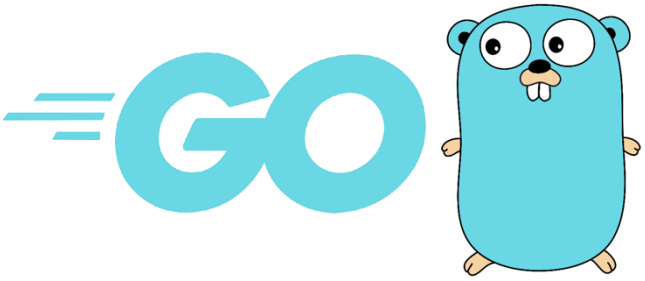
CSCI 308 Final Project

A Programmer's Guide

to



Including 33 programming design questions about Go

Lawrence Li

Spring 2021

Bucknell University

Hello World – How to run a Go program

First and foremost, download and install Go language environment from https://golang.org/ and follow the instructions specified on the website. The following instructions will be demonstrated on Mac terminal

Steps for running the Hello World program: [1]

1. Open terminal
2. Use `cd` command to move to the directory where the hello world program located
3. Type the command `go build filename.go` where `filename` is the name of the file
4. You should now have an executable file in the folder that has the same filename as your source code. For example, if you type `go build hello.go` then the output filename is defaulted to `hello` unless you specified the output name like this: `go build -o a filename.go` In this case you will get an executable file called `a`
5. Now type `./name_of_the_executable`, the program should now run.

Hello World:

```
// We declare main package, which is a way to group functions
// It's consisted of all files in the same directory
package main

// Import fmt package that contains functions for formatting text.
// This includes the Println function as shown below.
// fmt package is a standard library
import "fmt"

// We have a main function that executes by default at run
func main(){
  fmt.Println("Hello, World!");
  // We call the method from the fmt package that print the hello world
}
```

Test Run:

```
wifi173-168:hello lawrence$ ls
hello.go
wifi173-168:hello lawrence$ go build -o myprogram hello.go
wifi173-168:hello lawrence$ ls
hello.go        myprogram
wifi173-168:hello lawrence$ ./myprogram
Hello, World!
wifi173-168:hello lawrence$
```

[1] https://golang.org/doc/tutorial/getting-started Tutorial: Get Started with Go

Paradigm

Go is an imperative, compiled programming language because it has a list of operations to do in order to accomplish a task specified in a main class. [4] Go has loops, statements, and selections. For instance, Go has 'return' and 'goto' statement, as well the 'if else', 'switch', 'for', and 'select' statement, making Go an imperative language just like Python, C++, and Java. [4]

Although Go is not a strictly object-oriented language, it meets the fundamental concepts of object orientation with some terminology differences. [1] [2] Go uses 'struct' instead of objects, which is similar to what C language does. Go also does not implement any inheritances at all. Instead, go follows object composition over inheritance where classes should achieve polymorphic behavior and code reuse by their composition rather than inheritance from a parent class. To summarize, go utilizes structs as the union of data and logic just as C does. [3] It uses object composition where has-a relationships (one object belongs to another object) can be established between structs to minimize code repetition while keeping the code clear. Go uses interfaces to establish is-a relationships (one class is a subclass of another class) between types without unnecessary and counteractive declarations. [3]

Based on Go's object-oriented features, it is a post object-oriented language that borrows its structure from the Algol/Pascal/Modula language family. [1] [2] It is not a strong object-oriented language like Java. This is because that although Go allows types and methods, there is no class hierarchy but interfaces. [2] Methods can be defined for all types, including the primitive types such as int. The object in Go, which is 'struct', is so light-weighted that it looks more like C, which is not an object-oriented language. [1] [2] [3]

[1] https://spf13.com/post/is-go-object-oriented/ Is Go An Object Oriented Language?

[2] https://golang.org/doc/faq#Is_Go_an_object-oriented_language
Go Official FAQ: Is Go an object-oriented language?

[3] https://www.toptal.com/go/golang-oop-tutorial Well-structured Logic: A Golang OOP Tutorial

[4] https://golang.org/ref/spec#Statements Go Official Specifications: Statements

1. Background

The Go programming language was conceived in the late 2007 by Google. [1] The language can tackle some of the problems in software infrastructure development at Google. Go is a compiled language and was designed and developed to make working in a programming environment more productive. [1]

According to IEEE Spectrum and Stack Overflow, software engineers experienced with Go received an average of 9.2 interview requests, making it the most in-demand language. Most companies hire Go developers for webpage and front-end software products development, as well as high-performance computation. [2] [3]

[1] https://talks.golang.org/2012/splash.article#TOC_1
Go at Google: Language Design in the Service of Software Engineering

[2] https://spectrum.ieee.org/view-from-the-valley/at-work/tech-careers/go-language-tops-list-of-indemand-software-skills
Go Language Tops List of In-Demand Software Skills

[3] https://stackoverflow.com/jobs/developer-jobs-using-go
Developer Jobs Using Go

2. Primitive Types

Here is a table of all common primitive types in Go programming language [1] [2]:

Types	Values examples	Range of values
int	-2, -1, 0, 1, 2, …	OS dependent
int8		-128 to 127
int16		-32768 to 32767
int32		-2^{31} to $2^{31}-1$
int64		-2^{63} to $2^{63}-1$
uint (unsigned int)	0, 1, 2, 3, 4, …	OS dependent
uint8		0 to 255
uint16		0 to 65535
uint32		0 to 2^{32} - 1
uint64		0 to 2^{64} - 1
float32	-1.3, 4.0, 125.125, ...	Set of all IEEE-754 32-bit floating numbers
float64		Set of all IEEE-754 64-bit floating numbers
complex64	3+4i, 12-9i, 45.25+24.987i, …	Set of all complex numbers with float32 real and imaginary parts
complex128		Set of all complex numbers with float64 real and imaginary parts
string	"Hello World", …	Set of UTF-8 sequence of bytes
byte	0, 1, 2, 3, … (same as uint8)	0 to 255
error	`fmt.Errorf("oh no, there's an error")` Note: Special primitive AND interface type that wrappers around the string type	Set of UTF-8 sequence of bytes with error constructor
rune	-2, -1, 0, 1, 2, … (same as int32)	-2^{31} to $2^{31}-1$
bool	true, false	true, false

The following code tests for all primitive types of declarations:

```
package main

import (
   "fmt"
)

var (
   myBool              bool       = true        //1-bit boolean (True,False)
   myInt               int        = 4           //integer, (OS dependent bit)
   myInt8              int8       = 125         //8-bit integer
   myInt16             int16      = 32000       //16-bit integer
   myInt32             int32      = -230490     //32-bit integer
   myInt64             int64      = -9837468    //64-bit integer
   myuInt              uint       = 3           //Unsigned int (OS dependent)
   myuInt8             uint8      = 255         //8-bit unsigned int
   myuInt16            uint16     = 62323   //16-bit unsigned int
   myuInt32            uint32     = 7654332     //32-bit unsigned int
   myuInt64            uint64     = 2348282332  //64-bit unsigned int
   myFloat32           float32    = 3.14159265358979 //32-bit floating number
   myFloat64           float64    = 125.125125125125125 //64-bit floats
   myComplexNumber64   complex64  = 12.5-9.2i //64-bit complex number
   myComplexNumber128  complex128 = 23934.232323+986223.759823i //128-bit
   myString            string     = "Hello CSCI 308"  //string
   myError             error      = fmt.Errorf("oh no, there's an error")
   myByte              byte       = 255     //alias type of uint8
   myRune              rune       = 7654332 //alias type of int32
)

func main() {
   fmt.Printf("Type: %T Value: %v\n", myBool, myBool)
   fmt.Printf("Type: %T Value: %v\n", myInt, myInt)
   fmt.Printf("Type: %T Value: %v\n", myInt8, myInt8)
   fmt.Printf("Type: %T Value: %v\n", myInt16, myInt16)
   fmt.Printf("Type: %T Value: %v\n", myInt32, myInt32)
   fmt.Printf("Type: %T Value: %v\n", myInt64, myInt64)
   fmt.Printf("Type: %T Value: %v\n", myuInt, myuInt)
   fmt.Printf("Type: %T Value: %v\n", myuInt8, myuInt8)
   fmt.Printf("Type: %T Value: %v\n", myuInt16, myuInt16)
   fmt.Printf("Type: %T Value: %v\n", myuInt32, myuInt32)
   fmt.Printf("Type: %T Value: %v\n", myuInt64, myuInt64)
   fmt.Printf("Type: %T Value: %v\n", myFloat32, myFloat32)
   fmt.Printf("Type: %T Value: %v\n", myFloat64, myFloat64)
   fmt.Printf("Type: %T Value: %v\n", myComplexNumber64, myComplexNumber64)
   fmt.Printf("Type: %T Value: %v\n", myComplexNumber128, myComplexNumber128)
   fmt.Printf("Type: %T Value: %v\n", myString, myString)
   fmt.Printf("Type: %T Value: %v\n", myError, myError)
   fmt.Printf("Type: %T Value: %v\n", myByte, myByte)
   fmt.Printf("Type: %T Value: %v\n", myRune, myRune)
}
```

You should see the following output:

```
Type: bool Value: true
Type: int Value: 4
Type: int8 Value: 125
Type: int16 Value: 32000
Type: int32 Value: -230490
Type: int64 Value: -9837468
Type: uint Value: 3
Type: uint8 Value: 255
Type: uint16 Value: 62323
Type: uint32 Value: 7654332
Type: uint64 Value: 2348282332
Type: float32 Value: 3.1415927
Type: float64 Value: 125.12512512512512
Type: complex64 Value: (12.5-9.2i)
Type: complex128 Value: (23934.232323+986223.759823i)
Type: string Value: Hello CSCI 308
Type: *errors.errorString Value: oh no, there's an error
Type: uint8 Value: 255
Type: int32 Value: 7654332
```

The following code tests the actual size in memory in bytes for all primitive types.

```
package main

import "unsafe"
import "fmt"

var(
   b     bool
   i     int      //size is system dependent
   i_8   int8
   i_16  int16
   i_32  int32
   i_64  int64
   ui    uint     //size is system dependent
   ui_8  uint8
   ui_16 uint16
   ui_32 uint32
   ui_64 uint64
   f_32  float32
   f_64  float64
   c_64  complex64
   c_128 complex128
   s     string = "Hello World!"
   err   error
   by    byte
   r     rune
)
```

```
func main(){
   fmt.Printf("The size of bool is %d\n", unsafe.Sizeof(b))
   fmt.Printf("The size of int is %d\n", unsafe.Sizeof(i))
   fmt.Printf("The size of int8 is %d\n", unsafe.Sizeof(i_8))
   fmt.Printf("The size of int16 is %d\n", unsafe.Sizeof(i_16))
   fmt.Printf("The size of int32 is %d\n", unsafe.Sizeof(i_32))
   fmt.Printf("The size of int64 is %d\n", unsafe.Sizeof(i_64))
   fmt.Printf("The size of uint is %d\n", unsafe.Sizeof(ui))
   fmt.Printf("The size of uint8 is %d\n", unsafe.Sizeof(ui_8))
   fmt.Printf("The size of uint16 is %d\n", unsafe.Sizeof(ui_16))
   fmt.Printf("The size of uint32 is %d\n", unsafe.Sizeof(ui_32))
   fmt.Printf("The size of uint64 is %d\n", unsafe.Sizeof(ui_64))
   fmt.Printf("The size of float32 is %d\n", unsafe.Sizeof(f_32))
   fmt.Printf("The size of float64 is %d\n", unsafe.Sizeof(f_64))
   fmt.Printf("The size of complex64 is %d\n", unsafe.Sizeof(c_64))
   fmt.Printf("The size of complex128 is %d\n", unsafe.Sizeof(c_128))
   fmt.Printf("The size of string is %d\n", len(s))
   fmt.Printf("The size of error is %d\n", unsafe.Sizeof(err))
   fmt.Printf("The size of byte is %d\n", unsafe.Sizeof(by))
   fmt.Printf("The size of rune is %d\n", unsafe.Sizeof(r))
}
```

You should see the following output:

```
The size of bool is 1
The size of int is 8    //As indicated here my OS and compiler are 64-bit
The size of int8 is 1
The size of int16 is 2
The size of int32 is 4
The size of int64 is 8
The size of uint is 8   //As indicated here my OS and compiler are 64-bit
The size of uint8 is 1
The size of uint16 is 2
The size of uint32 is 4
The size of uint64 is 8
The size of float32 is 4
The size of float64 is 8
The size of complex64 is 8
The size of complex128 is 16
The size of string is 12
The size of error is 16
The size of byte is 1
The size of rune is 4
```

Note: The units are all in bytes.

I used Sizeof method from the unsafe package. Sizeof returns the size in bytes of a hypothetical variable. If the variable is a reference, then Sizeof will only return the size of the reference, not the actual memory size occupied by the object. However, since we are dealing with primitive types here, the Sizeof should give exact result with the exception of string, in which it gives size of the reference.

In Go, string is a sequence of bytes. A string literal represents a UTF-8 sequence of bytes. In UTF-8, ASCII characters are single-byte corresponding to the first 128 Unicode characters. All other characters are between 1 to 4 bytes. [3]

```
package main
import "fmt"

func main(){
   var s string = "Cookie!"
   fmt.Println(len(s))

   s = "Cookie§"
   //§ is not the first 128 Unicode characters. It has 2 bytes
   fmt.Println(len(s))
}
```

Output is:

```
7
8
```

In the above example, each string literal in the word `Cookie` is 1 byte, making a total of 6 bytes. Here, `!` has 1 byte and `§` has 2 bytes, which is why the result is different. Additionally, the function len() does not return the length of the string, it actually returns the memory size of the string. [3] Lastly, all primitive types in Go do not grow to fit large values. If the value surpassed the maximum value, then the compiler will throw an overflow error and you will need a larger type to use that value (e.g., int32 or int64).

```
package main

import (
   "fmt"
   "unsafe"
)

func main(){
   var a int = 999999999999999
   var b int = 1
   //var c int32 = 837648293933 //ERROR, integer surpassed 32-bit limit
   fmt.Println(unsafe.Sizeof(a),unsafe.Sizeof(b))
}
```

Output: `8 8`

[1] https://golang.org/ref/spec#Types Go Official Specification: Types

[2] https://www.callicoder.com/golang-basic-types-operators-type-conversion/
Golang Basic Types, Operators and Type Conversion

[3] https://golangbyexample.com/length-of-string-golang/ Length of string in Go

3. Composite and constructed types

Go supports 8 different composites, constructed types as specified by the language specification [1]: array, slice, struct, pointer, function, interface, map, and channel types.

Array

An array is a numbered sequence of elements of a single type, called the element type. [1] The length of an array is never negative. All indexes should be non-negative. Array is a mapping in Go with integer as indexes. An example is shown below.

```
package main
import "fmt"

func main(){
   var myIntArray [5]int
   fmt.Println(myIntArray)
}
```

You should see the following output:

```
[0 0 0 0 0]
```

Slice

A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. The value of an uninitialized slice is nil. [1] Slice is mapping type.

```
package main
import "fmt"

func main(){
   slice1 := make([]int, 5)
   slice1[0] = 5
   slice2 := slice1[0:2]
   fmt.Println(slice1)
   fmt.Println(slice2)
}
```

You should see the following result:

```
[5 0 0 0 0]

[5 0]
```

Notice in the example that the slice only holds the reference unlike the array type.

Struct

A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). [1] The struct is the cartesian product type in Go. Each element in struct has a name and a type. Struct does not have methods inside the declaration, just like C.

```
package main
import "fmt"

type apple struct{
   weight int  //Assumed weight is in grams
   name string
}

func main(){
   myApple := apple{weight : 80, name: "Rose"}
   fmt.Printf("The weight is %d grams\n", myApple.weight)
   fmt.Println("The name is", myApple.name)
}
```

You should see the following output:

```
The weight is 80 grams

The name is Rose
```

Pointer

A pointer type denotes the set of all pointers to variables of a given type, called the base type of the pointer. The value of an uninitialized pointer is nil (NULL). [1] Pointers in Go allow programmers to pass references in the program. Pointers are reference types.

```
package main
import "fmt"

func printPointer(p *int){
   fmt.Println("Value is:", *p)
}

func main(){
   var i *int = new(int) //First one declares as a pointer
   *i = 42
   printPointer(i)

   var j int = 10 //Second one declares as an integer but pass the address
   printPointer(&j)
}
```

You should see the following output.

```
Value is: 42

Value is: 10
```

Function

A function type denotes the set of all functions with the same parameter and result types. The value of an uninitialized variable of function type is nil. [1] They are first-class object type but can also be mapping type as shown in the example below.

```
package main
import "fmt"

type mapFunc func(int , int) int

func main() {
   functions := []mapFunc{
      func(a int, b int) int {return a+b},
      func(a int, b int) int {return a-b},
   }

   fmt.Println(functions[0](25, 10))
   fmt.Println(functions[1](25, 10))
}
```

You should see the following result:

```
35

15
```

Interface

An interface type specifies a method set called its interface. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to implement the interface. The value of an uninitialized variable of interface type is nil. [1] The interface is Cartesian product type as struct must implement all methods specified by the interface in order to actually implement that interface type.

The following shows an example of interface.

```
package main
import "fmt"

//This interface has 2 methods: study and play
type semester interface{
   study()
   play()
}
```

```
//This struct will have 2 member data
type student struct{
   grade int
   happiness int
}

func (s *student) study(){
   s.grade += 5
   s.happiness -= 3
}

func (s *student) play(){
   s.grade -= 7
   s.happiness += 6
}

//This is a method for the interface semester, where semester s is parameter
func day(s semester){
   s.study()
   s.play()
}

//To use the student struct as argument, you need to pass the address
func main(){
   myStudent := student{grade: 10, happiness: 10}
   fmt.Printf("Before day: grade = %d and happiness = %d\n", myStudent.grade,
myStudent.happiness)
   day(&myStudent)
   fmt.Printf("After day: grade = %d and happiness = %d\n", myStudent.grade,
myStudent.happiness)
}
```

You should see the following output:

```
Before day: grade = 10 and happiness = 10

After day: grade = 8 and happiness = 13
```

Map

A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. The value of an uninitialized map is nil. [1] They are called dictionary in other languages. This is mapping type.

```
package main
import "fmt"

func main() {
   favoriteFood := make(map[string]string)
   favoriteFood["lawrence"] = "apple"
   favoriteFood["edward"] = "butter"
   favoriteFood["wittie"] = "sushi"
   fmt.Println(favoriteFood)
   fmt.Println(favoriteFood["wittie"])
}
```

You should see the following output:

```
map[edward:butter lawrence:apple wittie:sushi]

sushi
```

Channels

A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The value of an uninitialized channel is nil. [1] Channels are built-in pipes for concurrent entities to communicate with each other. Since Go is a concurrent language, it has many nice features in their language to support concurrency. Channel is a special feature of Go and cannot be matched to any formal definitions of constructors. The following example is from Go official document website.

```
package main
import "fmt"

func main() {
   messages := make(chan string)
   go func() { messages <- "Hello World" }() //"ping" assigned to messages
   msg := <-messages //messages assigned to msg by channels
   fmt.Println(msg)
}
```

You should see the following output:

```
Hello World
```

[1] https://golang.org/ref/spec The Go Programming Language Specification

4. User defined types

Similar to C, Go can make type aliases as user defined types. [1] Here is one example.

```
package main

import (
   "fmt"
   "reflect"
)

func main() {
   type foof = int64 //define a new type alias named foof
   var x foof = 314 //assign a value to the new type
   fmt.Println("Value is:",x) //print the value
   fmt.Println(reflect.TypeOf(x)) //print the type of the variable
}
```

You should see the following output:

```
Value is: 314
int64
```

As shown here, the reflect still prints the original type name instead of the alias type name.

[1] https://yourbasic.org/golang/type-alias/ Type alias explained

5. Static vs Dynamic typing

Go is statically typed language, where variables hold the type. [1] Dynamic typing is not possible. Type checking is performed at compile time. Variable hold the type. Here is that you can declare a variable without declaring a type. But once you assigned a value to that variable the type of that variable is confirmed and will not be modified any further.

```
package main

import "fmt"

func main() {
   var x int = 3

   fmt.Printf("Type: %T\n",x)
   //x = "Lawrence"    //ERROR at compile time
   fmt.Println(x)
}
```

When the error line is commented, you should see the following output:

```
Type: int
3
```

[1] https://golang.org/doc/faq#creating_a_new_language FAQ: Why did you create a new language?

6. Implicit vs Explicit typed

As indicated in the section above (static vs dynamic typing), Go can offer both implicitly typed and explicitly typed. Since Go is a strongly typed language, it will give variable a default type as Python does. E.g., The default value for an integer is 0 as shown in the previous example (question 5)

```
package main
import "fmt"

func main() {
   //implicit typed
   var x = 3

   //explicit typed
   var y int = 5
   z := 7  //same as var z = 7, this is implicit typed

   fmt.Println(x,y,z)
}
```

You should see the following output:

```
3 5 7
```

Here, the := is a short assignment statement for a variable declaration with implicit type. [2]

The following about default type by implicit typed is quoted from the Go language specification: *An untyped constant has a default type which is the type to which the constant is implicitly converted in contexts where a typed value is required, for instance, in a short variable declaration such as i := 0 where there is no explicit type. The default type of an untyped constant is bool, rune, int, float64, complex128 or string respectively, depending on whether it is a boolean, rune, integer, floating-point, complex, or string constant.*[1]

[1] https://golang.org/ref/spec#Constants Golang Official Specifications: Constants

[2] https://tour.golang.org/basics/10 Short variable declarations

7. Nominal vs Structural typing

Go uses structural typing. [1] The programmer does not need to explicitly define the type for interpretation. Instead, the Go compiler does compile-time checks to ensure the integrity of the program. Go only has structural typing.

```
package main

type Bird interface{
   fly()
}

type Eagle struct{ //Eagle does not implement the Bird interface
}

func (e Eagle) fly(){
}

type Lion struct{
}

func (l Lion) run(){
}

func makeBirdFly(b Bird){
   b.fly()
}

func main(){
   makeBirdFly(Eagle{}) //Okay
   makeBirdFly(Lion{})  //ERROR!
}
```

There is no output for the above code and will give an error at compile time on the second line inside the main function. This is because that Lion does not implement the fly method.

In the above example, the programmer doesn't need to specify that Eagle is a Bird. The language compiler interprets Eagle to be a Bird because it has a fly function, and Lion is not a Bird because Lion does not fly. This demonstrated that Go is a structural-typed language. [1]

[1] https://medium.com/higher-order-functions/duck-typing-vs-structural-typing-vs-nominal-typing-e0881860bf10 Duck Typing vs Structural Typing vs Nominal Typing

8. Typing strength

Go is a strongly typed language [2], where variables are bounded to specific data types, and will result in type errors if types do not match up as expected. [1]

```
package main
import "fmt"

func main(){
   var x string = "Hello World!" //x is a string
   y := &x + 10 //Try to pass the address of x but ERROR
   y = int(x) + 10  //Try to convert x to integer but still ERROR
   y = x + 10 //Try to add string with integers and still ERROR
}
```

The code example will raise error at compile time.

Above example will give an error that we cannot convert string to integer, demonstrating that a string type cannot be concatenated with an integer. Here, variables are bounded to a particular data type and you cannot convert the type easily or concatenate with another type.

[1] https://medium.com/android-news/magic-lies-here-statically-typed-vs-dynamically-typed-languages-d151c7f95e2b Magic lies here - Statically vs Dynamically Typed Languages

[2] https://golang.org/ref/spec#Introduction Go Introduction

9. Reflection

Go has computational reflection. One simple reflection is to get the type and size of a variable. To use reflection, Go can import a "reflect" package. [1] Go can also get the size of the variable using Sizeof from "unsafe" package. [2] However, if the variable is a reference, then Sizeof will only return the size of the reference, not the actual memory size occupied by the object.

```
package main

import (
   "fmt"
   "reflect"
   "unsafe"
)

func main() {
   var x int32 = 175
   fmt.Println("type:", reflect.TypeOf(x)) //Get the type of the variable
   fmt.Println("size:", unsafe.Sizeof(x)) //Get the size of the variable
}
```

You will get the following output:

```
type: int32
size: 4
```

The above example shows how Go can get the type and memory size of the variable.

Go has computational reflection on interface and struct as well. Here is an example forked from the previous example:

```
package main

import (
   "fmt"
   "reflect"
)

type Bird interface{
   fly()
}

type Eagle struct{
   weight float64     //weight is in pounds
   name string
}

func (e Eagle) fly(){
}
```

```
func main(){
   var b Bird
   b = Eagle{14.52,"Hawk"}
   fmt.Printf("(%+v, %+v)\n", reflect.ValueOf(b), reflect.TypeOf(b))
}
```

The output should be:

```
({weight:14.52 name:Hawk}, main.Eagle)
```

As shown from the example, the reflection is able to give the name of the member variables of Eagle, as well the name of the struct class.

Unfortunately, I believe that structural reflection is not possible in Go because compiled languages are unable to modify the computational state of the system (i.e., the class's definition and methods, or converts the value of a variable into executable code). As far as I know, only interpreted language can natively support structural reflection.

[1] https://golang.org/pkg/reflect/ Official Go Package: Reflect

[2] https://golang.org/pkg/unsafe/ Official Go Package: Unsafe

10.Static vs Dynamic scope

Go is statically, lexically scoped using blocks. [1] Dynamic scoping is not possible.

```
package main
import "fmt"

var a int //global scope

//foo scope
func foo(){
   a = 2
   fmt.Println("a in foo:", a)
}

//main scope
func main(){
   var a = 1
   fmt.Println("Main a:", a)
   foo()
   fmt.Println("a after foo call:", a)
}
```

You will get the following output:

```
Main a: 1
a in foo: 2
a after foo call: 1
```

[1] https://golang.org/ref/spec#Declarations_and_scope
Go Official Language Specification: Declarations and scope

11. Polymorphism (adhoc/overloading alternatives)

Here is an example of adhoc/ overloading polymorphism for operators in Go:

```
package main

import "fmt"

func main(){
   fmt.Println(3+5) //+ on integers
   fmt.Println(3.4+5.2) //+ on floats
   fmt.Println("334"+"552") //+ on strings
}
```

The output should be:

```
8
8.6
334552
```

Above shows that operators exist in different scopes, where "+" can be used for integers, floats, and strings. [1] Another example for adhoc polymorphism alternative can be demonstrated using interface:

```
package main

import (
   "fmt"
   "math"
)

type Circle interface {
   Radius() float64
}

type SmallCircle struct {
}

type MiddleCircle struct {
}

type BigCircle struct {
}

func (s SmallCircle) Radius() float64 {
   return 1.5
}

func (m MiddleCircle) Radius() float64 {
   return 7.5
}
```

```
func (b BigCircle) Radius() float64 {
   return 37.5
}

func Areas(circle Circle) float64 {
   r := circle.Radius()
   result := r * r * math.Pi  //math.Pi is a constant from math package
   return result
}

func main() {
   small := new(SmallCircle)
   middle := new(MiddleCircle)
   large := new(BigCircle)
   //Create a list of Circle objects
   circle := [...]Circle{small,middle,large}


   //Using loop to print the result
   for _, i := range circle {
      fmt.Println(Areas(i))
   }
}
```

The output should be:

```
7.0685834705770345
176.71458676442586
4417.864669110647
```

Note: Floating result may vary between machines and compilers.

The above example is an adhoc polymorphism alternatives using interface and structs. Unfortunately, Go does not support overloading for methods, functions, and operators without interfaces. [2] The example only shows a workaround with the overloading of methods but can still count as an adhoc polymorphism.

Go does not have inclusion polymorphism as it does not support a data type that can refer to a set of types and form a union type. Go variables cannot hold values of its subtypes or children. Go has inheritance that could make several subtypes from parent class but inheritance in Go is very different from others. Go does not have parametric polymorphism either because programmers cannot implement a function that can take all types of input. (Note: `fmt.Println()` is one of the built-in functions that shows parametric polymorphism because it can print variables with any type so that could be one)

[1] https://www.golangprograms.com/polymorphism-in-go-programming-language.html Polymorphism in Go Programming Language

[2] https://golang.org/doc/faq#overloading Go Official Language FAQ: Overloading

12.Strict vs non-strict evaluation

In general, Go does strict/eager evaluation but for operands like && and || it does a lazy evaluation. The following code examples is shown below: [1]

```
package main
import "fmt"

func main() {
   fmt.Println(hi(true,foof(1,2),bar(1,2)))
   fmt.Println(hi(false,foof(1,2),bar(1,2)))
}

func foof(x int, y int) int {
   fmt.Println("foof is running!")
   return x + y
}

func bar(x int, y int) int {
   fmt.Println("bar is running!")
   return 2 * x + 3 * y
}

func hi(isTrue bool, a int, b int) int {
   if isTrue {
      return a
   }
   return b
}
```

You should see the following output:

```
foof is running!
bar is running!
3
foof is running!
bar is running!
8
```

As shown here, before printing the result from hi it executes both foof and bar for both arguments that were passed into the function. This shows that the foof and bar is both evaluated even though the function only requires to return either the result from foof or bar. This shows that Go is using strict evaluation.

Go has lazy evaluation on operands such as && and ||. [1]

Go can also do non-strict evaluation by using higher-order-functions

```go
package main
import "fmt"

func main() {
   fmt.Println(hi(true,foof,bar,1,2))
   fmt.Println(hi(false,foof,bar,1,2))
}

func foof(x int, y int) int {
   fmt.Println("foof is running!")
   return x + y
}

func bar(x int, y int) int {
   fmt.Println("bar is running!")
   return 2 * x + 3 * y
}

//Higher order functions shown in func hi
func hi(isTrue bool, a, b func(i int, j int) int, i int, j int) int {
   if isTrue {
      return a(i,j)
   }
   return b(i,j)
}
```

You should see the following output:

```
foof is running!
3
bar is running!
8
```

As shown here, now Go will not execute the functions if that function will not be returned. Hence, you can do non-strict evaluation in Go using higher order function.

[1] https://deepu.tech/functional-programming-in-go/
7 Easy functional programming techniques in Go

13.The string types

In Go, string is a primitive type. [2] A string is in effect a read-only slice of bytes. [2] Strings are immutable unlike C or C++, Go string is not null-terminated. The runtime representation constructor of string is StringHeader as shown in the following example [1]:

```
type StringHeader struct {
   Data uintptr    //The data pointer of string
   Len  int        //length of the string
}
```

One operation for Go string is the + operator for concatenating two strings together, another one is to print the first/second/third/etc. letter of that string. They are shown in the example below.

```
package main
import "fmt"

func main(){
   var myString1 string = "hello"
   var myString2 string = " world"
   var result = myString1 + myString2
   //var errorString1 = myString1 * 2  // ERROR at compile
   //var errorString2 = myString1 * myString2 // ERROR at compile
   var index = result[0]  //result[0] = 'h'
   fmt.Println(result, index) //Go will print 'h' as ASCII code, which is 104
}
```

Result is:

```
hello world 104
```

Go also has a package called *strings* to provide additional operations for strings. Here is an example program showing some string operations the Go package can do. The example below only shows some operations like Index, Contains, Compare, LastIndex, and Count.

```
package main

import (
   "fmt"
   "strings"
)

func main(){
   var myString1 string = "hello"
   var myString2 string = " world"
   var result = myString1 + myString2

   fmt.Println(result)
```

```
   //For Index, If there are multiple matches, then Go will choose the one
with first appearance
   fmt.Println(strings.Index(result, "o"))

   //Whether a string contains a given substring.
   fmt.Println(strings.Contains(result, "b"))
   fmt.Println(strings.Contains(result, "w"))

   //Compare strings lexicographically. Equal = 0, left < right = -1, left >
right = 1.
   fmt.Println(strings.Compare(result, result))
   fmt.Println(strings.Compare(myString1, myString2))

   //This returns the index of the last instance of the given substring.
Return -1 if the substring is not found
   fmt.Println(strings.LastIndex(result, "o"))

   //Count the number of given substring occurrence in a given string
   fmt.Println(strings.Count(result, "l"))
   fmt.Println(strings.Count(result, "d"))
}
```

Result:

```
hello world
4
false
true
0
1
7
3
1
```

There are lots of operations provided by the *strings* package in Go, programmers can check out all the operations at https://golang.org/pkg/strings/

[1] https://golang.org/pkg/reflect/#StringHeader
Package Reflect: String Header

[2] https://golang.org/ref/spec#String_types
Go Official Language Specification: String types

14.Math operations in numbers

Besides standard math operations, Go has the following operations on bitwise: [1]

```
&    bitwise AND            integers
|    bitwise OR             integers
^    bitwise XOR            integers
&^   bit clear (AND NOT)    integers

<<   left shift             integer << unsigned integer
>>   right shift            integer >> unsigned integer
```

Here is an example of using these math operations on bitwise level.

```
package main
import "fmt"

func main() {
   var x = 10
   fmt.Println(x & 3)
   fmt.Println(x | 3)
   fmt.Println(x ^ 3)
   fmt.Println(x &^ 3)

   fmt.Println(x << 2)
   fmt.Println(x >> 2)
}
```

Result:

```
2
11
9
8
40
2
```

Because Go is influenced by C, it can directly access or manipulate the bits of an integer variable as shown in the example above. The bitwise operators are mostly same as in C.

[1] https://golang.org/ref/spec#Arithmetic_operators
Go Official Language Specification: Arithmetic operators

15.Multi-dimensional arrays

Go supports multi-dimensional arrays. The limits of the number of dimensions in a Go array is implementation specific. Theoretically, you can have arrays with any number of dimensions. (The limits here is not really specified by the official documentation website. I believe that it could depend on implementation specific regarding memory sizes, operating systems, and compilers) The example below shows the making of a 2-D array, how to access the fields, and that they can be jagged. [1]

```
package main
import "fmt"
func main(){
   myArray := make([][]int, 5)
   for i := range myArray {
      //Make each row a different length!
      myArray[i] = make([]int, i)
      for j := range myArray[i] {
         //Assign values
         myArray[i][j] = i * j
      }
   }
   fmt.Println(myArray)
}
```

You should get the following output:

```
[[] [0] [0 2] [0 3 6] [0 4 8 12]]
```

As shown above, a 2-D array in Go can be jagged and accessed using index.

[1] https://www.tutorialspoint.com/go/go_multi_dimensional_arrays.htm
Go - Multidimensional Arrays in Go

16.Dangling Else

Go does not have dangling else as it requires block in curly brackets. Each "if", "for", and "switch" statement is considered to be in its own implicit block. [1]

```
package main
import "fmt"

func main(){
   var a = 1
   var b = 2
   f(a,b)
   b = 1
   f(a,b) //This will not print anything
   a = 2
   f(a,b)
}

func f(a int, b int) {
   if a == 1 {
      if b == 2 {
         fmt.Println("Yes")
      }
   } else {
      fmt.Println("No")
   }
}
//func errorf(a int, b int) {
// if (a == 1) if (b==2) fmt.Println("Yes") else fmt.Println("No") //ERROR
won't compile
//}
```

The following will output:

```
Yes
No
```

[1]https://golang.org/ref/spec#Blocks
Go Official Language Specification: Blocks

[2]https://golang.org/ref/spec#If_statements
Go Official Language Specification: If statements

17.Coersion

As stated in question 8, Go is a strongly typed language. This shows that Go can only do explicit conversion and not implicit ones. With explicit conversion, Go can have possible lossy conversion. [1]

```
package main
import "fmt"

func main(){
   var i = 10
   fmt.Println("int created:",i)
   //var j float64 = i    //ERROR, won't work
   var j = float64(i)  //Explicit conversion will work
   fmt.Println("float created:",j)
   //var k rune = i      //ERROR, won't work
   fmt.Println("string created:",rune(i))
}
```

Output:

```
int created: 10
float created: 10
string created: 10
```

Here is a table of explicit type conversions for general primitive types:

from \ to	int	float (32/64)	uint	string	complex (64/128)	rune*	byte*	bool
int		Yes	Partial*	Partial*	No	Yes	Yes	No
float	Yes*		Partial*	No	No	Yes	Yes	No
uint	Yes	Yes		Partial*	No	Yes	Yes	No
string	No	No	No		No	No	No	No
complex	No	No	No	No		No	No	No
rune*	Yes	Yes	Partial*	Partial*	No		Yes	No
byte*	Yes	Yes	Yes	Partial*	No	Yes		No
bool	No	No	No	No	No	No	No	

*Conversion from int/uint/rune/byte to string interprets that value as a code point.
*Conversion from int/float/rune to uint will have unexpected result if the int/float/rune has a negative value.
*Conversion from float to int is a lossy conversion but it is allowed explicitly.
*Also, since rune and byte correspond to int32 and uint8, the value to be converted should be within the respective range of numbers.
Note: This table does not show error primitive type since it is a wrapper around the string type. (See question 2: primitive types)

[1] https://tour.golang.org/basics/13 Type conversions

18.Garbage Collecting

Go has a garbage collector. [1] Just like other languages, it tracks heap memory allocations, frees up allocations that are no longer needed, and keeps allocations that are still in-use. [2] The garbage collector in Go is a mark and sweep collector at runtime. [1] You can trigger a garbage collection by yourself by calling the function `runtime.FC()` to force one. [3] However, by doing so it will block the entire program. [3] Go does garbage collector automatically and programmers cannot manually deallocate memory. Go does not insert manual collection code during compile time. [2]

[1]https://golang.org/doc/faq#garbage_collection
Garbage Collection in Go FAQ

[2]https://docs.google.com/document/d/16Y4IsnNRCN43Mx0NZc5YXZLovrHvvLhK_h0KN8woTO4/edit
Go 1.4+ Garbage Collection Plan and Roadmap

[3]https://golang.org/pkg/runtime/#GC
Go runtime package: GC

19.Short circuit evaluation

Go uses short circuit evaluation. Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally. [1]

```
package main
import "fmt"

func foof() bool {
   fmt.Println("foof is called")
   return true
}

func bar() bool {
   fmt.Println("bar is called")
   return false
}

func main() {
   if bar() && foof() {
      fmt.Println("bar and foof")
   } else {
      fmt.Println("not foof and bar")
   }
   if foof() || bar() {
      fmt.Println("foof or bar")
   }
}
```

You should get the following result:

```
bar is called
not foof and bar
foof is called
foof or bar
```

In the above example, when doing bar and foof statement, it calls bar first and the program already knows that the statement will be false. Same as the bar or foof statement, it calls foof first and the program already knows that the statement will be true. This demonstrates short circuit evaluation.

[1] https://golang.org/ref/spec#Logical_operators
Logical operators in Go

20. BNF grammar

The Go language specification is written in BNF. Here are some examples. [1]

For loops:

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
```

If statements:

```
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .
```

Return statements:

```
ReturnStmt = "return" [ ExpressionList ] .
```

Array types:

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

[1] https://golang.org/ref/spec
The Go Programming Language Specification

21.Exceptions

Go does not have exceptions like Java. [1] However, Go has a predefined error interface type [2]:

```
type error interface {

    Error() string

}
```

Go does error handling by defining a primitive type called error (a very weird programming design decision). The following example is an error handling when opening a file: [2]

```
package main

import (
   "fmt"
   "log"
   "os"
)

func main() {
   file, err := os.Open("no_exist.txt")
   if err != nil {
      log.Fatal(err)
   } else {
      fmt.Println(file)
   }
}
```

If you do not have the file "no_exist.txt" in your working directory, you should get the following output:

```
2021/04/23 13:56:52 open no_exist.txt: no such file or directory
```

Note: The date and time printed here will be different.

For this example, the err variable will store whether if there is an error when opening the file. (In this case, it will be the file not found error)

As seen here, Go has no try and catch block like Python, just two variables getting some data from opening a file, and if the err variable does receive some data, then there is an error. Here, Go uses multi-value returns to try catch error. Error handling in Go have struct as equivalent to exceptions in Java.

You can also create custom errors like the following:

```go
package main

import (
   "errors"
   "fmt"
   "log"
)

type CustomError struct {
   Name string
}

func (e *CustomError) Error() error {   //error is a primitive type
   return errors.New(e.Name + " encountered an error") // strings based error
}

func Foo(x float64, y float64) {
   if x / y == 1.5 {
      customError := CustomError{Name: "The program"}
      log.Fatal(customError.Error())
   } else {
      fmt.Println("Result is",x/y)
   }
}

func main() {
   x := 3.0
   y := 5.0
   Foo(x,y)
   y = 2.0
   Foo(x,y)
}
```

Output:

```
Result is 0.6
2021/04/23 14:14:38 The program encountered an error
```

Note: The date and time printed here will be different.

[1] https://golang.org/doc/faq#exceptions
FAQ Exceptions

[2] http://blog.golang.org/error-handling-and-go
Error handling and Go

22. Pointers

There are pointers in Go just as C and C++. The uninitialized pointer is set to `nil` by default. (which is NULL symbol). You can access them by references but cannot manipulate them directly (you cannot do pointer arithmetic in Go) [1]

```
package main
import "fmt"

func Foo(p *int){
   *p = 10
}

func main(){
   var i = 5
   var p1 = &i
   var p2 = p1
   //p1++  //ERROR
   fmt.Println(*p2)
   Foo(p2)
   fmt.Println(*p2)
   fmt.Println(i)
}
```

Output:

```
5
10
10
```

[1] https://golang.org/ref/spec#Pointer_types
Go Pointer Types

23. Stack vs Heap

Go is special in terms of stack and heap usage: each variable exists as long as there are references to it. [1] If a variable has its address taken, that variable is a candidate for allocation on the heap. [1] However, a basic escape analysis recognizes some cases when such variables will not live past the return from the function and can reside on the stack. [1] To illustrate above, here is the program for testing.

```
package main
import "fmt"

type Foo struct{
   x int
   y float64
}

func main(){
   a := 5
   b := 3.14
   var foo = Foo{a,b}
   fmt.Println(foo.x)
   fmt.Println(foo.y)
}
```

To know where the variables are allocated, we can build this program by a special flag "-m":

```
$go run -gcflags -m test.go          //test.go is the example filename
```

Now the output will be:

```
./test.go:13:13: inlining call to fmt.Println
./test.go:14:13: inlining call to fmt.Println
./test.go:13:17: foo.x escapes to heap
./test.go:13:13: []interface {} literal does not escape
./test.go:14:17: foo.y escapes to heap
./test.go:14:13: []interface {} literal does not escape
<autogenerated>:1: .this does not escape
5
3.14
```

[1] https://golang.org/doc/faq#stack_or_heap
Stack or Heap FAQ

24. Inheritance

Go adopts a completely different way of object inheritance than C and Java. It uses a concept called "embedding" to solve inheritance in a way that looks like object composition. [1]

```
package main
import "fmt"

//A parent object
type Parent struct{
   name string
   i int
}

//A parent method
func (p Parent) Foo(){
   fmt.Println(p.name,p.i)
}

//A child object inheriting parent
type Child struct{
   Parent
}

func main(){
   var p = Parent{"Apple", 10}
   p.Foo()
   //Inherit parent by embeddings and composition
   var c = Child{Parent{"Butter", 5}}
   c.Foo()
}
```

Output:

```
Apple 10
Butter 5
```

Here, we embedded Parent class into Child struct as an anonymous field. Although it is similar to struct composition, we are using the parent method Foo from a subclass. Unfortunately, there is no way to override the parent methods as all the functions are non-virtual, but it is possible to shadow the method. Since there is no public, private and protected instances, Go will say that the child has all the same access control as the parent had. There is no public/private inheritance in Go, only the compositional inheritance. Not all objects in Go automatically inherit from one base class. In addition, Go supports multiple inheritance and forbids any overlapping/conflicting methods. [2] The example below shows multiple inheritance:

```
package main
import "fmt"
```

```
type ParentA struct{
   name string
   f float64
}

type ParentB struct{
   name string
   i int
}

//A parentA method
func (p ParentA) Bar(){
   fmt.Println(p.name,p.f)
}

//A parentB method
func (p ParentB) Foo(){
   fmt.Println(p.name,p.i)
}

func (p ParentA) h(){
}

func (p ParentB) h(){
}

//A child object that inherits from both ParentA and ParentB
type Child struct{
   ParentA
   ParentB
}

func main(){
   var p = ParentA{"Apple", 3.14}
   p.Bar()
   var p2 = ParentB{"Butter",10}
   p2.Foo()
   //Inherit parent by embeddings and composition
   var c = Child{p,p2}
   c.Bar()
   c.Foo()
   //c.h() //ERROR: ambiguous method selector
}
```

Output:

```
Apple 3.14
Butter 10
Apple 3.14
Butter 10
```

Here is another example showing the parent methods shadowing:

```
package main
import "fmt"

//A parent object
type Parent struct{
   name string
   i int
}

//A parent method
func (p Parent) Foo(){
   fmt.Println(p.name,p.i)
}

//A child shadow method
func (c Child) Foo(){
   fmt.Println("This is a child")
}

//Another parent method Bar
func (p Parent) Bar(){
   j := p.i + 20
   fmt.Println(p.name,j)
}

//A child object inheriting parent
type Child struct{
   Parent
}

func main(){
   var p = Parent{"Apple", 10}
   //Inherit parent by embeddings and composition
   var c = Child{p}
   c.Foo() //c will use the Child shadow method
   c.Bar() //c will still use the parent method
}
```

Output:

```
This is a child
Apple 30
```

[1] https://golang.org/ref/spec#Struct_types
Struct Types in Go

[2] https://golangtutorials.blogspot.com/2011/06/multiple-inheritance-in-go.html
Multiple Inheritance in Go

25. Regular Expression

The regular expression syntax is specified in https://golang.org/pkg/regexp/syntax/.

The regular expression in Go is built-in and close to Python. All the functionalities are supported by a built-in package called `regexp`. [1] Just like Python, programmers need to compile the regular expression before using. Here are the examples.

```
package main
import "fmt"
import "regexp"

func main() {
   //This is match
   var r, _ = regexp.Compile("a?[bce]+[^a]c")
   var stringList = [...]string{"abcabc","abababab","abbbbbbbbbbbc"}

   for _, s := range stringList {
      if r.MatchString(s){
         fmt.Printf("%s matches\n", s)
      }else{
         fmt.Printf("%s doesn't match\n", s)
      }
   }

   r2, _ := regexp.Compile("b.") //This section is search and replace
   fmt.Println(r2.FindAllString("bucknell", -1))
   fmt.Println(r2.FindAllString("babbles", -1))
   fmt.Println(r2.FindAllString("none", -1))
   fmt.Println(r2.ReplaceAllString("babbles","P")) //Replace and substitute

   r3, _ := regexp.Compile("a+?")
   fmt.Println(r3.FindAllString("aaaaaaa",-1)) //Checking greedy or non-greedy
}
```

Output:

```
abcabc doesn't match
abababab doesn't match
abbbbbbbbbbbc matches
[bu]
[ba bb]
[]
PPles
[a a a a a a a]
```

As shown from above, Go can match whole string, search the string and find inside, find all non-overlapping searches inside, and can do replace substitution. As shown in the last line, the search is non-greedy, where the FindAllString method in Go matches as few characters as possible (shortest possible match). [2]

Go also do capturing and grouping for regular expressions and substitute those groups with specific instructions. Here is an example of capturing, grouping and substitution.

```
package main

import (
   "fmt"
   "regexp"
)

//An example of grouping in Go and swap the first name and last name of a
Bucknell email address
//This example is adapted from Regular Expression Lab in CS308
func main() {
   r2, _ := regexp.Compile("([A-Za-z]+).([A-Za-z]+)@([A-Za-z0-9]+).edu")
   fmt.Println(r2.ReplaceAllString("hello.world@bucknell.edu","$2.$1@$3.edu"))
   fmt.Println(r2.ReplaceAllString("foo.bar@bucknell.edu","$2.$1@$3.edu"))
}
```

Output:

```
world.hello@bucknell.edu
bar.foo@bucknell.edu
```

As shown in the example, Go uses brackets for grouping just like Python. You can specify the groups using the dollar sign as shown.

[1] https://golang.org/pkg/regexp/syntax/
Go Package Syntax

[2] https://www.computerworld.com/article/2786107/regular-expression-tutorial-part-5--greedy-and-non-greedy-quantification.html
Regular Expression Tutorial Part 5: Greedy and Non-Greedy Quantification

26. Anonymous functions

Go supports anonymous functions. [1] Programmers can call them on an argument without storing them in a named variable, or store them in a variable, or pass them to other functions as uncalled parameters. [2] [3] You can also pass them to other functions as uncalled parameters [2] [3]

```
package main
import "fmt"

//Anonymous function stored in a variable
var multiply = func(a int, b int) int {
      return a * b
   }

//Anonymous as first-class
func mul(x int) func(int) int {
   return func(y int) int {
      return x * y
   }
}

func main() {
   //Call function on an argument without storing as a named variable
   fmt.Println(
      func(i int, j int) int {
         return i + j
      }(7, 5))

   //Call function with a named variable
   fmt.Println(multiply(8,9))

   //Call function as first-class
   var mul2 = mul(2)
   fmt.Println(mul2(3))
}
```

Output:

```
12
72
6
```

[1] https://blog.golang.org/first-class-functions-in-go-and-new-go First Class Functions in Go

[2] https://www.golangprograms.com/anonymous-functions-in-golang.html
Anonymous Functions in Golang

[3] https://yourbasic.org/golang/anonymous-function-literal-lambda-closure/
Anonymous functions and closures

27. Parameter passing (pass by value)

Go is a strictly evaluated language, and pass everything by value. [1] In Go, a function always gets a copy of the thing being passed. For instance, passing an int value to a function makes a copy of the int, and passing a pointer value makes a copy of the pointer, but not the data it points to. The following is extracted from the official Go FAQ:

In Go, map and slice values behave like pointers: they are descriptors that contain pointers to the underlying map or slice data. Copying a map or slice value doesn't copy the data it points to. Copying an interface value makes a copy of the thing stored in the interface value. If the interface value holds a struct, copying the interface value makes a copy of the struct. If the interface value holds a pointer, copying the interface value makes a copy of the pointer, but again not the data it points to. [1]

The following example shows how Go pass things by value and to fake pass by reference (pass in the address as an alternative way for pass by reference):

```
import "fmt"

func Foo(x int){
   x = 10
}

func bar(x *int){
   *x = 10
}

func main(){
   var i = 5
   fmt.Println("main:", i)
   Foo(i)
   fmt.Println("after foo:", i)
   //This is similar to C: pass the address to fake pass by reference
   bar(&i)
   fmt.Println("after bar:", i)
}
```

Output:

```
main: 5
after foo: 5
after bar: 10
```

[1] https://golang.org/doc/faq#pass_by_value
FAQ: Pass by value in Go

28. Higher order functions

Go supports higher order functions, which is a function that receives a function as an argument or returns the function as output. Here is an example of higher order functions derived from question 26, anonymous function. [1]

```
package main
import "fmt"

func mul(x, y int) int {
   return x * y
}

func partialMul(x int) func(int) int {
   return func(y int) int {
      return mul(x, y)
   }
}

func main() {
   partial := partialMul(5)
   fmt.Println(partial(6))
}
```

Output:

```
30
```

[1] https://www.golangprograms.com/higher-order-functions-in-golang.html
Higher order functions in Golang

29. Interfaces

This is also one of the composite and constructed types in Go (See question 3). Although Go is a half object-oriented language, it supports interfaces as described below:

An interface type specifies a method set called its interface. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to implement the interface. The value of uninitialized variable of interface type is nil. [1]

```
package main

import (
   "fmt"
   "math"
)

type Shape interface{
   area() float64
}

type Rectangle struct{
   width float64
   length float64
}

type Circle struct{
   radius float64
}

type Triangle struct{
   width float64
   height float64
}

func (r Rectangle) area() float64 {
   return r.width * r.length
}

func (c Circle) area() float64 {
   return math.Pi * c.radius * c.radius
}

func (t Triangle) area() float64 {
   return t.width * t.height / 2
}

func calculate(s Shape) {
   fmt.Println(s.area())
}
```

```
func main() {
   x := Rectangle{4.5, 9.3}
   y := Circle{6.7}
   z := Triangle{2.5, 4.0}
   calculate(x)
   calculate(y)
   calculate(z)
}
```

Output:

```
41.85
141.02609421964584
5
```

The above example uses an interface that implements the area method, where each struct implements the interface.

Note: The actual floating number result may not be exact.

[1] https://golang.org/ref/spec#Interface_types
Go Interface Types

30. Explicit vs Implicit Declared Variables

Go can do either way. Here is a table:

Go variables	Explicit type	Implicit type
Explicit declaration	`var a int = 5`	`var a = 5`
Implicit declaration	`a := int(5)`	`a := 5`

Inside a function, the := short assignment statement can be used in place of a var declaration with implicit type or declaration. [1]

```
package main

import "fmt"

func main(){
   var a int = 5
   var b = 10
   c := int(3)
   d := 6
   fmt.Println(a,b,c,d)
}
```

Output:

```
5 10 3 6
```

[1] https://tour.golang.org/basics/10
Short variable declaration in Go

31.Concurrency, channels, and pipelines

Go is designed to be a concurrent programming language. The prominent feature of Go is the support of pipelines, which are a series of stages connected by channels, where each stage is a group of goroutines running the same function [1]. A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. [2] Here is an example of ping channels by concurrency:

```
package main

import (
   "fmt"
   "log"
   "time"
)

//See here we use channel keyword (chan), indicating that the string is a
channel
func pingProcessor(c chan string) {
   //An infinite loop
   for {
      c <- "hello world"
   }
}

//Print the message
func print_msg(c chan string) {
   //An infinite loop
   for {
      msg := <- c
      fmt.Println(msg)
      time.Sleep(time.Millisecond * 1000) //Time sleep for 1 second
   }
}

func main() {
   var c = make(chan string)

   go pingProcessor(c)
   go print_msg(c)

   var input string //Declare an input variable
   //Channels Error handling. We scan in the message
   _, err := fmt.Scanln(&input)
   if err != nil {
      log.Fatal(fmt.Errorf("error occured when scan in"))
   }
}
```

The output will print "`hello world`" forever until ENTER is pressed.

[1] https://blog.golang.org/pipelines
Go Concurrency Patterns: Pipelines and cancellation

[2] https://golang.org/ref/spec#Channel_types
Channel Types in Go

Additional References:

Cover Page Picture from:

https://medium.com/javarevisited/7-online-courses-to-learn-golang-or-go-programming-languages-in-2020-f599a25cf14a
Top 10 Online Courses to learn Golang/Go in 2021 — Best of Lot

Bibliography:

Book Title: **The Way to Go: A Thorough Introduction to the Go Programming Language**
Author: Ivo Balbaert
Publisher: iUniverse, Inc. Bloomington
ISBN: 978-1-4697-6916-5
Year: 2012

https://kuree.gitbooks.io/the-go-programming-language-report/content/
The Go Programming Language Report