BTW: This booklet is in progress. If you would like to contribute email me alan.khosro at gmail.

Elegant Go

Go is simple by "design". Go codes are elegant but you may find it difficult at the beginning due to its minimalistic approach to develop a robust, fast, safe, and scalable software.

There are many "good" programming languages such as my second favorite one, Python, but if your team starts writing (and thinking) in GO, your team will gradually adapts a new paradign in software development, which I call "Elegant Software Development" mthodology. That is what happened to me and I would like to fast forward you in this journey.

Python is easy and convenient (comformist), C++/C#/Java are theoritically advanced (academician), C/Pascal are old-style (orthodox), R/Matlab/SQL are for special purpose (niche), JS/VBasic/Ruby are supporting lnaguages (auxulary) but Go is for elegant programming (Zen).

Some people might say Go is an orthodox religion in software developmet that denies most of the recent advancement and conventions. But its simplicity resembles Zen.
If you prefer elegance over luxury, GO is your language; if you prefer feature set and convenience over quality, you better go with Python or Java.

To learn Go, you need to think simple like a Zen master. Forget complicated concepts such as object, class, polymorphism, asynchronism, generics, encapsulation, etc.
Go has a genuis way to implement parallelism, inheritance, safety, and modularity: four destruptive concepts in the last four decades after the inception of C.

This booklet encourages you to use the following resources along with this booklet:

- A Tour Of Go: It is an amazing source to learn basics of Go. I assume you already had a tour of go, if not, do not miss it.
- Go Playground: Whenever you want to test and learn, you can use this playground. We use it throughout this booklet.
- Effective Go: After reading this booklet, whenver you need to learn more, effective Go is an effective source.
- Go Standard Packages: Go's documentation for standard packages is developed to show how Go works in practice. After you advanced a little, use this source as an ultimate learning resource.

Hello World

Let us write our first Go program and print "Hello, World":

```go
package main
import "fmt"
func main () {
    fmt.Print("Hello, World.")
```

```
}
```

There you go, we printed Hello, World.
We needed standard "fmt" package to Print. Go is very small and most of functions and methods are packaged separately in standard packages.

As you noticed, Go is very specific about how to code. Modularity in Go starts with good (obligatory) practices, clear packaging, and smart naming.
For instance, using main is mandotary. main declares this is the main package (so it is not a package to be used by other packages) and the function main contains to-be-executed statements, which shows the flow of the program.
Every Exported identifier, such as Print in fmt package, has to start with a capital letter. The identifiers that starting with lower case are for internal use and they are hidden outside the package.

Let us expand the "Hello, World" program to print in a few international languages and to print the current time and to use \n as the line breaker in function "Println", which ends the prints with a line break.

```go
package main
import (
    "fmt"
    "time"
)
func main () {
    fmt.Println("Hello, World\n","你好, 世界\n", "سلام دنیا", "\nNow time is ",
time.Now())
}
```

Variables and Types

Go has a variety of primitive (and immutable) data types:

```go
bool // to store boolean variable true or false
string // to store string: usual texts
int  int8  int16  int32  int64 rune // to store integers
uint uint8 uint16 uint32 uint64 uintptr byte // to store unsigned integer
float32 float64 // to store decimal numbers
complex64 complex128 // to store complex numbers
/*
uint32 means unsigned integer of 32 bits which covers 0 to 2^32-1 integer
numbers.
rune is alias for int32 and represents a Unicode characters. byte is alias for
unit8 and represents a byte.
int, unit, and unitptr are 32 bits in 32-bit operating system and 64 bits in
64-bit operating system.
*/
```

These are primitive data types and they have negligible footprint on memory. When passing them around (for instance, as a function argument), pass them directly and do not pass their pointers. We talk about it later.

> BTW: Comment a line with // a line and comment a block of texts with /* texts */.

Go syntax is very readable, you write as you read: to declare
variable x, y of type int:

```
var x, y int
```

variable s of type string with initial value "Hello, World!":

```
var s string = "Hello, World!"
```

But it is idiomatic (and safer) to declare:

```
x, y := 3, 4
s := "Hello, World!"
i, v, name := 3, 4.0, "Scruffy" // i=3 is integer, v=4.0 is float32,
name="Scruffy" is string
```

We can create new types. To declare a new *type grade that is float32*:

```
type grade float32
```

It may seem unnecessary to create new types but it is very common in Go, especially to create new structs and to create types that have methods. We will explore them in the next chapters.

The following program takes math and PE grades to print the average grades for the given student:

```
package main
import (
    "fmt"
    "os"
)
type student string
type grade float32
var math, pe grade //variables math and pe of type grade
var name student

os.Read ()
```

```
```

Every type has a zero value that means the base of that type; often if we do not assign any value to a variable of that type, the variable takes zero value of that type.
For numeric types, off course, zero is 0 and for strings, zero is "", and for boolean type, it is false.

Go is statically-typed language, means variables must match their allocated types. Every variable and its type must be declared so that Go knows how to allocate memory to that variable.
Go program panics if you mix the types. Try the following failed program:

```
package main
import "fmt"
func main () {
    type grade float32
    var math, pe grade = 90, 85
    height := 166.5 // idiomatic for float32 decleration
    average := (math + pe + height)/3.0
    fmt.Println ("the meaningless average of your height and grades is ",
average)
}
```

Go has standard functions to convert types (or *cast types*) that we can use to resolve the above issue:

```
average := (float32(math) + float32(pe) + height)/3.0
```

Now all variables types match. Besides the primary function such as int(), float64(), string(), ... to convert types, there is a package called "con" that you can use to parse numbers from string type.

```
s := "3.14"
f := conv.String.ParseFloat64 (s) //f is float64 with value 3.14
```

But beware that wrong conversion would cause Go panic. To avoid it, you can check if the conversion is successful through checking the second return value of the above functions.

```
s := "3.14"
if f, ok := con.String.ParseFloat64 (s); !ok {
    fmt.Println ("Float64 could not be parsed")
}
```

BTW: Go functions can return multiple values and Goephers take advantage of it often. One of usual return values for many functions is error or similar values that shows if the function execution was successful. Error handling in Go is nothing but returning and checking this extra value.

> Example: We start building a real world application to store and fetch financial data.

struct

First, we need to define type Firm as a structure that is a collection of fields such as Company Name, Exchange Market, and IPO Year:

```
type Firm struct {
    Name string
    Exchange string
    IPO int // represents IPO year
}
```

Then we can define variables of type Firm to store information:

```
var goog Firm = {"Alphabet Inc", "NASDAQ", 2004}
```

We also could use the fields names and use idiomatic decleration:

```
appl := Firm{Name: "Apple Inc", Exchange: "NASDAQ", IPO:1980}
msft := Firm{Name: "Microsoft", Exchange: "NYSE"}
amzn := Firm{Exchange: "NASDAQ"}
```

If a field is left out with no value, the zero of that type will be filled.

Interface: simple way to dynamic typing

How does a function know what to do *in future* when facing a new struct? How to write a function that works for a variety of data types? The answer is via interface*. It is nothing but a collection of function signatures (or interfaces). Under the hood, it is a placeholder for a tuple (type, value) so it can be replaced with *anything*.

> **Example:** Some creatures speak, we want to write a function, called Introduce, that prints what they like to say.

We do not need to know anything about these creatures but the fact that they Speak() string. We define type Vocal as *anything* that Speak() string:

```
type Vocal interface{
    Speak() string
}
```

Then, we write Introduce function that prints what Vocals want to say:

```
func Introduce (v Vocal) {
    fmt.Println("Yeah! I can talk and I wanted to say ", v.Speak())
}
```

Done! Later, any data type that implements Speak() string can call Introduce. For instance, human can speak(), dog can Speak() too but in different ways:

```
func (d dog) Speak () string {
    return ("Woof Woof Woof "+d.name+" Woof Woof Woof.")
}
func (h human) Speak () string {
    return ("My name is "+h.name+" and I do "+h.job+" for living.")
}
```

Both types dog and human implement Introduce without further notion, just because they Speak() string.

Type Assertion: to check type compatibility

But some creatures may not Speak() string. How do we check if a type is compatible with our Vocal? Answer: by type assertion. This is espcailly useful for error handling that might happen in run-time.

Let us edit our Introduce function to accept *anything* (or interface{}), then check if *anything* is Vocal to print its saying, otherwise to print an error message on screen.

```
func introduce (anything interface{}) {
    if v, ok := anything.(Vocal); ok {
        fmt.Println ("Yeah! I can talk and I wanted to say ", v.Speak())
    }else{
        fmt.Println ("Error! This is not Vocal and does not Speak().")
    }
}
```

In comparison to other OOP languages, Go's approach in using interface is simple, beautiful, and powerful: the power of minimal design.

Let us put all pieces together and write the full program, seperated in two packages:

- **vocal** to define type Vocal and function Introduce
- **main** to use vocal package for types human and dog

```
/*
Package vocal contains functions for Vocal types. Vocal is a type that Speak()
```

```
string; Introduce(v Vocal) prints what vocal want to say.
*/
package vocal
import (
    "fmt"
)
// type Vocal is anything that Speak()
type Vocal interface{
    Speak() string
}
// Introduce() prints what Vocal wants to say
func Introduce (v Vocal) {
    fmt.Println("Yeah! I can talk and I wanted to say ", v.Speak())
}
```

Now we use package vocal in package main:

```
/*
Two types `dog` and `human` do `Speak() string` so are `Vocal`s and can use
`vocal.Introduce()` to introduce themselves but `type tree` does not `Speak()
string` and `vocal.Introduce()` prints error for tree.
*/
package main
import "vocal"
// human has name and job and speaks about them
type Human struct {
    Name string
    Job string
}
func (h Human) Speak() string {
    return ("My name is "+h.Name+" and I am a "+h.Job+".")
}
// dog has name and woof when speaking
type Dog struct {
    Name string
    age int
}
func (d Dog) Speak() string {
    return ("Woof woof woof "+d.Name+" Woof Woof Woof.")
}
// tree has age and height and fruit but does not Speak() string
type tree struct {
    age int
    height float
    fruit string
}
// human and dog implement Introduce with no further notion just because they
Speak()
func main () {
    h := Human{"Alan", "Developer"}
```

```
    d := Dog{"Scruffy", 10}
    t := tree{999, 45.0, "Maple"}
    vocal.Introduce (h)
    vocal.Introduce (d)
    vocal.Introduce (t)
}
```

Though interface{} is a very handy tool but its overuse is a sign of bad Gopher. You may need it here and there but do not think "I can write everything for an interface and implement those interfaces later." We talk about this later.

Gopher Zen

The approach we teach here is more than just Go's syntax and concepts. It takes Go philosophy to beyod coding: to the software development. Look at the comments, names, modularity, and flow of the above program.

Let us develop a software as a teamwork:

- *Architect* decide about packageing and writes a comment right before each package about what it is about.
- *Designer* writes all data types and structs and the signiture of the funcs that are supposed to be Exported to be used by other packages. Exported names start with capital letter. A brief comment right before each Exported package explains the functionality.
- *Developer* implements each func that modeler wrote its signiture. Developer might add more local and not-exported data types and funcs.
- *Tester* verifies developer code to make sure it implements the design specification. Tester might add error handling that developer had skip. In Go, functions usually return error as one of their return variables. If developer skipped them with _, tester will complete to keep the integrity of the code.

Indeed, godoc will extract the package name and all Exported names along with their preceding comments as the official documentation of that package. This way, godoc presents the *architect* and *designer* job to the outside world and hides internal implementation by *developer* and *tester*. That is all others need to know to use that package in their programs.

Gophers use brief names with all lower case for package names and package path shows the packaging hierarchy. When a package imported, the entire path will be mentioned but the last part is enough to call. For instance package decoding\json has brief, clear, lower case names with path showing the packaging hierarchy. Other programs import "decoding\json" but write json.Marchal() to call its Marshal function.

Gophers use brief and clear names for packages, funcs, types, and vars; Each package name is brief with all lower case and package path shows the packaging hiarachy;

Gophers use brief names that starts with capital letter for Exported names in each package and CamelCase is prefered to Camel_Case or other naming conventions if you need to use multiple words. Use brief and simple names, Speak() is enough to declear a method, using

Please pay attention how packaging, brief names, modularity, comments, and the flow of program is expressive; this is the Go way in software development. It is more than just coding. If Go program is written in a correct and elegant way, the teamwork and collaboration and project management (from architecture to design to develop to test) is integrated and need no further tools. This is compatible with the body of knowledge that has accumulated in software development in the past few decades. We encourage you to think and act in the tradition of **eXtreme Programming** in which the followings are the pillars:

- **Code Is Design**:
- **Reverse Engineer For Artifacts**
- **

and local camelCase are prefered to camel_case or other naming conventions.

When sharing the packages, godoc extracts the package name and its proceding comments, all `exported

and all type and the signiture of the func that are exported (starts with Capital letter) because other packages will need them.

In Go, functions (behaviors) and data structures (states) are separate.