

Introduction to Go

Programs

- Instructing computer in binary or assembly is tedious
- High level languages help us in writing instructions in a human-readable format
- Go is:
 - A High level programming language compared to binary and assembly
 - A Low level programming language compared to Python
 - A Systems level programming language
- All programs have two states:
 - Compile: Source code verified and translated to machine executable code
 - Run: Run source code executable
- All programs have access to two types of storage at runtime:
 - Stack based:
 - Data structure which follows LIFO(Last In First Out), consider example of a stack of plates
e.g. let us have 5 plates, a, b, c, d and e
If I want to place these on top of each other, this will be the order:
plate e // this plate will come out at first(LIFO)
plate d
plate c
plate b
plate a
 - Heap based:
 - Data structure which is based on a tree data structure i.e, each entry points to another entry
 - Unordered
 - Ordering can be enforced by using min-heap(stores the smallest item at the root node), max-heap(opposite of min-heap)
 - Stack is faster, but memory size is limited
 - Heap is slower, but memory size is larger: [Values on the heap can be accessed only through their memory address](#), if address is lost, value is lost even though it might exist on the heap

Programming Languages

Programming languages can be categorized on different parameters. One of those is the typing convention that a language opts for, and supports.

Statically Typed Languages

- Types are static after they have been defined for a variable

- In the following code, type of variable x is static after being defined for a scope

Note: Scope in this case is defined as the area bounded by the curly braces.

```
{  
var x = 0 // x is declared as an integer (int)  
x = 4.3 // Invalid semantics: Trying to assign a floating-point value  
(float) to an integer  
x = "Shashank" // Invalid semantics: Trying to assign a string value to an  
integer  
}
```

Dynamically Typed Languages

- Types are dynamic i.e, types of variables can change within the same scope based on value
- Following code will work in JavaScript(dynamically typed), but not in Golang(statically typed):

```
var x = 0; // x is initially declared as a number (int equivalent in Go)  
x = 4.3; // This is valid, but x now holds a floating-point number (float  
equivalent in Go)  
x = "Shashank"; // This is also valid; JavaScript dynamically changes x's  
type to a string
```

The Go development environment(Linux)

Writing Go code and running it requires the following:

Go installation

- Go installation will add the go binary to your system
- This go binary has the go compiler and includes other helper packages
- Some jobs of this compiler are:
 - Verify your code based on syntax and semantics
 - Transform code to machine executable code
- `go build` or `go run` can be used for building and running Go source code
- Go also provides other tools to for standalone syntax checking and other checks without building and running your code
- Go installation can be verified using the `go version` command
- Location of the Go installation can be verified using the `which go` command
- Go compiler itself is written using Golang: this process is called bootstrapping

A code editor like VS Code, Goland, etc

Go workspace

A Go Workspace, contains source code, binaries, external libraries, and caches necessary for a Golang development project

GOROOT(Part of Go installation, not workspace)

- **GOROOT** environment variable has value on a Linux system like: `/usr/local/go`
- Location in file system where go installation lives
- This directory contains multiple subdirectories, but the most important ones are:
 - `bin`: Contains the go binary
 - `src`: Contains the packages of the Go standard library, like `fmt`, `errors`, `os`, `time`, etc

GOPATH

- **GOPATH** environment variable has value on a Linux system like: `/home/user@domain/go`
- **Go workspace containing go source files, dependencies of source file, etc.**
- This directory contains three subdirectories:
 - `pkg`: Dependencies of go source files contained in the `src/`
 - `bin`: Go binaries
 - `src`: Contains go source code
- Value of **GOROOT** and **GOPATH** should not be the same

GOBIN

- **GOBIN** environment variable has value on a Linux system like: `/home/user@domain/go/bin`
- This directory contains all go binaries including those downloaded using `go install` command

Packages and modules

Packages

- Packages are collection of functions
- Each function has a dedicated task
- To declare a package, use `package ${package name}` in the first line of a file with a `.go` extension
- Every application needs a starting point: in Go, that starting point is the `func main()` of the main package in a file with a `.go` extension
- Different packages can be imported based on requirement

Modules

- Modules are collections of packages
- To declare a module, use `go mod init` or use `module ${module name}` in the first line of a file with name `go.mod`

Variables and Data Types

A variable in any programming language is an identifier, a name, for a memory location.

Once a variable has become valid in a particular scope, it behaves according to its name i.e, the value that it stores keeps on changing until the variable expires, or goes out of scope.

Lifecycle of a variable involves four steps:

- Declaration: Creating a new name to be used as a variable
- Definition: Defining some property of the variable like a data type

- Initialization: Putting some value into the variable
- Expiration: The variable expires(goes out of scope), and the corresponding memory location becomes available for use.

Note: Scope of a variable is the duration for which the variable is valid in a program.

Scalar data types: Numbers, Byte, Rune, Complex, Boolean, Strings, Pointers

Numbers

Number types include int, byte, rune, float and complex in Golang.

- Integers

```
var x int      // Declaring and defining a variable, go lang self initializes
                the variable with default 0 value for the type
var y int = 0  // Declaring, defining and initializing a variable manually
z := 0        // Using shorthand to declare and initialize a variable,
                go lang self defines the variable as an integer based on value

x = 5
y = 10
z = 3000000000

// Size of integer data types
var a int8  // Integer values from -128 to 127
var b int16 // -32768 to 32767

var c uint8  // Integer values from 0-255
var d uint16 // 0 to 65535
```

Note: Other sizes include 32 and 64: int32 and int64, which are applicable for unsigned integers as well.

Note: If the bit size of an integer variable is not specified, it is decided based on the bit size of the processor: int32 on 32-bit systems, int64 on 64-bit systems.

- Float
 - Variables of type float are used to store decimal values
 - Variables of type float need to be defined with the bit size i.e, cannot be float, but either float32 or float64

```
var x, y, z float32 // Golang allows defining multiple variables of the
                    same type in a single statement

x = 5.74
y = 10.68
z = 3000000000.00000000
```

Strings

- Strings in Golang are immutable
- "Shashank": After saving this value in memory, this value cannot be modified
- This value can be read, copied and a new value created out of it
- String in Go are stored as their ASCII representations: this is a special type in Go called rune
- "Shashank" will be stored as an array of runes on memory, each rune representing the ASCII value of a character
- "A" = 65, "B" = 66 this ASCII representation of a character is called a rune

```
var str string = "Shashank"
```

Pointers

- Pointers are special types in Go
- Any variable of type pointer does not point to a value
- Variable of pointer type points to an address of a memory location
- Pointers have two signs associated with them:
 - *: Used to define type of variable as a pointer, or to get the actual value stored at the memory address stored in the pointer variable
 - &: Used to get the memory address of a value stored in a variable

```
var x *int // This means that x will contain address of a memory location
that can only contain integer values

y := 5
x = &y // & is used to get the memory location of a variable

fmt.Println("value of y: ", y) // Prints 5
fmt.Println("value of x: ", x) // Prints address where 5 is stored

fmt.Println("dereferenced value of x: ", *x) // dereferencing // This
prints 5, as * is used to declare pointer types and to get value contained
at a memory location
```

```
// If we take an example of a house, x is the name of the house: Shashank's
House
x := "Shashank" // Each variable has a lifecycle: Declaration, Definition,
Initialization, in this statement only Declaration and Initialization are
explicit, Definition is implicit based on value

// s is the address of the house: Village: Narnia, Telangana - 500000
// In computing terms, an address is represented as a hex value, e.g
0xc000014070, 0xc00004a020, etc
var s *string // s will point to a memory location: s = some memory
location, and this memory location can only contain strings
s = &x // Returns memory address of the value in x
// When & is used, it is an instruction to return the actual address, e.g
```

```
give me the address for Shashank's House which will return "Village:
Narnia, Telangana - 500000"

fmt.Println(s) // Prints gibberish memory address i.e, s = 0xc000014070 //
this value points to "Shashank"

fmt.Println(*s) // Prints value: in this case "Shashank"

fmt.Println(&s) // Since s stores a memory address, the value of s = memory
address, and this value is also stored somewhere
// When we do &s, we get the memory address of the value of s which is a
memory address
&s = 0xc00004a020 // this value points to "0xc000014070"

fmt.Println(*&s) // Since * is used to get the value stored at a memory
address, *&s prints the value stored at &s(0xc00004a020)
```

Vector data types: Arrays, Slices and Maps

Composite data type: Struct

any/interface