All of the values that we define in our program are stored in the computer memory and have their own unique memory address. Pointers are special variables that hold the memory address of values.  
In Go, we declare a pointer using a **\***

**var p \*int Now, p is a pointer to an integer value.**

We know how to define a pointer, but how do we assign it a memory address? This is done using the & operator, which returns the memory address of a variable.

**x := 42  
p := &x**

Now p is a pointer and holds the memory address of x.

If we want to access the underlying value of a pointer, we can use the \* operator

**x := 42  
p := &x  
fmt.Println(\*p)**

The \* operator can also be used to change the value of the memory address the pointer holds:

**x := 42  
p := &x  
\*p = 8  
fmt.Println(\*p)  
fmt.Println(x)**

The \* operator is called the dereferencing operator.

We have used pointers with functions in a previous lesson, when taking input from the user:

**var input string  
fmt.Scanln(&input)  
fmt.Println(input)**

Here, we pass the memory address of the input variable (a pointer to input) to the Scanln() function, which uses it to store the input value.

We can pass pointers as function parameters.

func change(val int) {

val = 8

}

func change\_ptr(ptr \*int) {

\*ptr = 8

}

func main() {

x := 42

change(x)

fmt.Println(x)

change\_ptr(&x)

fmt.Println(x)

}

The **change()** function takes an integer argument and changes its value. The **change\_ptr()** function does the same using a pointer.  
When you run the code, you will see that the **change()** function did not change the value of our x variable, because the argument is just a copy of its value, while the **change\_ptr()** did change the actual value of x, because it used its memory address as the argument.

Note that we need to pass the memory address using the & operator to functions that take a pointer as their argument.

**Structs**

Go does not support classes. Instead, it has structs. Structs are collections of fields that allow you to group data together.

let's make a struct to store the data of Contacts

**type Contact struct {  
 name string  
 age int  
}**

Our Contact struct has two fields, a string and an integer. Now, we can create a new Contact using the following syntax:

**x := Contact{"James", 42}**

**x** is now a structs object that is initialized with the data provided in the curly braces.

We can also provide the names of the fields when creating a new struct. This makes it easier to read the code. For example:

**x := Contact{name: "James", age: 42}**

We can access the struct fields using the name of the struct and a dot:

**x := Contact{"James", 42}  
x.age = 8  
fmt.Println(x.age)  
fmt.Println(x.name)**

Note, that we are able to change the values of the fields by simply assigning them new values.

**Pointers to Structs**

Similar to simple pointers, we can also make pointers to structs using the & operator:

**x := Contact{"James", 42}  
p := &x**

Pointers to structs are automatically dereferenced, meaning we can access the field values by simply using a dot:

**x := Contact{"James", 42}  
p := &x  
fmt.Println(p.age)**

We could use **(\*p).age** to access the age field of the struct, but that looks complicated and hard to read. Go allows you to shorten that syntax and simply use **p.age** instead. We can also use pointers when creating a new struct:

**p := &Contact{"John", 15}  
fmt.Println(p.name)**

Now **p** is a pointer to the newly created struct. Pointers to structs are useful, as they allow you to pass them to functions as arguments.

We can add functionality to our structs using methods. Methods are simply functions with a special **receiver argument**. Let's have a look at an example:

**func (x Contact) welcome() {  
 fmt.Println(x.name)  
 fmt.Println(x.age)  
}**

The receiver appears between the **func** keyword and the method name. In the example above, the receiver is the Contact struct. Note that we can access the receiver structs fields in the method.

type Contact struct {

name string

age int

}

func (contact Contact) show() {

fmt.Println(contact.name)

fmt.Println(contact.age)

}

func (ptr \*Contact) increase(val int) {

ptr.age += val

}

func main() {

contact := Contact{"James", 42}

contact.increase(8)

contact.show()

}

Since methods are just functions with a receiver argument, we can achieve the same functionality using a regular function that takes a struct as its argument.  
In case we need to change the data of the struct in a method, we can use **pointers as** **method receivers**. Go automatically dereferences the pointers, so we simply call the method with the struct name, just as we did with a simple receiver. Since methods often need to modify their receiver, **pointer receivers** are more common than **value receivers**.

We can also use **type** on the existing datatypes and attach receiver function on them.   
**type Cards []string**We can use this cards anywhere like **cards := Cards{}**

**Arrays**

An array is a sequence of elements of the same type. An array is defined using square brackets which define the number of elements the array will hold.   
An array type definition specifies a length and an element type. For example, the type [4]int represents an array of four integers. An array’s size is fixed; its length is part of its type ([4]int and [5]int are distinct, incompatible types). Arrays can be indexed in the usual way, so the expression s[n] accesses the nth element, starting from zero.  
Arrays do not need to be initialized explicitly; the zero value of an array is a ready-to-use array whose elements are themselves zeroed  
The in-memory representation of [4]int is just four integer values laid out sequentially  
Go’s arrays are values. An array variable denotes the entire array; it is not a pointer to the first array element (as would be the case in C). This means that when you assign or pass around an array value you will make a copy of its contents. (To avoid the copy you could pass a pointer to the array, but then that’s a pointer to an array, not an array.) One way to think about arrays is as a sort of struct but with indexed rather than named fields: a fixed-size composite value.

For example:

**var a [5]int**

Now a is an array of 5 integers.  
We can also define and initialize values of the array using the following syntax:

**a := [5]int{0, 2, 4, 6, 8}**

As we can see, we need to provide the size of the array when declaring it. This means that arrays have a fixed size.

After declaring the array, we can access its elements using square brackets and their indexes. For example:

**var a [5]int  
a[0] = 8  
a[1] = 42  
fmt.Println(a[1])**

Each element of an array has its unique index, starting with 0. The first element has the index 0, the second element has the index 1, and so on. We can assign values to array elements, as well as retrieve their value. By default, the elements of the array are initialized to the zero value of the given type.

**Slices**

The slice type is an abstraction built on top of Go’s array type.  
An array has a fixed size, meaning once defined, you cannot change the number of elements it holds. To overcome this, Go provides the slice, which is a dynamically-sized view into the elements of an array. A slice is based on an array and is defined by specifying two indices, a low and high bound, separated by a colon:

**a := []int{0, 2, 4, 6, 8}**

A slice does not store any data; it just describes a section of an underlying array. Changing the elements of a slice modifies the corresponding elements of its underlying array.  
Arrays have their place, but they’re a bit inflexible, so you don’t see them too often in Go code. Slices, though, are everywhere. They build on arrays to provide great power and convenience.  
The type specification for a slice is []T, where T is the type of the elements of the slice. Unlike an array type, a slice type has no specified length.  
A slice literal is declared just like an array literal, except you leave out the element count  
We can have multiple slices of the same array. A change in any of them will be seen in all slices, as it will affect the underlying array.   
A slice can be created with the built-in function called **make**, which has the signature,

**func make([]T, len, cap) []T**

**a := make([]int, 5)**

The length and capacity of a slice can be inspected using the built-in **len** and **cap** functions.The make function creates an array of the given type and size, and returns a slice that refers to that array. After creating a slice, we can add append new elements to it using the append() function:

**a := make([]int, 5)  
a = append(a, 8)  
fmt.Println(a)**

The **append()** function takes the slice as its first argument and the elements to be added to the end of the slice as its next argument. It then returns a new slice, containing the old slice plus the new elements appended. We can append multiple values at once by just comma separating the values as arguments, for example:

**append(s, 1, 2, 3)**The zero value of a slice is nil. The len and cap functions will both return 0 for a nil slice.

A slice can also be formed by “slicing” an existing slice or array. Slicing is done by specifying a half-open range with two indices separated by a colon. For example, the expression b[1:4] creates a slice including elements 1 through 3 of b (the indices of the resulting slice will be 0 through 2).

For more reference check <https://go.dev/blog/slices-intro>

**Range**

Now that we know how to create arrays and slices, let's learn how to iterate over their elements using a loop. The range form of the for loop allows you to iterate over a slice. During each iteration of the loop, it returns two values: the index of the element and its value. If you want only the values, you can skip the index using an underscore:

func main() {

a := make([]int, 5)

a[1] = 2

a[2] = 3

for \_, v := range a {

fmt.Println(v)

}

}

func main() {

a := make([]int, 5)

a[1] = 2

a[2] = 3

for i, v := range a {

fmt.Println(i, v)

}

}

You can use ranges for slices as well as arrays. The range can also be used to iterate over the characters of a string. This program outputs Unicode code points of the characters. If you want to output the actual characters, you can use the Printf() function. The Printf() function is similar to the one in C, taking the format of the output as its argument. %c in our case denotes a character, while \n defines a new line.

func main() {

x := "hello"

for \_, c := range x {

fmt.Printf("%c ", c)

}

}

func main() {

x := "hello"

for \_, c := range x {

fmt.Print(c)

}

}

**Maps**

**Maps** are used to store **key:value** pairs. The key is always unique. We can create a **map** using the **make()** function, similar to arrays.

**mp := make(map[string]int)**

You can also initialize a map using the following syntax. If the requested key does not exist in the map, a zero value will be returned. Maps are also called dictionaries, associative arrays, or hash tables. You can use the delete function to remove an element from the map. Maps are printed in the form **map[key:value key:value]** when output with **fmt.Println().**

func main() {

m := map[string]int{

"James": 42,

"Amy": 24}

delete(m, "James")

fmt.Println(m)

}

func main() {

m := map[string]int{

"James": 42,

"Amy": 24}

fmt.Println(m["Amy"])

}

**Concurrency**

Concurrency means multiple computations are happening at the same time. It is used when your program has multiple things to do. Concurrency is about creating multiple processes executing independently.  
For example, imagine a shooting game, where many things are happening at the same time: enemies are running and shooting, points are being calculated, weapons are being unlocked, etc.  
All of these things need to be happening concurrently.  
In order to use concurrency, the program is broken into parts, which are then executed separately.  
When using concurrency, we are able to achieve the intended results in less time, thus increasing the overall performance and efficiency of our programs.

To achieve concurrency, Go provides **Goroutines**.  
A **Goroutine** is much like a **thread** to accomplish multiple tasks, but it consumes fewer resources than OS threads.  
When a program is broken down into separate tasks, each **Goroutine** can be used to accomplish a task, enabling concurrency in our program.

Goroutines are not **OS threads**, they are **virtual threads**, managed by **Go**. You can have thousands of **Goroutines** running in a Go program. Let's have a look at the following program:

import (

"fmt"

"time"

)

func out(from, to int) {

for i := from; i <= to; i++ {

time.Sleep(50 \* time.Millisecond)

fmt.Println(i)

}

}

func main() {

out(0, 5)

out(6, 10)

}

The **out()** function simply outputs numbers in the given range. We use a **time.Sleep()** to emulate work being done between the outputs just for demonstration purposes. It simply waits for the provided time (50ms) before continuing the execution.  
Now, if we call the function in main two times, the first call will execute first followed by the second call.  
This will generate the output of 0 to 5, then 6 to 10.  
This is called a **sequential program**, as the statements are executed one after the other. The first call needs to complete, before the second call starts.

When running concurrent programs, do not often want to wait for one task to finish before starting a new one.  
To achieve concurrency, let's start the function calls as Goroutines, using the **go** keyword:

**go out(0, 5)  
go out(6, 10)**

It is very simple to start a **Goroutine** -- we simply need to add a **go** keyword before the function call. If we run the program, we get **No Output**. This output happens because the **main()** function exits before the Goroutines complete. Our program has **3 virtual threads**. the **2 function calls, and main().** Our 2 function calls get executed concurrently, and main() does not wait for them to finish.

**go out(0, 5)  
go out(6, 10)  
time.Sleep(500 \* time.Millisecond)**

The **500ms** wait should be enough time for the Goroutines to finish executing and generating the output. Now when you run the code, you will see that the output is not sequential, each Goroutine worked independent and concurrently.

**Channels**

**Goroutines** run independently and they do not know when another one has finished executing. This causes, for example, the main() function to quit, before any started Goroutine has finished. To enable communication between **Goroutines**, Go provides **Channels**.

A **channel** is like a **pipe**, allowing you to send and receive data from Goroutines, and enabling them to communicate and synchronize.

Similar to how water flows from one end to another in a pipe, data can be sent from one end and received by the other end using channels. To use a channel, we first need to make one using the **make()** function.

**ch := make(chan int)**

The type after the **chan** keyword indicates the type of the data we will send through the channel.  
We can send data to the channel using the following syntax:

**ch <- 8**

Similarly, we can receive data from the channel using the following syntax:

**value := <- ch**

If we do not need the value as a variable, we can simply use:

**<- ch**

Data flows in the direction of the arrow.

We can use our channel and rewrite the previous example without a **time.Sleep()** in **main()**

import (

"fmt"

"time"

)

func out(from, to int, ch chan bool) {

for i := from; i <= to; i++ {

time.Sleep(50 \* time.Millisecond)

fmt.Println(i)

}

ch <- true

}

func main() {

ch1 := make(chan bool)

ch2 := make(chan bool)

go out(0, 5, ch1)

go out(6, 10, ch2)

fmt.Println(<-ch1)

fmt.Println(<-ch2)

}

We define a **bool channel** and pass it to our **out()** function as an **argument**. After the function finishes its task, we send the value **true** to the **channel**, which is received in **main()**.

Now, **main()** is waiting to receive data from the channel, making the program wait for the **Goroutines** to finish executing.

The receive operation blocks the code until and unless some data is sent by the send operation. If no data is received, a deadlock will occur, blocking the code from executing.

Let's use a **channel** to send data from a Goroutine and use it in main().

Our program needs to calculate and output the sum of all even numbers in a given range plus the sum of their squares and output the result:

**output = evenSum + squareSum**

We will use two Goroutines: one to calculate the **evenSum**, and the other to calculate the **squareSum**. We will get the data using channels in main(), then calculate and output the final sum.  
As you can see, our functions send the result via channels.  
Now we can call them as Goroutines in main() and output the resulting sum.  
We use the channels to get the result of each Goroutine and output their sum.

If you do not need to send data to a channel anymore, you can close it using **close(ch)**, where **ch** is the name of the channel. This is done in the sender.

func evenSum(from, to int, ch chan int) {

result := 0

for i := from; i <= to; i++ {

if i%2 == 0 {

result += i

}

}

ch <- result

}

func squareSum(from, to int, ch chan int) {

result := 0

for i := from; i <= to; i++ {

if i%2 == 0 {

result += i \* i

}

}

ch <- result

}

func main() {

evenCh := make(chan int)

sqCh := make(chan int)

go evenSum(0, 100, evenCh)

go squareSum(0, 100, sqCh)

fmt.Println(<-evenCh + <-sqCh)

}

**Select**

The select statement is used to wait on multiple channel operations.  
The syntax is similar to switch except that each of the case statements will be a channel operation.  
Let's use the same program from our previous example and select the channel that is ready first:

func evenSum(from, to int, ch chan int) {

result := 0

for i := from; i <= to; i++ {

if i%2 == 0 {

result += i

}

}

ch <- result

}

func squareSum(from, to int, ch chan int) {

result := 0

for i := from; i <= to; i++ {

if i%2 == 0 {

result += i \* i

}

}

ch <- result

}

func main() {

evenCh := make(chan int)

sqCh := make(chan int)

go evenSum(0, 100, evenCh)

go squareSum(0, 100, sqCh)

select {

case x := <-evenCh:

fmt.Println(x)

case y := <-sqCh:

fmt.Println(y)

}

}

The select statement waits for a channel to receive data and executes its case. This means that only one of the cases will execute -- the one that corresponds to the channel that receives data first. If both channels receive data at the same time, one of the cases is chosen randomly. Combining Goroutines and channels with select is a powerful feature of Go. Imagine a program that needs to execute some code whenever one of the concurrent operations completes -- this can be achieved using select. A select can have a default case, which will execute when no channel is ready. For example, we could have an infinite for loop, waiting for one of the channels to receive data:

select {

case x := <-evenCh:

fmt.Println(x)

case y := <-sqCh:

fmt.Println(y)

default:

fmt.Println("This is default case")

}

The for loop uses a select to check which channel got data. If none of them are ready, the default case will execute which will wait for 50ms.  
As soon as a channel gets data, the return statement will exit the loop.

A select statement blocks until at least one of its cases can proceed. The default case is useful in preventing deadlocks -- without it the select would wait for a channel forever, crashing the program if none of the channels received data.

**Interfaces**

In Go programming, we use interfaces to store a set of methods without implementation. That is, methods of interface won't have a method body. For example,

**type Shape interface {**

**area() float32**

**perimeter() float32**

**}**

Here, Shape is an interface with methods: area() and perimeter(). You can see both methods only have method signatures without any implementation.

to use an interface, we first need to implement it by a type (struct). To implement an interface, a struct should provide implementations for all methods of an interface. For example,

package main

import "fmt"

type Shape interface {

area() float32

}

type Rectangle struct {

length, breadth float32

}

func (r Rectangle) area() float32 {

return r.length \* r.breadth

}

func calculate(s Shape) {

fmt.Println("Area:", s.area())

}

func main() {

rect := Rectangle{7, 4}

calculate(rect)

}

When a struct implements an interface, it should provide an implementation for all the methods of the interface. If it fails to implement any method, we will get an error.

**cannot use r (type Rectangle) as type Shape in argument to calculate:**

**Rectangle does not implement Shape (missing perimeter method)**

We know that interfaces are used to store a set of methods without implementation. In Go, we can also create interfaces without any methods, known as empty interfaces. For example, **interface {}**

Here, we have created an empty interface without any methods.

In Go, we can create variables of the empty interface type.

package main

import "fmt"

func main() {

var a interface {}

fmt.Println("Value:", a)

}

Here, we have created a variable of type empty interface. When we print the variable, we get nil as output.

Empty Interface to Pass Function Argument of Any Type:

Normally in functions, when we pass values to the function parameters, we need to specify the data type of parameters in a function definition.

However, with an empty interface, we can pass parameters of any data type.

We can also use an empty interface to pass any number of arguments to the function definition.

package main

import "fmt"

func displayValue(i interface {}) {

fmt.Println(i)

}

func displayValues(I… interface {}) {

fmt.Println(i)

}

func main() {

a := "Welcome to Programiz"

b := 20

c := false

displayValue(a)

displayValue(b)

displayValue(c)

displayValues(a)

displayValues(a, b)

displayValue(a, b, c)

}

In the above example, we have used an empty interface i as the parameter to the displayValue() function.

Now the same function works for any type of parameters (string,numeric, boolean).

In the above example, we have used an empty interface i as the parameter to the displayValues() function.

Type Assertions:

Type assertions allow us to access the data and data type of values stored by the interface.

**var a interface {}**

**a = "Hello World"**

**a = 10**

Here, the **a** variable is of empty interface type, and it is storing both the string and integer value.

This seems like an important feature of an empty interface. However, sometimes this will create ambiguity on what data an interface is holding.

To remove this ambiguity, we use type assertions.

package main

import "fmt"

func main() {

// create an empty interface

var a interface {}

// store integer to an empty interface

a = 10

// type assertion

**interfaceValue := a.(int)**

fmt.Println(interfaceValue)

}

In the above example, we have stored the integer value 10 to the empty interface denoted by a. Notice the code

Here, (int) checks if the value of a is an integer or not. If true, the value will be assigned to interfaceValue.

Otherwise, the program panics and terminates

the type assertion statement actually returns a boolean value along with the interface value

**var a interface {}**

**a = 12**

**interfaceValue := a.(int)**

Here, the data type of value **12** matches with the specified type (int), so this code assigns the value of a to interfaceValue.

Along with this, a.(int) also returns a boolean value which we can store in another variable.

**interfaceValue, booleanValue := a.(int)**

we get both the data and boolean value.

We can use this to avoid panic if the data type doesn't match the specified type.

This is because, when the type doesn't match, it returns false as the boolean value, so the panic doesn't occur.

**Output :**

Interface Value: 0

Boolean Value: false

**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