Go Basics

I also recommend taking Go's tour at https://tour.golang.org/moretypes/

Basic Information

You may either use Go on Turing (I recommend) or download Go from github (you will need to also install C++ and npm if you are using your own laptop). This handout assumes that you are using Go on turing.

Turing Account

To begin, at your home directory, create a ***Go*** directory – make sure the file permission is 701 (from CLI, ***chmod 701 Go***). Within this directory create 3 subdirectories: **bin**, **src**, and **pkg**

In addition, you will need to edit the .bash_profile file in your home directory to include the following environment variables (I used nano to do this):
export GOBIN="$HOME/Go/bin"
export GOPATH="$HOME/Go/src"

NOTE: Everyone with a turing account ALREADY has a .bash_profile file. The dot in front means it is a hidden file. WinSCP does not show hidden files, Filezilla does. In either case, in Command Line, use an editor, as described above, to open .bash_profile and add the code to install the Go compiler.

Finally, to make the changes take effect, still in your home directory, execute the files as follows:
source /etc/profile
source ~/.bash_profile

Close the shell and re-open
Now, grab the mysql driver (we will use this at a later time). From your home directory on command line go get the driver:
go get -u github.com/go-sql-driver/mysql

Verify that you have the following folders Go > src > src > github.com → you should be good to Go (convenient pun)

Important Go Requirements

- Go files must end with .go and must be in the ***src*** folder.
- To both compile and execute: **go run *filename*.go**
  To only complie: **go build *filename*.go** To execute you will still need to run with ***go run*** …
- Statements DO NOT terminate with a semi-colon (;)
- When blocking code { } the open curly brace must be on the same line as the control structure!
  - func main() {
  - for i := 1; i < leng(os.Args); i++ {
- The only supported loop is the for-loop (no while and no foreach).
  - Loops DO NOT have ( ) in the header, but do have the open curly brace to block the body
- **&var** is a pointer to var (like a reference variable in Java); ***var** is an alias
  - m := 2
  - n := &m //n points to m and is 2. n is now considered an alias
  - *n = *n * 8 // BOTH are 16; again n is an alias of m
  - fmt.Printf("m is %d and n is %d\n", m, *n) // because it was defined as a pointer to m

- All programs must be part of a package – this helps when importing other packages. It's used as the default identifier. For example, math is a package in Go that may be imported. This allows you to use math members, such as math.Abs(v): **fmt.Printf("Value is %.2f", math.Abs(-15.333))**

  For us, most of the time, our files will be in **package main.** All executable programs begin in function main, which is part of package main (see Hello World program below).

- All functions within a package begin with uppercase
  - The ***fmt*** package allows for formatted output: fmt.Printf(…)
    The output is similar to Java's System.out.printf: **fmt.Printf("%.2f", x)** //value of x output to 2 decimal places. There is also a fmt.Println(…) similar to System.out.println <u>but all values must be strings or string literals</u>
- When importing a Go package, it MUST be used – otherwise it is a syntax error
  - To avoid a syntax error, use the blank identifier to create a blank import _ (underscore followed by a space)
  - A blank import runs the init() function for the import without having to explicitly use it (why you don't get a syntax error). For us, we will blank import the mysql driver that runs the init to register it:

    ```
    import (
       _ "github.com/go-sql-driver/mysql"
    )
    ```
  - NOTE: There is a space between the underscore (_) and double-quotes (")

Hello World Program

Begin by creating a Hello World program within the src folder – hello.go

```
package main
import (
  "fmt"
)

func main() {
  fmt.Printf("hello, world\n")
}
```

Navigate to your src folder and compile & execute: **go run hello.go**

<u>Comments</u>

Single line: //
Multi-line: /* */

<u>Operators</u>

Unary: ++, --
Binary: +, -, *, /, %
Compound Assignment: +=, -=, *=, /=, %=
Relational:
==, === (identical, including type)
!=, <>, !== (not identical)
>, >=, <, <=

Logical: &&, ||, !, and, or, xor
String:
+ (concatenation)
+= (concatenation append)

Variables

= → assignment. Begin with the keyword ***var***; must give a variable name AND a data type (int, int64, float64 and string the most common). May also initialize: var name type = expression

For example: var i, j, k int // NOTE that the data type FOLLOWS the variable name

Examples continued:

```
var err error
var name [ ]string      //slice – array dynamically expands and collapses
var p Point             //pre-defined structure
```

:= → short variable declaration; requires a variable name and initial value; data type is determined by initial value

For example: kelvin := 273.15 // a float64 (double in Java)

Constants

```
const var= value        // only one constant
const (                 // multiple constants
    var1 = value1
    var2 = value2
)
```

Tuple Assignment

Go allows several variables to be assigned at once. ALL right-hand expressions are evaluated BEFORE assigning to variables on the left

Examples:

```
a[ i ], a[ j ] = a[ j ], a[ i ]   //Swaps the i^th and j^th elements of array a
for y != 0 {                      //Determining gcd of x & y in a for-loop
    x, y = y, x%y
}
f, err = os.Open("myFile.txt")    //Function call to open a file returns two values: file handler and an
                                    error
```

Strings (must import strings package)

In ***Go***, strings are enclosed in double-quotes, while characters are enclosed in single-quotes. A *rune* is the integer equivalent of a character (For example 'A' is a rune with integer value 65).

strings.toLower(str)
strings.toUpper(str)
strings.Title(str) - all words uppercase
strings.TrimSpace(str) - trims all leading and trailing white space; you may also TrimLeft & TrimRight
strings.Replace(str, strStart, strReplace, n) - using str replaces n occurrences of strStart with strReplace. The variable n is the number of replacements. Let n = -1 to replace all occurrences.
len(str) - length of str
str2 = str1[start:finish] -creates a substring of str1 from index *start* up to but not including index *finish* if start is omitted, substring begins at index 0; if finish is omitted, substring continues to the end of the string
strings.Contains(str, substring) - returns true if the substring is contained within str (may also use a character)

strings.Repeat(str, n) - repeats str n number of times
strings.Index(str, substring) - returns index of first occurring substring in str
strings.Compare(str1, str2) - compares ASCII (so uppercase are smaller than lowercase); if str1 < str2 → -1
if str1 > str2 → 1; if str1 == str2 → 0

Numerical/Math Functions (must import math package)

math.Abs(x) - absolute power of x
math.Pow(x, y) - x ^ y
math.Sqrt(x) - squareroot of x
math.Mod(x, y) - modulus; remainder after devision
rand.Int31n(x) - must import math/rand; returns a 32-bit integer from 0 (included) up to x (not included)
math.Round(x) - rounds up if above .5, otherwise rounds down
math.Ceil(x) - rounds up
math.Floor(x) - rounds down/truncates
math.IsNan(x) - for example, an imaginary number (squareroot of negative number)

unicode.IsDigit(v) - returns true if v can be represented as a Unicode (think ASCII) → single quotes
- boolean functions that require importing ***unicode***
unicode.IsNumber(v) - returns true if v is a number

Control Structures

Selection

```
if {            // else is optional BUT if there is an else, it MUST be on same line as closing if }
…
} else {
…
}
```

The if-statement also allows for a short statement to be executed BEFORE the condition. This variable is only visible within the if-block (includes if and else, assuming there is an else clause), and requires a semi-colon to separate this statement from the condition:

```
if myVal := math.Pow(a, b);  myVal < 100 {
        result := myVal
} else {
        result := 0
}
```

Go also supports the switch statement. No break is used; if a matching case is found, only that case is executed. It also supports a short variable assignment like the if-statement, and is also optional. A case may also have a function call rather than a constant (i.e., 0, 1 & 2 below would call functions instead):

```
switch myVar := x % 3; myVar  {
        case 0:
                …
        case 1:
                …
        case 2:
                …
        default:
```

```
                    …
}
```

Without a condition, the switch behaves the same as an if/else-if statement

```
Switch age {
        case age < 21:
        case age < 65:
        default:
}
```

Looping

Go only supports the for-loop and its header is similar to Java. Parentheses DO NOT surround the header and { } are ALWAYS required:
for initialVar := val; initialVar condition; increment:

```
for i := 0; i < len(mySlice); i++ {
…
}
```

Alternatively, to traverse through an entire slice/array use ***range***, which returns <u>two</u> values, the index AND the element:

```
for i, v:= range mySlice {                  // i is the index
        fmt.Printf("%d and %d\n", i, v)     //assuming mySlice is a slice of int
}
```

If you do not need the index, use a blank identifier:

```
for _ , v:= range mySlice {                 // i is the index
        fmt.Printf("%d\n", v)               //assuming mySlice is a slice of int
}
```

The initialVar and increment are optional. When these are dropped, the for-loop works the same as a while-loop:

```
for myVal < 10 {
…
myVal += 2
}
```

<u>Arrays/Slices</u>

A slice has an underlying array associated with it – it is said to describe a piece of an array (arrays are fixed-length). For us, we will primarily use slices which work similarly to ArrayList in Java in that they can dynamically expand and shrink, though it can never expand beyond the length of the array it is underlying. Instead, to expand the capacity, you must create a new slice – see second way to define a slice using ***make***.

There are three primary ways to define a slice:

```
a := var [ ] datatype{value1, value2, etc. }
or
a := make([ ] datatype, length, capacity)        //capacity is an optional field; if left off capacity = length
```

or
b := a[low: high] //low index included; high index NOT included

Use append to add a value to a slice (must be same type as the slice): ***mySlice = append(mySlice, value)*** where value is the data type of the slice
Consider the following slices a-f

```
a := make([]int, 5)                  // a is defined with a length of 5; all elements are 0
        printSlice("a", a)           // a = [0, 0, 0, 0, 0]

for i:= 0; i < 5; i++ {              //a now has the following values:  a = [0, 1, 2, 3, 4]
  a[i] = i
}
fmt.Printf("\nNEW A\n")
printSlice("New a", a)

b := make([]int, 0, 5)               //b's length is initially 0, but a capacity of 5
printSlice("b", b)

b = append(b, 100)                   //100 & 250 added to b:  b = [100, 250]
b = append(b,250)
fmt.Printf("\nNEW B\n")
printSlice("New b", b)

c := a[:2]                           //c is a's slice beginning at index 0
printSlice("c", c)                   //up to but not including index 2:  c = [0, 1]

d := c[2:5]                          //d is c's slice from index 2 up to but not including
printSlice("d", d)                   //index 5.  Because c is based on a, a's values are used
                                     //to fill in since c is only index 0 & 1 from slice a
                                     //d = [2, 3, 4]

e := a[3:]                           //e is a's slice beginning at index 3 to len(a)
printSlice("e", e)                   //e = [3, 4]

f := b[:cap(b)]                      //f uses b's capacity for its length, which is 5
printSlice("f", f)                   //f = [100, 250, 0, 0, 0]

}
func printSlice(s string, x []int) {
        fmt.Printf("%s len=%d cap=%d %v\n",
                s, len(x), cap(x), x)
}
```

Structs & Maps (Associative Arrays)

Struct

Technically a struct is an aggregate data type. You may use this like a key-value pair in an associative array instead of indexing in a standard array:

```
type Name struct{        - where Name is the actual name of your struct
}
```

Example 1

```
type Person struct {      //Fname & Lname are the fields. Note that fields begin with Uppercase
        Fname  string
        Lname  string
}

me := Person{Fname:"Kristi", Lname:"Davidson"}
fmt.Println(me.Fname + " " + me.Lname)
me.fname = "Gregg"
fmt.Println(me.Fname + " " + me.Lname)
```

Example 2

```
type Point struct {       //Again, X & Y are the fields. Note that fields begin with Uppercase
        X, Y int
}

type Circle struct {
        Center Point
        Radius int
}
type Wheel struct {
        Tire Circle
        Spokes int
}
//To use these 3 structures
var w Wheel
w.Tire.Center.X = 8
w.Tire.Center.Y = 8
w.Tire.Radius = 5
w.Spokes = 20

w2 := Wheel{Circle{Point{3, 3}, 2}, 15}
fmt.Printf("Center at (%d, %d) Radius = %d Spokes = %d\n", w2.Tire.Center.X, w2.Tire.Center.Y,
w2.Tire.Radius, w2.Spokes)

w3 := Wheel{
    Tire: Circle{
      Center: Point{X: 3, Y: 3},
      Radius: 2,
    },
```

```
        Spokes: 15,    //NOTE: Trailing comma required
      }
    fmt.Printf("My Third Wheel: %#v\n", w3)   //# allows the %v, verb, to print all values of w3
```

Maps

A map maps keys to values:

```
func main () {
        var myNames map[string]Person
        myNames = make(map[string]Person)
        myNames["Instructor"] = Person{"Kristi", "Davidson"}
        fmt.Println(myNames["Instructor"])                    //Prints {Kristi Davidson}
}
```

Alternatively, when the map is defined it may also be populated (values pushed on a stack – FIFO)

```
myNames = map[string]Person{
   "Student 1": Person{"John", "Doe"},
   "Student 2": Person{"Suzy", "Smith"},
 }
 fmt.Println(myNames)                  //Prints map[Student 2:{Suzy Smith}  Student 1:{John Doe}]
```

Functions

A function in ***Go*** is defined as follows and must be at the package level (outside of func main):

```
func name(parameter-list)  data types of return values  {
        body
}
```

Example:

```
func twice(num int) int {
    return 2 * num
}
```

how ***twice*** is called in main

```
func main() {
        myVal := twice(5)
        fmt.Printf("Double %d is %d", 5, myVal)
```

Alternatively, ***Go*** also offers what is known as a ***Closure*** (Anonymous Function). Closures are written without a name and are the return value of a traditional, literal function:

Example (taken from A Tour of Go):

```
func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}
func main() {
    pos, neg := adder(), adder()
    for i := 0; i < 5; i++ {
        fmt.Println(
            pos(i),
            neg(-2*i),
```

```
Output to this closure:
0 0
1 -2
3 -6
6 -12
10 -20
```

```
                )
        }
}
```