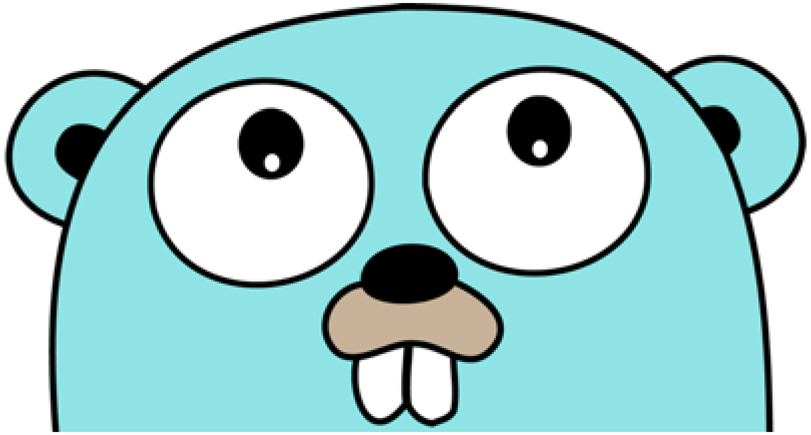
Go Basics

Daniel Hodan
Avocode

AVOCODE



What is Go?

"Go is an open source programming language that makes it easy to build simple, reliable, and efficient software."

golang.org (golang.org)

Why Go?

What are a typical Go applications and why?

- Distributed Systems
- Web Services
- Workers
- Tools

Go Environment

Install Go with Brew

```
> brew install go
```

Set GOROOT and GOPATH environment variables

```
> echo $GOROOT
/usr/local/Cellar/go/1.7.3/libexec

> echo $GOPATH
/Users/czertbytes/Avocode/

> cat ~/.zshrc
# Golang
export GOROOT=/usr/local/Cellar/go/1.7.3/libexec
export GOPATH=$HOME/Avocode/
export PATH=$PATH:$GOPATH/bin
export PATH=$PATH:$GOROOT/bin
```

Standard Library

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

- CSP Concurrency: Goroutines, Select, Channels
- RPC, HTTP(2) clients and servers
- JSON and XML encoding/decoding
- Encoding, cryptography, compress algorithms
- Mobile - Android and iOS

github.com/avelino/awesome-go (https://github.com/avelino/awesome-go)

Editor support

- Vim
- Emacs
- Sublime Text
- LiteIDE
- Plugin for IntelliJ

Autocomplete daemon

github.com/nsf/gocode (https://github.com/nsf/gocode)

The Easy way - Visual Studio Code

code.visualstudio.com/docs/?dv=osx (https://code.visualstudio.com/docs/?dv=osx)

Go Syntax

Hello Go

Go looks familiar.

```go
package main

import "fmt"

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

Run

Run examples remotely

play.golang.org (https://play.golang.org)

Run examples locally

Check your GOROOT and GOPATH

```
> echo $GOROOT
/usr/local/Cellar/go/1.7.3/libexec

> echo $GOPATH
/Users/czertbytes/Avocode/

> git clone https://github.com/czertbytes/workshop-samples

> go run main.go
Hello, Gophers!

> go build main.go
> ./hello
Hello, Gophers!
```

Packages, Types, Variables and Functions

Packages

The design of Go's package system combines some of the properties of libraries, name spaces, and modules into a single construct.

```
package mypkg
```

- Every Go file starts with keyword `package`
- Import dependecy in package with `import`

```
import "mypkg"

import "github.com/czertbytes/foo"
```

- Letter case sets visibility
- No circular or unused dependencies

Packages example

```
package main

import (
    _ "encoding/json" // blank identifier
    "fmt"
    m "math"
)

func main() {
    fmt.Printf("In Go π is defined as %f", m.Pi)
}
```

Run

Types

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64 uintptr
- byte
- rune
- float32 float64
- complex64 complex128
- interface{}

Types

Custom types with keyword `type`.

```
type ID uint64
```

The expression `T(v)` converts the value v to the type `T`.

```
id := ID(1433)
```

Variables

Declaration syntax is closer to Pascal's than to C's. Read from left to right.

```
var myVar string = "Avocode"
```

Idiomatic derived declaration (type inference)

```
myVar := "Avocode"
```

Default values

- Type `string` => ""
- Type `bool` => false
- Numeric types => 0

Constants

```
const myConst = "Avocode"
```

Variables example

```
package main

import (
    "fmt"
    "math/cmplx"
)

func main() {
    var (
        b        = true
        i uint64 = 1<<64 - 1
        z        = cmplx.Sqrt(-5 + 12i)
    )
    s := "Avocode and Gophers"

    fmt.Printf("b has type %T and value %t\n", b, b)
    fmt.Printf("i has type %T and value %d\n", i, i)
    fmt.Printf("z has type %T and value %f\n", z, z)
    fmt.Printf("s has type %T and value %s\n", s, s)
}
```

Run

Functions

A function can take zero or more arguments and can return any number of results.

```
func doSomeWork(arg1 string, arg2 int) (string, error) {
    ...
    return "Result is string", nil
}
```

Go has first-class functions and closures.

```
strLen := func(s string) int {
    return len(s)
}
```

Functions example

```go
package main

import "fmt"

func fnPrint(fn func() (string, string)) {
    a, b := fn()
    fmt.Printf("%s\n%s\n", a, b)
}

func main() {
    a := "First Gophers"
    b := "Second Avocode"
    fnPrint(func() (string, string) {
        return b, a
    })
    fnPrint(func() (string, string) {
        return a, ""
    })
}
```

Run

Flow control statements

For

A `for` is the only looping construct in language

```
for i := 0; i < 10; i++ {
    ...
}
```

Like while in C

```
for myCondition {
    ...
}
```

Infinite loop

```
for {
    ...
}
```

For example

```
package main

import "fmt"

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

Run

Range

A range form of the for loop iterates over a "stream"

```
// Without range
for i := 0; i < len(s); s++ {
    ...
}

for i, v := range s {
    ...
}
```

Not only strings! Slices, maps and read from channel.

Range example

```
package main

import "fmt"

func main() {
    for pos, char := range "Avocode" {
        fmt.Printf("[%d] = %c\n", pos, char)
    }
}
```

Run

If and Else

Basic conditional statement as we all know.

```
if myCondition {
    ...
} else {
    ...
}
```

Go has an initialization statement

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

If and Else example

```
package main

import "fmt"

func eval(i int) (string, error) {
    if i%2 == 0 {
        return fmt.Sprintf("*** %.2d", i), nil
    } else if i == 5 {
        return fmt.Sprintf("%b ###", i), nil
    }

    return "", fmt.Errorf("Ugly number")
}

func main() {
    for i := 1; i < 10; i++ {
        if res, err := eval(i); err == nil {
            fmt.Printf("%d => %s\n", i, res)
        }
    }
}
```

Run

Switch

A switch is if/else if/else statement on steroids.

```
switch myExpression {
    case a:
        ...
    default:
        ...
}
```

Switch on truthy case expression

```
switch {
    case len(a) == 0:
        ...
    default:
        ...
}
```

Switch example

```go
package main

import "fmt"

func main() {
    vars := []interface{}{"Avocode", func(a int) int { return a * 2 }, true, 34}
    for _, v := range vars {
        switch t := v.(type) {
        case string:
            fmt.Printf("String with value %q\n", t)
        case int:
            fmt.Printf("Number with value %d\n", t)
        case bool:
            fmt.Printf("Boolean with value %t\n", t)
        default:
            fmt.Printf("Unknown type %T\n", t)
        }
    }
}
```

Run

Defer

A defer statement defers the execution of a function until the surrounding function returns.

```
defer myFunc()
```

Defer example

```go
package main

import "fmt"

func main() {
    fmt.Println("Fn start")
    for i := 0; i < 10; i++ {
        fmt.Printf("For %d start\n", i)
        defer fmt.Printf("Defer %d\n", i)
        fmt.Printf("For %d end\n", i)
    }
    fmt.Println("Fn end")
}
```

Run

Pointers, Structs, Arrays, Slices and Maps

Pointers

Go has pointers. A pointer holds the memory address of a variable.

```
addr := &myVar

val = *addr
```

Unlike C, Go has no pointer arithmetic.

Pointers example

```
package main

import "fmt"

func main() {
    a := 7
    ptrA := &a
    fmt.Printf("ptrA has type %T, address %x and\nvalue %d\n", ptrA, ptrA, *ptrA)

    // b := ptrA + 3
    // will not work, ptrA is *int and 3 is int

    b := *ptrA + 3
    fmt.Printf("b: %d\n", b)
}
```

Run

Structs

A struct is a collection of fields.

```
type myStruct struct {
    Foo string
    ...
}
```

- No classes
- No inheritance - Embedding
- No constructors
- No annotations - Tags
- No user-defined generics

Structs

Allocation with new

```
a := new(myStruct) // returns pointer to struct

b := myStruct{
    Foo: "Gopher",
}
```

Setting the value

```
b.Foo = "Avocode"
```

Getting the value

```
fmt.Println(b.Foo)
```

Structs example

```
package main

import "fmt"

type Person struct {
    Name, Position string
    Company
}

type Company struct {
    Name string
}

func main() {
    p := Person{
        Name:     "Daniel Hodan",
        Position: "Gopher",
        Company: Company{
            Name: "Avocode",
        },
    }
    fmt.Printf("%s works at %s\n", p.Name, p.Company.Name)
}
```

Run

Arrays

The type `[n]T` is an array of n values of type `T`. Array has fixed size, cannot be resized.

```
var a [10]int

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

- Arrays are values.
- Assigning one array to another copies all the elements.
- The size of an array is part of its type.
- Arrays are just building blocks for slices.

Arrays example

```
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "Gopher"
    a[1] = "Avocode"
    fmt.Println(a[0], a[1])
    fmt.Println(a)

    primes := [...]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primes)
}
```

Run

Slices

The type []T is a slice with elements of type T. Slice is flexible view into the elements of an array.

```
sliceA := []int{1, 2, 3}
sliceB := [][]int{
    []int{1, 2, 3},
    []int{4, 5, 6},
    []int{7, 8, 9, 0},
}
```

A slice does not store any data, it just describes a section of an underlying array.

The zero value of a map is nil.

Slices

Allocation with `make`

```
slice := make([]int, 0, 5)
```

Function append

```
slice = append(slice, 10)
```

Function copy

```
copy(newSlice, slice)
```

Slices example

```go
package main

import "fmt"

func main() {
    a := []int{1, 2, 3, 4, 5}
    printSlice("a", a)
    b := make([]int, len(a))
    copy(b, a)
    b = append(b, 6)
    printSlice("b", b)
    c := b[2:4]
    printSlice("c", c)
}

func printSlice(s string, x []int) {
    fmt.Printf("%s: len=%d cap=%d %v\n",
        s, len(x), cap(x), x)
}
```

Run

Maps

A map is data structure that associate values of one type (the key) with values of another type (the element or value). The zero value of a map is nil.

The key can be of any type for which the equality operator is defined.

```
m := map[string]int{
    "Gopher": 34,
    "Avocode": 593,
}
```

Maps

Allocation with `make`

```
m := make(map[string]int)
```

Setting the value

```
m["key"] = 12
```

Getting value by key

```
val, found := m["key"]
```

Function `delete`

```
delete(m, "key")
```

Maps example

```go
package main

import "fmt"

type Person struct {
    Name, Position string
}

type ID uint

var m = map[ID]Person{
    1: Person{"Daniel Hodan", "Gopher"},
    2: Person{"Joe Doe", "React Developer"},
}

func main() {
    for k, v := range m {
        fmt.Printf("%d: %s\n", k, v.Name)
    }
}
```

Run

Methods, Pointer receivers and Interfaces

Methods

A method is just a function with a receiver argument. Limited to same package as receiver.

Does not modify the value of receiver, copies the value.

```
func (t MyType) Do(a, b int) {
    ...
}
```

What is the receiver?

```
(t MyType)
```

Methods example

```
package main

import "fmt"

type Person struct {
    Name, Position string
}

func (p Person) String() string {
    return fmt.Sprintf("Person: %s, %s", p.Name, p.Position)
}

func main() {
    persons := []Person{
        Person{"Daniel Hodan", "Gopher"},
        Person{"Joe Doe", "React Developer"},
    }

    for _, p := range persons {
        fmt.Println(p)
    }
}
```

Run

Pointer receivers

A pointer receiver is just a function with a pointer receiver argument. Limited to same package as receiver.

Modify the value of receiver and avoiding copying the value.

```
func (t *MyType) Do(a, b int) {
    ...
}
```

Pointer receivers example

```
func (p *Person) ConvertToGopherism() {
    p.Position = "Gopher"
}

func main() {
    persons := []Person{
        Person{"Daniel Hodan", "Gopher"},
        Person{"Joe Doe", "React Developer"},
    }

    for _, p := range persons {
        p.ConvertToGopherism()
        fmt.Println(p)
    }
}
```

Run

Interfaces

An `interface` type is defined as a set of method signatures.

```
type Shouter interface {
    Shout() string
}
```

Duck typing - If something can do this, then it can be used here.

A type can implement multiple interfaces.

Interfaces - sample

```
func (r FirstWordReader) Read(p []byte) (n int, err error) {
    v, err := ioutil.ReadAll(r.R)
    if err != nil {
        return 0, err
    }
    n = len(v)
    for i, c := range v {
        if c == ' ' {
            n = i
            break
        }
    }
    copy(p, v[:n])
    return n, io.EOF
}

func main() {
    //f, _ := os.Open("./read.txt")
    //fwr := FirstWordReader{f}
    fwr := FirstWordReader{strings.NewReader("Gophers in Avocode!")}
    v, _ := ioutil.ReadAll(fwr)
    fmt.Printf("FirstWordReader returned %q", string(v))
}
```

Run

Important Interfaces

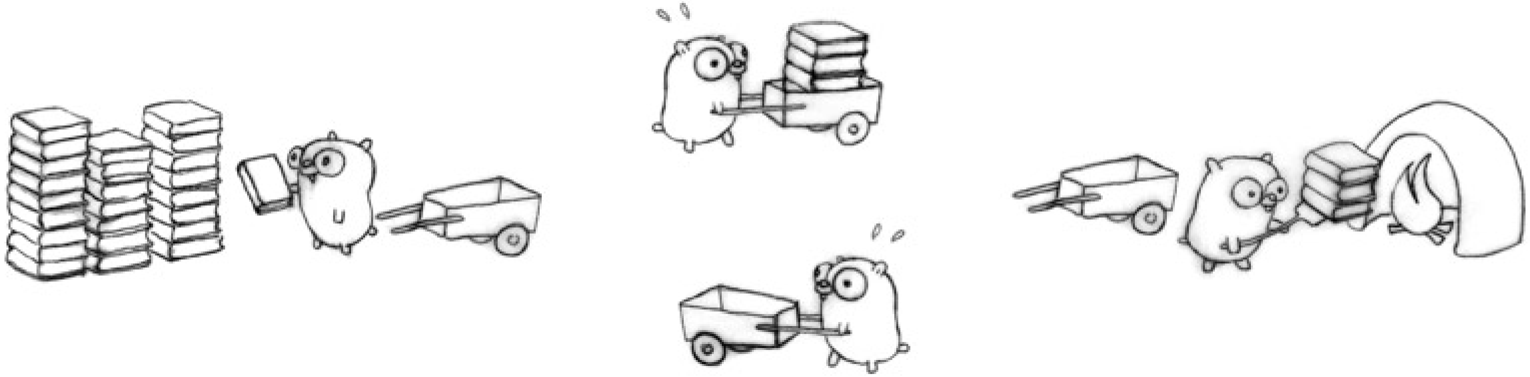
- error
- fmt.Stringer
- io.Reader, io.Writer
- http.Handler
- json.Marshaler, json.Unmarshaler
- interface{}

Concurrency

The Go approach

Concurrency derived from Tony Hoare's CSP (Communicating Sequential Processes) model.

Don't communicate by sharing memory, share memory by communicating.



Goroutines

It's an independently executing function, launched by a go statement.

```
go func() {
    ...
}()
```

- It has its own call stack, which grows and shrinks as required.
- It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.
- It's not a thread. But if you think of it as a very cheap thread, you won't be far off.
- There might be only one thread in a program with thousands of goroutines.
- Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

Goroutines example

```go
func downloadURLs(urls ...string) {
    for _, u := range urls {
        downloadUrl(u)
    }
}

func main() {
    s := time.Now()
    downloadURLs("https://avocode.com", "https://golang.org", "https://google.com")
    fmt.Printf("Took %s\n", time.Now().Sub(s))
}
```

Run

Goroutines example

```
func downloadURLs(urls ...string) {
    var wg sync.WaitGroup
    wg.Add(len(urls))
    for _, u := range urls {
        go func(u string) {
            downloadUrl(u)
            wg.Done()
        }(u)
    }
    wg.Wait()
}

func main() {
    s := time.Now()
    downloadURLs("https://avocode.com", "https://golang.org", "https://google.com")
    fmt.Printf("Took %s\n", time.Now().Sub(s))
}
```

Run

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate.

Allocationg new channel

```
c := make(chan int)
```

Sending on a channel

```
c <- 1
```

Reading from a channel

```
value = <-c
```

Closing a channel

```
close(c)
```

Select

The `select` statement lets a goroutine wait on multiple communication operations.

```
select {
    case x := <-c1:
        ...
    case y := <-c2:
        ...
    case <-c3:
        ...
}
```

Select example

```
func downloadURLs(urls ...string) {
    var wg sync.WaitGroup
    wg.Add(len(urls))
    for _, u := range urls {
        go func(url string) {
            defer wg.Done()
            lengthCh := make(chan int)
            go downloadUrl(url, lengthCh)

            timeout := time.After(300 * time.Millisecond)
            for {
                select {
                case length := <-lengthCh:
                    fmt.Printf("URL %s %d\n", url, length)
                    return
                case <-timeout:
                    fmt.Printf("Downloading %s too long\n", url)
                    return
                }
            }
        }(u)
    }
    wg.Wait()
}
```

Run

Common Patterns

- WaitGroup
- Generator
- Channel Fan-in
- Timeout
- Quit channel
- Context
- Mutex

Live coding

Summary

To be continued ...

Thank you

Daniel Hodan
Avocode
danielhodan@avocode.com (mailto:danielhodan@avocode.com)
http://avocode.com/ (http://avocode.com/)