Week 1

Basic Go Syntax

Agenda

- Introduction
- Hello world
- Program structure
- Comments
- Names
- Basic types
- Variables
- Constants
- Functions
- Types
- Methods
- Interfaces
- Control structure
- Errors

Introduction

Go is a general purpose language designed at Google in 2007

- Strongly, statically typed
- Interface types
- Compiled programming language
- Garbage collection
- Support for concurrent programming (goroutine, channel)

Want to do some experiments, try Go Playground https://play.golang.org.

An example of Go program

```go
package main

import (
  "fmt"
)

func main() {
  fmt.Println("Hello world")
}

// execute: go run main.go
```

Program structure

A Go program is constructed by one or multiple source files ending with *.go***.**

A Go program usually has:

- Package clause
- Import declaration
- Declarations of
    - Variables
    - Constants
    - Types
    - Functions
    - Methods

Comments

Same as some other programming languages, we have 2 ways to write comments:

- Block comments

```
/*
  content
*/
```

- Line comments

```
// content
```

Go use comments that appear before top-level declarations, as explanatory text for the item.

Names

Names are as important in Go as any other language.

Naming convention:

- Begin with letter.
- Follow with letters, digits.
- Use *MixedCaps* or *mixedCaps* rather than underscores to write multiword names.
- Applied for variables, constants, types, functions, methods, packages.

*If the name of variables, constants, types, functions, methods **begin with uppercase** letter, it’s exported, can be used by other packages*.

Basic data type

Go's basic types

```
bool

string

int  int8  int16  int32  int64

uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32, represents a Unicode code point

float32 float64

complex64 complex128
```

Variable declaration

The ***var*** statement declares a list of variables; as in function argument lists, the type is last.

A ***var*** statement can be at package or function level. We see both in this example.

Inside a function, the ***:=*** short assignment statement can be used in place of a var declaration with implicit type.

Outside a function, every statement begins with a keyword (***var***, ***func***, and so on) and so the ***:=*** construct is not available.

Variable declaration

```
var day int
month := 7 // function scope
var year int64 = 2019
var name = "Grab"

var (
  address = "Maple Tree Building, 1060 Nguyen Van Linh Street"
  level   = 18
)
```

Constant declaration

- Constants are declared like variables, but with the const keyword.
- Constants can be character, string, boolean, or numeric values.
- Constants cannot be declared using the **:=** syntax.

```
const Pi = 3.14
const defaultAddress = "Maple Tree Building"

const (
  defaultPort int    = 8088
  defaultHost string = "0.0.0.0"
)
```

Functions

A function has:

- Name
- Arguments (zero or more)
- Return values can be zero or more than one values

Functions

```
func min(a, b int) int{
  if a <= b{
    return a
  }
  return b
}
func parse(value string) (u *url.URL, err error){
  u, err = url.Parse(value)
  return
}
```

Functions

The *init* function

- Each source file can define its own *init* function
- It’s called
    - After all variables in the packages are evaluated
    - After all imported packages have been initialized

Functions

```go
const envPort = "CONFIG_PORT"
const defaultPort = "8088"
var listenAddress string
func init() {
  port := os.Getenv(envPort)
  if port != "" {
    listenAddress = fmt.Sprintf(":%s", port)
  } else {
    listenAddress = fmt.Sprintf(":%s", defaultPort)
  }
}
```

Types

A **type** declarations bind the type name to a type

There are 2 forms

- Alias declaration

  Bind an alias name to given type

- Type declaration

  Create a new type which is compose from existing type

Types

```
type Mode = string // alias

type String string

type Server struct {
  Mode Mode
  Host string
  Port uint
}
```

Methods

A method is a function with a receiver

```go
type Server struct {
  Mode Mode
  Host string
  Port uint
}

func (s Server) Address() string {
  return fmt.Sprintf("%s:%d", s.Host, s.Port)
}
```

Methods

A method receiver can be

- Value receiver

  No side-effect

- Pointer receiver

  When want to update receiver or type too large

`Nil` is a valid receiver

Methods

```
type Config struct {
  Host string
  Port uint
}

func (c *Config) Unmarshal(data []byte) error {
  if data == nil {
    return errors.New("invalid data")
  }
  if c == nil {
    c = &Config{}
  }
  return json.Unmarshal(data, c)
}
```

Interfaces

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

A value of interface type can hold any value that implements those methods

Interfaces

```go
type Abser interface {
  Abs() int
}

type Int int

func (v Int) Abs() int {
  value := int(v)
  if value < 0 {
    return -value
  }
  return value
}

var num Abser = Int(10)
```

```go
type Uint uint

func (v Uint) Abs() int {
  value := int(v)
  return value
}

var num Abser = Uint(10)
```

Interfaces

An empty interface is an interface with no methods.

An empty interface may hold value of any type.

Interfaces

```go
func do() {
  var value interface{}
  value = 10
  ok := check(value)
  if ok {
    fmt.Println("got valid value", value)
  } else {
    fmt.Println("got invalid value", value)
  }
}
```

Control structures

If

For

Switch

Control structures - If

```
if x < 0{
   fmt.Println("got x < 0")
} else {
   fmt.Println("got x >= 0")
}

if f, err := os.Open("dummy"); err != nil{
   return false
}
```

Control structure - For loop

Syntax:

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

Control structure - For loop

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

Control structure - For loop

```go
sum, i  := 0, 0
for {
  i++
  if i >= 10 {
    break
  }
  sum += i
}
```

Control structure - For loop

```go
// slice
items := []int{1, 2, 3}
for index, item := range items {
  fmt.Printf("item[%d] = %d\n", index, item)
}
// map
values := map[string]int{
  "a": 1,
  "b": 2,
}
for key, value := range values{
  fmt.Printf("values[%s] = %d\n", key, value)
}
```

Control structure - Switch case

```
switch num{
 case 1:
   return “A”
 case 2, 3, 4:
   return “B”
 case 5:
   return “C”
}
```

Errors

Errors are values (https://blog.golang.org/errors-are-values)

In Go, we handle failure by returning error values - built-in interface type **error**

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

Errors

```
type Config struct {
  Host string
  Port int
}

func Load(cfg *Config, r io.Reader) error {
  data, err := ioutil.Read(r)
  if err != nil {
    return err
  }
  return json.Unmarshal(data, cfg)
}
```

Errors

Create an error

- errors.New
- fmt.Errorf
- Create new type implement error interface

Errors

```go
type FileNotFound struct {
  Path string
}
func (f FileNotFound) Error() string {
  return fmt.Sprintf("file: not found %s", f.Path)
}
var err error = FileNotFound{Path: "dummy"}
```

Assignment (Thursday)

Write a simple REPL program (Read–Eval–Print–Loop).

Take a formula from a user then print out the result. The formula must be in this format:

```
<first number> <arithmetic: + - * / > <second number>
```

Example:

```
> 1 + 2
1 + 2 = 3
> 2 * 10
2 * 10 = 20
```

defer

panic

recover

blank identifier

type assertion

Thanks