September 16, 2020

Learn Go in Twenty Minutes

A lot of thanks goes to fasterthanlime for inspiring this article with A half-hour to learn Rust. I thought I'd try something similar, but instead of Rust, this article will try provide an overview of Go in twenty minutes.

Instead of focusing on one or two concepts, or spending half of this article trying to convince you why Go is awesome, I'll try to go through as many Go snippets I can and explain what the keywords and symbols they contain mean. This article can be used as a cheat sheet for developers familiar with Go, or a tutorial for developers new to Go but familiar with other languages.

Ready? Go!

Packages

Every Go program is made up of packages. Programs start by running the `main` function in the `main` package.

```
package main

func main() {
    // ...
}
```

From now on, the `main` function and package declaration will be omitted from examples for brevity.

You can use external packages by importing them:

```go
package main

import "fmt"
```

You can use exported names in an imported package by using the package name as a *qualifier*:

```go
import (
  "fmt"
  "math/rand"
)


fmt.Println("My favorite number is", rand.Intn(10))
```

You can create a local alias for any import:

```go
import (
  formatter "fmt"
  random "math/rand"
)


formatter.Println("My favorite number is", random.Intn(10))
```

Or use a dot to access it without any qualifier:

```go
import (
  . "fmt"
  . "math/rand"
)

Println("My favorite number is", Intn(10))
```

A name is exported if it begins with a capital letter. This is similar to `public` and `private` in other languages:

```go
var x string = "hello" // x is unexported, or private
var A string = "hello" // A will be exported, or public
```

Packages can have an `init` function. This will be executed as soon as the package is imported:

```
func init() {
  performSideEffects()
}
```

Scope

A pair of brackets declares a block, which has its own scope:

```
// This prints "in", then "out"
func main() {
  x := "out"
  {
    // this is a different `x`
    x := "in"
    println(x) // => in
  }
  println(x) // => out
}
```

A variable defined outside of any function is in the "global" scope.

```
var Global string = "I am a global variable!"
```

Variables

`var` declares a variable of a given type:

```
var x int // Declare variable "x" of type int
x = 42     // Assign 42 to "x"
```

This can also be written as a single line:

```
var x int = 42
```

The type of a variable can also be inferred:

```
var x = 42
```

Inside of a function body, this can also be written in short form:

```
x := 42
```

You can declare many variables at the same time:

```
var x, y int
x, y = 10, 20
```

If you declare a variable without initializing it, it will implicitly be assigned to the zero value of its type:

```
var a SomeStruct
var b bool
var x int
var y string
var z []int

fmt.Println(a) // => nil
fmt.Println(b) // => false
fmt.Println(x) // => 0
fmt.Println(y) // => ""
fmt.Println(z) // => []
```

Type Conversions

You can convert values between different types:

```
var f float64 = 1 // +1.000000
i := int(f)       // 1
z := uint(f)      // 1
```

Note that unlike in other languages, conversions between types must be done explicitly.

Constants

Unchanging values can be declared with the `const` keyword. Constants can be characters, strings, booleans, or numeric values:

```
const pi = 3.14
```

You can group the declaration of multiple constants or variables:

```
const (
  x = 2
  y = 4
)
```

Constants are *immutable* - they cannot be reassigned to:

```
const x = 1
x = 2 // error: cannot assign to x
```

Iota

The `iota` keyword represents successive integer constants:

```
const (
  zero = iota
  one  = iota
  two  = iota
)
```

```go
fmt.Println(zero, one, two) // "0 1 2"
```

This can be written in short form:

```go
const (
  zero = iota
  one
  two
)

fmt.Println(zero, one, two) // "0 1 2"
```

You can perform operations on an iota:

```go
const (
  four = iota + 4
  five
  six
)

fmt.Println(four, five, six) // "4 5 6"
```

And use the blank identifier to skip a value:

```go
const (
  zero = iota
  _     // skip 1
  two
  three
)
fmt.Println(zero, two, three) // "0 2 3"
```

Iota's are commonly used to represent enums, which Go does not support natively.

The Blank Identifier

The blank identifier `_` is an anonymous placeholder. It basically means to throw away something:

```
// this does nothing
_ = 42

// this calls `getThing` but throws away the
// two return values
_, _ = getThing();
```

You can use it to import a package solely for its `init` function:

```
import _ "log"
```

Or to avoid compiler errors during development:

```
var _ = devFunction() // TODO: remove in production
```

Looping

Go has only one loop, the `for` loop. It has three components:

- init: executed before the first iteration
- condition: evaluated before every iteration
- post: executed at the end of every iteration

Here's an example:

```
sum := 0
for i := 0; i < 10; i++ {
  sum ++
}
fmt.Println(sum) // 10
```

The init and post statements are optional:

```go
// this loop will run forever
yup := true
for yup {
  fmt.Println("yup!")
}
```

An infinite for loop can also be written like this:

```go
// this loop will also run forever
for {
    fmt.Println("yup!")
}
```

You can use break to `break` out of a loop:

```go
for {
  if shouldBreak() {
    break
  }
}
```

And `continue` to skip to the next iteration

```go
for i := 0; i < 5; i++ {
  if i == 2 {
    continue
  }
  fmt.Print(i)
}

// => 0 1 3 4
```

Control Flow

Go's `if` statements have a similar syntax to its loops:

```go
if 1 == 1 {
  fmt.Println("true")
}
```

`if` conditions can also have an init statement:

```
// this prints "foo"
if x := "foo"; x != "bar" {
  fmt.Println(x)
}
// "x" is now out of scope
```

Go also has `else` and `else if` statements:

```
if something {
  doSomething()
} else if somethingElse {
  doSomethingElse()
} else {
  return
}
```

If your `if - else` statement is getting long, switch to a `switch` statement :)

```
switch {
case something:
  doSomething()
case somethingElse > 10:
  doSomethingElse()
default:
  return
}
```

You can also switch on a condition:

```
switch x := 2; x {
// if x equals 1
case 1:
  doSomething()
// if x equals 2
case 2:
  doSomethingElse()
// otherwise
default:
    return
}
```

You can *fall through* to the case below the current case:

```
x := 1

switch x {
case 1:
    fmt.Print("1")
    fallthrough
case 2:
    fmt.Print("2")
}

// => 1 2
```

In a switch statement, only the first matched case is executed unless you *fall through*.

Go also has *labels* to help manage control flow:

```
MyLabel:
```

You can jump to a specific label with the `goto` keyword:

```
print(1)

// jump to the "Three" label
goto Three
fmt.Print(2)

Three:
fmt.Print(3)

// => 13
```

Labels have a very specific use case. They can often make code less readable and are avoided by many Go programmers.

Arrays

Arrays have a fixed length:

```
var a [2]string // an array of 2 strings
a[0] = "Hello"
a[1] = "World"
fmt.Println(a) // => [Hello World]
```

You create arrays using *array literals*:

```
helloWorld := [2]string{"Hello", "World"}
```

You cannot increase an array beyond its capacity. Doing so will cause an error:

```
helloWorld := [2]string{"Hello", "World"}
helloWorld[10] = "Space"

// panic: invalid array index 10 (out of bounds for 2-element array)
```

To get around this issue, Go provides slices.

Slices

Slices are more commonly used than arrays. They are flexible in that they do not have a fixed length:

```
nums := []int{1, 2, 3, 4, 5}
```

Slices can also be created with the `make` function:

```go
make([]string, initialLength)

// initialCapacity can be set to avoid allocations
make([]string, initialLength, initialCapacity)
```

You can also create a slice by *slicing* an existing array:

```go
nums := [6]int{1, 2, 3, 4, 5, 6}

// take elements 1 - 4 from `nums`
s := nums[1:4]

fmt.Println(s)
// => [2 3 4]
```

You can omit the start or end index when slicing an array, so for this array:

```go
var a [10]int
```

These expressions are equivalent:

```go
a[:]
a[0:]
a[:10]
a[0:10]
```

Modifying the elements of a slice will also modify its underlying array:

```go
nums := [6]int{1, 2, 3, 4, 5, 6}

s := nums[0:4]

// modify s
s[0] = 999

// which also modifies nums
fmt.Println(nums)
// => [999 2 3 4 5 6]
```

Slices are just a fancy way to manage an underlying array. You cannot increase a slice beyond its capacity. Doing so will cause a panic:

```go
nums := []int{1, 2, 3, 4, 5, 6}
nums[20] = 9

// panic: runtime error: index out of range [20]
// with length 6
```

However, Go has built-in functions to make slices feel like arrays. For examples, `append` can be used to add items to a slice:

```go
nums := []int{1, 2, 3, 4, 5}
nums = append(nums, 6)

fmt.Println(nums)
// => [1 2 3 4 5 6]
```

You can iterate over slices and arrays with `range`:

```go
names := []string{"john", "joe", "jessica"}
for index, name := range names {
  fmt.Println(index, name)
}

// 0 john
// 1 joe
// 2 jessica
```

You can use the blank identifier to omit the index, or the value from `range`:

```go
for _, name := range names { ... }
for index, _ := range names { ... }
```

If you only want the index, you can omit the second variable entirely:

```
names := []string{"john", "joe", "jessica"}

for index := range names {
    fmt.Println(index)
}

// => 0 1 2
```

To understand the inner workings of slices and arrays in detail, check out: [Go Slices: Usage and Internals]()

Maps

Maps are like hashes in ruby or dictionaries in python. You create them with a *map literal*:

```
mymap := map[string]string{"name": "ibraheem", "city": "toronto"}
```

Or the built-in `make` function:

```
mymap := make(map[string]string)
```

You can set the value under a particular key:

```
mymap["key"] = "value"
```

And access it:

```
val := mymap["key"]
fmt.Println(val) // "value"
```

Or delete it:

```
delete(mymap, "key")
```

You can check whether a key is present:

```
x, ok := mymap["key"] // => "value", true
x, ok := mymap["doesnt-exist"] // => "", false
```

And iterate over a map's keys and values:

```
m := map[int]string{1: "one", 2: "two"}
for key, value := range m {
  fmt.Println(key, value)
}

// 1 "one"
// 2 "two"
```

Functions

`func` declares a function.

Here's a void function:

```
func greet() {
  fmt.Println("Hi there!")
}
```

And here's a function that returns an integer:

```
func fairDiceRoll() int {
  return 4
}
```

Functions can return multiple values:

```go
func oneTwoThree() (int, int, int) {
  return 1, 2, 3
}
```

They can also take specified arguments:

```go
func sayHello(name string) {
  fmt.Printf("hello %s", name)
}
```

Or an arbitrary number of arguments:

```go
func variadic(nums ...int) {
  fmt.Println(nums)
}
```

These are called *variadic functions*. They can be called just like regular functions, except you can pass in as many arguments as you want:

```go
variadic(1, 2, 3) // => [1 2 3]
```

If a function takes two or more arguments of the same type, you can group them together:

```go
func multiply(x, y int) int {
  return x * y
}
```

The `defer` statement defers the execution of a function until the surrounding function returns:

```go
defer fmt.Println("world")
fmt.Println("hello")
```

```
// "hello"
// "world"
```

Defer is often used with an anonymous function:

```
defer func() {
  println("I'm done!")
}()
```

Anonymous Functions

Go supports anonymous functions. Anonymous functions are useful when you want to define a function inline without having to name it:

```
one := func() int {
  return 1
}

fmt.Println(one()) // 1
```

Functions can take functions as arguments

```
func printNumber(getNum func() int) {
  num := getNum()
  fmt.Println(num)
}

printNumber(func() int {
  return 1
})

// => 1
```

A function can also return an anonymous function:

```
func getOne() func() int {
  return func() int {
    return 1
  }
}

one := getOne()
fmt.Println(one())

// => 1
```

Anonymous functions allow us to dynamically change what a function does at runtime:

```
doStuff := func() {
  fmt.Println("Doing stuff!")
}
doStuff()

doStuff = func() {
  fmt.Println("Doing other stuff.")
}
doStuff()

// => Doing stuff!
// => Doing other stuff.
```

Anonymous functions can form *closures*. A closure is an anonymous function that references a variable from outside itself:

```
n := 0
counter := func() int {
  n += 1
  return n
}
fmt.Println(counter())
fmt.Println(counter())
fmt.Println(counter())

// => 1
// => 2
// => 3
```

Note how `counter` has access to `n`, even though it was never passed as a parameter.

Pointers

A pointer holds the memory address of a value.

```go
// "p" is a pointer to an integer
var p *int
```

The `&` operator generates a pointer to its operand:

```go
i := 42
var p *int = &i // point to i
```

The `*` operator lets you access the pointer's underlying value:

```go
i := 42
p := &i      // point to i

fmt.Println(p) // 0xc000018050
fmt.Println(*p) // => 42
```

You can use `*` to modify the original value:

```go
i := 42

p := &i   // point to i
*p = 21   // modify i through the pointer

fmt.Println(i) // 21
```

Custom Types and Methods

You can create a custom type with the `type` keyword:

```go
type UserName string
```

This is called a *type definition*. It defines a new, distinct type that is based on the structure of an underlying type.

You can define *methods* on your custom types. Methods are different than functions in that they take a *reciever* and are called on an instance of a specific type. The reciever is specified before the method name, and the arguments are specified after:

```go
type MyString string

// A method called `Print` with a reciever of type `MyString`
func (m MyString) Print() {
  fmt.Println(m)
}

var x MyString = "hello"
x.Print() // => "hello"
```

Methods receivers are copied by default, meaning their fields will not be mutated:

```go
func (m MyString) BecomeHello() {
  // m is a copy of the original MyString type
  // this modifies the copy
  m = "hello"
}

var x MyString = "not hello"
x.BecomeHello()

fmt.Println(x)
// => "not hello"
```

To mutate the receiver, use a pointer:

```go
func (m *MyString) BecomeHello() {
  *m = "hello"
}

var x MyString = "not hello"
x.BecomeHello()

println(x)
// => "hello"
```

Pointer receivers are used when you want to modify the value the receiver points to. They can also be used to avoid copying the value for each method call, which can be more efficient when dealing with large amounts of data.

Structs

Structs are a built-in Go type. You can declare one with the `struct` keyword. They can have any number of fields a specified type:

```
type MyStruct struct {
  x int
  y int
}
```

You can create a new struct with a *struct literals*:

```
s1 := MyStruct{ x: 1, y: 2 }

// a pointer to a struct
var s2 *MyStruct = &MyStruct{ x: 1, y: 2 }
```

You can omit the names of the fields and instead rely on the order:

```
s1 := MyStruct{ 1, 2 }
```

Struct fields are accessed using a dot:

```
s := MyStruct{ x: 1, y: 2 }
s.x = 1
```

If a field is not initialized, it defaults to the default value of its type:

```
s1 := MyStruct{}
fmt.Println(s1.x) // => 0
```

Struct fields can also be accessed and modified through a struct pointer just like a regular variable:

```
s := MyStruct{}

p := &s
(*p).x = 19 // modify `s` through `p`

fmt.Println(s.x) // => 19
```

Because this is so common, Go allows you to modify the underlying struct of a pointer without a `*`:

```
p := &s

// these are the same
(*p).x = 19
p.x = 19
```

You can declare methods on your own struct types just like with any other custom type:

```
type Number struct {
  value int
}

func (n Number) isStrictlyPositive() bool {
  return n.value >= 0
}
```

And use them as usual:

```
minusTwo := Number{
  value: -2,
}

minusTwo.isStrictlyPositive()
// => false
```

Interfaces

Interfaces are something multiple types have in common. They can contain any number of methods in their method set:

```
type Signed interface {
  isStrictlyNegative() bool
}
```

An interface is a type, so it can be used as a function argument, a struct field, or in the place of any other type:

```
type MyStruct struct {
  signed Signed
}
```

An interface value holds a value of the underlying concrete type. Calling a method on an interface value will call the method on its underlying type:

```
func SignedIsStrictlyNegative(s Signed) bool {
  // call a method on the underlying type of `s`
  return s.isStrictlyNegative()
}
```

A struct *implements* an interface if it has all of its methods:

```
type Number struct {
  value int
}

func (n *Number) isStrictlyNegative() bool {
  return n.value < 0
}
```

Now `*Number` implements the `Signed` interface.

So this works:

```
SignedIsStrictlyNegative(&Number{})
```

However, `Number` doesn't implement `Signed` because `isStrictlyNegative` is defined only on `*Number`. This is because `Number` and `*Number` are *different* types:

```
// this will not compile
SignedIsStrictlyNegative(Number{})

// but this will
SignedIsStrictlyNegative(&Number{})
```

Of course, this won't compile either:

```
// strings do not implement the Signed interface
SignedIsStrictlyNegative("a string")
```

`interface{}`

The interface type that specifies zero methods is known as the empty interface:

```
interface{} // literally anything
```

An empty interface may hold values of any type:

```
var x interface{} = Number{}
var y interface{} = "hello"
var z interface{} = 1
```

Values stored as an interface are type *erased*.

```
var x interface{} = Number{}

x.isStrictlyNegative() // error: x.isStrictlyNegative undefined (type
```

Because we know what `x` really is, we can use a type assertion:

```
var x interface{} = Number{}

n := x.(Number)

// now the compiler knows that n is a Number
// so this is fine
n.isStrictlyNegative()
```

If a type assertion fails (`x` doesn't really implement `Number`), it will trigger a panic:

```
var x interface{} = "not a number"

n := x.(Number)
// panic: interface conversion: interface {} is string, not Number
```

We can use the second return value of a type assertion to check if it was successful:

```
var x interface{} = "not a number"

// this code will not panic
n, ok := x.(Number)

// here, ok is false because the type assertion failed
if !ok { ... }
```

To perform multiple type assertions, we can use a *type switch*:

```
switch v := i.(type) {
case int:
  fmt.Println("i is an int")
case string:
  fmt.Println("i is an string")
default:
  fmt.Printf("I don't know about this type %T!", v)
}
```

Type Aliases

You can create an alias to another type:

```
type H = map[string]interface{}

// `map[string]interface{}` is a common pattern
// now we can use `H` as a short form
h := H{"one": 1, "two": 1}
```

This is different than a type definition:

```
// type definition
type H map[string]interface{}

// type alias
type H = map[string]interface{}
```

An alias declaration doesn't create a new type. It just introduces an alias name - an alternate spelling.

Struct Composition

Structs can be composed of one another through anonymous fields.

For example, here we have a struct called `Animal` that has a `Talk` method:

```
type Animal struct {
  feet int
}

func (a Animal) Talk() {
  fmt.Println("generic animal sound")
}
```

And a struct called `Cat` that embeds an `Animal`. All of `Animal`'s fields are promoted to `Cat`:

```
type Cat struct {
  Animal
}
```

You can instantiate a new `Cat`, setting the embedded `Animal's` fields:

```
cat := &Cat{ Animal{ feet: 4 } }
```

And use any `Animal` methods or fields:

```
cat.Talk() // => "generic animal sound"
cat.feet   // => 4
```

You can also override an embedded struct's methods:

```
func (c Cat) Talk() {
  println("meow")
}

cat.Talk() // => "meow"
```

To imitate a call to `super` as found in other languages, you can call the method through the embedded type directly:

```
cat.Animal.Talk() // "generic animal sound"
```

This behavior is not limited to struct fields. A struct can embed a primitive:

```
type MyStruct struct {
  string
  int
  bool
}
```

Or any other type, such as a pointer:

```
type Cat struct {
  *Animal
}
```

A collection (array, map, or slice):

```
// remember, this is a type *alias*
type Assignments = []string

type Homework struct {
  Assignments
}
```

Or an interface:

```
type MyInterface interface{}

type MyStruct struct {
  MyInterface
}
```

Interface Composition

Interfaces can also be composed of each other:

```
type Animal interface {
  Talk()
  Eat()
}

type Cat interface {
  Animal
  SomethingElse()
}
```

Here, Animal's method set is promoted to Cat. This is the equivalent to saying:

```
type Cat interface {
  Talk()
  Eat()
  SomethingElse()
}
```

Error Handling

Functions that can fail typically return an `error`, along with their regular return value:

```
file, err := os.Open("foo.txt")
```

Errors are values, so you can perform `nil` checks on them. You are going to be seeing *a lot* of this:

```
file, err := os.Open("foo.txt")
if err != nil {
  return err
}
DoMoreStuff(file)
```

You can create errors using the `errors` package:

```go
import "errors"

var err error = errors.New("I am an error")
println(err.Error()) // => "I am an error"
```

Packages often export common error values. You can perform equality checks between errors:

```go
import "errors"

err := makeDatabaseCall()
if errors.Is(err, database.ErrNotFound) {
  return 404
}
```

If code cannot continue because of a certain error, you can stop execution with `panic`:

```go
if err != nil {
    panic("OMG!!!")
}

// panic: OMG!!!
```

You can call `recover` to regain control of a panic. `recover` will return the value passed to panic:

```go
// if a panic occurs, this function will be called
defer func() {
  if r := recover(); r != nil {
    fmt.Println("Recovered from panic: ", r)
  }
}()

fmt.Println("Panicking")
panic("AAAHHH!!!")

// => Panicking
// => Recovered from panic:  AAAHHH!!!
```

Goroutines

Go is capable of concurrency through *goroutines*. A goroutine is a lightweight thread. To start a goroutine, you simple prefix a function call with the keyword `go`:

```
go DoSomething()
```

Goroutines are executed concurrently, not sequentially. For example, this code will take two seconds to complete:

```
time.Sleep(time.Second * 1)
time.Sleep(time.Second * 1)

// exec time: 2 seconds
```

But this code only takes one second, because both sleep calls are taking place at the same time!

```
go time.Sleep(time.Second * 1)
time.Sleep(time.Second * 1)

// exec time: 1 second
```

Goroutines are often used with anonymous functions:

```
go func() {
  fmt.Println("hello")
}()
```

Channels

Goroutines communicate through *channels*:

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

You can send values to a channel:

```
go func() { ch <- 1 }()
```

And receive values from a channel:

```
x := <-ch
```

Note that data flows in the direction of the arrow.

Here is an example of a simple multiplication function that communicates through channels:

```
func multiplyByTwo(num int, result chan <- int) {
  // calculate result
  sum := num * 2
  // and send it through the channel
  result <- sum
}
```

We can create a channel and pass it to the function:

```
result := make(chan int)
go multiplyByTwo(3, result)
```

The result channel now contains the number calculated by `multiplyByTwo`:

```
fmt.Println(<-result)
// => 6
```

Channels can be created with a limit. If too many values are sent to the channel, the channel will block:

```
limit := 1
ch := make(chan int, limit)

// this is fine
ch <- 1

// but now the channel is full
ch <- 2

// fatal error: all goroutines are asleep - deadlock!
```

Channels with a limit are called *buffered channels*.

You can also close a channel with the `close` method:

```
close(channel)
```

You can test whether a channel has been closed:

```
v, ok := <-ch
// if `ok` is true, the channel is still open
```

To receive all the values from a channel until it is closed, you can use `range`:

```
for value := range channel {
  fmt.Println(value)
}
```

Handling Multiple Channels

To handle communicating with multiple channels, you can use `select`:

```
func doubler(c, quit chan int) {
  x := 1
  for {
    select {
      // send x to channel and double it
      case c <- x:
        x += x
      // ... until a value is sent to "quit"
      case <- quit:
        fmt.Println("quit")
        return
    }
  }
}
```

Let's test our program out by creating a channel, passing it to `doubler`, and recieving the results:

```
func main() {
  // create the neccessary channels
  c := make(chan int)
  quit := make(chan int)

  go func() {
    // receive five values from the channel
    for i := 0; i < 5; i++ {
      println(<-c)
    }
    // and then quit
    quit <- 0
  }()

  // run doubler
  doubler(c, quit)
}
```

It works 🎉🎉🎉

```
1
2
4
8
16
quitting
```

And with that, we have hit 20 minutes estimated reading time. After reading this, you should be able to read most of the Go code you find online.

Thanks for reading!