Learning Objectives

Learners will be able to...

- **install Go**
- **compile and run a Go program**
- **create a Go Project structure**

info

Make Sure You Know

Python or other modern programming language.

Limitations

To keep versions consistent, learners will not actually be installing Go.

Hello Go!

Take a look at the simple Go program on the left.

A few boilerplate pieces of a `.go` file include:
*
* In Go, code must be defined in a specific `package`.
* The `main` package defines the executable file and must contain the main function.
*
* In Go, you `import` the packages you need.
* `fmt` implements formatted I/O functions (similar to `printf` or f-strings). See fmt docs for more.
*
* The `main` function is called when the Go program is executed.

Click the hyperlinks above to highlight the section of the code being described.

Installation

In Codio, Go has already been installed for you! To install on your local machine, we recommend following the installation instructions from the official Go site.

In order to make sure that the Go is installed, you can run the command in the terminal:

```
go version
```

If you don’t feel like typing or copy-pasting the above command, you can click the button below to run the above command.

The output will state the installed version of Go.

Running Go

To compile, you can use the command:

```
go build -o hello hello.go
```

Then you can run the application using:

```
./hello
```

You can also run the program without compiling it into an executable file with the following command:

```
go run hello.go
```

History and Features of Go

Go was designed by Robert Griesemer, Rob Pike and Ken Thompson at Google in 2007, followed by its first stable release in 2011.

Go addresses the needs of modern technology such as mutli-core, networked machines and large codebases with C-like runtime efficiency with high performance networking and multiprocessing support. However, unlike C-family languages, Go has a concise and simple syntax which emphasizes readability (like Python).

Features of Go

The main features of Go are it is/has:

- **compiled**
    - Source code is converted by the compiler into an executable file, making Go program execution faster than interpreted programming languages (e.g. Python, JavaScript).
    - Compiling a program in Go is quite fast and takes less time than, for example, in C.
- **statically typed**

  ```
  golang-hide-clipboard     var number int //declaration     number
  = 12 //initialization     number = "str" // type mismatch error
  will be thrown
  ```

    - Variable types, parameters, and return values of a function are set at the program compilation stage, eliminating many compile-time errors.
    - More relevant completion and error reporting is feasible.
    - Code is more readable and understandable.
- **automatic type inference**

  ```
  golang-hide-clipboard     number := 12
  fmt.Println(reflect.TypeOf(number)) // int     str := "string"
  fmt.Println(reflect.TypeOf(str)) // string
  ```

    - If the variable is initialized when declared, then the type of the assigned expression becomes the type.

definition

### Duck Typing

Automatic type inference is sometimes called **duck typing**.

The term comes from the English “duck test”:
> If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck.

- **a garbage collector**
  - Unlike C and C++ where the programmer has to keep track of memory usage and clean it up manually, Go has a special process that periodically frees memory by deleting unnecessary objects.
- **support for multithreading and parallel programming**
  ~~~
  ```golang-hide-clipboard
  func main() {
  go exampleFunc()
  go exampleFunc()
  }

  func exampleFunc() {
  // ...
  }
  ```
  ~~~
  - Parallel programming in Go is natively supported by **goroutines**, lightweight threads that consume little memory. A goroutine kicks off a separate thread when you use the `go` keyword before calling the function.
  - **Channels** are used to communicate between goroutines.

  ```
  exampleChannel <- sentValue
  receivedValue := <-exampleChannel
  ```
- **pointer support**
  ```
  golang-hide-clipboard     var name string = "Codio"  // define a
  variable     var ptr *string                // define a pointer
  ptr = &name                     // the pointer gets the address of
  the variable     fmt.Println(ptr)              // 0xc000114020
  fmt.Println(*ptr)              // Codio
  ```
  - The arguments in the Go function are passed by value (with some exceptions), but when you need to change a variable directly, you

    can pass the address of the variable to the function.
  - Go also has pointer arithmetic, using the standard `unsafe` module, but as the name implies, this is not the recommended way to work with pointers.

- **cross-platform**

  - Go allows you to compile an application for different operating systems and architectures. To compile an application for a specific architecture and operating system, it is enough to specify the `GOOS` and `GOARCH` environment variables. For example, to get a binary file for execution in the Windows system on the AMD64 architecture, you should use the command:

  ```
  GOOS=windows GOARCH=amd64 go build main.go
  ```

Makefile

As mentioned on the previous pages, Go is a compiled language. Similar to C and C++, Go supports `makefiles` to simplify program building, testing, and execution.

important

## No `make` on Windows

`make` is a UNIX toolchain, so it is not natively supported by Windows. You can install a Linux environment inside Windows to use `make` commands.

Writing `Makefiles`

Take a look at the example `Makefile` on the top left:
1. The structure of the file is:

```
bash-hide-clipboard     target: prerequisites          command
...         command
```

* Common **targets** are:
*
*
*
* test
* all
* **Prerequisites** are targets that are run before the listed commands (e.g. the `run` target starts by calling the `build` target).
* **Commands** are Bash commands like you would run on the terminal.

1. Variables work like in Bash
   - Syntax to set variable:

     ```
     bash-hide-clipboard      variable_name=value
     ```

     ■
   - How to access a variable

     ```
     bash-hide-clipboard      ${<variable_name>}
     ```

     ■
2. You can build binaries for different OSes using `GOARCH` and `GOOS`:

   ```
   bash-hide-clipboard     GOARCH=amd64 GOOS=darwin go build -o
   hello-darwin hello.go     GOARCH=amd64 GOOS=linux go build -o
   hello-linux hello.go     GOARCH=amd64 GOOS=windows go build -o
   ```

```
hello-windows.exe hello.go
```

Running a `Makefile`

To run a specific set of commands from your `Makefile`, you would use:

```
make target_name
```

info

What happens when you run `make`?

You do not have to specify a target after the `make` keyword – there is a **default goal** or target.

The default target is the first one listed in the file (but can be overridden using the `.DEFAULT_GOAL` variable). This means if you are going to be running one target more than the others, you should list it first in your make file.

Read the manual for more details

Try it out!

Run each of the following commands in the terminal on the bottom left:
- `make`
- `make build`
- `make run`
- `make clean`

Project Structure

The two main folders in a Go project structure are `internal` and `cmd`.

### /internal

`internal` holds the main application code. This code cannot be used in other applications and libraries.

```
├── internal/
│   ├── app/
│   │   └── app.go
```

Typically, you will have a file called `internal/app/app.go` which has a `Run()` method called from the `main` function (see Example `main.go` file below).

> **`internal` or `pkg`?**
>
> Some application code might live in the `pkg` directory as an indication that this code *is* safe for external use in other applications or libraries.

### /cmd

`cmd` indicates the main application of the project.

The `cmd` directory should contain a directory whose name matches the name of the executable file that will be built (e.g. `cmd/myappname`).

It is good practice for files with a main function to import and call code from the `/internal` and `/pkg` directories.

Example file `/cmd/myappname/main.go`:

```
package main

import (
    "myproject/internal/app"
)

func main() {
    if err := app.Run(); err != nil {
        log.Fatalf("error: %s\n", err.Error())
    }
}
```

There is a more full project structure discussed below, but the `internal` and `cmd` folders are the two directories you will see in all non-trivial Go applications. For some examples, take a look at the source code for:
* the Go website
* the Package section of the Go website
* the Go playground on the Go website

Full Project Structure

As your Go applications grow in complexity, you might need more than the `internal` and `cmd` directories.

Below is a recommended project structure:

```
myproject/
├── api/
├── assets/
├── build/
├── cmd/
│   └── myappname/
│       └── main.go
├── configs/
├── deploy/
├── docs/
├── internal/
│   ├── app/
│   │   └── app.go
│   └── utils/
│       └── utils.go
├── pkg/
├── scripts/
├── test/
├── tools/
├── vendor/
├── website/
└── README.md
```

Here is a description of the directories listed above:

Directory	Description
`/api`	OpenAPI/Swagger specs, JSON schema files, protocol definition files
`/assets`	Assets (e.g. images, fonts)
`/build`	Build configurations (e.g. cloud/container/OS package configurations and CI configurations and scripts)
`/configs`	Application configuration files (e.g. `confd` or `consul-template`)
`/deploy` or `/deployment`	Deployment configurations such as for containers (e.g. docker-compose, kubernetes/helm, terraform)
`/docs`	Documentation for the project (e.g. design docs, user-facing docs)
`/pkg`	Code that can be safely used by others
`/scripts`	Scripts for building, installing, analyzing, etc.
`/test`	Test apps and test data
`/tools`	Tools supporting the project
`/vendor`	Application dependencies, typically managed by the `go mod` feature
`/web` or `/website`	Web-specific components (e.g. static web assets, server side templates)

important

### /src

Note that **there is no `/src` folder at the main project level**. This is common in other languages such as Java and you will sometimes see Go projects with one because the developer is carrying over the pattern - but it is not best practice in Go.

Visit golang-standards/project-layout for more details and examples.