Learning Objectives

Learners will be able to...

- **create packages in Go and import them**
- **create and import modules**
- **manage dependencies and versions**

info

Make Sure You Know

Python or other modern programming language.

Creating a Package

Go programs are organized into **packages** or collections of source files in a single directory that are compiled together.

> info
>
> ### ### Types of Packages
>
> There are two types of packages:
> 1. **executable**
> 2. **libraries**
>
> To create an executable package, the package must be named `main` and contain a `main` function. All other packages are libraries.

Packages in Go are similar to modules in Python or packages in Java in that they are **namespaces** (i.e. functions, types, variables, and constants defined in one source file are visible to all other source files in the same package). You can think of the Go compiler concatenating all the source files within that package – similar to how python modules are defined by the file name.

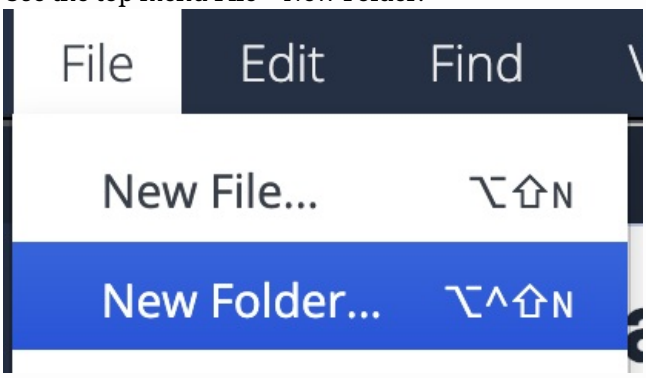
Create a package directory

The name of the package usually matches the name of the directory.

> xdiscipline
>
> ### ## Code Style: Package names
>
> Package names should be short and contain only lowercase letters.
>
> Read more about package names on the Go blog.

Try it out by creating a folder named `user`.

Use the top menu File > New Folder.



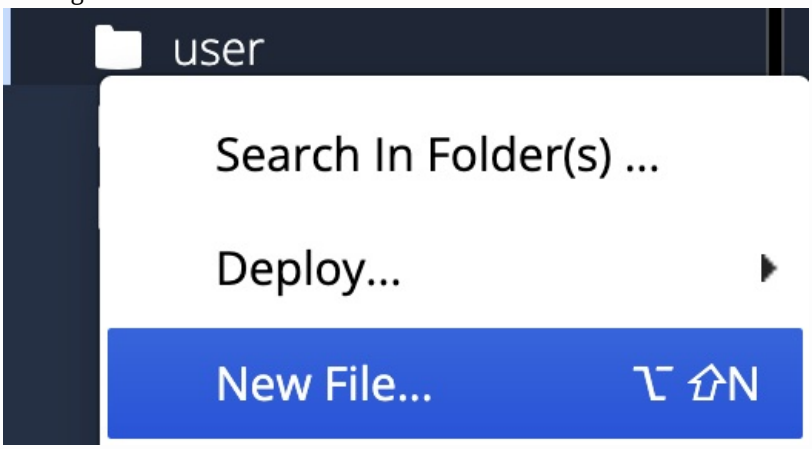
All files for the `user` package should be inside the `user` directory.

Use `package` keyword in source files

All source files located within a package (and therefore within the package directory) will start with a `package` statement specifying which package it is a part of.

Create your first source file `Account.go`

You can use the top menu File > New File to create a new file - just be sure to check you are creating the file **inside** your `user` folder. Alternatively, you can right-click on the `user` folder in the file tree and select New File.



Add the following code into your `Account.go` file:

```
package user

type Account struct {
    FirstName string
    LastName  string
  ID string
}

// since the function is exportable it can be used in other
packages
func GetUserAccount(id string) (Account, error) {
  ...
}

// the function is not exported, it can only be used in the
users package
func validateID(id string) (bool, error) {
  ...
}
```

Note that:

*
*
*

Importing Packages

If there are packages with ready-made functionality that we want to use, we can `import` them into our Go file.

Specifying import path

The **import path** is the string used to import the package:

```
import "github.com/user/department"
```

The import path of a package is the path of its module, concatenated to its subdirectory within the module. The management of modules will be discussed on the following pages.

Packages in the **standard library** (such as `fmt`, `io` and others) do not have a module path prefix.

```
import "fmt"
```

info

Standard libraries

A complete list of the Go standard library packages can be found at https://golang.org/pkg/.

Often, programs connect several external packages at once. In this case, you can import each package in sequence:

```
import "fmt"
import "io"
import "github.com/user/department"
```

Or, to shorten the package import definition, you can enclose all packages in parentheses:

```
import (
  "fmt"
  "io"
  "github.com/user/department"
)
```

Naming, ignoring, and other imports

There are times when you need to use **named or aliased imports**. This may be required to avoid collisions:

```
import f "fmt"

f.Println("Hello")
```

Now, when using the package, you call `f` instead of `fmt`.

If you don't want to have to put even a shortened version of the package, you can use a dot or `.` to importing the entire package into the same namespace:

```
import . "fmt"

Println("Hello")
```

This has some drawbacks associated with it since it mixes up the packaging and can cause namespace collisions.

If an import is temporarily not being used, you can preface it with _ so Go ignores the import (preventing the unused import error):

```
import _ "fmt"
```

Creating Modules

Modules are a collection of related Go packages that can be published for use in other projects. These are similar to packages in python (e.g. pandas, matplotlib).

Starting with Go 1.16, to use external modules, you must first define your own module.

Create a Module

First, create a directory where our module will be located. For this example, let's say our module will be called `myapp`.

Create and navigate into `myapp` directory:

```
mkdir myapp
cd myapp
```

To initialize a module, use the `go mod init` command, optionally followed by the name of the module.

Initialize the `myapp` module using `go mod init` command:

```
go mod init myapp
```

If the module path is not passed to the `go mod init` command, it initializes and writes a new go.mod file in the current directory.

Try initializing the `myapp` module again:

```
go mod init
```

You should see an error because the go.mod file must not already exist.

`go.mod` file

Once you have successfully created a module, a `go.mod` file will appear.

Open your `go.mod` file by clicking on it in the file tree.

You can think of the `go.mod` file similar to the `__init.py__` file for defining python packages.

A simple `go.mod` file will look something like:

```
module myapp

go 1.21.0
```

The first line is the `module` directive which defines the main module's path. The second line is the `go` directive which indicates the minimum version of Go required to use this module (mandatory as of Go 1.21).

The other directives you might see include:
Directive	Description
———	———–
`require`	Declares a minimum required version of a given module dependency.
`exclude`	Prevents a module version from being loaded by the go command.
`replace`	Replaces the contents of a specific version of a module, or all versions of a module, with contents found elsewhere (e.g. a local file instead of a repository).
`retract`	Indicates that a version or range of versions of the module should not be depended upon.
`toolchain`	Declares a suggested Go toolchain to use with a module which cannot be less than the required Go version declared in the `go` directive.

The toolchain directive only has an effect on main modules where the default toolchain's version is less than the suggested toolchain's version. |

Click here to read more about the go.mod file and it's directives in the Go docs.

The `go.mod` file is designed to be human readable and machine writable – meaning you don't typically edit the file directly. The `go` command provides several subcommands that change go.mod files (e.g. `go mod init`, `go get`, `go mod edit`, `go mod tidy`).

Managing Modules

In order to use an external module, you can use the `go get` command to `require` it in the `go.mod` file.

For example, we want to include `google.golang.org/grpc`.

Run the command:

```
go get google.golang.org/grpc
```

After that, we can look at the contents of `go.mod` again (by clicking on it in the filetree to open it) and make sure that the necessary dependencies are added:

```
module myapp

go 1.21.0

require (
    github.com/golang/protobuf v1.5.3 // indirect
    golang.org/x/net v0.12.0 // indirect
    golang.org/x/sys v0.10.0 // indirect
    golang.org/x/text v0.11.0 // indirect
    google.golang.org/genproto/googleapis/rpc v0.0.0-
20230711160842-782d3b101e98 // indirect
    google.golang.org/grpc v1.58.2 // indirect
    google.golang.org/protobuf v1.31.0 // indirect
)
```

The `require()` directive has been added containing all the dependencies that are needed to work with the “google.golang.org/grpc” package.

A `go.sum` file is also created which contains the checksum for the included packages.

Now we can use this module in our project by including it using the `import` directive.

```
import "google.golang.org/grpc"
```

info

Clean up dependencies with `go tidy`

Instead of `go get`, you can `import` the module in the application code and then execute `go mod tidy`. This command will update and add dependencies in the project.

The goal of `go mod tidy` is also to add any dependencies needed for other combinations of OS, architecture, and build tags so it should be run before every release.

Adding a specific version module

If necessary, you can specify a specific version of the modul using the `@` symbol:

```
go get github.com/userName/module@v0.0.1 // by tag
go get github.com/userName/module@master // by branch name
go get github.com/userName/module@fad23rf // by commit
go get github.com/userName/module@latest // latest version by
tags
```

Alternatively, you can directly edit the `go.mod` file.

info

List module dependencies with `go list -m`

Lists all modules that are dependencies of the current module:

```
go list -m all
```

Use the `-u` flag to get information on the latest versions of the modules:

```
go list -m -u all
```

You can also show the latest version available for a specific module:

```
go list -m -u example.com/userName/module
```

Replacing a dependency

In addition to a specific version, you can specify where the module should be retrieved from. For example, you can specify that the required module code is on the same local drive as the code that requires it, which is useful if you are developing your own module and want to test it.

To use a local copy of a module, a `replace` directive is added in your `go.mod` file after the module's existing `require` directive:

```
require example.com/userName/module v0.0.0

replace example.com/userName/module v0.0.0 => ../module
```

To set up a require/replace pair, use the `go mod edit` and `go get` commands to ensure that the requirements described in the file remain consistent:

```
go mod edit -
replace=example.com/userName/module@v0.0.0=../module
go get example.com/userName/module@v0.0.0
```

For another example, you can replace a module with your fork of the module:

```
replace example.com/userName/module v0.0.0 =>
example.com/myfork/module v0.0.0-fixed
```

Removing a Dependency

If for some reason you no longer use a module, you can stop tracking it. To do this, use the `go mod tidy` command which will remove all unused dependencies.

To explicitly remove a module, you can use `go get` and set the version to `none`:

```
go get example.com/userName/module@none
```

The `go get` command will also downgrade or remove other dependencies that depend on the removed module.