**Modules**

A module is a collection of packages stored in a file tree under $GOPATH/pkg folder with a go.mod file at its root. This file defines the module's path which is also the import path used for the root directory and its dependency requirements.

The Go command automatically checks and adds dependencies required for imports provided the current directory or the parent directory has a go.mod fie.

A Go module will have several Go files or packages in addition to two important files in the root, the go.mod file and go.sum file. These files are maintained by the Go tool, and it is used to track the module's configuration.

Before creating a module, you need to identify a directory where the module will reside. This directory can be anywhere on the computer and need not be in any specific Go directory. You can use an existing directory or create a new one.

**go mod init MyModule**

This will create the go.mod file under the MyModule folder.

The newly created go.mod file will have the module name and the go version which the module is targeting. This file will expand as more information is added to the module.

Now, you can start adding files to the newly created module. First, create the **main.go** file to run the go module. The main.go file is the starting point of a go program. The name of the file is not important (can have any name) but the **main()** function within this file is the entry point for the program. So having the file name as **main.go** makes it easier to find the starting point.

A specific version of the Go module can be used as Go modules are distributed from a version control repository and they can use version control features like tags, branches, and commits. You can specify the version of the module that will be used in the dependency with **@** symbol at the end of the module path as shown below.

**go get sample.com/sales@latest**

The function names of the inside any file in the go module should be starting with capital letter.

For more info: <https://go.dev/blog/using-go-modules>

**Bits, Bytes, and Byte Slices**

A bit references a single binary value, either a 1 or a 0. For a computer to process information the information must be written in binary. If all we could send to a computer would be a single 1 or a 0, we wouldn’t be as far as we are today with computers. Luckily, bits can be added together to reference much larger numbers than 1 and 0.

Bits can be added together, and their values are equal to 2 to the power of the place of the bit (from right to left). When the bit is 1, the bit value is added to the binary value. When the bit is 0, the value is not added. The binary value consisting of 8 bits equal to00000001 would equal 1 (2 to the 0th power = 1), the value 00000010 would equal 2 (2 to the 1st power), and the value 00000100 would equal 4 (2 to the 2nd power). Once you start combining these, you start to see how they can soon represent a large number. The value of 00000011 is 3 (2 to the 0th power is 1 + 2 to the 1st power is 2 = 3), the value of 10010010 is 146 (2 to the 2nd power + 2 to the 4th power + 2 to the 7th power) and 11111111 is 256.

**Unicode and UTF-8**

Before diving into byte slices, we need to briefly visit encoding. [Unicode](https://home.unicode.org/) is “the universal character encoding”. Unicode supports over 137,000 characters and 150 different languages. Each character has a specific Unicode code point that represents the character. For example, the Unicode code point for the capital letter “T” is U+0054*,*the Unicode code point for the lowercase letter “t” is U+0074*,*and the Unicode point for the lower left triangle “◺” is U+25FA*.*

The “problem” with Unicode is that it is most often represented in software as an int32 (32-bit integer). Most characters in widespread use could fit into a much smaller number than required for a 32-bit data type. This is where UTF-8 enters the picture.

UTF-8 is a variable length encoding of Unicode code points. For each Unicode code point, it uses between 1 and 4 bytes. All the most common characters can be represented using 1–2 bytes (all ASCII characters can be represented in 1 byte). UTF-8 allows us to use all the characters defined by Unicode but allows us to save some extra space and only reach for the 3rd or 4th byte when we really need it. To again list the same examples, we could represent the Unicode code point “T” as 84, “t” as and “◺” as 226 151 186*.*

This is a very brief overview of encoding, and I would highly recommend reading the article [The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets](https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/).

**Byte Slices**

Byte slices are a list of bytes that represent UTF-8 encodings of Unicode code points. Taking the information from above, we could create a byte slice that represents the word “Go”:

**bs := []byte{71, 111}**

**fmt.Printf("%s", bs) // Output: Go**

You may notice the %s used here. This converts the byte slice to a string. Strings are *literally* made up of arrays of bytes. This makes converting strings to byte slices and byte slices to strings very easy and convenient. %d is also often used with byte slices and prints the UTF-8 decimal value of each byte.

**s := "Wow look at me"**

**bs := []byte(s)**

**fmt.Printf("%s", bs) // Output: Wow look at me**

**fmt.Printf("%d", bs) // Output: [87 111 119 32 108 111 111 107 32 97 116 32 109 101]**

To view something a little more interesting, here is an example of a byte slice representing a non-ASCII value:

**bs := []byte("◺")**

**fmt.Println(bs) // Output: [226 151 186]**

**s := string(bs)**

**fmt.Println(len(s)) // Output: 3**

The length of this string may seem confusing at first, but remember a string is made up of a byte slice array. The length of this string is 3 because the UTF-8 value of “◺” is 226 151 186, which means that UTF-8 uses 3 bytes to represent the Unicode code point. To get the Unicode code point (or rune) count in a string, we can use the utf8 package in the standard library:

**import (**

**"fmt"**

**"unicode/utf8**

**)**

**bs := []byte("◺")**

**s := string(bs)**

**fmt.Println(utf8.RuneCountInString(s)) // Output: 1**

This is important to remember if Unicode character count is important to your software. Luckily, we do not need to handle this within range, because Go will implicitly loop over strings by their Unicode code points. The only catch here is that the index will still be incremented by the string’s byte count. Notice how i in the below example jumps from 3 to 6 after printing the triangle:

**func main() {**

**for i, b := range "Hi ◺ there" {**

**fmt.Printf("i: %d. b: %q\n", i, b)**

**}**

**}**

**// i: 0. b: 'H'**

**// i: 1. b: 'i'**

**// i: 2. b: ' '**

**// i: 3. b: '◺'**

**// i: 6. b: ' '**

**// i: 7. b: 't'**

**// i: 8. b: 'h'**

**// i: 9. b: 'e'**

**// i: 10. b: 'r'**

**// i: 11. b: 'e'**

Using %c here prints the character represented by the corresponding Unicode code point. Since each of the characters in the strings of the rune type (Unicode code point) we cannot use %s to print here.

Going back to the example I gave earlier: string(42) // Output: \* . This may start to be making some sense. The UTF-8 decimal value for \* is 42. So when we pass the integer 42 into a string, it is creating a byte slice containing 42, which is equal to \*.

**Byte Slices vs Strings**

So now that we know about byte slices, when should we use them over strings? The main difference in strings and byte slices in Go is the way they are handled in memory. Strings are *immutable,*meaning they cannot be changed within memory. So, every time you add or remove from a a string, Go is creating a brand-new string in memory. On the contrary, byte slices are *mutable*. So, whenever you are adding to a byte slice, you are not creating a new object in memory. Depending on the circumstance, this can effect application speed.

Hopefully this helps “demystify” some of these concepts. There are quite a large number of additional resources, and this post has only scratched the surface of the topic,

Some additional resources

* [The Go Programming Lanuage by Alan A. A. Donovan and Brian W. Kernighan](https://www.amazon.com/Programming-Language-Addison-Wesley-Professional-Computing/dp/0134190440/ref=sr_1_2?crid=2H8RLE0WGE9BF&keywords=the+go+programming+language&qid=1579457325&sprefix=the+go+programming+lan%2Caps%2C287&sr=8-2) (specifically chapter 2 section 3.5)
* [The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)](https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/)
* [Unicode Website](https://home.unicode.org/)
* [UTF-8 Character Set](http://www.fileformat.info/info/charset/UTF-8/list.htm) (note: the decimals are listed in their hexadecimal value. You can print the hexadecimal value of a byte slice by using % x when printing)

**type conversions casting type assertions.**

Interfaces in Go provide a way to specify the behavior of an object: if something can do this, then it can be used here.

Interfaces are a big deal in Go. If a variable’s type is that of an interface, then you can be confident that the object referenced by the variable implements the methods prescribed by the interface.

**type Stringer interface {**

**String() string**

**}**

**var x Stringer**

**The only guarantee an interface provides is that the object it points to will implement its prescribed methods. Nothing more.**

That is also the reason why the empty interface (**interface{})** is almost useless because it doesn’t guarantee anything! Its primary usefulness is only due to the lack of Generics.

**Type Assertions**

Although the variable is an interface, it will still reference an object that contains fields and other methods (exported or unexported).

To access those fields and methods, you need to **type assert**. Type assertion basically proclaims that the object is *something* else (either another interface or struct).

type A struct {

name string

}

// A implements Stringer interface

func (a A) String() string {

return "Hello"

}

func main() {

var x Stringer

x = A{name: "sam"}

fmt.Println(x.String()) // Output: Hello

fmt.Println(x.(A).name) // Output: sam

}

You can type assert by using this syntax: x.(A) , where x is the (interface) variable and (A) is the type you are proclaiming x to *really being.*

fmt.Println(x.name)

// Error: x.name undefined (type Stringer has no field or method name)

If you try and access name without type asserting, then you will get a compile-time error.

Of course, if you type assert incorrectly (assert that x is something it’s not), the application will crash with a run-time panic. To avoid this, you can use a [type switch](https://tour.golang.org/methods/16) or the [comma, ok idiom](https://golang.org/doc/effective_go.html#interface_conversions).

**Type Conversions**

In Go, all variable types are distinct from one another — even if behind the scenes, they are stored with exactly the same structure in memory, or are aliases of each other.

This means an int type is distinct from an int64 type, even though on a 64-bit machine they are stored in memory the same way.

Type conversion is required when you need to convert one variable type into another, usually because a particular variable or function argument requires it. If you want to convert an int64 to an int, the syntax is: x := int(y), where y is an int64 and x is an int.

Type conversion will create a *copy* of the original data, so it’s not necessarily a “free” operation.

Go’s type conversion will do it’s best to maintain the same value in the new data type *if possible*.**To do that, it may transform the underlying bit structure.**

It can also fail to convert accurately on some occasions: converting from a larger to smaller data type, or from a signed to an unsigned, or from a large int64 to a float64 are common culprits.

**Casting is seldom used in Go.**  Even advanced developers will rarely (if ever) see explicit casting syntax in their code.

When coupled with the unsafe package, there is one elegant\* use-case of casting. It is potentially dangerous, and there are safer ways to achieve the same objective. It’s when you want to convert between 2 different types of structs which have precisely the same underlying data structure.

Type Conversion doesn’t work for this scenario, but this does:

import "unsafe"

type Z struct {

A int

B string

}

type Y struct {

C int

D string

}

func main() {

z := Z{A: 1, B: "sam"}

y := \*(\*Y)(unsafe.Pointer(&z))

}

Attempting to *convert* z (of type Z) to type Y will not work with the conversion syntax y := Y(z). In this example, casting is required to bypass Go’s type safety checks.

**Generics**

Generics allow our functions or data structures to take in several types that are defined in their generic form.  
To truly understand what this means, let's take a look at a very simple case.  
Let's say you need to make a function that takes one slice and prints it. Then you might write this type of function:

**func Print(s []string) {**

**for \_, v := range s {**

**fmt.Print(v)**

**}**

**}**

Simple, right? What if we want to have the slice be an integer? You will need to make a new method for that:

**func Print(s []int) {**

**for \_, v := range s {**

**fmt.Print(v)**

**}**

**}**

These solutions might seem redundant, as we're only changing the parameter. But currently, that's how we solve it in Go without resorting to making it into some interface.  
And now with generics, they will allow us to declare our functions like this:

**func Print[T any](s []T) {**

**for \_, v := range s {**

**fmt.Print(v)**

**}**

**}**

**In the above function, we are declaring two things:**  
1. We have T, which is the type of the any keyword (this keyword is specifically defined as part of a generic, which indicates any type)  
2. And our parameter, where we have variable s whose type is a slice of T .

We will now be able to call our method like this:

**func main() {**

**Print([]string{"Hello, ", "playground\n"})**

**Print([]int{1,2,3})**

**}**

There are limitations on how far generics can take us. Printing, for example, is pretty simple since Golang can print out any type of variable being thrown into it.  
What if we want to do more complex things? Let's say that we have defined our own methods for a structure and want to call it:

type worker string

func (w worker) Work(){

fmt.Printf("%s is working\n", w)

}

func DoWork[T any](things []T) {

for \_, v := range things {

v.Work()

}

}

func main() {

var a,b,c worker

a = "A"

b = "B"

c = "C"

DoWork([]worker{a,b,c})

}

And we will get this:

**type checking failed for main  
prog.go2:25:11: v.Work undefined (type bound for T has no method Work)**

It fails to run because the slice processed inside the function is of type any and it doesn't implement the method Work, which makes it fail to run.

We can actually make it work, though, by using an interface

Well it works with the interface, but just having an interface without the generic works well, too

type Person interface {

Work()

}

type worker string

func (w worker) Work(){

fmt.Printf("%s is working\n", w)

}

func DoWork[T Person](things []T) {

for \_, v := range things {

v.Work()

}

}

func DoWorkInterface(things []Person) {

for \_, v := range things {

v.Work()

}

}

func main() {

var a,b,c worker

a = "A"

b = "B"

c = "C"

DoWork([]worker{a,b,c})

var d,e,f worker

d = "D"

e = "E"

f = "F"

DoWorkInterface([]Person{d,e,f})

}

And it will print out this:

**A is working**

**B is working**

**C is working**

**Reading and Writing files**

In order to read from files on your local filesystem, you’ll have to use the io/ioutil module. You’ll first have to pull of the contents of a file into memory by calling ioutil.ReadFile("/path/to/file.ext") which will take in the path to the file you wish to read in as it’s only parameter. This will return either the data of the file, or an err which can be handled as you normally handle errors in go.

func retrieveFromFile(filename string) string {

bs, err := ioutil.ReadFile(filename)

if err != nil {

fmt.Println("unable to get the file", err)

return ""

}

return string(bs)

}

In order to write content to files using Go, we’ll again have to leverage the io/ioutil module. We’ll first have to construct a byte array that represents the content we wish to store within our files.

**mydata := []byte("all my data I want to write to a file")**

Once we have constructed this byte array, we can then call ioutil.WriteFile() to write this byte array to a file. The WriteFile() method takes in 3 different parameters, the first is the location of the file we wish to write to, the second is our mydata object, and the third is the FileMode, which represents our file’s mode and permission bits

func saveToFile(filename, data string) error {

e := ioutil.WriteFile(filename, []byte(data), 0666)

if e == nil {

fmt.Println("data saved to file")

} else {

fmt.Println("data is not saved to file")

}

return e

}

**Writing to Existing Files**

func appendToFile(filename, newData string) {

f, err := os.OpenFile(filename, os.O\_APPEND|os.O\_WRONLY, 0600)

if err != nil {

panic(err)

}

defer f.Close()

if \_, err = f.WriteString(newData + "\n"); err != nil {

panic(err)

}

}

**Testing in Go**

**Steps for writing test suite in Golang:**

* Create a file whose name ends with \_test.go
* Import package testing by import “testing” command
* Write the test function of form*func TestXxx(\*testing.T)* which uses any of Error, Fail, or related methods to signal failure.
* Put the file in any package.
* Run command go test
* Create go module
* Exported method names should be starting with capital case
* We should write all the clean-up codes in t.Cleanup method

**Note:** test file will be excluded in package build and will only get executed on go test command.

import (

"fmt"

"testing"

)

func TestReturnGeeks(t \*testing.T) {

actualString := SayHello()

expectedString := "hello"

if actualString != expectedString {

t.Errorf("Expected String(%s) is not same as"+

" actual string (%s)", expectedString, actualString)

}

t.Cleanup(func() { fmt.Println("This is cleanup function") })

}

———————————————————————————————

import "fmt"

func SayHello() string {

return "hello"

}

// main function of package

func main() {

fmt.Println(SayHello())

}

**Golang Embedded Structs**

Golang doesn’t have the concepts of objects and inheritance. The design of the language is strongly opinionated and follows object-oriented principles (OOP)very closely, hence it favors composition over inheritance. Golang uses structs and methods to emulate objects.

Golang allows you to compose structs inside of other structs. There are two ways of composing structs: **anonymous embedded** fields and **explicit** fields.

# Anonymous Embedded Fields

An anonymous embedded field is a struct field that doesn’t have an explicit field name. Structs that have an anonymous field obtain all of the methods and properties of the nested struct. These methods and properties are called “promoted” methods and properties. Anonymous embedded fields can also be accessed by the type name directly.

The nuance comes from when structs declare methods and properties that also exist on the embedded struct. In this case, the methods and properties from the embedded struct are “shadowed” and the methods and properties on the struct you are calling will be used. You can still access the shadowed methods and properties by specifically accessing the embedded field and then calling the method or property.

type Bird struct {

}

func (b Bird) makeSound() {

fmt.Println("chirp")

}

type Eagle struct {

Bird // anonymous embedded field

}

e := Eagle{name: "Baldie"}

e.makeSound() // chirp

e.Bird.makeSound() // chirp

# Explicit Fields

You can also explicitly name a field to avoid confusion with promoted and shadowed fields.

Here, in order to call Bird’s makeSound() method, one has to explicitly access the nested Bird object using .b. It is no longer possible to access the underlying .Bird field.

type Eagle struct {

b Bird // explicit field

}

func (e Eagle) makeSound(){

fmt.Println("caw")

}

e := Eagle{name: "Baldie"}

e.makeSound() // caw

e.b.makeSound() // chirp

e.Bird.makeSound() // error

**Conclusion**

In summary, there are two ways of composing structs in Golang: **anonymous embedded** fields and **explicit** fields.

Anonymous embedded fields allow methods and properties from the embedded struct to automatically get promoted to the nesting struct, but there’s a possibility of those methods and properties getting shadowed if the nesting struct declares methods of properties of the same name.

Explicit fields directly associate a field name with the nested struct, so that accessing the nested struct’s methods and properties are unambiguous. In production code, it’s often better to use explicit fields over anonymous embedded fields to make the code more readable and not surprise yourself with shadowed fields.

# Maps vs. Structs

**For map:**  
- All key and value are of same type.  
- When keys are indexed and we can iterate over them.  
- Closely related and significant value type.  
- Don’t need to know all the keys at compile time.  
- Key are indexed- we can iterate over them.  
- Reference type  
- Zero value for a map is empty map

**For struct:**  
- All values can be of different type.  
- Need to know all the different fields at compile time.  
- Keys don’t support indexing  
- Value type.  
- Zero value for struct will be according to the struct field type

# When to use?

When to use structs? If we have close set of keys means the fixed data size with keys we will be using structs. Using structs are safe way and easy way while working with JSON data also.

When to use maps? If we are creating some kind of relationship between keys and values and we don’t really know what that collection of values going to be at compile time or as we are writing our code then we got the great use-case of using a map.

Most of cases, vast majority of time we mostly, use structs than maps. But it all depends on nature and type of the application and requirement of the project.

**Note:**

Small case means that is private field or member, Cap case means that is public field or member