Custom Packages and Visibility

This lesson provides a detailed description on how to create and successfully run a custom package in any Go IDE.

Packages are the primary means in Go for organizing and compiling code. A lot of basic information about packages has already been given in Chapter 2, most notably the *Visibility rule*. Now we will see concrete examples of the use of packages that you write yourself. By **custom packages**, we mean self-written packages or packages otherwise external to the standard library.

When writing your packages, use short, single-word, lowercase names without _ for the filename(s).

The `go` command

The `go` command is used as a uniform (OS independent) tool to fetch, build, and install Go packages and applications. Everything from how to get a package, how to compile, link and test it, is deduced by looking at the source code to find dependencies and determine build conditions. A single environment variable: GOPATH (which is discussed in Appendix) tells the `go` command (and other related tools) where to find and install the Go packages on your system.

This command has the same syntax as the system's PATH environment variable, so a GOPATH is a list of absolute paths, separated by *colons* (:) on Unix-like machines and by *semicolons* (;) on Windows machines.

Unix-like machines and by *semicolons* (;) on Windows machines.

On Unix a typical value could be:

```
GOPATH=/home/user/ext:/home/user/go_projects
```

On Windows a typical value could be:

```
GOPATH=d:\go_projects;e:\go_projects
```

Each path in the list (in this case e:\go_projects or/home/user/go_projects) specifies the location of a workspace.

A workspace contains Go source files and their associated package objects, and command executables. It has a prescribed structure of three subdirectories:

• **src** contains a structure of subdirectories each containing Go source files (.go)

• **pkg** contains installed and compiled package objects (.a)

• **bin** contains executables (.out or .exe)

> **Note**: There is no specific need to have several workspaces; one workspace can often meet all needs. **GOPATH** should not normally contain **GOROOT** which is a folder where the Software Development Kit(SDK) for Go is downloaded on your systems.

Subdirectories of the **src** directory hold independent packages, and all source files (.go, .c, .h, and .s) in each subdirectory are elements of that subdirectory's package.

When building a program that imports the package `widget`, the go command looks for **src/pkg/widget** inside the Go *root*, and then, if the package source isn't found there, it searches for **src/widget** inside each workspace in order. Multiple workspaces can offer flexibility and convenience, but for now, we'll concern ourselves with only a single workspace.

Schematically:

```
workspace /src
              /pkg1

              /pkg2
              ...
          /pkg
              ...
          /bin
              ...
```

Example: A project containing a package

Here is a simple example as to how `go` command works for building/running and how the visibility works. Create a folder **e:\go_projects or $HOME/go_projects** as workspace.

For Windows, set the system variable GOPATH to the value **e:\go_projects** and check that **GOPATH\bin** is added to the PATH variable.

For Linux, do the same by adding the following lines to **$HOME/.profile** (or equivalent):

```
export GOPATH=$HOME/go_projects
export PATH=$PATH:$GOPATH/bin
```

Suppose we want to make a Go project named **book**. Inside the workspace directory, create a **src** folder, and within it a folder **book**. Our base import path in Go source files will then be **book/.** In __book, we want to create a package `pack1`. This can be downloaded from a subfolder **pack1** of the folder **book**. This subfolder contains a program **pack1.go**, which indicates it belongs to package `pack1` at its first line.

> **Remarks**: It is not necessary that the subfolder and the source file have the same name as the package. For the subfolder, it is considered good practice, but in most cases, the package will be spread across more than one source file. Our project could contain more than one package, but the way of working is the same for additional packages.

By convention, there is a close relationship between subdirectories and packages: each package (all the go-files belonging to it) resides in its subdirectory, which has the same name as the package. For clarity, different packages also reside in different directories.

```go
package pack1

var Pack1Int int = 42
var pack1Float = 3.14
func ReturnStr() string {
  return "Hello main!"
}
```

It exports an int variable `Pack1Int`, a float32 variable `pack1Float` and a function `ReturnStr`, which returns a *string*. It is a good practice to type all exported values explicitly: the API is typed. Moreover, in a real project, all the exported objects should be documented. This program does not do anything when it is run because it does not contain a `main()` function.

Building and installing the package: `go install`

This is done via the subcommand install of go:

```
go install importpath
```

or in this case:

```
go install book/pack1
```

from a terminal or command-line (opened in the home folder or another folder). This command builds and installs the package specified by `importpath` and its dependencies. It writes the package object **pack1.a** to the **pkg\windows_amd64\book** subdirectory of the workspace in which the source resides. On Linux 64-bit this would be **pkg\linux_amd64\book**. If the **pkg** folder does not exist, it is created. If everything builds and installs correctly, then the command will produce no output. If you are in the folder of the package (**cd $GOPATH/src/book/pack1**), then issue:

```
go install
```

So the resulting workspace directory tree looks like this:

```
src/
    book/                         # the sources for the project book
    book/                     # the sources for the project book
         pack1/
               pack1.go       # package source
pkg/
    windows_amd64/            # or linux_amd64 on a Linux 64 bit OS
        book/                 # contains the object
             pack1.a          # package object
```

Building and installing the command executable: `go install`

The go command treats code belonging to package `main` as an executable command and installs the package `binary` to the GOPATH's bin subdirectory. A package is imported in another package by its `importpath`, which is the pathname of the directory it is in starting after the **src** component. The general format of the import is:

```
import "path or url to the package"
```

like:

```
import "github.com/org1/pack1"
```

Let us create the startup program in a source file **main.go**; this is preferably put in a subfolder **book_main** of book, with the package being imported in our case via the statement:

```
import "book/pack1"
```

(Working this way the startup program is clearly put aside from the packages; it could also allow for more than 1 startup program in our book project.)

```
package main
import (
"fmt"
"pack1"
)

func main() {
  var test1 string
  test1 = pack1.ReturnStr()
  fmt.Printf("ReturnStr from package1: %s\n", test1)
  fmt.Printf("Integer from package1: %d\n", pack1.Pack1Int)
  // fmt.Printf("Float from package1: %f\n", pack1.pack1Float)
}
```

Issue the command:

```
go install book/book_main
```

If it is an executable command it is written to the **bin** subdirectory of the workspace in which the source resides. This compiles **main.go**, links in the package `pack1.a` and installs the executable (or binary) **book_main.exe** (or pack1_main on Linux) to **$GOPATH/bin**. If `pack1.a` was not yet built, it is made with this command, so it is not necessary to build all dependent packages beforehand. The workspace directory tree now looks like this:

```
bin/
    book_main.exe           # command executable (pack1_main on Linux)
pkg/
    windows_amd64/          # or linux_amd64 on a Linux 64 bit OS
        book/               # contains the object files of the packages i
n the project book
            pack1.a         # package object
src/
    book/                   # the sources for the project book
        book_main/
            main.go         # command source
    pack1/
         pack1.go           # package source
```

Because the bin directory is in our PATH variable, we can execute the program on the command-line from any folder with **book_main** which produces the output:

```
ReturnStr from package1: Hello main!
Integer from package1: 42
```

Initialization of a package: `init()`

Program execution begins by importing the packages, initializing the `main` package and invoking the function `main()`. A package with no imports is initialized by assigning initial values to all its package-level variables and then calling any package-level `init()` function defined in its source. A package may contain multiple `init()` functions, even within a single source file. They are executed in unspecified order. It is best practice if the determination of a package's values only depend on other values or functions found in the same package. The `init()` functions cannot be called. Imported packages are initialized before the initialization of the package itself, but initialization of a package occurs only once in the execution of a program.

Now that you know how to make your package, in the next lesson, you'll learn how to document your package.