Go

Go is a programming language that was originally designed to solve various problems in large-scale software development in the real world, initially within Google. It addresses slow program construction, out-of-control dependency management, complex code, and difficult cross-language construction.

Each language tries to address these problems in a different way, either by restricting the user or by being as flexible as possible. Go's team chose to tackle these problems by targeting modern engineering: removing the constraint of dealing with memory and making it simple to run parallel pieces of code.

Go was built for concurrency and networked servers, which explains the fast adoption of the language in software companies of all sizes in the past few years. Go was designed to improve productivity during a time when multicore networked machines and large codebases were becoming the norm.

> Go started in September 2007 when Robert Griesemer, Ken Thompson, and I began discussing a new language to address the engineering challenges we and our colleagues at Google were facing in our daily work.
> —Rob Pike, coauthor of Go

Go was designed with simplicity and productivity in mind, making it a natural fit for backend services, APIs, and modern cloud computing needs.

Learning Resources

What follows is a list of resources where you can learn Go.

- Go by example.
- Efective Go.

Code organization

Go programs are organized into **packages**. A package is a **collection of source files** in the **same directory** that are compiled together. Functions, types, variables, and constants defined in one source file are visible to all other source files within the same package.

> Go convention is to name the package the same as the directory that contains it.

A **module** is a **collection of** related Go **packages** that are released together. A Go repository typically contains only one module, located at the root of the repository. A file named `go.mod` there declares the **module path**. This module path is the **prefix** we need to write to import any package of that module. The module contains the packages in the directory containing its `go.mod` file as well as subdirectories of that directory, up to the next subdirectory containing another `go.mod` file (if any).

```
mkdir first
go mod init example/hello
```

In the previous block we are creating a directory to organize de code of our module. The name of the módulo will be `example/hello`.

Inside the `first` directory we would write the code of our packages. Usually we will create a directory for each package.

Running code

If we want to run our code we need to have an **entry point**. The entry point for a Go application will be the `main` function in the `main` package:

```
package "main"

import "fmt"

func main() {
  // Our code will be here.
  fmt.Print("Hello world!")
}
```

The `main` package is a special package. It defines a standalone executable program, not a library. The program execution begins by calling the `main` function of the `main` package. **We do not follow the naming convention of naming the package the same as the directory that contains it for the `main` package.**

Once we have a file with the `main` package and a `main` function we have three options to run our code:

1. Using `go run` command with the name of the file that contains the `main` function: `go run main.go`.
2. Using `go build` command to create an executable file and then running that file: `go build .` will create an exacutable file with the name of the module (in this case `hello`), then we can run it with `./hello.exe`.
3. Using `go install` command to create an executable file in the `$GOPATH/bin` directory and then running that file. In our case the path will be `C:/Users/<User>/go/bin/hello.exe`.

```
go run main.go
```

Visibility

Everything that starts with a capital letter is **exposed** and can be used outside the package where it is defined. Everything that starts with a lowercase letter is **unexposed** and can only be used inside the package where it is defined.

In any case, this only matters when we are using multiple packages. In a single package, everything is visible.

Basic syntax

- `;` at the end of the line is optional. The Go compiler automatically inserts semicolons at the end of lines.
- Code blocks are defined using `{}`.
- Tabs are used for indentation. Spaces are not allowed for indentation.

Variables

Go is a statically typed language, which means that the type of a variable is known at compile time. We can declare a variable using the `var` keyword followed by the variable name and type:

```
var name string
name = "Manuel"
```

Or we can declare and initialize a variable in the same line leaving the type to be inferred by the compiler:

```
name := "Manuel"
```

"Inferred by the compiler" means that the compiler will determine the type of the variable based on the value we assign to it.

Go allows for multiple variable declarations in the same line:

```
var x, y, z int = 1, 2, 3
```

Or via inference:

```
x, y, z := 1, 2, 3
```

Data types

Basic data types

Go have three different basic types: boolean, numeric, and string.

Boolean data types can be either `true` or `false`:

```
var bigger bool

bigger = 3 > 2 // true
```

Numeric types are further divided into integer and floating-point types:

- `int`: Integer type.
- `float32` and `float64`: Real number types single or double precision.

```
var i int

i = 10

var f float64

f = 3.14
```

Strings are a sequence of characters:

```
var str string

str = "Hello, World!"
```

In Go we don't usually have to specify the type of a variable, the compiler infers it from the value we assign to the variable:

```
bigger := 3 > 2 // bool

x := 5 + 5 // int

pi := 3.14 // float
```

Data structures

Objects in Go

Go does not have objects in the traditional sense. Instead, it has structs and interfaces.

We will see these concepts in more detail later, as we need them.

Flow control

Conditionals

Go has only two types of conditional statements: `if` and `switch`.

`if`

The conditional statements in Go are similar to those in other programming languages. The primary conditional statements are `if`, `else if`, and `else`.

```
// Code before...

x = 10

if x > 0 {
    fmt.Println("x is positive")
} else if x < 0 {
    fmt.Println("x is negative")
} else {
    fmt.Println("x is zero")
}

// Code after...
```

`switch`

The second conditional statement is `switch`, which is used to select one of many code blocks to be executed.

```
import (
    "fmt"
    "runtime"
)
// Code before...

l := "linux"

switch os := runtime.GOOS; os { // We can initialize a variable in the switch statement.
case "darwin":
    fmt.Println("OS X.")
    // break is implicit.
case l: // We can use variables as case values.
    fmt.Println("Linux.")
    // break is implicit.
default:
    // freebsd, openbsd,
    // plan9, windows...
    fmt.Printf("%s.\n", os)
}

// Code after...
```

Loops

Go has only one looping construct, the `for` loop. This `for` loop can be used in several ways.

C-style

This for loop works similarly to the Java for loop. We have an initialization statement, a condition, and a post statement.

- Initialization statement: executed before the first iteration `i := 0`.
- Condition: evaluated before each iteration, `i < 10`, to determine if the loop should continue.
- Post statement: executed at the end of each iteration `++i`.

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

While-style

This loop will work like a while loop in other languages. We only have a condition that is evaluated before each iteration. The loop will continue as long as the condition is true.

```
condition := true
i := 0

for condition {
    fmt.Println(i)
    condition = i < 10
    ++i
}
```

Loop over a collection

This last versions of the for loop is similar to the **for-each** loop in Java and other languages. We can use it to iterate over arrays, slices, maps, strings, and channels. (We will be seeing these data structures later).

```
// We declare a collection (a map) with pairs of key-value.
m := map[int]string{
    1: "one",
    2: "two",
    3: "three",
}

for key, value := range collection {
    fmt.Println("key:", key, "=>", "value: ", value)
}
```

Functions

In Go, functions are **first-class citizens**. This means that functions can be assigned to variables, passed as arguments to other functions, and returned from other functions.

To define a function in Go, we use the `func` keyword followed by the function name, parameters, and return type.

```
func addition(a int, b int) int {
    return a + b
}

// We can call the function like this:
result := addition(3, 4)

fmt.Printf("The result is: %d", result)
```

We can also assign functions to variables (and pass them as arguments to other functions):

```
// We can assign a function to a variable.
add := addition

// And we can call the function using the variable.
result := add(3, 4)

fmt.Printf("The result is: %d", result)
```

Function in go can return multiple values:

```
func swap(a, b string) (string, string) {
    return b, a
}
a, b := "world", "hello"
fmt.Println(a, b) // world hello

a, b := swap(a, b)

fmt.Println(a, b) // hello world
```

To control for errors, Go uses a common pattern where functions return a value and an error. If the function is successful, the error will be `nil`. If there is an error, the error will contain an error message.

```
value, err := strconv.ParseBool("true")

if err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("Value:", value)
}
```

The previous example shows how to convert a string to a boolean. The `ParseBool` function returns two values: the parsed boolean value and an error value. If the conversion is successful, the error value will be `nil`. If the conversion fails, the error value will contain an error message.

If we want to ignore one of the returned values, we can use the blank identifier `_`:

```
value, _ := strconv.ParseBool("true")
```

Function arguments

Arguments in Go are passed by value. This means that when we pass a variable to a function, we are passing a **copy of the variable**, not the original variable.

```
// previos code...

a := 10

func modify(x int) {
    x = 20
}

modify(a)

fmt.Println(a) // 10
```

If we need to modify the original variable, we can pass a **pointer to the variable**. A pointer is a reference to a memory location where the variable is stored. We can create a pointer using the `&` operator and dereference it using the `*` operator.

```
// previous code...

a := 10

func modify(x *int) {
    *x = 20
}

modify(&a)

fmt.Println(a) // 20
```

Function parameters vs function arguments

Function parameters are the variables listed in a function's definition.
Function arguments are the actual values passed to the function when it is called.

In the following code `x` and `y` are function parameters:

```
func add(x int, y int) int {
    return x + y
}
```

And here `a` and `b` are function (invocation) arguments:

```
a, b := 10, 20

z := add(a, b)
```