Week 1, Golang

Azamat Serek, PhD, Assist.Prof.

Go

Addresses issue of slow compilation time and poor dependency management.

Install

https://go.dev/doc/install

Compile the code into executable

Go build main.go

Running

Go run main.go

Create module

Go mod init folder_name

Packages

import "fmt"

Hello Go

```
package main

import "fmt"

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

Declarations

```
var foo int // declaration without initialization
var foo int = 42 // declaration with initialization
var foo, bar int = 42, 1302 // declare and init multiple vars at once
var foo = 42 // type omitted, will be inferred
foo := 42 // shorthand, only in func bodies, omit var keyword, type is always implicit
const constant = "This is a constant"
```

Functions

Functions

```
// a simple function
func functionName() {}

// function with parameters (again, types go after identifiers)
func functionName(param1 string, param2 int) {}

// multiple parameters of the same type
func functionName(param1, param2 int) {}

// return type declaration
func functionName() int {
    return 42
}

// Can return multiple values at once
func returnMulti() (int, string) {
    return 42, "foobar"
}
var x, str = returnMulti()

// Return multiple named results simply by return
func returnMulti2() (n int, s string) {
    n = 42
    s = "foobar"
    // n and s will be returned
    return
}
var x, str = returnMulti2()
```

```go
func main() {
    // assign a function to a name
    add := func(a, b int) int {
        return a + b
    }
    // use the name to call the function
    fmt.Println(add(3, 4))
}

// Closures, lexically scoped: Functions can access values that were
// in scope when defining the function
func scope() func() int{
    outer_var := 2
    foo := func() int { return outer_var}
    return foo
}

func another_scope() func() int{
    // won't compile because outer_var and foo not defined in this scope
    outer_var = 444
    return foo
```

Functions

```
// Closures
func outer() (func() int, int) {
    outer_var := 2
    inner := func() int {
        outer_var += 99 // outer_var from outer scope is mutated.
        return outer_var
    }
    inner()
    return inner, outer_var // return inner func and mutated outer_var 101
}
```

Type Conversions

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)

// alternative syntax
i := 42
f := float64(i)
u := uint(f)
```

Packages

- Package declaration at top of every source file
- Executables are in package `main`
- Convention: package name == last name of import path (import path `math/rand` => package `rand`)
- Upper case identifier: exported (visible from other packages)
- Lower case identifier: private (not visible from other packages)

If

```
func main() {
        // Basic one
        if x > 10 {
                return x
        } else if x == 10 {
                return 10
        } else {
                return -x
        }

        // You can put one statement before the condition
        if a := b + c; a < 42 {
                return a
        } else {
                return a - 42
        }

        // Type assertion inside if
        var val interface{} = "foo"
        if str, ok := val.(string); ok {
                fmt.Println(str)
        }
}
```

Loops

```
    // There's only `for`, no `while`, no `until`
    for i := 1; i < 10; i++ {
    }
    for ; i < 10;  { // while - loop
    }
    for i < 10  { // you can omit semicolons if there is only a condition
    }
    for { // you can omit the condition ~ while (true)
    }

    // use break/continue on current loop
    // use break/continue with label on outer loop
here:
    for i := 0; i < 2; i++ {
        for j := i + 1; j < 3; j++ {
            if i == 0 {
                continue here
            }
            fmt.Println(j)
            if j == 2 {
                break
            }
        }
    }
```

Switch

```go
// switch statement
switch operatingSystem {
case "darwin":
    fmt.Println("Mac OS Hipster")
    // cases break automatically, no fallthrough by default
case "linux":
    fmt.Println("Linux Geek")
default:
    // Windows, BSD, ...
    fmt.Println("Other")
}

// as with for and if, you can have an assignment statement before the switch value
switch os := runtime.GOOS; os {
case "darwin": ...
}

// you can also make comparisons in switch cases
number := 42
switch {
    case number < 42:
        fmt.Println("Smaller")
    case number == 42:
        fmt.Println("Equal")
    case number > 42:
        fmt.Println("Greater")
}
```

Arrays

```
var a [10]int // declare an int array with length 10. Array length is part of the type!
a[3] = 42     // set elements
i := a[3]     // read elements

// declare and initialize
var a = [2]int{1, 2}
a := [2]int{1, 2} //shorthand
a := [...]int{1, 2} // elipsis -> Compiler figures out array length
```

Slices

```
var a []int                             // declare a slice - similar to an array, but length is
var a = []int {1, 2, 3, 4}              // declare and initialize a slice (backed by the array give
a := []int{1, 2, 3, 4}                  // shorthand
chars := []string{0:"a", 2:"c", 1: "b"} // ["a", "b", "c"]

var b = a[lo:hi]        // creates a slice (view of the array) from index lo to hi-1
var b = a[1:4]          // slice from index 1 to 3
var b = a[:3]           // missing low index implies 0
var b = a[3:]           // missing high index implies len(a)
a =  append(a,17,3)     // append items to slice a
c := append(a,b...)     // concatenate slices a and b

// create a slice with make
a = make([]byte, 5, 5)  // first arg length, second capacity
a = make([]byte, 5)     // capacity is optional

// create a slice from an array
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // a slice referencing the storage of x
```

Operations on Arrays and Slices

`len(a)` gives you the length of an array/a slice. It's a built-in function, not a attribute/method on the array.

```
// loop over an array/a slice
for i, e := range a {
    // i is the index, e the element
}

// if you only need e:
for _, e := range a {
    // e is the element
}

// ...and if you only need the index
for i := range a {
}

// In Go pre-1.4, you'll get a compiler error if you're not using i and e.
// Go 1.4 introduced a variable-free form, so that you can do this
for range time.Tick(time.Second) {
    // do it once a sec
}
```

Maps

```
m := make(map[string]int)
m["key"] = 42
fmt.Println(m["key"])

delete(m, "key")

elem, ok := m["key"] // test if key "key" is present and retrieve it, if so

// map literal
var m = map[string]Vertex{
    "Bell Labs": {40.68433, -74.39967},
    "Google":    {37.42202, -122.08408},
}

// iterate over map content
for key, value := range m {
}
```

Exercise (will not given grade for it)

Practice on the above mentioned syntactic constructs and show results.

References

https://github.com/a8m/golang-cheat-sheet