Go Application Structure

Ambush Journey Program

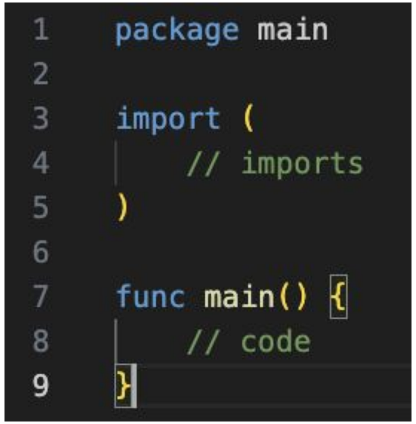
Ambush

1. The basic structure of a Go program
2. Packages and modules
3. Folder structure and visibility

The basic structure of a Go program

The basic structure

Any **executable** Go program should have the base structure shown below:



```go
package main

import (
    // imports
)

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

- Package definition
- Imports*
- Main function

The basic structure

A Go file can have any of the following:

- Variables
- Functions
- Type declarations
- Method declarations

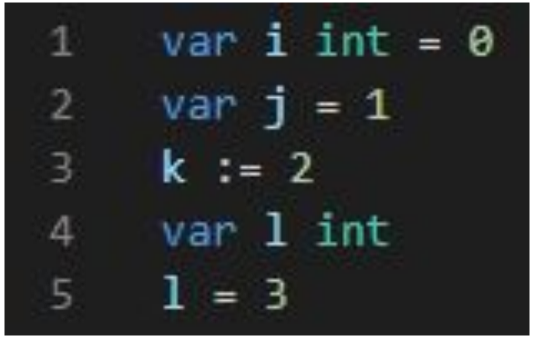
The basic structure

- Semicolons are not necessary at line (instruction) endings; line breaks automatically separate the instructions
- Go is **strongly** and **statically** typed
- Code blocks are represented by brackets

Variables

Variables in Go can be defined used either of the syntax options shown below

- If the declaration + assignment operator is used, the type of the variable doesn't need to be specified

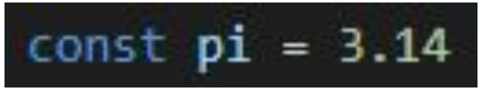


```go
var i int = 0
var j = 1
k := 2
var l int
l = 3
```

Constants

Besides variables, programmers can also define constants

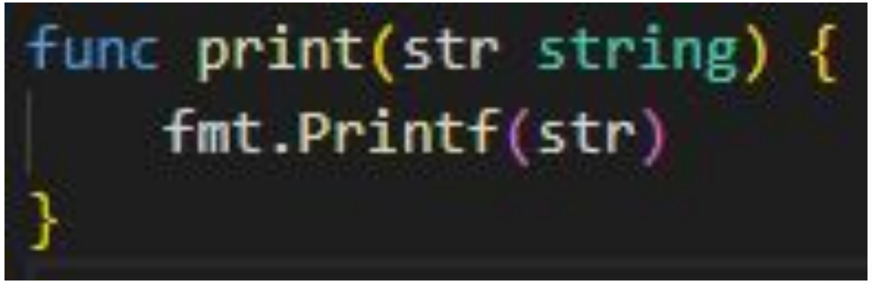
- Which can only be *boolean*, *numbers* or *strings*
- Constants are not stored in memory; instead, compiler copies those values and replaces them in every reference



```
const pi = 3.14
```

Functions

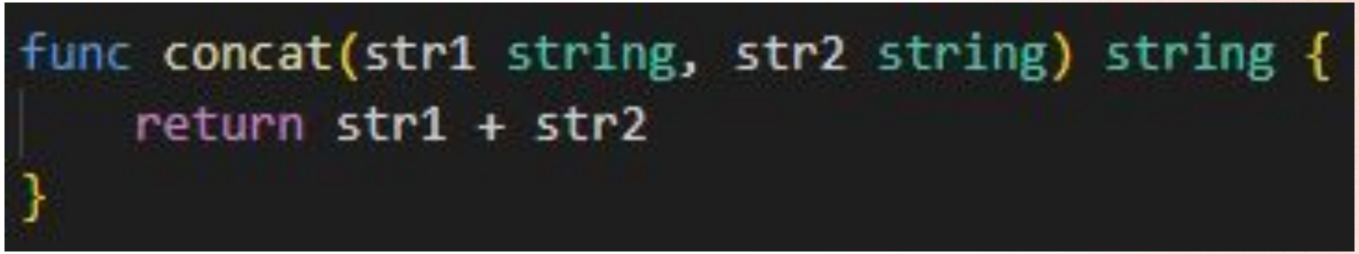
Functions in Go can be defined using the `func` keyword, as follows



```
func print(str string) {
    fmt.Printf(str)
}
```

Functions

If a function is supposed to return something, the type of the return value must be specified



```
func concat(str1 string, str2 string) string {
    return str1 + str2
}
```

Data Types

Go has the following basic data types:

- `string`
- `int, int8, int16, uint,` …
- `float32, float64`
- `bool`

And also the following operators: `==, !=, <, >, <=, >=, &&, ||, !`

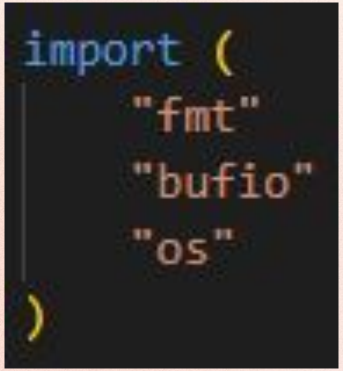
Practice

1. Modify the Hello World program you made to contain a variable with your name, and then print “Hello, <name>” where “name” contains that value
2. Write a function that takes two integers as parameters and returns the sum of those two integers. Print this in the console as well.

Packages and modules

Packages

- A package is a set of files with Go code in the same **folder** or **URL**
- External packages can be imported into the program using the `import` instruction
- Multiple packages can be imported by invoking `import` with their names separated by line breaks



```
import (
	"fmt"
	"bufio"
	"os"
)
```

Packages

- Go has a **standard library**, which consists of packages that are automatically shipped with the Go installation and don't need to be manually installed before importing to your project
    - Go Standard Library

Modules

- A module is a group of packages
- Our project is also a module
- It contains a `go.mod` file which has all the settings for that module
- It may also contain a `go.sum` file, which stores dependency checksums for that module
- A module can be initialized with `go mod init <name>`, where name is the name of that module

Modules

- It contains a `go.mod` file which has all the settings for that module
  - This file defines the module's path, which is used when it's imported by other projects
  - It also contains all the dependencies (packages that are used in the project) needed for the project to run
  - Similar to `package.json` when working with JS projects
    - But more simple

Modules

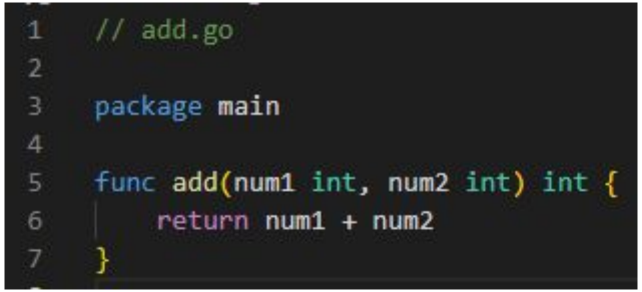
- It may also contain a `go.sum` file, which stores dependency checksums for that module
    - The checksums are used to ensure that each **direct** and **indirect** dependency on the project is already present in your Go installation, and doesn't need to be installed in subsequent runs of the program
    - Managed by the Go tools, generated after a `go mod tidy`

Modules

- With a module, you should be able to run the project without specifying the file name to `go run`.
    - Keep in mind it will always look for `main.go`
- Packages with the same name under a module **share** their resources
- Having a module allows you to add external packages with the `go get` command
    - We will practice this more in the future

Modules

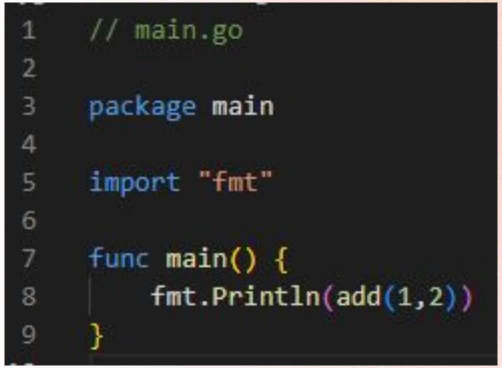
- Packages with the same name under a module **share** their resources



```go
// add.go

package main

func add(num1 int, num2 int) int {
	return num1 + num2
}
```



```go
// main.go

package main

import "fmt"

func main() {
	fmt.Println(add(1,2))
}
```

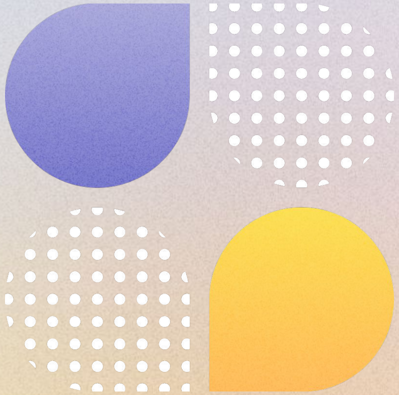
Workspaces

- Special type of concept, where an app contains multiple modules
- New from version 1.18 (March 2022)
- Can be manipulated through the command line interface in a very similar way to modules
    - `go work init`
- Although it’s not used often today in a microservices architecture, it’s still sometimes used in a monolithic application

Practice

1. Take the program you made to sum two integers and print it on the console and transform it into a module. Make it run using only the following command: `go run .`
2. Move the function to a separate file and invoke it from the main function (file).

Folder structure and visibility

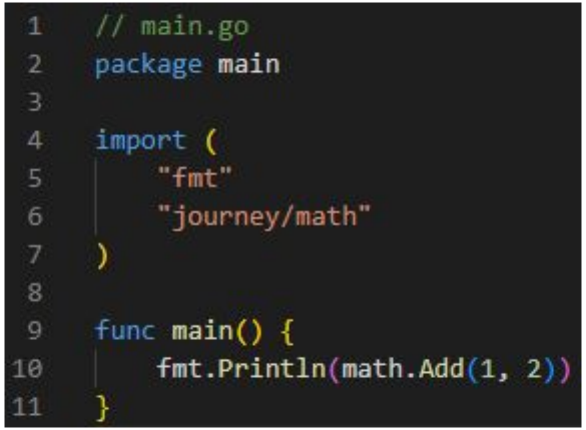


Package visibility

- Any method that starts with a capital letter is a public method
  - `fmt.Println`
- In order to import a local package into your project, you must add the module's name to the prefix of the import
  - To access methods or objects from it, use the name of the package as prefix as well

Package visibility

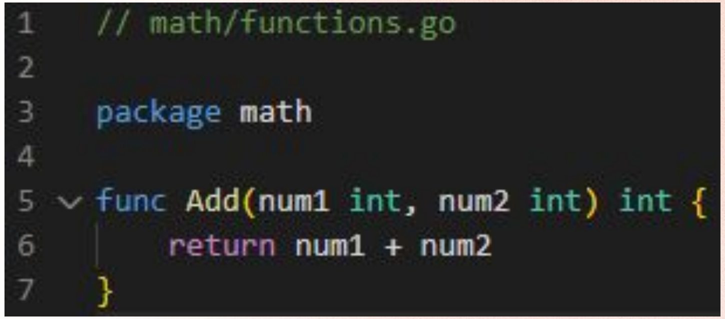
- In order to import a local package into your project, you must add the module's name to the prefix of the import



```go
// main.go
package main

import (
	"fmt"
	"journey/math"
)

func main() {
	fmt.Println(math.Add(1, 2))
}
```



```go
// math/functions.go

package math

func Add(num1 int, num2 int) int {
	return num1 + num2
}
```

Folder structure

- Any executable Go module must contain a `main.go` file at the root of the project
- Local packages can be placed inside a common folder with its name, where multiple files can be created according to roles or necessities
    - All must contain the same package name at the top of the file

Folder structure

- When a project (module) is hosted at github, and there's an intent to import it into other projects, the module name should follow this template:

```
module github.com/<user>/<module_name>
```

- It can later be installed into other projects with the following line on the CLI:

```
go get github.com/<user>/<module_name>
```

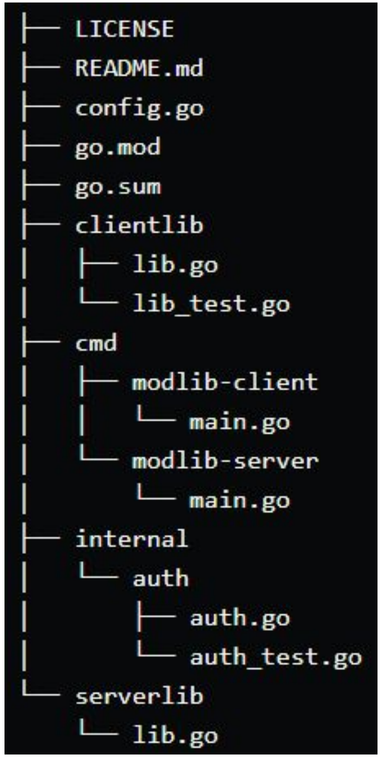
Folder structure

- In order to structure a module and its local packages, they should be separated according to roles, responsibilities and objects
  - This is subjective, but there are common guidelines and patterns

Folder structure

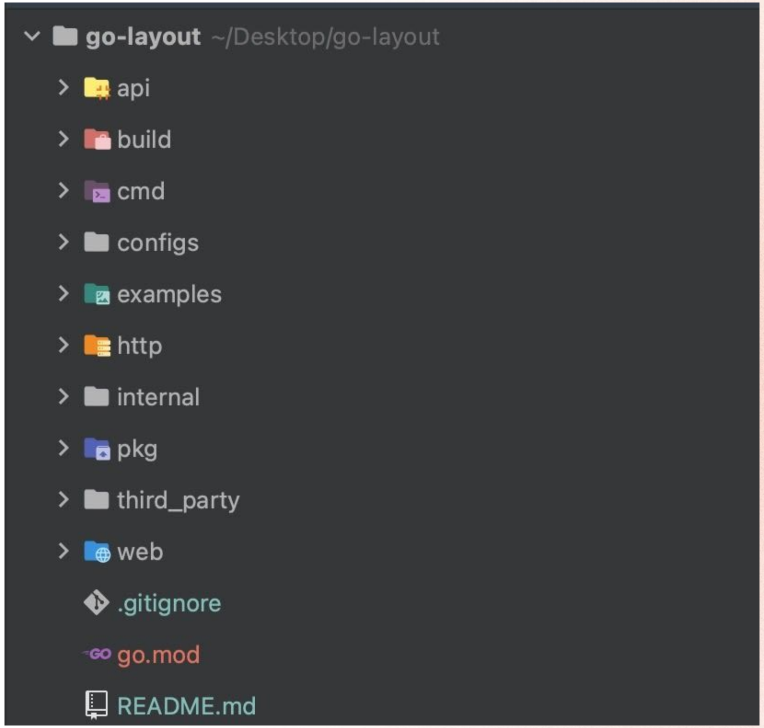
- The following folders at the root directory of the module are very common:
  - **/cmd** - for application entry points (and main functions)
  - **/pkg** - for code and services that may be exposed (consumed)
  - **/internal** - for code and packages that should be accessible only to the project itself
- Other folders may be created according to necessities and complexity

Folder structure



```
├── LICENSE
├── README.md
├── config.go
├── go.mod
├── go.sum
├── clientlib
│   ├── lib.go
│   └── lib_test.go
├── cmd
│   ├── modlib-client
│   │   └── main.go
│   └── modlib-server
│       └── main.go
├── internal
│   └── auth
│       ├── auth.go
│       └── auth_test.go
└── serverlib
    └── lib.go
```

Source: dev.to (Go - Project Structure and Guidelines)



```
go-layout ~/Desktop/go-layout
  > api
  > build
  > cmd
  > configs
  > examples
  > http
  > internal
  > pkg
  > third_party
  > web
    .gitignore
    go.mod
    README.md
```

Source: dev.to (A practical approach to structuring Golang applications)

Folder structure

- A more complex organization is defined at the golang-standards repository, which contains directories for API swagger files, web-specific components, build scripts, test files, deployment files, among others.

Go Application Structure

Ambush Journey Program

Ambush