An inspirational introduction to the Go Programming language

Session 01

Golang course by Exadel

03 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Introduction

▶️ A few words about myself:

- Started learning Go at the end of 2015 (Go toolchain version 1.5)
- **Before:** Java, Python, C'99, Kotlin, Groovy, Scala, Perl etc.
- **Not:** JS, C++, TypeScript, PHP, Ruby, Rust (yet), Closure (yet)

▶️ *How* do I start my journey with Go?

- Encountered a mind-blowing article: "Moving a team from Scala to Golang" (http://jimplush.com/talk/2015/12/19/moving-a-team-from-scala-to-golang/)
- An article from the technical director's point of view.
- Enterprise business scalability, team growth, new hiring, standards, etc.
- **Example:** Scala path, C++ path

Agenda

- Goals
- Origins of Go
- Go Language Benefits
- Language Original Goals
- Go Building Blocks
- Documentation
- Language implementation
- Tools
- Pros / Cons
- Next time...

Goals

- Course structure is based on official Golang specification
- Provide enough information to pass interview (theory)
- Philosophical view into Go ecosystem
- What can we learn from Go ecosystem and history?

▶️ *Assumptions:*

- Disclaimer: this course is based on **personal** experience, very opinionated
- In my mind: **language is a tool to express (and share) ideas**
- Language related to a philosophy and design
- Idiomatic vs Non-idiomatic approach
- You have to finish tutorial at "A tour of Go" (https://go.dev/tour/welcome/1)
- You have to know about basic CS concept, like "pointers" (https://dave.cheney.net/2017/04/26/understand-go-pointers-in-less-than-800-words-or-your-money-back), stack/heap, code compilation/interpretation, etc.

Origins of Go 🤓

▶️ Concurrency in 2022 is still a **key**.

▶️ Go was started in 2007 by Rob Pike, Robert Griesemer, Ken Thompson. Russ Cox and Ian Lance Taylor joined soon after.

▶️ There is a story that Go was designed while waiting for a C++ program to compile. 🤔

▶️ In reality, Go was a response to many of the **frustrations** that Rob, Robert and Ken experienced with the languages in use at Google at that time; C++, Java, and Python.

▶️ Rob Pike's 2012 paper, Less is exponentially more (http://commandcenter.blogspot.com.au/2012/06/less-is-exponentially-more.html) gives an excellent overview of the environment that gave rise to Go.

▶️ Go is an open source project.

▶️ All the commits, code review, design, debate, etc, are all done in the open on a public mailing lists.

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Go Language Benefits

▶️ Go language is rather simple, but not easy.

▶️ The Go ecosystem is highly mature, with many stable open-source libraries to help build robust and efficient applications.

▶️ The Go programming language has withstood the test of time and has been adopted by big-name and trusted companies (https://go.dev/solutions/#case-studies).

▶️ Go promises V1 compatibility: code written ten years ago still works with new compiler versions without significant and fatal changes.

▶️ Now, Go is the first cloud-native language of choice.

- check the cloud-native landscape (https://landscape.cncf.io/)

▶️ The main language for DevOps (majority of tooling have been developed using Go, such as Docker, Kubernetes, Terraform, Packr, etc.)

▶️ With Go, it is possible to utilize hardware and deliver highly concurrent solutions fully

The Go programming language 😎

👍 Modern 😂

👍 Compact, concise, general-purpose

👍 Imperative, statically type-checked, dynamically type-safe

👍 Garbage-collected 🤔

👍 Opinionated, **no warnings**, unused local vars and imports are an error 🤔

👍 Strong support for **concurrency**: Go concurrency model based on CSP: C. A. R. Hoare: Communicating Sequential Processes (CACM 1978) 🤔

👍 Compiles to native code, statically linked - single binary, easy to transfer and run

👍 Fast compilation, efficient execution

👍 **Conventions** first!

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

What's next?

▶️ Go 1.0 marked a line in the sand (great for the business) 🤔

- API **stability**. The Go 1 compatibility document (http://golang.org/doc/go1compat) 🙏
- **No** major language changes (there have been a few minor additions) + go generics since go1.18

▶️ Rough **6** month release cycle; **3** month change window, **3** month stabilisation 🤔

▶️ Big ticket features are discussed before the change window opens

▶️ Today we have Go **1.19**

▶️ Go can offer stability ⚖️

- **Pro**: Your code you write today will continue to work with newer versions of Go
- **Con**: The bar for adding a new feature is now insurmountably high

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Language Original Goals

Original design goals (pt. 1/3)

- regular syntax (don't need a symbol table to parse) 🤔
- garbage collection (only)
- no header files
- explicit dependencies
- no circular dependencies 🤔
- constants are just numbers
- int and int32 are distinct types 🤔
- letter case sets visibility 🤔
- methods for any type (no classes) 🤔
- no subtype inheritance (no subclasses) 🤔
- package-level initialization and well-defined order of initialization
- files compiled together in a package

Original design goals (pt. 2/3)

- package-level globals presented in any order
- no arithmetic conversions (constants help) 🤔
- interfaces are implicit (no "implements" declaration) 🤔
- embedding (no promotion to superclass) 🤔
- methods are declared as functions (no special location) 🤔
- methods are just functions
- interfaces are just for methods (no data) 🤔
- methods match by name only (not by type)
- no constructors or destructors
- postincrement and postdecrement are *statements*, not *expressions*
- no preincrement or predecrement
- assignment is not an expression

Original design goals (pt. 3/3)

- no pointer arithmetic
- memory is always zeroed 🤔
- legal to take address of local variable 🤔
- no "this" in methods
- segmented stacks
- no const or other type annotations 🤔
- no exceptions 🤔
- builtin string, slice, map 🤔
- array bounds checking

Go Building Blocks

Hello, World! 👍

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界!")
}
```

Run

- Unicode
- Programs are organized in packages
- A package is a set of package files
- A package file expresses its dependencies on other packages via import declarations
- The remainder of a package file is a list of (constant, variable, type, and function) declarations

Hello, World! Internet-style 😳

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func HelloServer(w http.ResponseWriter, req *http.Request) {
    log.Println(req.URL)
    fmt.Fprintf(w, "Hello, 世界!\nURL = %s\n", req.URL)
}

func main() {
    fmt.Println("please connect to localhost:7777/hello")
    http.HandleFunc("/hello", HelloServer)
    log.Fatal(http.ListenAndServe(":7777", nil))
}
```

Run

Program elements ⚒

- ▶️ Variables
- ▶️ Constants (Typed or without type, evaluated at compile-time) 🤔
- ▶️ Predeclared types, the usual suspects.
- ▶️ Composite types:

  `array, struct, pointer, function,`
  `slice, map, channel`

- ▶️ Abstract type - **interface**
- ▶️ Type constraints, aka **Generics**

Control structures ⚒

- Curly braces (C style)
- Many cleanups: mandatory braces, **no** parentheses for conditionals, **implicit** break in switches, no semicolons, etc.
- Unified **for** syntax 🤔
- Modern **switch** statement
- **range** over arrays, slices, maps and channels
- Multiple assignments
- Well-known: **break**, **goto**, **continue**, **fallthrough**

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Functions ⚒

- Regular functions

```
func Sin(x float64) float64
func AddScale(x, y int, f float64) int
```

- Multiple return values - hard to chain method invocations 🤔

```
func Write(data []byte) (written int, err error)
```

- Variadic parameter lists without magic

```
func Printf(format string, args ...interface{})
```

- Functions are first-class values 🤔

```
var delta int
return func(x int) int { return x + delta }
```

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Methods ⚒

Methods are functions with a *receiver* parameter: 🤔

```
func (p Point) String() string {
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
}
```

The receiver binds the method to its *base_type* (Point):

```
type Point struct {
    x, y int
}
```

Methods are invoked via the usual dot notation:

```
func main() {
    p := Point{2, 3}
    fmt.Println(p.String())
    fmt.Println(Point{3, 5}.String())
}
```

Run

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Methods can be defined for any user-defined type!

For the Weekday type:

```
type Weekday int
```

Define String method on Weekday:

```
func (d Weekday) String() string { // ...
```

```
func main() {
    fmt.Println(Mon.String())
    fmt.Println()

    for d := Mon; d <= Sun; d++ {
        fmt.Println(d.String())
    }
}
```

Run

Method calls via non-interface types are *statically* dispatched. 🤔

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Structs vs Interfaces

Type declarations

- Composition from left-to-right (Pascal style), like in all modern languages 🤔:
- A *type* declaration defines a *new* type 🤔:

```
type Weekday int

type Point struct {
    x, y int
}
```

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Interface types

- Abstract (no data!) 🤔
- Define (possibly empty) set of method signatures 🤔
- Values of *any_type* that implement all methods of an interface can be assigned to a variable of that interface.

Examples:

```
type Anything interface{}  // empty interface

type Stringer interface {
    String() string
}

type Sorter interface {
    Len() int
    Swap(i, j int)
    Less(i, j int) bool
}
```

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

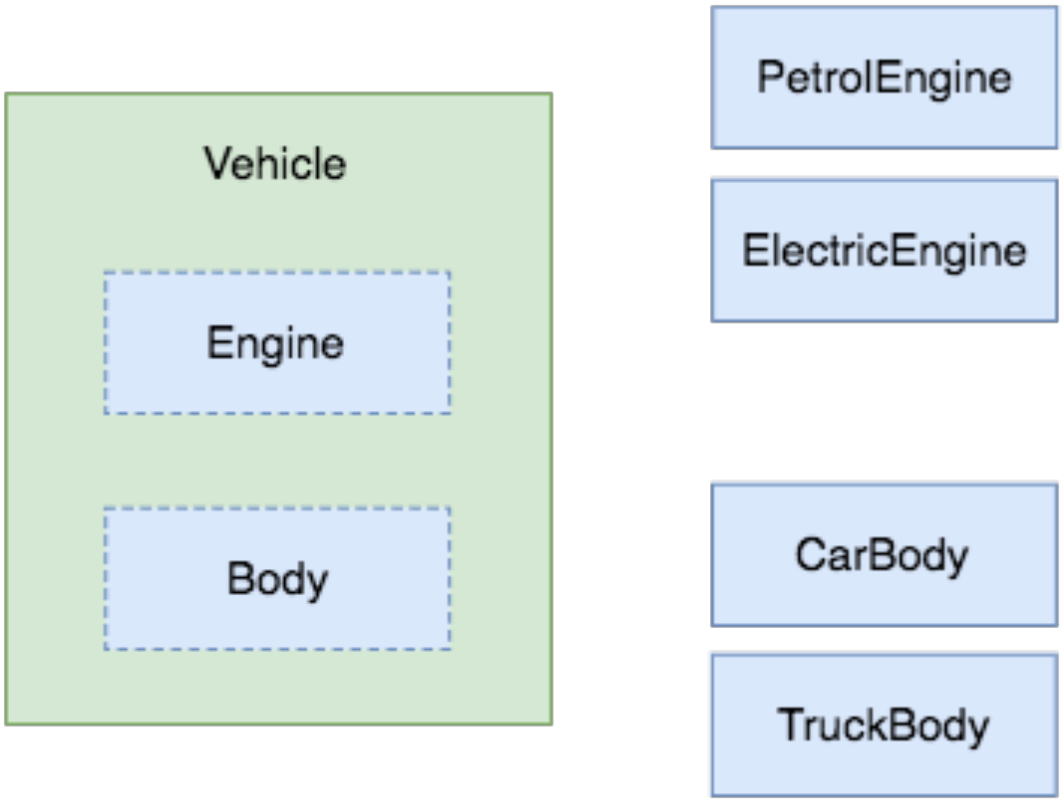
What to choose?

▶️ Clear separation:

- Structures for data!
- Interfaces for behavior!

Practical example design overview

Let's model a vehicle that can have different types of engines and bodies:



Source: "5 things about programming I learned with Go" by MICHAŁ KONARSKI

(http://mjk.space/5-things-about-programming-learned-with-go/)

Practical example implementation

```
type Engine interface {
    Refill()
}

type Body interface {
    Load()
}

type Vehicle struct {
    Engine
    Body
}
```

Function f is launched as 3 different goroutines, all running concurrently:

```
func main() {
    vehicle := Vehicle{Engine: PetrolEngine{}, Body: TruckBody{}}
    vehicle.Refill()
    vehicle.Load()
}
```

Run

Source: "5 things about programming I learned with Go" by MICHAŁ KONARSKI

(http://mjk.space/5-things-about-programming-learned-with-go/)

Error handling

Always handle errors

```
type error interface {
    Error() string
}
```

- ▶️ Go Proverb: **errors are just values**
- ▶️ Go Proverb: **don't just check errors, handle them gracefully**
- ▶️ **Linters** will help
- ▶️ Error root cause? Use https://github.com/pkg/errors instead!

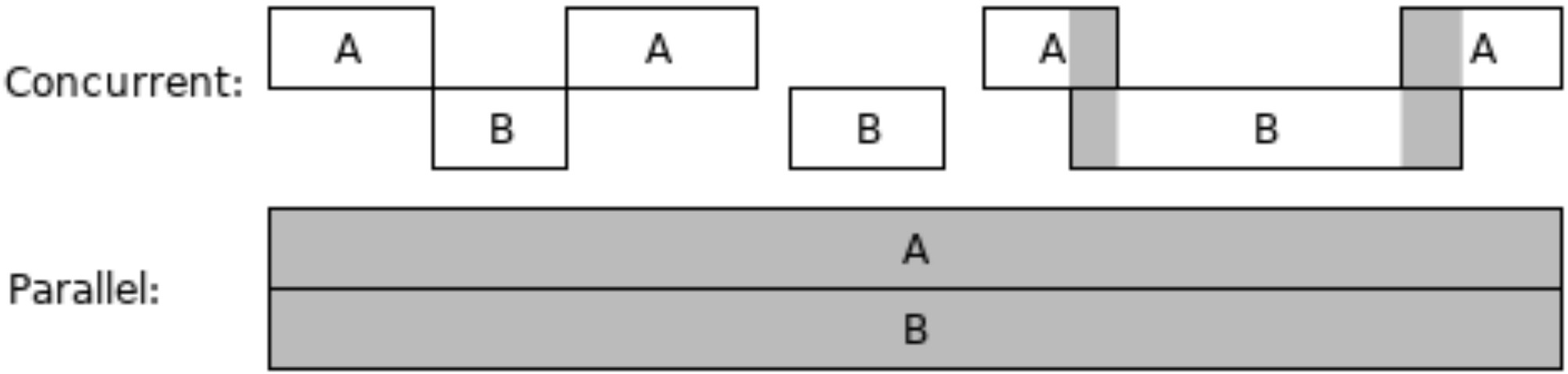

if err != nil {
return nil, err
}
Strg

Concurrency 💎

Concurrency vs. parallelism

- Concurrency is about dealing with lots of things at once
- Parallelism is about doing lots of things at once
- Concurrency is about structure, parallelism is about execution

Source: https://talks.golang.org/2012/waza.slide#8 (https://talks.golang.org/2012/waza.slide#8)



Source: https://go101.org/article/channel.html (https://go101.org/article/channel.html)

Goroutines 💣

- The go statement launches a function call as a goroutine

```
go f()
go f(x, y, ...)
```

- A goroutine runs **concurrently** (but not necessarily in **parallel**)
- A goroutine is a thread of control within the program, with its own local variables and stack.
- Much cheaper to create and schedule than operating system threads.

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

A simple example

```
func f(msg string, delay time.Duration) {
    for {
        fmt.Println(msg)
        time.Sleep(delay)
    }
}
```

Function f is launched as 3 different goroutines, all running concurrently:

```
func main() {
    go f("A--", 300*time.Millisecond)
    go f("-B-", 500*time.Millisecond)
    go f("--C", 1100*time.Millisecond)
    time.Sleep(20 * time.Second)
}
```

Run

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Communication via channels

A channel type specifies a channel value type (and possibly a communication direction):

```
chan int
chan<- string  // send-only channel
<-chan T       // receive-only channel
```

A channel is a variable of channel type:

```
var ch chan int
ch := make(chan int)  // declare and initialize with newly made channel
```

A channel permits *sending* and *receiving* values:

```
ch <- 1   // send value 1 on channel ch
x = <-ch  // receive a value from channel ch (and assign to x)
```

Channel operations synchronize the communicating goroutines.

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Communicating goroutines

Each goroutine sends its results via channel ch:

```
func f(msg string, delay time.Duration, ch chan string) {
    for {
        ch <- msg
        time.Sleep(delay)
    }
}
```

The main goroutine receives (and prints) all results from the same channel:

```
func main() {
    ch := make(chan string)
    go f("A--", 300*time.Millisecond, ch)
    go f("-B-", 500*time.Millisecond, ch)
    go f("--C", 1100*time.Millisecond, ch)

    for i := 0; i < 100; i++ {
        fmt.Println(i, <-ch)
    }
}
```

Run

Documentation

Documentation: Godoc vs Javadoc

- hystrix-go (Based on the java project of the same name, by Netflix):

  https://godoc.org/github.com/afex/hystrix-go/hystrix (https://godoc.org/github.com/afex/hystrix-go/hystrix)

- Netflix Hystrix: Latency and Fault Tolerance for Distributed Systems:

  http://netflix.github.io/Hystrix/javadoc/ (http://netflix.github.io/Hystrix/javadoc/)

- Feel the difference

Language implementation

Small, well polished standard library 🗡

- Great support for testing, fuzzing, benchmarking, coverage reporting.
- The usual libc/libm cast of characters.
- Support for networking, HTTP, TLS.
- Support for common encodings, XML, JSON.
- Support for compression, tar, etc.

golang.org/pkg/ (http://golang.org/pkg/)

To a first order approximation, the std lib provides only the things needed to build Go itself.

Source: "Introduction to Go" by Dave Chaney (https://github.com/davecheney/introduction-to-go)

Excellent cross-platform support

- Windows, OS X, Linux, {Free,Open,Net,DragonFly}BSD, Solaris
- NaCl, Google's Native Client runtime
- Mobile application development (iOs/Android) (example: Tailscale (https://tailscale.com/))

▶️ Cross platform is defined as the super set of all platforms with some not supporting a particular feature, rather than the lowest common denominator.

▶️ Two production compiler implementations

- gc toolchain
- gccgo toolchain

▶️ Other implementations, llgo, tardis-go, etc, keep us honest by writing to the spec.

Tools

Basic tooling

`/usr/bin/go`

- ▶️ Simple, zero configuration, build tool.
- ▶️ Supports conditional compilation via **suffixes** and **build tags**.
- ▶️ Support for cross compilation.

```
GOARCH=386 GOOS=windows go build mycmd
```

More tools

- Read package documentation

```
godoc $PKG
```

- Lint your code

```
go vet $PKG
```

- Solve code formatting issues, once and for all

```
go fmt $PKG
```

- Built in testing framework (http://golang.org/pkg/testing/), including benchmarking and coverage reports (http://blog.golang.org/cover)

```
go test $PKG
go test -cover $PKG
```

- Built in race detector (http://blog.golang.org/race-detector)

```
go {build,test,install} -race $PKG
```

PROS / CONS

Go PROS 🤓

- ▶️ Easy to get back to your project after some time
- ▶️ Easy to introduce project to new members, even they are not aware of Golang
- ▶️ Easy to read your and others code
- ▶️ Good learning materials available
- ▶️ Security checks for dependencies
- ▶️ Great tooling, IDE/editor support

Go CONS ☠️

- ▶️ No immutability
- ▶️ No enums
- ▶️ Variable shadowing - use linters
- ▶️ Go forces you to write some low-level code (over and over again)

Next time...

▶️ Session02: How to start writing Go code, structure, modules introduction

- Go toolchain installation
- Basic interactions with go toolchain
- Environment variables
- Managing different go toolchain versions using gvm

Thank you

Golang course by Exadel

03 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)