Programming In Go (GoLang)

By Jiten Palaparthi (JP)
Twitter: @Jiten_1981
Linkedin: https://www.linkedin.com/in/jpalaparthi/

Introduction

Go also known as Golang is an open source, compiled and statically typed programming language developed by Google. The key people behind the creation of Go are Rob Pike, Ken Thompson and Robert Griesemer. Go was made publicly available in November 2009.

Go is a general-purpose programming language with a simple syntax and is backed by a robust standard library. One of the key areas where Go shines is the creation of highly available and scalable web apps. Go can also be used to create command-line applications, desktop apps and even mobile applications.

Why Go

- Concise Syntax
- Best Suited for concurrent programs
- Compiled language
- Fast compilation
- Static linking
- Go Tooling
- Garbage collection
- Simple language specification
- Opensource

Profound Products written in Go

- Docker
- Kubernetes
- Nomad
- etcd
- NATS.io
- DropBox
- SendGrid(Many of their products)
- VMware Tanzu
- Many more opensource projects

Install Go

- https://golang.org/doc/install
- For Linux, Mac and Windows
- Supports Different Architectures
- Verify the installation

HelloWorld

- Creating workspace
- Setting up the Code Editor (VS Code)
- Install required tools (Go tools)
- go install
- go build
- go run XXXX.go
- Explaining “Hello World” program
- https://play.golang.org/p/VtXafkQHYe

Variables and DataTypes

- Variable is the name given to a memory location to store a value of a specific type. There are various syntaxes to declare variables in Go. Let's look at them one by one.
- Declaring a single variable
- https://play.golang.org/p/bycpQlMWyw-
- https://play.golang.org/p/dP8hG83Gj3K
- Declaring a variable with an initial value
- https://play.golang.org/p/zWtpqINuJlA
- Type inference?
- https://play.golang.org/p/i99Xzzz9XBD

More on Variables

- Multiple variable declaration
- https://play.golang.org/p/4aOQyt55ah
- https://play.golang.org/p/sErofTJ6g-
- https://play.golang.org/p/aPegNR4MAA_q
- https://play.golang.org/p/7pkp74h_9L
- Shorthand declaration
- https://play.golang.org/p/L5_8aru7VQM
- https://play.golang.org/p/ctqgw4w6kx
- https://play.golang.org/p/wZd2HmDvqw

More on Variables

- Shorthand declaration
- https://play.golang.org/p/MSUYR8vazB
- https://play.golang.org/p/EYTtRnlDu3
- https://play.golang.org/p/Kk84pOyFqQB
- https://play.golang.org/p/K5rz4qxjPj

Types

- These are main types in Go
    - bool
    - Numeric Types
    - int8, int16, int32, int64, int
    - uint8, uint16, uint32, uint64, uint
    - float32, float64
    - complex64, complex128
    - byte
    - rune
    - string

More on Types

- Bool
- https://play.golang.org/p/v_W3HQ0MdY
- Signed integers
- int8: represents 8 bit signed integers
- size: 8 bits
- range: -128 to 127
-
- int16: represents 16 bit signed integers
- size: 16 bits
- range: -32768 to 32767

More on Types

- int32: represents 32 bit signed integers
- size: 32 bits
- range: -2147483648 to 2147483647
-
- int64: represents 64 bit signed integers
- size: 64 bits
- range: -9223372036854775808 to 9223372036854775807

More on Types

- int: represents 32 or 64 bit integers depending on the underlying platform. You should generally be using int to represent integers unless there is a need to use a specific sized integer.
- size: 32 bits in 32 bit systems and 64 bit in 64 bit systems.
- range: -2147483648 to 2147483647 in 32 bit systems and -9223372036854775808 to 9223372036854775807 in 64 bit systems

More on Types

- int: represents 32 or 64 bit integers depending on the underlying platform. You should generally be using int to represent integers unless there is a need to use a specific sized integer.
- size: 32 bits in 32 bit systems and 64 bit in 64 bit systems.
- range: -2147483648 to 2147483647 in 32 bit systems and -9223372036854775808 to 9223372036854775807 in 64 bit systems
- https://play.golang.org/p/NyDPsjkma3
- https://play.golang.org/p/mFsmjVk5oc

More on Types

- Unsigned integers
- uint8: represents 8 bit unsigned integers
- size: 8 bits
- range: 0 to 255
-
- uint16: represents 16 bit unsigned integers
- size: 16 bits
- range: 0 to 65535

More on Types

- Unsigned integers
- uint32: represents 32 bit unsigned integers
- size: 32 bits
- range: 0 to 4294967295
-
- uint64: represents 64 bit unsigned integers
- size: 64 bits
- range: 0 to 18446744073709551615

More on Types

- Unsigned integers
- uint : represents 32 or 64 bit unsigned integers depending on the underlying platform.
- size : 32 bits in 32 bit systems and 64 bits in 64 bit systems.
- range : 0 to 4294967295 in 32 bit systems and 0 to 18446744073709551615 in 64 bit systems

More on Types

- Floating point types
- float32: 32 bit floating point numbers
- float64: 64 bit floating point numbers
- https://play.golang.org/p/upwUCprT-j

More on Types

- Complex types
- complex64: complex numbers which have float32 real and imaginary parts
- complex128: complex numbers with float64 real and imaginary parts
- The builtin function complex is used to construct a complex number with real and imaginary parts. The complex function has the following definition
- func complex(r, i FloatType) ComplexType

More on Types

- Complex types
- It takes a real and imaginary part as a parameter and returns a complex type. Both the real and imaginary parts must be of the same type. ie either float32 or float64. If both the real and imaginary parts are float32, this function returns a complex value of type complex64. If both the real and imaginary parts are of type float64, this function returns a complex value of type complex128
- Complex numbers can also be created using the shorthand syntax
- c := 6 + 7i
- https://play.golang.org/p/kEz1uKCdKs

More on Types

- Other numeric types
- byte is an alias of uint8
- rune is an alias of int32

- String Type
- https://play.golang.org/p/CI6phwSVel

Type Conversion

- Go is very strict about explicit typing. There is no automatic type promotion or conversion.
- https://play.golang.org/p/m-TMRpmmnm
- To fix the error , there should be type conversion
- https://play.golang.org/p/mweu3n3jMy
- https://play.golang.org/p/Y2uSYYr46c

Constants

- Constants
- Like in other programming languages in Go constant is a fixed value.That cannot be changed during runtime
- Declaring a constant
- https://play.golang.org/p/mv3B-q3h0zh
- Declaring a group of constants
- https://play.golang.org/p/KvZ6zNz4A04
- https://play.golang.org/p/b2J8_UQobb (Why?)
- The value of a constant should be known at compile time
- https://play.golang.org/p/GBlQDcbqfn-

More on Constants

- String Constants, Typed and Untyped Constants
- https://play.golang.org/p/oFv_cFuEucL
- https://play.golang.org/p/1Q-vudNn_9
- Boolean Constants
- https://play.golang.org/p/h9yzC6RxOR
- Numeric Constants
- https://play.golang.org/p/a8sxVNdU8M
- https://play.golang.org/p/0-eVCbJ76B5
- https://play.golang.org/p/_zu0iK-Hyj
- Numeric Expressions
- https://play.golang.org/p/Nsak9scUAWq

Functions

- What is a function?
- A function is a block of code that performs a specific task. A function takes an input, performs some calculations on the input, and generates an output.
- Function declaration
- The syntax for declaring a function in go is

```
func FunctionName(parametername type) returntype {
 //function body
}
```

- https://play.golang.org/p/FtjhPcx3ySa

More on Functions

- Multiple return values
- https://play.golang.org/p/qAftE_yke_

- Named return values
- func rectProps(length, width float64)(area, perimeter float64)

```
{
    area = length * width
    perimeter = (length + width) * 2
    return //no explicit return value
}
```

More on Functions

- Blank Identifier
  - _ is known as the blank identifier in Go. It can be used in place of any value of any type. Let's see what's the use of this blank identifier.
  - https://play.golang.org/p/IkugSH1jIt

Conditional statements

- Conditional Statements
- if is a statement that has a boolean condition and it executes a block of code if that condition evaluates to true. It executes an alternate else block if the condition evaluates to false. In this tutorial, we will look at the various syntaxes and ways of using if statement.
- if condition { }
- Unlike in other languages like C, the braces { } are mandatory even if there is only one line of code between the braces{ }.
- https://play.golang.org/p/RRxkqK07ul4

More on Conditional statements

- if else
- https://play.golang.org/p/bMevwhJhguO

- If ... else if ... else statement
- https://play.golang.org/p/VNPbCiK9eXT

- If with assignment
- There is one more variant of if which includes an optional shorthand assignment statement that is executed before the condition is evaluated. Its syntax is
- https://play.golang.org/p/_X9q4MWr4s

Loops

- A loop statement is used to execute a block of code repeatedly. for loop syntax

```
    for initialisation; condition; post {
}
```

- https://play.golang.org/p/mV6Zgcx2DY
- break
- https://play.golang.org/p/sujKy92f--
- continue
- https://play.golang.org/p/DRLN2ZHwVS
- Nested loops
- https://play.golang.org/p/0rg8fWjVDLb

More on Loops

- Labels
- Labels can be used to break the outer loop from inside the inner for loop. Let's understand what I mean by using a simple example.
- https://play.golang.org/p/BI10Rmp_Z3y
- https://play.golang.org/p/PNXliGINku
- https://play.golang.org/p/UYiz-Wtnoj
- https://play.golang.org/p/e7Pf0UDjj0
- infinite loop

  ```
  for {
  }
  ```

Switch Statement

- What is switch Statement
- A switch is a conditional statement that evaluates an expression and compares it against a list of possible matches, and then executes the block of code.
- https://play.golang.org/p/94ktmJWlUom
- Duplicate cases are not allowed
- https://play.golang.org/p/7qrmR0hdvHH
- Default Case
- https://play.golang.org/p/Fq7U7SkHe1
- Multiple Expressions in a case
- https://play.golang.org/p/AAVSQK76Me7

More on Switch Statement

- Expression less switch
- https://play.golang.org/p/KPkwK0VdXII
- fallthrough
- https://play.golang.org/p/svGJAiswQj
- fallthrough with false case
- https://play.golang.org/p/sjynOMXtnmY
- break switch
- https://play.golang.org/p/UHwBXPYLv1B
- break outer loop
- https://play.golang.org/p/0bLYOqs2TUk

Arrays and Slices

- Array
- An array is a collection of elements that belong to same type
- Declaration
- https://play.golang.org/p/Zvqh82u0ej
- Assign values
- https://play.golang.org/p/WF0Uj8sv39
- Shorthand declaration
- https://play.golang.org/p/NKOV04zgI6
- Size of array is part of the type
- https://play.golang.org/p/kBdot3pXSB

More on Arrays and Slices

- Arrays are value types
- https://play.golang.org/p/-ncGk1mqPd
- Arrays are passed by values to functions
- https://play.golang.org/p/e3U75Q8eUZ
- Length of an array
- https://play.golang.org/p/UrIeNlS0RN
- Iterate through an array
- https://play.golang.org/p/80ejSTACO6
- Range loop
- https://play.golang.org/p/Ji6FRon36m

More on Arrays and Slices

- Multi-Dimensional arrays
- https://play.golang.org/p/InchXI4yY8
- Slice
- Slice is a flexible type of an array
- https://play.golang.org/p/Za6w5eubBB
- make and new
- append, copy
- Update by reference
- Copy Slice
- Copy by reference
- Remove an element from slice

Variadic functions

- Functions in general accept only a fixed number of arguments. A variadic function is a function that accepts a variable number of arguments. If the last parameter of a function definition is prefixed by ellipsis ..., then the function can accept any number of arguments for that parameter
- https://play.golang.org/p/7occymiS6s
- https://play.golang.org/p/7occymiS6s

Maps

- A map is a built in type in Go that is used to store key-value pairs. Maps are similar to dictionaries in other languages such as Python.
- https://play.golang.org/p/-IUSnvdgF2I
- https://play.golang.org/p/oR_j4jkJf1f
- https://play.golang.org/p/qthGPO6pj0Z

More on Maps

- Checking if a key exists
- https://play.golang.org/p/Y4n1p4yfdVi
- Iterate over all elements in a map
- https://play.golang.org/p/rz8U_g2slb0
- Deleting items from a map
- https://play.golang.org/p/u0WCB-Ta_dB

Pointers

- What is a pointer?
- Declaring pointers
- Zero value of a pointer
- Creating pointers using the new function
- Dereferencing a pointer
- Passing pointer to a function
- Returning pointer from a function
- Do not pass a pointer to an array as an argument to a function. Use slice instead.
- Go does not support pointer arithmetic

Pointers

- A pointer is a variable that stores the memory address of another variable.
- Declaring pointers
- *T is the type of the pointer variable which points to a value of type T.
- https://play.golang.org/p/A4vmlgxAy8
- Zero value of a pointer
- https://play.golang.org/p/yAeGhzgOE1
- Creating pointers using the new function
- https://play.golang.org/p/BNkfB3RZCOY

More on Pointers

- Dereferencing a pointer:Dereferencing a pointer means accessing the value of the variable to which the pointer points. *a is the syntax to dereference a.
- https://play.golang.org/p/m5pNbqFwbM
- https://play.golang.org/p/cdmvlpBNmb
- Passing pointer to a function
- https://play.golang.org/p/3n2nHRJJqn
- Returning pointer from a function
- https://play.golang.org/p/I6r-fRx2qML
- Go does not support pointer arithmetic
- https://play.golang.org/p/WRaj4pkqRD

Packages

- What is a package
- Using Packages
- go mod
- Creating user defined package
- Calling package from the program

Structs

- What is a struct?
- A struct is a user-defined type that represents a collection of fields. It can be used in places where it makes sense to group the data into a single unit rather than having each of them as separate values.
- Declaring a struct

```
type Employee struct {
    firstName string
    lastName  string
    age       int
}
```

More on Structs

- Creating named structs
- https://play.golang.org/p/WPlLuPy0Lty
- Creating anonymous structs
- https://play.golang.org/p/m_7UoICTiMy
- Accessing individual fields of a struct
- https://play.golang.org/p/iggKCd8xUMy
- Zero value of a struct: https://play.golang.org/p/jiCEH1tFvgW
- Pointers to a struct: https://play.golang.org/p/Rli_WqmE9_H
- Anonymous fields: https://play.golang.org/p/zDkb0EbLqyJ
- Nested structs : https://play.golang.org/p/ZwfRatdwc4p
- Promoted fields : https://play.golang.org/p/0sFliXv2FqV

More on Structs

- Exported structs and fields
- Write a demo about it
- Structs Equality
- Structs are value types and are comparable if each of their fields are comparable. Two struct variables are considered equal if their corresponding fields are equal
- https://play.golang.org/p/ntDT8ZuOVK8

Methods

- A method is just a function with a special receiver type between the func keyword and the method name. The receiver can either be a struct type or non-struct type.
- The syntax of a method declaration is provided below.

```
func (t Type) methodName(parameter list) {
}
```

- Sample Method : https://play.golang.org/p/rRsI_sWAOZ

More on Methods

- Go is not a pure object-oriented programming language and it does not support classes. Hence methods on types are a way to achieve behaviour similar to classes. Methods allow a logical grouping of behaviour related to a type similar to classes. Methods with the same name can be defined on different types whereas functions with the same names are not allowed.
- Example : https://play.golang.org/p/0hDM3E3LiP
- Pointer Receivers vs Value Receivers
- The difference between value and pointer receiver is, changes made inside a method with a pointer receiver is visible to the caller whereas this is not the case in value receiver.

More on Methods

- https://play.golang.org/p/tTO100HmUX
- Methods of anonymous struct fields
- https://play.golang.org/p/vURnImw4_9
- Methods with non-struct receivers
- https://play.golang.org/p/sTe7i1qAng

Interfaces

- In Go, an interface is a set of method signatures. When a type provides definition for all the methods in the interface, it is said to implement the interface.
- Declaring and implementing an interface
- https://play.golang.org/p/F-T3S_wNNB
- Practical use of an interface
- https://play.golang.org/p/3DZQH_Xh_Pl
- Interface internal representation
- https://play.golang.org/p/kweC7_oELzE

More on Interfaces

- Empty interface
- An interface that has zero methods is called an empty interface. It is represented as interface{}. Since the empty interface has zero methods, all types implement the empty interface.
- https://play.golang.org/p/Fm5KescoJb
- Type assertion : is used to extract the underlying value of the interface.
- i.(T) is the syntax which is used to get the underlying value of interface i whose concrete type is T
- https://play.golang.org/p/YstKXEeSBL

More on Interfaces

- https://play.golang.org/p/0sB-KlVw8A
- Type switch: is used to compare the concrete type of an interface against multiple types specified in various case statements. It is similar to switch case. The only difference being the cases specify types and not values as in normal switch.
- https://play.golang.org/p/XYPDwOvoCh
- https://play.golang.org/p/o6aHzIz4wC
- Implementing interfaces using pointer receivers vs value receivers
- https://play.golang.org/p/IzspYiAQ82

Concurrency

- Go is a concurrent language and not a parallel one. Before discussing how concurrency is taken care in Go, we must first understand what is concurrency and how it is different from parallelism.
- What is concurrency?
- Concurrency is the capability to deal with lots of things at once. It's best explained with an example.
- Concurrency is not parallelism
- Support for concurrency in Go

More on Concurrency

- Concurrency is an inherent part of the Go programming language. Concurrency is handled in Go using Goroutines and channels. We will discuss them in detail in the upcoming tutorials.
- What are Goroutines?
- Goroutines are functions or methods that run concurrently with other functions or methods. Goroutines can be thought of as lightweight threads. The cost of creating a Goroutine is tiny when compared to a thread. Hence it's common for Go applications to have thousands of Goroutines running concurrently.

More on Concurrency

- Advantages of Goroutines over threads
- Goroutines are extremely cheap when compared to threads. They are only a few kb in stack size and the stack can grow and shrink according to the needs of the application whereas in the case of threads the stack size has to be specified and is fixed.
- Goroutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using Goroutines. Channels can be thought of as a pipe using which Goroutines communicate.

More on Concurrency

- The Goroutines are multiplexed to a fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines. If any Goroutine in that thread blocks say waiting for user input, then another OS thread is created and the remaining Goroutines are moved to the new OS thread. All these are taken care of by the runtime and we as programmers are abstracted from these intricate details and are given a clean API to work with concurrency.

More on Concurrency

- How to start a Goroutine?
- https://play.golang.org/p/zC78_fc1Hn
- https://play.golang.org/p/U9ZZuSql8-
- Starting multiple Goroutines
- https://play.golang.org/p/oltn5nw0w3

More on Concurrency

- Channels
- What are channels
  - Do not communicate by sharing memory; instead, share memory by communicating.
- Channels can be thought of as pipes using which Goroutines communicate. Similar to how water flows from one end to another in a pipe, data can be sent from one end and received from the other end using channels.
- Declaring channels : https://play.golang.org/p/QDtf6mvymD
- Sending and receiving from a channel
- data := <- a // read from channel a
- a <- data // write to channel a
- Sends and receives are blocking by default

More on Concurrency

- Channel example program
- https://play.golang.org/p/U9ZZuSql8-
- https://play.golang.org/p/I8goKv6ZMF
- Better understanding blocking
- https://play.golang.org/p/EejiO-yjUQ
- Another example for channels
- https://play.golang.org/p/4RKr7_YO_B
- Deadlock : https://play.golang.org/p/q1O5sNx4aW
- Unidirectional channels
- https://play.golang.org/p/PRKHxM-iRK
- https://play.golang.org/p/aqi_rJ1U8j

More on Concurrency

- Closing channels and for range loops on channels
- https://play.golang.org/p/XWmUKDA2Ri
- https://play.golang.org/p/JJ3Ida1r_6
- https://play.golang.org/p/oL86W9Ui03
- What are buffered channels?
- So far we have discussed unbuffered channels. Channels without buffer is unbuffered channel.
- It is possible to create a channel with a buffer. Sends to a buffered channel are blocked only when the buffer is full. Similarly receives from a buffered channel are blocked only when the buffer is empty.

More on Concurrency

- `ch := make(chan type, capacity)`
- https://play.golang.org/p/It-em11etK
- https://play.golang.org/p/bKe5GdgMK9
- Deadlock : https://play.golang.org/p/FW-LHeH7oD
- Length vs Capacity : https://play.golang.org/p/2ggC64yyvr
- A WaitGroup is used to wait for a collection of Goroutines to finish executing. The control is blocked until all Goroutines finish executing. Let's say we have 3 concurrently executing Goroutines spawned from the main Goroutine. The main Goroutines needs to wait for the 3 other Goroutines to finish before terminating. https://play.golang.org/p/CZNtu8ktQh

More on Concurrency

- The select statement is used to choose from multiple send/receive channel operations. The select statement blocks until one of the send/receive operations is ready. If multiple operations are ready, one of them is chosen at random. The syntax is similar to switch except that each of the case statements will be a channel operation.
- https://play.golang.org/p/3_yaJSoSpG
- Default case : https://play.golang.org/p/8xS5r9g1Uy
- Deadlock and default case
- https://play.golang.org/p/za0GZ4o7HH
- https://play.golang.org/p/Pxsh_KlFUw

More on Concurrency

- https://play.golang.org/p/IKmGpN61m1
- Random selection : https://play.golang.org/p/vJ6VhVl9YY
- Mutex : is used to provide a locking mechanism to ensure that only one Goroutine is running the critical section of code at any point in time to prevent race conditions from happening.
- Mutex is available in the sync package. There are two methods defined on Mutex namely Lock and Unlock. Any code that is present between a call to Lock and Unlock will be executed by only one Goroutine, thus avoiding race condition.

  ```
  mutex.Lock();  x = x + 1
  ```
- mutex.Unlock()

More on Concurrency

- Solving the race condition using a mutex
- https://play.golang.org/p/VX9dwGhR62
- https://play.golang.org/p/M1fPEK9lYz

Defer

- Defer statement is used to execute a function call just before the surrounding function where the defer statement is present returns.
- https://play.golang.org/p/IlccOsuSUE
- Arguments evaluation
- https://play.golang.org/p/sBnwrUgObd
- Stack of defers
- https://play.golang.org/p/x05NlTsDPuT

Error handling

- Errors indicate an abnormal condition occurring in the program. Let's say we are trying to open a file and the file does not exist in the file system. This is an abnormal condition and it's represented as an error.
- Errors in Go are plain old values. Errors are represented using the built-in error type. We will learn more about the error type later in this tutorial.
- https://play.golang.org/p/yOhAviFM05
- Asserting the underlying struct type and getting more information from the struct fields

Error handling

- https://play.golang.org/p/dWpiCuC0wXz
- Do not ignore errors
- https://play.golang.org/p/2k8r_Qg_lc

Panic

- An unrecoverable error where the program cannot simply continue its execution or A programmer error.
- panic function func panic(interface{})
- https://play.golang.org/p/xQJYRSCu8S
- https://play.golang.org/p/__PAabvchxt
- Defer Calls During a Panic
- https://play.golang.org/p/oUFnu-uTmC

Recover

- recover is a builtin function that is used to regain control of a panicking program.
- Recover is useful only when called inside deferred functions. Executing a call to recover inside a deferred function stops the panicking sequence by restoring normal execution and retrieves the error message passed to the panic function call. If recover is called outside the deferred function, it will not stop a panicking sequence.
- https://play.golang.org/p/enCM-dd5DUr
- https://play.golang.org/p/Bth98An9Ah0