Go is an open-source programming language that makes it easy to build simple, reliable, and efficient software.

Go was designed at Google in 2007 to improve programming productivity in an era of multicore, networked machines and large codebases.

The language is often referred to as Golang because of its domain name, golang.org, but its proper name is Go.

package main

import "fmt"

func main() {

fmt.Println("Hello World")

}

To import a package, we use the import statement

One of the most popular packages is "fmt", which stands for format, and provides input and output functionality.

Package main is the entry point for go program.

You can import multiple packages at once, using parentheses. For example: **import ("fmt" "math")**

**Variables**

In Go, the var keyword is used to declare variables.

For example: **var i int**

The code above declares a variable named i of type int. int stands for integer and represents whole numbers.

We can assign the variable a value and output it:

var i int = 8

fmt.Println(i)

You can also declare multiple variables on one line and assign values to them:

var i, j int = 8, 42

fmt.Println(i)

fmt.Println(j)

If you assign a value to the variable, you can omit the type declaration, as Go will automatically take the type of the value assigned. Go supports short variable declaration using **:=**

For example : **i := 8**

The variable **I** will be a integer variable and the value of the **i** will be **8**

It can also be used to declare and initialize multiple variables on one line. **x, y := 10, 5**

**Data Types**

We used int to declare integer values in our previous lesson. **var i int = 8**

Let's see what other common types Go supports.

float32 - a single-precision floating point value.

float64 - a double-precision floating point value.

string - a text value.

bool - Boolean true or false.

The difference between float32 and float64 is the precision, meaning that float64 will represent the number with higher accuracy, but will take more space in memory.

var x int = 42

var y float32 = 1.37

var name string = "James"

var online bool = true

Since we used initializers when declaring the variables, we could have omitted the types, as they would be taken from the values assigned. We included them to demonstrate the different data types.

Another interesting feature of Go are zero values: variables that are declared without a value take the zero value of their type. 0 for numeric types, false for the boolean type,"" for strings.

**Constants**

A variable can change its value during the program bit in some cases your program may need values that are preserved during the program. These are called constants and they cannot be changed from their initial value.

Constants are declared like variables, but with the const keyword and need to be assigned a value:

const pi = 3.14

fmt.Println(pi)

Now, pi is a constant and cannot be changed.

**Constants cannot be declared using the := syntax.**

**Input**

The "fmt" package also allows you to take input from the user of the program.

To take input, we need to use the Scanln() function and provide it with the variable which should hold the input value:

var input string

fmt.Scanln(&input)

fmt.Println(input)

Now the input variable will hold the value which the user enters when running the program.

**Note the ampersand & before the variable name -- it is used to return the address of a variable.**

The previous example took the input from the user as a string.

If we need to take a number as input, such as an integer, we can simply declare the type of the input variable as int and Go will automatically convert the input to that type:

var input int

fmt.Scanln(&input)

fmt.Println(input\*2)

**Decision Making**

The **if** statement can be used to make decisions and run code when a given condition is true.

For example:

x := 42

if x > 18 {

fmt.Println("Allowed")

}

The code of the if statement should be enclosed in curly braces { } and can include multiple lines of code.

The code in the curly braces of an if statement will run only if the condition evaluates to true.

The else statement can be used to run code when the condition of an if statement is false.

For example:

x := 14

if x > 18 {

fmt.Println("Allowed")

} else {

fmt.Println("Not allowed")

}

**There is no ternary if in Go, so you'll need to use a full if statement even for basic conditions.**

**Sometimes you need a variable only for the if/else statements.**

For this, the if statement in Go can start with a variable declaration before the condition:

if x := 42; x > 18 {

fmt.Println("Allowed")

} else {

fmt.Println("Not allowed")

}

**Note the semicolon after the variable declaration - it is used to separate the statements.**

Variables declared in a if statement are only available in the if/else blocks.

**Switch**

A **switch** statement is a shorter way of writing a sequence of **if/else** statements.

For example, imagine having the following if/else statements. The code checks the value of the num variable and outputs the corresponding text version. We can achieve the same result using a switch statement.

num := 3

switch num {

case 1:

fmt.Println("One")

case 2:

fmt.Println("Two")

case 3:

fmt.Println("Three")

default:

fmt.Println(num)

}

num := 3

if num == 1 {

fmt.Println("One")

} else if num == 2 {

fmt.Println("Two")

} else if num == 3 {

fmt.Println("Three")

} else {

fmt.Println(num)

}

The switch statement makes the code shorter and easier to read. Each case statement includes the value to compare with and the code to run after the colon **:**

A simple way to replace an if/else chain is to use a switch without an expression. That way, each case statement can include a condition:

x := 2

switch {

case x>0 && x<10:

fmt.Println("something")

case x > 10:

fmt.Println("something else")

}

The optional default case runs if none of the cases match.

In other programming languages, such as C++ or Java, each case needs to have a break statement, in order to stop the execution of the switch statement. In Go, the break statement is not needed, as the switch cases evaluate from top to bottom, stopping when a case succeeds.

Similar to if statements, switch can also have a short variable declaration before the conditional expression.

**For Loop**

Loops are used to repeat code until a certain condition holds.

The only loop construct in Go is for, which has three components: init, condition, post statement.  
For example:

for i:=0; i<5; i++ {

fmt.Println(i)

}

The code above will output the numbers **0 to 4**, as we initialize the value of **i to 0** and increment it while the condition **i<5** holds. **i++** is a shorter version of **i = i+1** and is called the increment operator.

The init and post statements can be omitted.

For example:

num, sum := 1, 0

for num <= 1000 {

sum += num

num++

}

fmt.Println(sum)

The code above will calculate the sum of numbers from 1 to 1000. Note that with just the condition, for becomes similar to while loops available in other programming languages.

**Functions**

Functions are an important part of Go. They allow you to define a block of code, give it a name and call it in the code. This makes the function code reusable, as you can call it multiple times in different parts of your program.

We have already seen functions in our previous examples. For example, the Println() function outputs text. Println is the name of the function, while "Hello" is the argument which we provide to the function using the parentheses. Remember, to call a function, we always use the name of the function followed by parentheses.

Function arguments are a way for our functions to take input parameters. For example, we can add a name argument to our welcome() function and use it in the output.

As you can see, the argument is provided in the parentheses and includes the name followed by the type. The argument behaves like a variable inside the function's body, meaning you can use its value by its name. When calling the function, we need to pass it a string argument inside the parentheses.

We are able to reuse the code in our function by calling it with different arguments. Remember, when defining the function, the type of the argument comes after the variable name.

To make a function take multiple arguments, separate them using commas. For example, let's create a function that takes two integer arguments and outputs their sum.

If the arguments are of the same type, you can omit the types for the arguments and define it only for the last one.

All of our function examples so far have resulted in a simple output. Oftentimes, the result of the function needs to be assigned to a variable so that your code can use it further.

The return statement terminates the function and returns the provided value to the code that called it. Notice that we need to define the return type of the function after the arguments definition. In our case, it is int.

package main

import "fmt"

func main() {

fmt.Println("Hello")

result := sum( 1 , 3 )

fmt.Println( result )

}

func sum(a int, b int) int {

fmt.Println(a+b)

return a+b

}

func sum(a, b int){

fmt.Println(a+b)

}

package main

import "fmt"

func welcome(name string) {

fmt.Println("Welcome", name)

}

func main() {

welcome("David")

welcome("James")

}

}

A useful feature of Go is that functions can return multiple values. The swap() function takes two integer arguments and returns them in swapped order. Note that the return type of each value should be declared: in our case it is (int, int). We can call our function and assign it result to variables:

func swap(x, y int) (int, int) {

return y, x

}

func main() {

a, b := swap(42, 8)

fmt.Println(a)

fmt.Println(b)

}

**Returning multiple values** from a function is handy! For example, it can be used to return both the result and the error values of an operation.

A **Defer** statement ensures that the function is called only after the surrounding function returns. The code will first output "Hey" and only after that output the result of the welcome() function. This happens because the call to welcome() is deferred, meaning it waits until main() finishes execution and only then calls it. **defer** is often used for clean-up, for example, to release resources used by the code, such as files, connections, etc. If you have deferred multiple function calls, they will execute in last-in-first-out order. **The defer calls are stacked on top of each other, which is why they are executed in last-in-first-out order.**

func main() {

fmt.Println("start")

for i := 0; i < 5; i++ {

defer fmt.Println(i)

}

fmt.Println("end")

}

func welcome() {

fmt.Println("Welcome")

}

func main() {

defer welcome()

fmt.Println("Hey")

}

Scope is where a variable can be used. There are two main scopes in Go: local and global.  
A variable defined in the function is called a local variable. Their scope is only in the function body, which means they only exist within their function.

func test() {

var x = 8

}

Now, x is a local variable and is available only in the body of the test() function. Trying to access it in another function, for example main(), will cause an error.

A variable defined outside the local scope is called a global variable. Global variables can be used throughout the package.

var x = 8

func test() {

fmt.Println(x)

}

func main() {

fmt.Println(x)

}

The variable x is declared outside of the functions, making it a global variable, which is accessible anywhere in the package. Global variables are often considered a bad practice. It is better to pass variables as function arguments.

**Pointers**

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**.

**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. 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**

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 := [5]int{0, 2, 4, 6, 8}  
var s []int = a[1:3]**

This code selects the elements with index 1 to 3 from the a array, including the first given index, but excluding the last. So, the slice s will now include the values [2, 4]:

**a := [5]int{0, 2, 4, 6, 8}  
var s []int = a[1:3]  
fmt.Println(s)**

We can omit the low or the high bound. Omitting the low bound will take the value 0, while omitting the high bound will take the length of the array. For example: a[:3] will take the first 3 elements of the array. We can access the values of the slice using the same syntax as with arrays:

**a := [5]int{0, 2, 4, 6, 8}  
var s []int = a[1:3]  
fmt.Println(s[1])**

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.

**a := [5]int{0, 2, 4, 6, 8}  
var s []int = a[1:3]  
s[0] = 8  
fmt.Println(a)**

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. Go provides a make() function to create slices. This is how you create dynamically-sized arrays. For example:

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

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)**

**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.