
# Module Thirteen

## *Packages*

*It's hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn't a book on software design*

Bruce Tognazzini

# 13.1 Packages

One of the motivating issue behind the development of Go was the problem of having to manage a number of large builds on a tight schedule. In previous modules we have mentioned packages and worked with them in passing so you should have a sense generally how they work. In this module, we will look at the package and build systems in more depth. We haven't needed the to use build system so far because the examples and labs have been simple enough that we do not need that sort of project organization. But is not generally true out there in the real world and more specifically, for the kinds of large scale projects that motivated the development of Go.

The Go package system bears some similarity to other programming languages so while many of the concepts may be familiar, there are a number of features that are quite unique to Go and which contribute to the speed with which Go programs compile and build.

As Brian Kernighan points out:

> *There are three main reasons for the compiler's speed. First, all imports must be explicitly listed at the beginning of each source file, so the compiler does not have to read and process an entire file to determine its dependencies. Second, the dependencies of a package form a directed acyclic graph, and because there are no cycles, packages can be compiled separately and perhaps in parallel. Finally, the object file for a compiled Go package records export information not just for the package itself, but for its dependencies too. When compiling a package, the compiler must read one object file for each import but need not look beyond these files.*

## *13.1.1 Package Visibility and Naming*

The naming system of packages and symbols follows a set of very simple rules designed to simplify the organization of project source files and project structures. As mentioned earlier in the course, symbols are either public or private in Go. Pubic symbols begin with a capital letter, private symbols begin with a lowercase letter. Symbols referenced in code that belong to a different package have the package name prefixed. This is the full extent of how the visibility is managed in Go.

### *Package naming*

Each package is named by its import path, which is a string that is used by the Go tool (not the compiler) to determine the physical location of the package. While it is not required by the language, the recommended naming for custom packages is to use the organization name as a prefix to the custom package. The protocol is the same for using name spaces in XML, if we are going to reuse or share or otherwise make the package available for general use, we want the package name to be globally unique.

# 13.2 Workspace Organization

While we can work out of an arbitrary directory structure, as we have been doing in this course so far since we have been working with small examples of code, in order to streamline the use and development of the Go build process and Go tool, there is a standard Go project structure and a set of assumptions made about how a Go project is organized.

These are:

1. Go programmers typically keep all their Go code in a single workspace.
2. A workspace contains many version control repositories (managed by Git, for example).
3. Each repository contains one or more packages.
4. Each package consists of one or more Go source files in a single directory.
5. The path to a package's directory determines its import path.

In other words, there is a single location dedicated to Go projects and each project's source files are under version control. The go tool looks for this workspace location in the GOPATH variable. This means we can have a number of workspaces that we use for Go and we can switch between them by resetting the value of the GOPATH variable.

## *13.2.1 Project Organization*

Within a given project, the following directory structure is expected. Notice that the structure is a generalization of what programmers generally do in terms of organizing their projects.

A workspace is a directory hierarchy with three directories at its root:

**`src`** A directory that contains all of the Go source files,

**`pkg`** A directory that contains package objects, and

**`bin`** A directory that contains executable commands.

Having a standard directory structure for a project allows a simplified implementation of the tool-set rather than having to specify the directory structure in a make file or other kind of description file.

As a point of clarification, the pkg directory does not contain the source for the packges, that is still maintained under the src directory. The pkg directory contains subdirectories generated by the go tool which correspond to the target architecture of a specified platform. In these subdirectories are the compiled binary versions of the packages, what is referred to above as the package objects.

### *13.2.2 Hello World Example*

To illustrate how this works, we will redo the Hello World example from module one as a Go project. The first thing we do is set the Go workspace. In this example, I have chosen “/home/projects/gospace” as my Go workspace. Since I am using a Unix file system, the path names will be in a Unix format, but you can make the appropriate translations to Windows where necessary.

The Go tool uses the GOPATH environmental variable to know where the workspace is located.

```
[Module13]$ export GOPATH=/home/projects/gospace
[Module13]echo $GOPATH
/home/projects/gospace
```

Now we add the hello project to the src directory and run go install hello from anywhere and we get the binary “installed” in the bin directory. Nothing new so far, but now we will add a library or package to the project called stringutil.

We add the package directory stringutils to the src directory and add the file listed on the next page to that directory (this example is from the golang.org website). Remember the code itself is not that important for this example, what is important the organization of the files and dependencies of the packages.

Our project now has the following structure:

```
src\
  hello\
       hello.go
  stringutil\
       stringutil.go
pkg\
bin\
```

```
// Example 13-01 Stringutils package

package stringutil

// Reverse returns its argument string reversed
// rune-wise left to right.
func Reverse(s string) string {
  r := []rune(s)
  for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
       r[i], r[j] = r[j], r[i]
  }
  return string(r)
}
```

The package import statement looks like this in the main package.

```
// Example 13-01 Hello.go

package main

import (
  "fmt"
  "stringutil"
)

func main() {
  fmt.Println(stringutil.Reverse("Hello World!"))
}
```

We can now test out the project by compiling the file and see if compiles correctly. As seen before, we locate to the package directory and run go build. No output is created if the compilation is successful and no executable is created because there is no main package.

```
[Module13]$ cd $GOPATH/src/stringutil
[stringutil] go build
[stringutil]
```

However when we execute go build hello, because of the import statement in the hello.go file, both the hello.go and stringutil.go files are compiled and the executable is deposited in the directory that we ran the build command from.

The go build command is primarily used as a quick check on how well the compilation works. All of the intermediate files are deleted right after so as not to pollute the project environment.This means that the build command will recompile all a file's dependencies even if there have been no changes to any of the dependencies.

```
[Module13]$ cd $GOPATH/src/hello
[stringutil] go build hello
[stringutil] ls
hello.go hello
```

### *13.2.2 Go install and Hello World*

Now if we run the go install command on our hello project, two things happen that are different than what happened with go build.

1. The executable is now "installed" in the bin directory
2. A binary version of stringutil is now "installed" in the pkg directory. A sub directory has also been created in the pkg directory corresponding to the binary platform that the stringutil.go file was compiled to.

Our project now has the following structure:

```
src\
  hello\
       hello.go
  stringutil\
       stringutil.go
pkg\
  linux_amd64\
       stringutil.a
bin\
  hello
```

If we made a change to the hello.go file and ran install again, the Go tool would not re-compile the stringutil.go file since it has not changed since the binary form (stringutil.a) of the package was created. If we have a large project with only a couple of changes in one file, install becomes a lot faster than build since only the changed files are actually rebuilt.

Just as a note, this process also works best when we have a dev environment and a production environment. Once our dev version works and passes its tests, we can move only the changed files to the production environment where install creates the new version in optimal time. The further exploration of that aspect of Go project management is beyond the scope of this course.

### *13.2.3 Package aliases*

Generally, the import path name for the package corresponds to the directory of package. However when we have a multi-part import path, we generally only use the last part of the path name as the prefix in our code. If our stringutil package had been a sub package of a "util" while the import path would be “util/stringutil” we still would have only used the last part as the package prefix for symbols – more specifically, our actual code remains unchanged.

Suppose though that we have two stringutil packages one of which is in $GOPATH/src/srtingutil and the other in $GOPATH/src/util/stringutil, and to make things even more worst case scenario, each one of these has a Reverse function, the only difference is that one just reverses the string and the other also converts the string to uppercase (just so we can tell which one we are calling in the examples).

```
// Example 13-03 util/stringutil package

package stringutil
import "strings'

// Reverse returns its argument string reversed
// rune-wise left to right.
func Reverse(s string) string {
  r := []rune(s)
  for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
       r[i], r[j] = r[j], r[i]
  }
  return strings.ToUpper(string(r))
}
```

Notice there is no path information in the package statement about where the source file is located. The actual location of the package source code is not bound to the code in any way. This means I can relocate my packages wherever I want to without having to modify the package source code.

Suppose that I want to reference each of these Reverse functions. Because of some poor code planning on my part, I have created an ambiguity.

```
// Example 13-04 Hello.go with abiguities

package main
import (
  "fmt"
  "stringutil"
  "util/stringutil"
)

func main() {
  fmt.Println(stringutil.Reverse("Hello World!"))
  fmt.Println(stringutil.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello.go:7: stringutil redeclared as imported package name
  previous declaration at ./hello.go:6
Error: process exited with code 2.
```

Just like using name spaces in XML, we can assign local aliases to packages to remove these ambiguous references.

```
// Example 13-05 Hello.go with aliases

package main
import (
  "fmt"
  s1 "stringutil"
  s2 "util/stringutil"
)

func main() {
  fmt.Println(s1.Reverse("Hello World!"))
  fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
!dlroW olleH
!DLROW OLLEH
```

## 13.3 Package Initialization

When a file is loaded, we can execute initialization code to do any initialization that cannot be handled through standard declarations. In each file there can be one or more functions named init() (this is the only exception to functions having unique names) which are all executed after:

1. All of the package variable have been initialized
2. All of the init() functions from the imported packages have run.

This is illustrated in the example below where we have added two init() functions to the hello main package.

```
// Example 13-06 Hello.go with init functions
package main

import (
  "fmt"
  s1 "stringutil"
  s2 "util/stringutil"
)

func init() {
  fmt.Println("Main init 1")
}
func init() {
  fmt.Println("Main init 2")
}

func main() {
  fmt.Println(s1.Reverse("Hello World!"))
  fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
Main init 1
Main init 2
!dlroW olleH
!DLROW OLLEH
```

Now adding an init() function into the package util/stringutil and re-running the build on on the same hello project would produce this output.

```
// Example 13-07 Dependent init functions
package stringutil

import "strings"
import "fmt"

func init() {
  fmt.Println("util/stringutil init")

... and so on
}
```

```
[hello]$ go build hello
./hello
util/stringutil init
Main init 1
Main init 2
!dlroW olleH
!DLROW OLLEH
```

According to the golang documentation one of the primary uses of the init() function is to verify and repair program state before execution as in the following example:

```
func init () {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath may be overridden --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath,
                                "override default GOPATH")
}
```

Sometimes we may want the init() function of an importedf package to execute but we don't want to use any of the symbols in the package. To avoid a compile time error, we use the blank variable as a package alias so the compiler will not panic if it can't find a symbol from that imported package.

```
// Example 13-08 Hello.go blank alias
package main

import (
  "fmt"
  "stringutil"
  _ "util/stringutil"
)

func main() {
  fmt.Println(stringutil.Reverse("Hello World!"))

}
```

```
[hello]$ go build hello
./hello
util/stringutil init
!dlroW olleH
```

# 13.4 The go get Option

Programming languages like Python, R and Ruby all allow for access to an extended set of resources over Internet type connections. The go tool also provides the capability to import packages and utilities from other remote locations.

In this example, we start with a clean project space so we can see the get command at work without any additional clutter.  Before we begin, the src, bin and pkt directories are all empty.

Suppose we want to add a utility “golint” to our workspace. The utility is located in a standard Go project structure at https://github.com/golang/lint  The “go get” command goes to the specificed location (in this case the url) and fetches the project, then runs “install” on the downloaded package or project.

```
go get github.com/golang/lint/golint
```

Once this command is finished, then the directory tree looks like this

```
bin\
  golint
pkg\
  linux_amd64\
       github.com\
            golang\
                 lint.a
       golang.org\
            x\
                 tools\
                      go\
                           gcimporter15.a
src\
  github.com\
       golang\
            lint\
  golang.org\
            x\
                 tools\
```

There is an additional package that needs to be installed because it is a dependency that is required to make golint, specifically the gcimporter15.a library.

Looking at the source for the lint tool, we can see why this dependency was included.

The go get operation went to the specific repository, downloaded the source code into the scr directory, then downloaded all the dependencies that are required, then compiles all of the source to the appropriate binary package objects.

There is a rich set of options that govern the go get tool which are documented in the go tool documentation.

```
package lint

import (
  "bytes"
  "fmt"
  "go/ast"
  "go/parser"
  "go/printer"
  "go/token"
  "go/types"
  "regexp"
  "sort"
  "strconv"
  "strings"
  "unicode"
  "unicode/utf8"

  "golang.org/x/tools/go/gcimporter15"
)
```

The last line of the import path refers to a local directory that is a copy of a repository at golang.org. When this code is compiled, Go tries to find the specified package locally, then a build error is generated. At this point the go get command has to be used to create the local copy.

# 13.5 Vendor Management

The go get tool creates a copy of the remote repository in the the local GOPATH workspace which is necessary for build to be repeatable. If the go install tool went out and copied the remote repository automatically, then changes to either the contents or location of the remote repository would cause the code to break. Go avoids this by creating a local copy to work with, a process called “vendoring.”

The import statement

```
import “github.com/golang/examples”
```

Would look only for the directory $GOPATH/src/github.com/golang/examples and if this directory does not exist, then any attempt to build an executable will fail because of a missing dependencies.

## *13.5.1 Problems with Versions*

While this creates a nice clean build environment, there is a versioning problem. You can only have one version of a package in the workspace specified by GOPATH. In the examples we have been seeing, this has not been a problem, but consider the two following cases:

1. Two different developers are using two different versions of the same package.
2. A single developer has two projects, each of which use a different version of the same package.

In both of these cases, if we use go get for get the packages remotely, we generally are going to get the latest versions so that we may find ourselves breaking one of the projects.

## *13.5.2 Import Rewriting*

One way that can be used to use different versions of a package is to do path rewriting, so that the installed path “github/golang/examples” can be changed to something like “github/golang/ver2/examples." A similar rewrite can take place for a different version. Al that needs to be done is to then rewrite any import statements necessary to ensure that we are pointing to the right location.

While this does solve the problem, it is not an effective solution. There is a lot of low level work being done that is tedious, critical and error prone. The whole solution works is quite against the whole philosophy of Go.

### 13.5.3 Third Party Vendoring Tools

An example of this sort of vendoring tool is godep. We won't exploring the third party tools in the course but it is illustrative to see how one of them typically works.

After executing go get command like

```
go get github/golang/examples
```

then godep snapshots the dependencies by running

```
godep save
```

which creates a json file that looks something like this

```
"ImportPath": "github.com/golang/prog",
    "GoVersion": "go1.6",
    "Deps": [
            {
            "ImportPath": "github.com/golang/examples/ex1",
            "Rev": "e0e1b550d545d9be0446ce324babcb16f09270f5"
            },
            {
             "ImportPath": "ithub.com/golang/examples/ex2",
             "Rev": "a1577bd3870218dc30725a7cf4655e9917e3751b"
                },
)
```

The Go commands are then executed via godep like this:

```
godep go build
```

which does the appropriate fixups.

## *13.5 The Vendor Folder*

One of the innovations of the Go 1.6 release is the use of the vendor folder. The idea is that instead of sharing a common location for an external dependency, each project can declare have a vendor folder in the project tree.

Under the vendor folder is a local copy of an external dependency, however the difference is that when Go is building the project, it will look first in the local vendor folder to resolve the dependency rather than outside the project.

```
src\
  hello\
       vendor\
            stringutil\
                 version 1 code
  bye\
       vendor\
            stringutil\
                 version 2 code
  stringutil\
       version3 code
  whatever\
pkg\
bin\
```

In the example above, when resolving the stringutil dependency, the hello project will import the version 1 code, the bye project will import the version 2 code and the whatever project will import the version 3 code.

Currently most of the third party vendoring tools support the vendor folder.