Basic Go Syntax

Learning Objectives

In this lesson, we'll cover the basics of Go, its features, program structure, data types, syntax, and more.
To put these concepts into practice, we'll end session two with a simple coding exercise.

Prerequisites

- Basic understanding of computer programming terminologies

Session 1

What is Go?

*“**Go**, also known as **Golang**, is a statically typed, compiled programming language designed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing, and communicating sequential processes style concurrency.”* - Wikipedia

Installation Guide

See instructions in the Golang website

Features of Go

- Imperative language
- Statically typed
- Syntax tokens similar to C (but less parentheses and no semicolons)
- Compiles to native code (no JVM)
- No classes, but uses structs with methods
- Interfaces
- No implementation inheritance. There's type embedding.
- Functions are first class citizens
- Functions can return multiple values
- Has closures
- Has pointers, but not pointer arithmetic
- Built-in concurrency primitives: Goroutines and Channels

Let’s go

Hello world

Source code

```
// file main.go
package main

import (
  "fmt"
)

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

Execute it

```
$ gofmt -s -w main.go    // optional, rewrites code into the standard format
$ goimports -w main.go // optional, manages the insertion and removal of
import declarations as needed, install via command go get
golang.org/x/tools/cmd/goimports. It also format the code
$ go run main.go           // execute it
```

Program Structure

A Go program is constructed by one or more source files, usually ending in `.go`. Each source file usually consists of the following parts:

- Import Packages
- Declaration for the following:
    - Packages
    - Constants
    - Variables
    - Types
    - Functions
    - Methods

Packages and Files

Declarations

A Go source file begins with a package declaration, which defines the package it belongs to.

Go packages share the same purpose as libraries or modules in other languages.

A package in Go usually consists of one or multiple source files.

package name

```
// file demo.go
package demo

const (
  Description = "Public constant"  // public
  description = "private constant" // private, visible to only its
own package
)

var (
  Content string = "Public"
  content string = "private"
)

func Sum() int { // exported
  return 0
}

func sum() int { // unexported
  return 0
}
```

Names

The names follow a simple rule: begins with a letter or an underscore and may have any number of additional letters, digits, and underscores. Case matters: *heapSort* and *Heapsort* are different names.

This naming convention applies for: `functions, variables, constants, types, statement labels`, and `packages`.

If the name of `var, const, type, func` begins with an upper-case letter, it is exported, which means that it is visible and accessible outside of its own package and may be referred to by other parts of the program.

There are 25 *reserved keywords:*

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Comments

When adding comments, use the following syntax:

Line comment	// this is a line comment
Block comments	/* This is for block comments; it can have many lines */

Variables

A *var* declaration creates a variable of a particular type, attaches a name to it, and sets its initial value.

```
var name type = expression
```

When declaring variables, the `type` or `expression` must be indicated. Both cannot be omitted. If the `type` is omitted, the `type` is determined by the initializer expression. If the `expression` is omitted, the initial value is the *zero value* for the type.

```
var num int64 = 10

var zero int64

var ten = 10
```

Constants

A `const` declaration defines named values that syntactically look like variables, but whose value is constant. This prevents accidental (or nefarious) changes during program execution.

The underlying type of every constant is a basic type: `boolean, string, or number.`

`const` *`name type = expression`*

```
const(
  Pi = 3.14
  Company = “Grab”
)

type Name string
const(
  Grab Grab = “grab”
)
```

Type Declarations

The variable or expression type defines the characteristics of the values it takes on, such as the size (number of bits or number of elements), how they are represented internally, the intrinsic operations that can be performed on them, and the methods associated with them.

```
type Age uint
type Student struct{
  ID   int
  Name string
}

// group declaration
type (
  Name string
  User struct{
    ID   int
    Name string
  }
)
```

Scope

A declared identifier's scope refers to the identifiable range of the identifier.

The scope definitions:

- The scope of a predeclared/built-in identifier is the universe block.
- The scope of an import package's identifier is the file block containing the package import declaration.
- The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at package level is the package block.
- The scope of an identifier denoting a method receiver, function parameter, or result variable is the corresponding function body (a local block).
- The scope of a constant or variable's identifier declared inside a function begins at the end of the constant or variable's specification (or the end of the declaration for a short declared variable) and ends at the innermost containing block.
- The scope of a defined type's identifier declared inside a function begins at the identifier in the type's specification that ends at innermost containing block.
- The scope of an alias' identifier declared inside a local block begins at the end of the type declaration and ends at the innermost containing block.
- The scope of a label is the innermost function body block containing the label declaration, but excludes all the anonymous functions' bodies nested in the containing function.
- Blank identifiers have no scope.

Basic Data Types

Here's a list of the basic data types:

type	sample
int int8 int16 int32 int64	-1 0 1 2
uint uint8 uint16 uint32 uint64	0 1 2
float32 float64	0 1.5
char (alias of int8)	'x'
rune (alias of int32)	
string	"Go" "走"
Complex64 complex128	

Integers

See in Golang website

Floating point number

See in Golang website

Boolean

See in Golang website

Complex numbers

See in [Golang website]

Strings

A string is an immutable sequence of bytes.

Declarations

Strings are defined between two double quotes or two backquotes.

```
s := “Grab”
hello := `Welcome`
```

Lengths

The built-in len function returns the number of bytes in a string instead of the number of characters in a string.

```
s := “Grab”
fmt.Println(len(s)) // 4

S := “Chào”
fmt.Println(len(s)) // 5
```

Count characters

The strings in Go are UTF-8 encoded by default. Go represents a character (code point) in `int32` data type. A `rune` is Go terminology for a single Unicode code point.

```
func count(value string) int{
  var total int
  for range value{
    total++
  }
  return total
}

S := “Chào”
fmt.Println(count(s)) // 4
```

Library

Here are some important standard library packages for working with strings:

- strings implements string searching, splitting, case conversions
- bytes has the same functionality as `strings` package but operates on `[]byte` slices
- strconv for converting strings to integers and float numbers
- unicode/utf8 decodes from UTF-8-encoded strings and encodes to UTF-8-encoded strings
- regexp implements regular expressions
- text/scanner is for scanning and tokenizing UTF-8-encoded text
- text/template is for generating larger strings from templates
- html/template has all the functionality of `text/template` but understands the structure of HTML for HTML generation that is safe from code injection attacks

Composite Types

Arrays

An array is a fixed length sequence of a particular type. Arrays are rarely used in Go.

```
var a [3]int

var b [...]int{1, 2, 3}



fmt.Println(a[0], a[1], a[2]) // print 0 0 0

fmt.Println(len(a), len(b)   // print 3 3
```

Slices

A slice is a variable length sequence of a particular type. A slice has three components:

- A pointer points to the first element of the array that is reachable through the slice
- The length is the number of slice elements, it can't exceed the capacity (`len`)
- The capacity is the number of elements between the start and the end of the underlying array (`cap`)

```
var s []int              // len = 0, cap  = 0

a :=  make([]int, 3, 10) // len = 3, cap = 10
```

Multiple slices can share the same underlying array and my refer to overlapping parts of that array.

```
var arr = [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

var x = arr[0:2] // len = 2, cap = 10
var y = arr[2:5] // len = 3, cap = 8
```

Append

The build-in `append` function appends items to slices.

```
var a []int
var b = []int{1, 2, 3}
a = append(a, 0) // a = [0]
a = append(a, b … ) // a = [0, 1, 2, 3]
```

Note: Since a slice contains a pointer to an array element, passing a slice to a function permits the function to modify the underlying array elements.

Maps

A map is an unordered collection of key/value pairs in which all key are distinct. All of the keys in a given map are of the same type.

- The key type must be comparable using ==
- There are no restrictions on the value type

```
var m map[string]int
var a = map[string]interface{}{
   “name”: “foo”,
   “age” : 10,
}
x := map[string]struct{} {
  “foo”: struct{}{},
  “bar”: struct{}{},
}

for key, value := range a{
  fmt.Println(key, value )
}

/* will print
name foo
age 10
*/
```

Structs

A struct is an aggregate data that groups together zero or more fields. Each field will have a name and a data type for that name.

```
type Circle struct{
```

```
  X      int
  Y      int
  Radius float64
}

var x Circle
x.X = 1
x.Y = 2
x.Radius = 10

var y = Circle { X: 1, Y: 1, Radius: 0.5 }
```

Anonymous structs

An anonymous struct is a struct without a name.

```
x := struct {
  Name string
} {
  Name: "Grab"
}
```

Composition and embedding

Go allows field definitions for a nameless type, such a field is called anonymous field. The field type must be a named type or pointer to a named type. In effect, the outer struct type gains not just the embedded type fields, but its methods too.

```
type Animal struct {
  Name string
}

type Cat struct {
```

```
  Animal
}

type Dog struct {
  *Animal
}

var x Cat
var y = Cat{
  Animal: Animal{
    Name: "Foo",
  },
}
var d = Dog{
  Animal: &Animal{
    Name: "Bar",
  },
}
```

Functions

A function is a sequence of statements known as a unit, which can be called from anywhere else in the program.

The function declaration has the following elements:

- A name
- A list of parameters, which specifies the names and types of the function's parameters
- The return list specifies the names (optional) and types of the function's returns. If the name is specified, then it is declared as a local variable for the function.

Declaration

```
func name(parameter list) (return list) {

  body

}
```

```
func sum(a, b int) int{
  return a + b
}

func dump(value int) {
  fmt.Println(value)
}

func get(url string) (io.Reader, error){
  return nil, nil
}

func lowercase (value string ) (s string) {
  s = strings.ToLower(value)
  return
```

```
}
```

Return values

A function can return more than one results.

You can name returned values that act as implicit local variables.

```
func parse(value string) (u *url.URL, err error){
  u, err = url.Parse(value)
  return
}

func read(filename content) ([]byte, error){
  f, err := os.Open(filename)
  if err != nil {
    Return nil, err
  }
  data, err := ioutil.ReadAll(f)
  if err != nil{
    Return nil, err
  }

  return data, nil
}
```

First class

Functions are the `first-class` values in Go. It has types and can be assigned to variables or passed to or returned from functions. A function value can be called like any other functions.

```
func sum(a, b int) int { return a + b }

var decrease = func(value int) int { return value - 1 }

func main() {
  s := sum
  total := s(1, 2)
  fmt.Println(total)) // 3
  fmt.Println(decrease(10)) // 9
}
```

Variadic functions

A variadic function can be called with varying numbers of arguments.

```
func sum(values ...int) int{

  var total int

  for _, value := range values{

     total += value

  }

  return total

}



// usage

total := sum(1, 2, 3) // 6

total := sum()        // 0
```

Defer statements

A defer statement is an ordinary function or method call prefixed by the keyword `prefer.` The function and argument expressions are evaluated when the statement is executed, but the actual call is deferred until the function that contains the defer statement has finished.

If a function has multiple defer statements, they are executed in reverse order.

```
func increase(value int) int{
  defer func(v int) {
    fmt.Println(“receive:”, v)
  }(value)
  return value++
}
const DEBUG = true
func decrease (value int) int{
  If DEBUG {
      defer func(v int) {
        fmt.Println(“receive:”, v)
      }(value)
  }
  return value--
}
func hello() {
```

```
  defer fmt.Print(1)

  defer fmt.Print(2)

  defer fmt.Print(3)

}

// output: 321
```

Methods

A method is a function tied to a type, most commonly a struct.

Declarations

A method is declared with an ordinary function declaration variant in which an extra parameter appears before the function name. The parameter attaches the function to that parameter type.

```
type Point struct{
   X int
   Y int
}

func (p Point) Distance(value Point) float64{
  return math.Hypot(p.X-value.X, p.Y - value.Y)
}

func (p* Point) Reset() {
  p.X = 0
  p.Y = 0
}

// usage

a := Point( X: 1, Y: 1}
b := Point{ X: 4, Y: 5}
fmt.Println( a.Distance(b) ) // 5
a.Reset()
fmt.Println(a.X, a.Y) // 0, 0
```

Value vs pointer receiver

Method receiver can be either a value and a pointer. When calling a function, Go will make a copy of each argument value. Ensure to use:

- Pointer receiver to update the receiver variable or type when it's too large. Note that nill is a valid receiver value.
- Value receiver when there are no side effects.

```
type Ints struct{
  values []int
}

func (i *Ints) Sum() int{
  if i == nil {
    return 0
  }
  var total int
  for _, value := range i.values{
    total += value
  }
  return total
}
```

Interfaces

An interface describes a set of methods on a type. It is used to abstract behavior.

Interfaces can only contain methods, not data.

```
type Printer interface{
  fmt.Print(w io.Writer)
}

type Int int

func (i Int) Print(w io.Writer){
  fmt.Fprintf(w, “%d”, i)
}
```

Type switch

To determine underlying type from interface, we can use a type switch.

```
type Point struct {
  X int
  Y int
}

type Square struct {
  TopLeft           Point
  BottomRight Point
}

func (s Square) Paint() {
  // todo
}

type Painter interface{
  Paint()
}

func typecheck(painter Painter) {
  switch value := painter.(type) {
   case Square:
     fmt.Println(“Square”, value)
   default :
    fmt.Println(“Unknown”)
  }
}
```

Empty interface

The empty interface is an interface with no method. Since every type conforms to `interface{}`, you can assign any value to a variable of `interface{}` type.

```
var x interface{}
x = 1
x = struct{}{}
x = Point{}
```

Type assertion

Type assertion allows you to check an empty interface value's given type.

```
func check(value interface{}){
  num, ok := value.(int)
  If ok {
    fmt.Println("Given value is a number", num)
  } else{
    fmt.Println("Given value is not a number")
  }
}
```

Control structures

If

```
If num > 0 {
  return num
}

if f, err := os.Open(“grab.doc”); err != nil {
  return nil, err
}

_, err := os.Create(“grab.doc”)
if err != nil {
  // todo
} else if err = os.EOF {
  // todo
} else {
  // todo
}
```

For

The for syntax consists of three different forms, only one of which has semicolons:

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

// Like a C while
for condition { }

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

For example:

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

```
nums := []int{}
for index, num := range num{
  // todo
}

m := map[string]int{}
for key, value := range m{
  // todo
}
```

Switch

Switch statements provides multi-way execution

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

switch {
case a < b:
  return -1
case a > b:
  return 1
default:
  return 0
}
```

Go only runs the matched case. If you want to continue with the rest of the code execution, use `fallthrough`.

```
n := 1
switch n {
  case 1:
    fmt.Println(“case 1”)
  case 2:
```

```
   fmt.Println(“case 2”)
}
// output
case 1

switch n {
  case 1:
    fmt.Println(“case 1”)
    fallthrough
  case 2:
   fmt.Println(“case 2”)
}
// output
case 1
case 2
```

Goto

A goto statement transfers control to the statement with the corresponding label within the same function.

```
func printAB(num int) {
  if num%2 == 1 {
    goto odd
  }
  fmt.Println("A")
  return
odd:
  fmt.Println("B")
}
```

Error handling

In Go, errors are values, just like strings and integers. Go handles failures by returning error values. If a function or method returns an error, it should always be the last returned value.

```
func divide(a, b int) (int, error){
  If b == 0{
    return 0, errors.New(“invalid value”)
  }
  return a / b, nil
}
```

Error interface

In Go, the error type is a built-in interface with a single method `Error() string`.

Creation

You can create an error with the following approaches:

- `errors.New`
- `fmt.Errorf`
- Custom error type

```
var err error = errors.New(“not found”)

var err error = fmt.Errorf(“not found %d”, 1)


type MyError struct{
  Message string
}

func (m MyError) Error() string {
  return m.Message
}

var err error = MyError{Message: “Hello my first custom error”}
```

Panic and recover

Panic and recover are similar to exception handling in other languages like C#, Java, etc:

- `panic` is the equivalent of the `throw error`
- `recover` is the equivalent of the `catch error`

A `panic` halts normal execution flow and exits the current function, while a `recover` attempts to recover from a panic. The recover statement must appear directly within the deferred function enclosure.

```
func get(url string) {
  defer func() {
    If err := recover(); err != nil {
      fmt.Println(“recover with err”, err)
    }
  }()
  u, err := url.Parse()
  if err != nil {
    panic(err)
 }
 // todo
}
```

Exercises

- Effective Go
- Wiki
- Go Concurrency Patterns: Context
- Go Concurrency Patterns: Pipelines and cancellation
- Profiling Go Programs

Session 2

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
> 100 / 10
100 / 10 = 10
```