated: %v", c.total, c.lastUpdated)
}

func main() {
        var c Counter
        fmt.Println(c.String())
        c.Increment()
        fmt.Println(c.String())
        c.Increment()
        fmt.Println(c.String())
}
```

Go methods - pointer- and value-receivers

- Officially the Increment method should have been called as `(&c_).Increment()` instead of `c.Increment`
- However Go automatically converts it to a pointer type for you if needed
- For a pointer instance Go considers both pointer and value receiver methods to be in the method set for a pointer instance
- For a value instance only the value receiver methods are in the set

A Method Set is a set of methods available to a data type

```
1  package main
2
3  import "fmt"
4
5  type Student struct {
6          name              string
7          courses_entered   int
8          courses_completed int
9  }
10
11 func (s Student) Display() {
12         fmt.Println(s.name)
13 }
14
15 func (s Student) Progress() {
16         fmt.Printf("Progress: %d/%d\n", s.courses_entered, s.courses_completed)
17 }
18
19 func main() {
20         s := Student{"Niilas", 1, 3}
21         s.Display()
22         s.Progress()
23 }
```

GO METHODS - METHODS ARE JUST FUNCTIONS

As methods are functions, when to choose which?

- If your function's logic only depends on it's arguments -> use a plain function
- If your function's logic (also) depends on data outside the function > use a method

GO METHODS - TYPE DECLARATIONS ARE NOT INHERITANCE

- Declaring a user defined type based on another type looks like inheritance, but it isn't
- Both types will have the same underlying type, but no more than that

Go Interfaces

GO INTERFACES

Interfaces are actually the only abstract type in Go

- Interfaces are `named collections of method signatures`
- Remember: a signature is a function/method declaration
- Interfaces are type-safe duck-typing
- By convention interfaces are user with trailin '`-er`'s
  - fmt.Stringer
  - io.Reader
  - json.Marshaler
  - http.Handler
- Alternatives: `-able` like in Recoverable or start it with an `I`

GO INTERFACES - EXAMPLE

```
1  package main
2
3  import (
4          "fmt"
5          "math"
6  )
7
8  type shape interface {
9          area() float64
10         perimeter() float64
11         scale(factor float64)
12 }
13
14 func (r *rectangle) scale (factor float64) {
15         r.width = r.width * factor
16         r.length = r.length * factor
17 }
18
19 type square struct {
20         side float64
21 }
22
23 type rectangle struct {
24         length,width float64
25 }
26
27 func (r rectangle) area() float64 {
28         return r.length * r.width
29 }
30
```

GO INTERFACES - EXPLAINED

- Interfaces are satisfied implicitly
- Multiple types can implemented the same interface
- A type can implement multiple interfaces
- A type that implements an interface can also have other functions
- An interface type can contain references to instances of any of the types that implement this interface

Go Concurrency

Concepts

- Concurrency
- Parallelism
- Important: Concurrency != Paralllism

Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once

GO CONCURRENCY - CONCEPTS

- G: Go routine
- M: OS Thread (Machine)
- P: CPU Core (Processor)

A Go routine can be in one of the following 3 states:

- Executing
- Runnable
- Waiting

- OS Scheduler is always in the lead
- Global RUNQ with all Go routines
- Local RUNQ with Go routines per Processor
- Multiple Go routines can be on a local RUNQ
- Work stealing is possible
- Since Go 1.14 the scheduler is pre-emptive
- A waiting Go routine will be replaced by a Runnable one

GO CONCURRENCY - GO ROUTINES

- Go routine 1 is the main function
- Extra Go routines can be launched using `go` prefix
- GC etc also run as hidden Go routines
- Go routines are cheap (Memory 2k, setup 200ms CPU)

GO CONCURRENCY - WAITGROUPS

- WaitGroups allow function to sync up with Go routines
- WaitGroup.add() to add nr of Go routines
- WaitGroup.done() to signal completion in a Go routine
- WaitGroup.wait() to sync outstanding Go routines

- Go motto: share memory by communicating not communicate by sharing memory
- Channels can be used to share information between Go routines
- Channels can support all kinds of data types
- Channels can be unbufferd of buffered (capacity)
- Channels will block writers if buffer is full
- Channels will block readers if buffer is empty
- Use select to 'probe' channels

- Critical sections
- Mutex -> lock for Read and Write
- RWMutex -> different locks for Read and Write