Theory Questions

The goal of the questions below is to assess your critical thinking and being able to communicate them clearly. All the questions are open-ended. You are not expected to write whole essays for each question. However, by leaving them open, we can guarantee that students demonstrate critical thinking, or at the very least, do some basic research, before answering.

1. **What is the relationship between an array and a slice in Go? Why do Go developers say that they “use slices instead of arrays most of the time?”**

Arrays are an important building block of any programming language that allows us to represent and manipulate collections of homogenous data. In Go, we work rarely directly with arrays, instead, we most often use slices. This is because the difference between them.

From one hand arrays are value types that have a predefined length. An array gets created in memory and once its length has been set, it cannot be changed. The value type means that every assignment of passing of an actual array as a function argument will result in the copying of the value of the array and will not affect the original one.

From other hand slices are dynamically resizable collection, so we do not need specify the length of the array from the start. And also the most important distinctions between arrays and slices is that slices are reference types, so changing will affect the original one.

In slices and arrays trying to access an item outside of the bounds will result in a panic. In slices we cannot directly assign values to slice elements outside the boundaries, we can append (add) new elements to a nil-slice. By default, append will allocate a backing array with a capacity of 1. Once his capacity has been outgrown, it will have to copy the array into a new one (with twice the capacity), and point the slice to it. So that is way slices can self-increase his size.

1. **In one of his conference talks, Rob Pike talks about “making the zero-value useful.” Find that talk and describe what he means in your own words.**

According to Rob Pike talks, when memory is allocated to store a value, either through a declaration or a call of make or new, and no explicit initialization is provided, in Go the memory is given a default initialization. This property of always setting a value to a known default is important for safety and correctness of program, but can also make your Go programs simpler and more compact. Af course you can use constructor instesad.

Each element of such a value is set to the zero value for its type:  boolean - false , integers - 0,  floats – 0.0, strings - "" , and  for pointers, functions, interfaces, slices, channels, and maps - nil . This initialization is done recursively, so for instance each element of an array of structs will have its fields zeroed if no value is specified.

He gives an exmple using sync.Mutex, which is designed to be usable without explicit initialization. The sync.Mutex contains [two unexported integer fields](http://golang.org/src/pkg/sync/mutex.go?s=697:745#L8). Thanks to the zerovaluethose fields will be set to will be set to 0 whenever a sync.Mutex is declared.

1. **What is one unique characteristic of Go’s interfaces? How do they differ from, interfaces in languages like Java or C#?**

As in many other programming languages interfaces it’s a contract and defines behavior. Go language interfaces are different from other languages. First of all the interface is a custom type that is used to specify a set of one or more method signatures. Second the interface is abstract, so we are not allowed to create an instance of the interface. But we are allowed to create a variable of an interface type and this variable can be assigned with a concrete type value that has the methods the interface requires. Or in other words, the interface is a collection of methods as well as it is a custom type. So we just declare the interface, and then any type that happens to have those methods can be used anywhere that interface is required.

Unlike Java, an implementing type does not need to explicitlysay that it is implementing an interface. Any type in Go automatically matches any interface in any package, as long as it has methods that conform to the interface contract (have the same method signature). This allows us decouple code from one another and connect thing together in main package.

Another unique characteristic of interfaces is that, interfaces are pointers themselves. So we you never need to use a pointer to an interface type.

Instead of inheritance, Go uses interfaces to achieve polymorphismthrough what is called structural subtyping. Interfaces are the most basic way of making code  
generic, allowing others to reuse it, by providing their own implementation of certain functionality.

The empty interface (interface{}) is an interface implicitly implemented by every type in Go, like object in C#. In other words, interface{} is a placeholder for situations where we have no idea what type of data we are working with.

1. **The creators of Go decided to make variable pointers *explicit*. Other languages, such as Java and C#, have decide to make them *implicit*, abstracting them away from the programmer. What do you think is one advantage of having to deal with pointers explicitly?**

Having explicit pointers gives us more control over allocation. So Go is a Garbage Collector(GC) language, and there are no guarantees of whether allocation will occur on the stack or the heap. But generally, non-pointer values will be stack-allocated. This reduces pressure on the GC, since the values are just discarded along with the stack frame on return.

For this reason, it's usually considered good form to pass values to functions instead of pointers, even if the values are larger. This allows the compiler to prove through escape analysis that the arguments don't need to be heap-allocated. In some cases pointers can be proven not to escape, but in a limited set of circumstances.

Having explicit pointers also resolves type system awardness between simple and complex data types. Consider Java, which is also a pass-by-value language, but has "reference" types, except for int, float, etc. So everything is a reference except this bag of simple data types, effectively creating a 2-tiered type system. Java has dealt with this by implementing boxing and unboxing rules for int/Integer and friends, which is fine, but adds complexity.

1. **In which situations would you use a pointer to a variable, instead of a copy of its value?**

The general consensus is to usevalues for as long as you can, but are four major use cases in programming using pointers:  
1. Sharing access to the same value, so multiple parts of the program can read it, or modify it.

2. Saving the effort of copying large values of data structures (large string, struct, or an array, storing millions of bytes of information)all the time. By simply creating a pointer to the value and using the pointer, instead of copy all those bytes around, every time you want to work with it.

3. Express optionality - telling the consumer of the value, that it may not exist (nil).

4. Allow the creation of recursive and graph-style data structures (e.g. Parent->Child->Parent)

1. **What is the one unique property of unbuffered channels? In simpler terms, what makes channels so interesting?**

A channel gives us a way to “connect” the different concurrent parts of our program. In this case, we can represent this connection between our two concurrent blocks of code. Channels can be thought of as “pipes” or “arteries” that connect the different concurrent parts of our code.

I think that the major things that describes unbuffered channels is combine communication—the exchange of a value—with synchronization—guaranteeing that two calculations (goroutines) are in a known state.

Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

Channels can be directional - which means that you can restrict a channel to only send or receive data. This is specified by the arrow (<-) accompanied with the channel declaration

Statements that send or receive values from channels are blocking inside their own goroutine. This means:

* A statement to receive data from a channel will block until some data is received
* A statement to send data to a channel will wait until the sent data has been received

For example, when we try to print the value received (in the main function) :

fmt.Println(<-out)

The <-out statement will block the code *until* some data is received on the out channel. It helps to then visualize this by splitting the main block into two parts : the part that runs until its time to wait for the channel to receive data, and the part that is run after.

1. **Can you think of a situation where you would use a buffered channel?**

In the previous questition, we saw that channel statements block until data is either sent into or received from a channel. This happens because a channel doesn’t have anywhere to store the data going into it, and so needs to wait for a statement to receive data.

**A buffered channel** is a type of channel that has storage capacity within it. To create a buffered channel, we add a second argument to the make statement to specify capacity:

out := make(chan int, 3)

Now out is a buffered channel with a capacity of three integer variables. This means that it can intake upto three values before it blocks:

Buffered channels are used in cases where we don’t want the channel statement to block if there are no available receivers. Adding a buffer allows us to wait for some of the receivers to get freed without blocking the sending code.

But buffered channels never have unlimited buffers. Proper use of a buffered channel means that you must handle the case where the buffer is full and your writing goroutine blocks waiting for a reading goroutine. Buffered channels are useful when you know how many goroutines you have launched, want to limit the number of goroutines you will launch, or want to limit the amount of work that is queued up.They are also helpful for managing the amount of work a system has queued up, preventing your services from falling behind and becoming overwhelmed.

1. **Briefly describe struct embedding. Why do we say that it is different from inheritance (as in Java/C#)?**

Go doesn't support inheritance in the classical sense instead, it encourages composition as a way to extend the functionality of types. Embedding is an important Go feature making composition more convenient and useful.

There are different three kinds of embedding Go supports:

1. Structs in structs
2. Interfaces in interfaces
3. Interfaces in structs

### Inheritance

Inheritance defines an is-a relationship between two entities, where one entity is the base and the other is derived. A derived entity inherits properties and behavior from the base entity.

## Golang embedding

Inheritance, when used incorrectly, can lead to less flexible and highly coupled designs. It is recommended to use association over the inheritance.

In say you should favor object composition over class inheritance. There is a lot of debate and conflict on whether Golang is truly an object-oriented language, because it does not support inheritance, although polymorphism can be achieved through interfaces.

1. **Could you explain the difference between errors and panics?**

A typical program can have two types of errors. The first ones are those anticipated by the programmers. We can handle them using the error interface. The second type of errors are those that programmers may not have anticipated. In other words, they are unforeseen errors.

A panic is an unforeseen error. It can lead to abnormal termination of the program. Panics are a usually a result of common programming mistakes that might creep in while development. However, the important point is not why panics occur in a program but how we can handle them before they can terminate our program unexpectedly.

There are some operations that automatically return a panic and stop the program from further execution. For example, accessing an array index that is out-of-bounds, calling methods on nil pointer and so on. As you can see, these are mostly a result of programming mistake. Therefore, a compiler cannot detect them during compilation step.

A panic contains two important points. First is a message containing the cause of the panic. Second is a stack-trace that helps us identify where the panic occurred. Together, these pieces of information can help a developer find the error and fix it.

Panics have only one recovery mechanism and that is by using the recover() function. Basically, this function allows us to intercept a panic on its way up the call stack.

From other hand when something occurs which is not supported by any means then an error occurs. Errors help to write clean code that increases the maintainability of the program.

An error is a well developed abstract concept which occurs when an exception happens. That is whenever something unexpected happens an error is thrown.

Tere are multiple methods for creating errors:

New function- can be used to create errors

Errorf function - the fmt package has an Errorf() method that allows formatted errors

To check for an error we simply get the second value of the function and then check the value with the nil.

What the Go language calls errors are what in most other languages goes for a side  
effect. Side effects are secondary, usually unexpected and potentially undesirable effects of  
the execution of some code. For example:  
— The file we try to read from isn't there  
— The DB cannot find the record we need  
— Someone called us with invalid input.  
— The API we try to call is down, or extremely slow

1. **Please describe a situation where you would use a mutex, instead of a channel. What are mutexes good for?**

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine. Channels are best suited in cases like**passing ownership of data, distributing units of work, and communicating async results.**

The Mutex (mutual exclusion lock) is an invaluable resource when synchronizing state across multiple goroutines.

Channels are high level concept in Go which somewhere inside use Mutex only. Go channels are appealing because they provide built-in thread safety and encourage single-threaded access to shared critical resources. Channels, however, incur a performance penalty compared to mutexes. Using mutexes is helpful when you just need locks over a few shared resources. A memory-based race condition occurs if two or more threads (at least one of which performs a write operation) have uncoordinated access to the same memory location.

We have used mutex.Lock() and mutex.Unlock() to create a synchronous lock over a shared resource. And to manage multiple logs we create a goroutine every time and added them to sync.WaitGroup. These are an important synchronization primitive.

1. **I have a function with one parameter. What should I do to make that function accept arguments from different possible types?**

The furst approach is to add as many as needed functions with different type of parameters. So this is the more difficult approach.

Another one is till GO 1.18 to use something else as the language does not support generics that allow us to define variables whose type can be one of several. There are situations where a developer would like to define just one function, since the task at hand would be the same across several different types.

You can use interface types as arguments, in which case you can call the function with any type that implements the given interface. In Go types automatically implement any interfaces if they have the interface's methods. So if you want to accept all possible types, you can use empty interface (interface{}) since all types implement that. No other modification needs to be done to your function.

However, **any** type is not the best… the compiler cannot help you check and validate your code. So there are two aapproaches:

-use type

func print\_out\_type(x interface{}) string {

switch v := x.(type) {

case string:

return "A string"

case int32:

return "An Integer"

default:

return "A default"

}

}

You can also use the reflect package to study the type of an interface variable

func print\_out\_type(x interface{}) string {

return reflect.TypeOf(x).String()

}

func main() {

fmt.Println(print\_out\_type(42))

fmt.Println(print\_out\_type("foo"))

}

After Go 1.18:

You’ll add a single generic function that can receive a variable containing either integer or float. To support values of either type, that single function will need a way to declare what types it supports. Calling code, on the other hand, will need a way to specify whether it is calling with an integer or float variable.

To support this, you’ll write a function that declares typeparameters in addition to its ordinary function parameters. These type parameters make the function generic, enabling it to work with arguments of different types. You’ll call the function with typearguments and ordinary function arguments.

func newGenericFunc[age int64 | float64](myAge age) {

fmt.Println(myAge)

}

func main() {

fmt.Println("Go Generics Tutorial")

var testAge int64 = 23

var testAge2 float64 = 24.5

newGenericFunc(testAge)

newGenericFunc(testAge2)

}

1. **Describe an example, where you would have a function take a reference to another one as one of its parameters. You can use code alongside your description if it helps you illustrate your example better.**

Functions as TypesFunction signatures can serve as types, for convenience reasons. One often most widely used example in Go is the http package’s:

HandlerFunctype HandlerFunc func(ResponseWriter, \*Request)

Having that type written down helps with reducing the possible errors when copying the function declaration.