Go with Examples

Introduction: Hello World, values, and variables

Go code layouts

A Go project is also called a module. A module is just a collection of packages.

A package is just a group of related .go files. You would declare the .go files that belong in the same package with the line

```
package <package name>
```

For example: If you use the `main` package it is used to make the package an executable program (you get a binary) because it contains your main function. The `main` package tells the Go compiler that the package will be compiled as an executable program rather than a library which will not produce an executable.

Otherwise, the package name can be whatever you want. However, keep the package name that you are declaring the same as the directory that it is under. For example:

```
src
helper
    helper.go
    main.go
```

If you have a directory like such keep the package name that you use in `helper.go` as `helper` because if do `package lol` which doesn't match the directory name. You would be importing the helper package in `main.go` as

```
import (
"module/path/helper
)
```

But when you want to call the function from the `lol` package it would be

```
lol.helperFunc()
```

So keeping the directory name and the package name the same would make it easier for yourself and for others to maintain.

Hello World

```
package main

import "fmt"

func main() {

    var a = "initial"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)

    f := "apple"
    fmt.Println(f)
}
```

As you can see the `fmt` module that is imported is the built-in module in Golang for printing things out to the consoles.

`Println` is just one of the functions inside `fmt` module to print things, there are many others.

> Notice that the function is capitalized. This is not arbitary but rather telling Go that the Println function from the `fmt` module is exported for others to use. If it is lower-case then it cannot be used by others.

Variables

Couple of ways of declaring a variable:

1. This is the most verbose way of declaring and assigning it, you have the type of the variable on the right and then assignment to the variable of the expected type to the right of the assignment operator

```
var x int = 10
```

2. A simpler variable declaration and assignment of the previous method, you can skip out on the type if the type on the right is what you want of the assignment. This is possible because the compiler can infer the type in compile time

```
var x = 10
```

3. If you just want to declare the variable without assigning it here it is. The variable will have it's zero value assigned.

```
var x int
```

4. You can also declare multiple var in a block. Also provide them with initialization as well

```
var (
	x int
	y int
	z string = "default"
)
```

5. Go also have a short hand declaration that you can only use within a function. This method cannot be used outside of the function, but `var` can be. You can just skip out the `var` keyword and add in a colon. The compiler will use type inference to infer the type on the right.

```
x := 10
```

6. There is also constants in Go but they are pretty limited. You can only assign them the values that compiler can figure out at compile time. Type with constant is allowed as well.

```
const y = "hello world"
```

var vs :=

There are limitation on `:=`. Such as you cannot use it outside of functions to declare a global variable like you can with `var`.

You cannot have a zero value if you are using `:=`. You must assign something to it in order to make it work.

Zero value

Type	Zero value
Boolean	false
Numeric types	0
Float	0
String	"" (Empty string)
Structs	Each of struct's key is set to their respective zero-value

If/else

If/else

If else is very similar to C except you just take out the parenthesis.

```
func main() {
	x := 10

	if x > 10 {
		fmt.Println("It is greater than 10")
	} else if x < 10 {
		fmt.Println("It is less than 10")
	} else {
		fmt.Println("It is equal to 10")
	}
}
```

Notice that the `else` and `else if` MUST be after the closing bracket of the previous statement. Otherwise, it will not compile.

Temporary variable

If/else generate it's own block, with this you can do something very interesting. You can declare and assign a variable that is only valid within that particular if/else block like such:

```
func main() {
	if x := 10; x > 10 {
		fmt.Println("It is greater than 10")
	} else if x < 10 {
		fmt.Println("It is less than 10")
	} else {
		fmt.Println("It is equal to 10")
	}
}
```

Ternary operator?

No ternary operator, must use a full if/else branch for assignment of the variable.

Switch Statement

Switch Statement

Switch in Go is pretty nice, it function similar to the switch statement in C but better.

```
x := 20


switch x {
  case 10, 20:
    fmt.Println("It is either 10 or 20")
  default:
    fmt.Println("It is not 10 and is not 20")
}
```

You can put case on multiple values rather than only just one. In addition, the default case is not mandatory, you can leave it out of your program.

Switch on conditions

Furthermore, instead of switching on a value, in the previous example it was switching on the variable `x` however, if you leave the value to switch on out, you can switch on other variables conditionally.

```
x := 22


switch {
  case x > 10:
     fmt.Println("It is greater than 10")
  case x < 10:
     fmt.Println("It is less than 10")
  case x == 10:
     fmt.Println("It is equal to 10!")
}
```

Arry and Slices

Array

An array can be declared with the following syntax:

```
var <arr_name> [length]type
```

For example if I want to create a list of array with length of 5:

```
var a [5]int
```

The array is initialized with zero value of the type. So since it is integer it will be initialized to be 0. If it were `bool` it will all be `false`.

Indexing

Indexing the array is just as the same as with other array

```
arr_name[index]
```

There are no negative indexing like Python to prevent unintentional bugs.

Slices

Array is fine on it's own, but it comes with a down side, in the sense that it is very rigid. If you want to pass an array to a function, the size must be specified, otherwise, it cannot be passed.

Therefore, slices which are references to array are created to combat that issue. Passing slices of array allows the size to vary depending on the array.

The function below can only take in an array of 3 elements, if you want this function to work with any array, then you will have to make it to take in a slice.

```
func printArray(nums [3]int) {
	i := 0
	for i < len(nums) {
		fmt.Printf("The number is %d\n", nums[i])
```

```
    i += 1
  }
}
```

This variation which just took out the number becomes a slice, which will allow it to take in any slice of varying sizes.

```
func printArray(nums []int) {
  i := 0
  for i < len(nums) {
    fmt.Printf("The number is %d\n", nums[i])
    i += 1
  }
}
```

Loops and range

For loops

The only looping construct in Go is the for loop. There is no while loop, but it is actually just merged into the for loop.

Loop on condition

Basically while loop except you replace the while with for

```
i := 1


for i <= 3 {
  fmt.Println(i)
  i = i + 1
}
```

Normal for loops

The same for loop that you see in C, Java, or JavaScript without the parenthesis.

```
for j := 0; j < 5; j ++ {
  fmt.PrinItn(j)
}
```

You can break on condition as well.

Range

The range keyword is used to iterate through an iterable object such as an Array or Map.

```
nums := []int{1, 2, 3, 4}
for i, num := range nums {
  fmt.Println(i)
  fmt.Println(num)
}
```

If you don't want the index then you can replace it with `_` to ignore that variable.

The `range` keyword can also be applied on Maps to iterate through the key-value pairs.

Functions

Functions

To write a function you would need to use the `func` keyword, provide the name of the funciton, provide in the argument of the function, and finally the return type of the function:

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

Multiple return values

You can return multiple values in a function, however, you would need to explicit state what those return values are:

```
func vals() (int, int) [
  return 10, 20
}
```

In this case, the function `vals` will return two return values which are just simply two integers. Notice that the return types are surrounded by parenthesis, this is needed, if you only have one return value then you skip it.

Functions that takes in variable number of arguments

In Python, Java, JavaScript there is the concept of making function accept in a variable number of arguments, in Go there is that concept too.

And surprisingly it is also done with the same operator `...` in most languages.

```
func sum(nums ...int) int {
  total := 0
  for _, num := range nums {
    total += num
  }
  return total
}
```

The type of `nums` will be `[]int`.