COS418 Precept 1

Feb 5th 2021

Go Resources

Go tutorial: https://tour.golang.org/list

Go Playground: https://play.golang.org

Basic syntax code in playground: https://tinyurl.com/y7rdgqj3

Agenda for Today

- Go Basics
  - Program Structure
  - Variables
  - Functions
  - Loops
  - Composite Data Types
  - OOP in Go
- Exercise Time

Program Structure

A basic Go program contains

- ❏ Package specification: serves as a separate namespace, like modules or libraries in other languages
- ❏ Import other packages
- ❏ Package-level declarations: var, func, const, type

```
// All files start with a package declaration
package main

// Import statements, one package on each line
import (
    "errors"
    "fmt"
)

// Main method will be called when the Go executable is run
func main() {
    fmt.Println("Hello world!")
    basic()
    add(1, 2)
    divide(3, 4)
    loops()
    slices()
    maps()
    sharks()
}
```

Variables

- Variable Declaration
  - The General form
    var name type = expression

  "*= expression*" may be omitted. The variable will take zero value for the type, e.g 0 for numbers, false for boolean, "" for strings, and nil for the rest

  - Short Variable Declaration
    name := expression

  Only for local variables within a function

Note: Keep in mind := is a declaration, whereas = is an assignment

```
// Declare a package-level variable
var msg string = "Hello World"

// Function declaration
func basic() {
    // Declare x as a variable, initialized to 0
    var x int
    // Declare y as a variable, initialized to 2
    var y int = 2
    // Declare z as a variable, initialized to 4
    // This syntax can only be used in a function
    z := 4

    // Assign values to variables
    x = 1
    y = 2
    z = x + 2 * y + 3

    // Print the variables; just use %v for most types
    fmt.Printf("x = %v, y = %v, z = %v\n", x, y, z)
    // Print the package-level string variable
    fmt.Println(msg)
}
```

Functions

- Function Declaration
  - The General Form:

func name (parameter-list) (result-list)
{
body
}

Named functions are declared only at the package level.

```
// Function declaration; takes in 2 ints and outputs an int
func add(x, y int) int {
    return x + y
}

// Function that returns two things; error is nil if successful
func divide(x, y int) (float64, error) {
    if y == 0 {
        return 0.0, errors.New("Divide by zero")
    }
    // Cast x and y to float64 before dividing
    return float64(x) / float64(y), nil
}
```

Functions

- Anonymous Functions
  - Define such a function at its point of use
  - Declare without a name following the *func* keyword

```
// squares() returns an anonymous function
// that itself returns the next square number each
time it is called
func squares() func() int {
    var x int
    return func() int {
      x++
      return x*x
    }
}


func main() {
   // Assign a function variable to func squares()
    f := squares()
    fmt.Println(f()) // “1”
    fmt.Println(f()) // “4”
    fmt.Println(f()) // “9”
}
```

Loops

- In Go, while loops are represented via for loops

```
func loops() {
    // For loop
    for i := 0; i < 10; i++ {
        fmt.Print(".")
    }
    // While loop
    sum := 1
    for sum < 1000 {
        sum *= 2
    }
    fmt.Printf("The sum is %v\n", sum)
}
```

Composite Data Types

- Composite types are based on basic data types (e.g integers, floating point numbers, strings, and booleans). In Go, some common composite types are:
  - Array: fixed-length, elements of same type
  - Slice: variable-length, elements of same type
  - Map: hash table of key value pairs
  - Struct: contain arbitrary fields and types

```
func slices() {
	slice := []int{1, 2, 3, 4, 5, 6, 7, 8}
	fmt.Println(slice)
	fmt.Println(slice[2:5]) // 3, 4, 5
	fmt.Println(slice[5:]) // 6, 7, 8
	fmt.Println(slice[:3]) // 1, 2, 3

	slice2 := make([]string, 3)
	slice2[0] = "tic"
	slice2[1] = "tac"
	slice2[2] = "toe"
	fmt.Println(slice2)
	slice2 = append(slice2, "tom")
	slice2 = append(slice2, "radar")
	fmt.Println(slice2)
	for index, value := range slice2 {
		fmt.Printf("%v: %v\n", index, value)
	}
	fmt.Printf("Slice length = %v\n",
len(slice2))
}
```

Composite Data Types: Map

```
func maps() {
    // Declare a map whose keys have type string, and values have type int
    myMap := make(map[string]int)
    myMap["yellow"] = 1
    myMap["magic"] = 2
    myMap["amsterdam"] = 3
    fmt.Println(myMap)
    myMap["magic"] = 100
    delete(myMap, "amsterdam")
    fmt.Println(myMap)
    fmt.Printf("Map size = %v\n", len(myMap))
}
```

Object-Oriented Programming (OOP) in Go

- Go also provides programmers with an OOP paradigm. We can view:
  - Object: a value or variable that has methods
  - Method: a function associated with a particular type

- Methods in Go
  - Method Declaration
    Similar to function declaration, but add an extra parameter between *func* and *name*. This will attach the function to the type of the parameter.
  - Example

```
import “math”
// Declare a struct named Point with x, y positions
type Point struct { X, Y float64}

// Implement a method that find Hypotenuse distance
between one Point and another
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X - p.X, q.Y - p.Y)
}

// standard function
func Distance(p Point, q Point) float64 {
    return math.Hypot(q.X - p.X, q.Y - p.Y)
}

func main() {
   p := Point{1, 2}
   q := Point{4, 6}
   fmt.Println(p.Distance(q)) // “5” method call
   fmt.Println(Distance(p, q)) // “5” function call
}
```

Exercise Time

Sharks and Their Methods

```
// Object oriented programming
// Convention: capitalize first letter of public fields
type Shark struct {
    Name string
    Age int
}

// Declare a public method
// This is called a receiver method
func (s *Shark) Bite() {
    fmt.Printf("%v says CHOMP!\n", s.Name)
}

// Because functions in Go are pass by value
// (as opposed to pass by reference), receiver
// methods generally take in pointers to the
// object instead of the object itself.
func (s *Shark) ChangeName(newName string) {
    s.Name = newName
}
```

Output:
Bruce says CHOMP!
Lee says your majesty
Sharkira says yo what's up Lee

```
// Receiver methods can take in other objects as well
func (s *Shark) Greet(s2 *Shark) {
    if (s.Age < s2.Age) {
        fmt.Printf("%v says your majesty\n",
s.Name)
    } else {
        fmt.Printf("%v says yo what's up %v\n",
            s.Name, s2.Name)
    }
}

func sharks() {
    shark1 := Shark{"Bruce", 32}
    shark2 := Shark{"Sharkira", 40}
    shark1.Bite()
    shark1.ChangeName("Lee")
    shark1.Greet(&shark2) // pass in pointer
    shark2.Greet(&shark1)
}
```

Go Routines

```
// Launch n goroutines, each printing a number
// Note how the numbers are not printed in order
func goroutines() {
    for i := 0; i < 10; i++ {
        // Print the number asynchronously
        go fmt.Printf("Printing %v in a goroutine\n", i)
    }
    // At this point the numbers may not have been printed yet
    fmt.Println("Launched the goroutines")
}
```

Possible Output:

```
Printing 4 in a goroutine
Printing 8 in a goroutine
Printing 9 in a goroutine
Printing 0 in a goroutine
Printing 1 in a goroutine
Printing 6 in a goroutine
Printing 2 in a goroutine
Printing 3 in a goroutine
Launched the goroutines
```

(Unbuffered) Channels

```
// Channels are a way to pass messages across goroutines
func channels() {
    ch := make(chan int)
    // Launch a goroutine using an anonymous function
    go func() {
        i := 1
        for {
            // This line blocks until someone
            // consumes from the channel
            ch <- i * i
            i++
        }
    }()
    // Extract first 10 squared numbers from the channel
    for i := 0; i < 10; i++ {
        // This line blocks until someone sends into the channel
        fmt.Printf("The next squared number is %v\n", <-ch)
    }
}
```

Output:

```
The next squared number is 1
The next squared number is 4
The next squared number is 9
The next squared number is 16
The next squared number is 25
The next squared number is 36
The next squared number is 49
The next squared number is 64
The next squared number is 81
The next squared number is 100
```

Buffered Channels

```
// Buffered channels are like channels except:
//   1. Sending only blocks when the channel is full
//   2. Receiving only blocks when the channel is empty
func bufferedChannels() {
    ch := make(chan int, 3)
    ch <- 1
    ch <- 2
    ch <- 3
    // Buffer is now full; sending any new messages will block
    // Instead let's just consume from the channel
    for i := 0; i < 3; i++ {
        fmt.Printf("Consuming %v from channel\n", <-ch)
    }
    // Buffer is now empty; consuming from channel will block
}
```

Output:

```
Consuming 1 from channel
Consuming 2 from channel
Consuming 3 from channel
```