Go Academy

#1 Introduction

www.3fs.si

3fs go academy

- An internal learning session
- Open to the community
- Will consist of 6 - 7 sessions
- Internally attendees will have ~4 hours weekly to complete the homework

History of go

- Open sourced by Google in 2009
- Envisioned / Designed as a typed, scalable, productive and readable language
- 1.0 released in March 2012 (right now at 1.9)
- Self-hosting since 1.5

How to install

- Visit https://golang.org/dl and find the suitable package / archive
  - On a mac? `brew install golang`
-
```
$ go version
go version go1.9 darwin/amd64
```
- Setup your workspace (`$GOPATH`, `$GOROOT`)

$GOPATH / $GOROOT

- `$GOPATH (~/go)` contains your code and its dependencies
- `$GOROOT` contains all built-in packages / libraries
- `./bin` binaries
- `./pkg` packages built as shared objects
- `./src` source files
- `$ export PATH=$GOPATH/bin;$PATH`

Hello World example

```
package main

import "fmt"

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

About the language

- C like
- Has a lot of builtin types (byte, int64, float32, string, rune, struct, array, map, etc.)
- Any type can have methods associated to it (`String()`)
- Provides two features to replace class inheritance
  - Embedding
  - Interfaces

About the language

- Has built-in packaging / library system with support for public and private items inside the package
- Has built-in support for concurrent programming using goroutines and channels
- It intentionally does not support generics, implicit types, assertions, etc.

Builtin tools

- go get
- go build
- go install
- go run
- go test
- go doc
- godoc
- go generate
- go vet
- go fmt

go get <import path>

- Downloads the packages named by the import paths, along with their dependencies and installs it
- Will put files inside `$GOPATH/src/<import path>`
- Code can be inside a Bazaar (.bzr), Git (.git), Mercurial (.hg) or Subversion (.svn) repository
- ```
  $ go get github.com/A/B
  $ go get github.com/A/B/...
  $ go get ./...
  ```
- Does not support vendor folder
  ```

vendor folder

- For when you care about version locking etc.
- General rule of thumb is that libraries should not use it
- Commit your dependencies along side your code
- Requires an additional tool to manage it
- `$ go get -u github.com/kardianos/govendor`
- `$ go get -u github.com/golang/dep/cmd/dep`

go build

- compiles the packages named by the import paths, along with their dependencies.
- everything with `_test.go` suffix is ignored
- by default the binary requires libc installed but can be omitted
- build is OS specific, can be changed with `$GOOS` and `$GOARCH`

go install

- Install compiles and installs the packages named by the import paths, along with their dependencies.

- places executables built from main packages into `$GOPATH/bin` and archives into `$GOPATH/pkg`

go run

- Compiles and runs the main package using the defined go source files.
- If your project has many files it might be better to run
  - `$ go build your_project && ./your_project`
  - `$ go install your_project && your_project`

go test

- recompiles each package along with any files with names matching the file pattern `*_test.go` and runs the binary
- `TestXXX` for test functions
- `BenchmarkXXX` for benchmarking functions

go doc

- Doc prints the documentation comments associated with the item identified by its arguments (a package, const, func, type, var, or method).

- Great for quick offline documentation checks

```
$ go doc json

package json // import "encoding/json"

Package json implements encoding...

func Compact(dst *bytes.Buffer, src []byte) error
func HTMLEscape(dst *bytes.Buffer, src []byte)
...
```

godoc

- locally running version of godoc.org
- should be checked for packages you plan on sharing
- get an offline play.golang.org with it

```
$ godoc -http=:8080 -play
```

go generate

- allows you to run external commands to potentially generate some content (perhaps code)
- ```
  package main

  //go:generate echo hello
  ```
- ```
  $ go generate
  ```
- will iterate all files in the current folder or specified project
- needs to be run manually

go vet

- examines Go source code and reports suspicious constructs
- can find errors not found by the compiler
- `$ go vet my/package`
- can detect issues like:

```
fmt.Printf("Test %s …")
```

go fmt

- silently ensures all go code looks the same
- it’s probably executed in your file editor anyway

Hello world

```
$ echo 'package main

import "fmt"

func main() {
  fmt.Printf("Hello World!")
}' > main.go

$ go build -o hello
./hello

$ ./hello
Hello world!
```

Hello world

```
$ go run main.go
Hello world!

$ go install
$GOPATH/bin/hello

$ hello
```

Types

- **`bool`** true, false
- **`int`** 0..2^32, 0..2^64, int32, int64
- **`byte`** 0..255
- **`float32`**, **`float64`** floating-point numbers
- **`string`** an array of characters
- **`array`** numbered sequence of elements of a single type with a fixed size

Types

- **`slice`** a portion of an array
- **`map`** unordered key-value structure
- **`struct`** object composed of other types
- **`func`** first class citizens, have typed input and output definition
- **`interface`** used for abstraction
- **`channel`** communication primitive enabling concurrency

Variables, constants and operators

- `var explicitNumber int = 10`
- `implicitNumber := 10`
- `const shallNotChange = "value"`
- Arithmetic `+ - * / % ++ --`
- Comparators `== != > < >= <= && ||`

Flow control

- **if..else**

- ```
  if x == 1 {
      fmt.Println("is one!")
  } else if x == 2 {
      fmt.Println("is two!")
  } else {
      fmt.Println("whatever")
  }
  ```

- ```
  if v := calculate(X); v > 0 {
      fmt.Println("v is high!")
  }
  ```

- Does not support ternary conditions (`COND ? true : false`)

- **switch**

- ```
  switch x {
  case 1:
       fmt.Println("is one!")
  case 2, 3:
       fmt.Println("is two or three!")
  default:
       fmt.Println("whatever")
  }
  ```

- No condition defaults to `true`
  ```

Flow control

select

- Handles communication of multiple goroutines
- ```
  select {
  case <-done:
      // close down
  case x <-chItems:
      // process it
  }
  ```

for ... range

- ```
  for i := 0; i <= 10; i++ {
      fmt.Println(i)
  }
  ```
- ```
  for i, v := range someArray {
      //
  }
  ```
- ```
  for {
      // ever and ever
  }
  ```

Flow control

- `break`, `continue`, `fallthrough`
- Label break

```
Loop:
for {
    select {
    case <-ch:
        //
        break Loop
    }
}
```

Arrays, slices

- Array has a fixed size and value set to “zero”
  - `var arr [10]string`
  - `var arr [2]int{1, 2}`
- Slice’s size can be modified
  - ```
    slice := []string{"first"}
    slice = append(slice, "second")
    ```
  - `slice[1:], slice[:1]`
    ```

Maps

- Dictionary object
- Value assigned to each key
- Anything comparable can be a key
- `m := map[int]string{1: "First"}`
- `m[2] = "Second"`
- `value := m[2]`
- `value, ok := m[2]`
- map values are intentionally randomised when iterating over them

Custom types

- Any type can be "aliased"
- `type age int`
- `type KV map[string]string`
- Can be used to assign methods to them

Structures

- Implement the “Composition over inheritance” design goal in the language
- An object with multiple fields
- Initialised in memory as a single instance
- ```
  type Person struct {
      Name string
      key int
  }
  ```
  ```

Structures

- ```
  pLong := Person{
      Name: "Me",
      key: 1,
  }
  pShort := Person{"You", 2}
  ```

- Usual initializing constructor

```
func NewPerson(name string, key int) *Person {
    return &Person{Name: name, key: key}
}
```
```

Pointers

- Represent the address where the value is stored
- Prevents unnecessary copying of data
- `&` returns the address of the value, `*` returns the value behind the address
- There is no support for pointer arithmetic
- ```
  func fetchRow(d *Database, id string) {…}
  myDatabase := &Database{…} // 10GB of data
  fetchRow(myDatabase, "Row1")
  ```
  ```

Functions

- Named functions: `func strRepeat(s string, i int) string {…}`
- Anonymous functions: `strRepeat := func(s string, i int) string {…}`
- Error handling
  - ```
    func doFail(input string) (string, error) {
        return "", errors.New("Something went wrong")
    }
    output, err := doFail(input)
    if err != nil { … }
    ```
- Variadic functions: `func sprintf(format string, args …inteface{}) {…}`
    ```

Methods

- Every custom type can have methods assigned to it

- ```
  type age int
  func (a age) canVote() bool {
      return a >= 18
  }
  ```

- Without the pointer the method is operating with a copy of the value and not the value itself
  ```

Interfaces

- Abstraction level where a type only needs to implement a specific set of methods
- ```
  interface Driver {
      Drive()
  }

  type Stig struct {}

  func (s *Stig) Drive() {
      fmt.Println("wroom")
  }

  func Race(d []*Driver) {
      //
  }
  ```
- Empty interface: `interface{}`
  ```

Goroutines

- Asynchronously executes given function
- Returns no value
- Does not indicate when the function completes
- `go heavyComputation()`

Channels

- “Share memory by communicating, don't communicate by sharing memory.”
- Connect together concurrent goroutines
- Can send and receive values
- ```
  ch := make(chan int)
  go func() {
      time.Sleep(5 * time.Second)
      ch <- 1
  }()
  myI := <-ch

  ch2 := make(chan int, 10)
  ```
  ```

Defer

- Postpones execution of a function until the end of current scope
- Will execute no matter what (return, panic)
- Handy for ensuring resources are not left locked for example

```
defer database.Disconnect()
```

Visibility

- ```
  package example

  const secret := "My internal secret"
  var ErrFailure := errors.New("Something went wrong")
  type Person struct { Name string }

  func Execute(i int) error {
       return ErrFailure
  }
  ```

- ```
  package example2

  import "example"

  if err := example.Execute(1); err == example.ErrFailure {…}
  ```

Scope

- var v = 1

```
func main() {
    v := 2
    fmt.Println(1, v) // 1 2
    {
        v := 3
        fmt.Println(2, v) // 2 3
        {
            v = 4
            fmt.Println(3, v) // 3 4
        }
    }
    fmt.Println(4, v) // 4 2
}
```

Slightly more hands on

- A command line tool for saying hello
- Name passed as a command line argument or piped in (unix approach)
  - `greet –name Janez`
  - `echo Janez | greet`

Reading flags

- Package “flags” (https://golang.org/pkg/flag/)
- `flag.TYPE("FLAG", "DEFAULT", "DESCRIPTION")`
  - `name := flag.String("name", "", "Name of the person you’d like to greet")`
- flag.Parse() parses the command line into the defined flags

Example (hello.go)

```
package main

import (
    "flag"
    "fmt"
)

func main() {
   name := flag.String("name", "", "Name of the person you’d like to
greet")
   flag.Parse()

   fmt.Printf("Hello %s!\n", *name)
}
```

Example

- `go build -o hello hello.go`

- ```
  ./hello -h
  Usage of ./hello:
    -name string
       Name of the person you’d like to greet
  ```

- ```
  ./hello -name Dominik
  Hello Dominik!
  ```

- ```
  ./hello -name=Dominik
  ./hello --name="Dominik"
  ```
  ```

Reading from Standard Input

- Reader, Writer interfaces
  - Part of the io package (do have a read at https://golang.org/pkg/io)
- `os.Stdin` is a `Reader`, that exposes the Standard Input (Read method)
- bufio.Reader convenient utility for reading a `Reader`
  - ReadLine reads a line of text

Example (print.go)

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    in, _ := readStdin()
    fmt.Printf("Hello %s!\n", in)
}

func readStdin() (string, error) {
    b, _, err := bufio.NewReader(os.Stdin).ReadLine()
    return string(b), err
}
```

Example

- `echo something | go run input.go`
  `something`

- `go run input.go`
  `… and nothing happens`

Example

- `cat /dev/stdin`
  `… nothing happens`

- `echo something | cat /dev/stdin`
  `something`

- `os.Stdin` is a `File` object that is always generated in the os package

- Every File has `FileMode` flags set one of which is `` `ModeNamedPipe` ``

- If set, Standard Input was present when our code was executed

Example (print.go)

```
func readStdin() (string, error) {
    fi, err := os.Stdin.Stat()
    if err != nil {
        return "", err
    }

    if fi.Mode()&os.ModeNamedPipe == 0 {
        return "", errors.New("StdIn not a named pipe")
    }

    b, _, err := bufio.NewReader(os.Stdin).ReadLine()
    if err != nil {
        return "", err
    }

    return string(b), nil
}
```

Converting between different types

- Bytes -> string (`[]byte(“test”)`, `string([]byte{’t’,’e’,’s’,’t’})`)
- Number -> number (`int(3.5)`, `float32(3)`)
- String -> number?
  - strconv package
  - `i, err := strconv.Atoi("42")`
- Conversion between objects
  - `k, ok := unknownTypeOrEmptyInterface.(*knownType)`

Challenge to tackle at home

- CLI calculator that can handle
  - `./calc –c “10 + 7”`
    `./calc –c “8 – 8”`
  - `echo “5+5” | calc`
- Feel free to add support for more operators