CHAPTER 10

Modules, Packages, and Imports

Most modern programming languages have a system for organizing code into namespaces and libraries, and Go is no exception. As you've seen while exploring other features, Go introduces some new approaches to this old idea. In this chapter, you'll learn how to organize code with packages and modules, how to import them, how to work with third-party libraries, and how to create libraries of your own.

Repositories, Modules, and Packages

Library management in Go is based around three concepts: repositories, modules, and packages. A *repository* is familiar to all developers. It is a place in a version control system where the source code for a project is stored. A *module* is a bundle of Go source code that's distributed and versioned as a single unit. Modules are stored in a repository. Modules consist of one or more *packages*, which are directories of source code. Packages give a module organization and structure.

While you can store more than one module in a repository, it is discouraged. Everything within a module is versioned together. Maintaining two modules in one repository requires you to track separate versions for two different modules in a single repository.

Unfortunately, different programming languages use these words to represent different concepts. While packages in Java and Go are similar, a Java repository is a centralized place to store multiple *artifacts* (the analogue of a Go module). Node.js and Go swap the meanings of the terms: a Node.js package is similar to what Go calls a module, and a Go package is similar to a Node.js module. The terminology certainly can be confusing at first, but as you get more comfortable with Go, the terms will seem more familiar.

Before using code from packages outside the standard library, you need to make sure that you have a properly created module. Every module has a globally unique identifier. This is not unique to Go. Java defines globally unique package declarations by using the reverse domain name convention (`com.companyname.projectname.library`).

In Go, this name is called a *module path*. It is usually based on the repository where the module is stored. For example, you can find Proteus, a module I wrote to simplify relational database access in Go, at *https://github.com/jonbodner/proteus*. It has a module path of *github.com/jonbodner/proteus*.

Back in "Your First Go Program" on page 3, you created a module whose name was `hello_world`, which is obviously not globally unique. This is fine if you are creating a module for local use only. If you put a module with a nonunique name into a source code repository, it cannot be imported by another module.

Using go.mod

A directory tree of Go source code becomes a module when there's a valid *go.mod* file in it. Rather than create this file manually, use the subcommands of the `go mod` command to manage modules. The command `go mod init` *`MODULE_PATH`* creates the *go.mod* file that makes the current directory the root of a module. The *`MODULE_PATH`* is the globally unique name that identifies your module. The module path is case-sensitive. To reduce confusion, do not use uppercase letters within it.

Take a look at the contents of a *go.mod* file:

```
module github.com/learning-go-book-2e/money

go 1.21

require (
    github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-18...
    github.com/shopspring/decimal v1.3.1
)

require (
    github.com/fatih/color v1.13.0 // indirect
    github.com/mattn/go-colorable v0.1.9 // indirect
    github.com/mattn/go-isatty v0.0.14 // indirect
    golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c // indirect
)
```

Every *go.mod* file starts with a `module` directive that consists of the word `module` and the module's unique path. Next, the *go.mod* file specifies the minimum compatible version of Go with the `go` directive. All source code within the module must be

compatible with the specified version. For example, if you specify the (rather old) version `1.12`, the compiler will not let you use underscores within numeric literals, because that feature was added in Go 1.13.

Use the go Directive to Manage Go Build Versions

What happens if the `go` directive specifies a version of Go that's newer than what's installed? If you have Go 1.20 or earlier installed, the newer Go version is ignored and you get the features of the installed version. If you have Go 1.21 or newer installed, the default behavior is to download the newer version of Go and use it to build your code. You can control this behavior in Go 1.21 and newer with the `toolchain` directive and the `GOTOOLCHAIN` environment variable. You can assign them the following values:

- `auto`, which downloads newer versions of Go. (This is the default behavior of Go 1.21 and later.)
- `local`, which restores the behavior of Go releases before 1.21.
- A specific Go version (such as `go1.20.4`), which means that specific version will be downloaded and used to build the program.

For example, the command line `GOTOOLCHAIN=go1.18 go build` will build your Go program with Go 1.18, downloading it if necessary.

If both the `GOTOOLCHAIN` environment variable and the `toolchain` directive are set, the value assigned to `GOTOOLCHAIN` is used.

Complete details on the `go` directive, the `toolchain` directive, and the `GOTOOLCHAIN` environment variable are found in the official Go toolchain documentation.

As discussed in "The for-range value is a copy" on page 81, Go 1.22 introduces the first backward-breaking change to the language. When using Go 1.22 or later, if the `go` directive is set to `1.22` or higher, a `for` loop creates a new index and value variable on each iteration. This behavior is applied per module. The value of the `go` directive in each imported module determines the language level for that module. ("Importing Third-Party Code" on page 240 describes how to use and manage multiple modules in your programs.)

You can see this difference with a short example. You can find the code in the *sample_code/loop_test* directory in the Chapter 10 repository.

The code in *loop.go* is simple:

```
func main() {
    x := []int{1, 2, 3, 4, 5}
    for _, v := range x {
        fmt.Printf("%p\n", &v)
```

```
    }
}
```

If you haven't seen it before, the `%p` verb in the `fmt` formatting language returns the memory location of a pointer. The repository has the `go` directive in its *go.mod* file set to `1.21`. Building and running the program gives the following output:

```
140000140a8
140000140a8
140000140a8
140000140a8
140000140a8
```

When built using older versions of Go, the program prints out the same memory address five times. (The memory addresses might differ from what's shown here, but all will be the same value.)

Change the value of the `go` directive in *go.mod* to `1.22` and rebuild and rerun the program. You will now get this output:

```
1400000e0b0
1400000e0b8
1400000e0d0
1400000e0d8
1400000e0e0
```

Notice that every memory address value is different, indicating that a new variable was created on each iteration. (The memory addresses might differ from what's shown here, but each one will be a different value.)

The require Directive

The next sections in a *go.mod* file are the `require` directives. The `require` directives are present only if your module has dependencies. They list the modules that your module depends on and the minimum version required for each one. The first `require` section lists the direct dependencies of your module. The second one lists the dependencies of the dependencies of your module. Each line in this section ends with an `// indirect` comment. There's no functional difference between modules marked as indirect and those that aren't; it's just documentation for people when they look at the *go.mod* file. Direct dependencies are marked as indirect in one situation, and I will discuss it when I talk about the ways to use `go get`. In "Importing Third-Party Code" on page 240, you'll learn more about adding and managing the dependencies of your module.

While the `module`, `go`, and `require` directives are the ones most commonly used in a *go.mod* file, there are three others directives as well. I will cover the `replace` and `exclude` directives in "Overriding Dependencies" on page 254, and I'll cover the `retract` directive in "Retracting a Version of Your Module" on page 255.

Building Packages

Now that you've learned how to make your directory of code into a module, it's time to start using packages to organize your code. You'll start by seeing how `import` works, move on to creating and organizing packages, and then look at some of the features of Go's packages, both good and bad.

Importing and Exporting

The example programs have been using the `import` statement in Go even though I haven't discussed what it does and how importing in Go differs from other languages. Go's `import` statement allows you to access exported constants, variables, functions, and types in another package. A package's exported identifiers (an *identifier* is the name of a variable, constant, type, function, method, or a field in a struct) cannot be accessed from another current package without an `import` statement.

This leads to the question, how do you export an identifier in Go? Rather than use a special keyword, Go uses *capitalization* to determine whether a package-level identifier is visible outside the package where it is declared. An identifier whose name starts with an uppercase letter is *exported*. Conversely, an identifier whose name starts with a lowercase letter or underscore can be accessed only from within the package where it is declared. (Identifiers in Go can't start with a digit but can contain them.)

Anything you export is part of your package's API. Before you export an identifier, be sure that you intend to expose it to clients. Document all exported identifiers and keep them backward compatible unless you are intentionally making a major version change (see "Versioning Your Module" on page 252 for more information).

Creating and Accessing a Package

Making packages in Go is easy. Let's look at a small program to demonstrate this. You can find it in the *package_example* directory for this book. Inside *package_example*, you'll see two additional directories, *math* and *do-format*. In *math*, there's a file called *math.go* with the following contents:

```
package math

func Double(a int) int {
    return a * 2
}
```

The first line of the file is called the *package clause*. It consists of the keyword `package` and the name for the package. The package clause is always the first nonblank, noncomment line in a Go source file.

In the *do-format* directory is a file called *formatter.go* with the following contents:

```
package format

import "fmt"

func Number(num int) string {
    return fmt.Sprintf("The number is %d", num)
}
```

Note that the package name is `format` in the package clause, but it's in the *do-format* directory. How to interact with this package will be covered very shortly.

Finally, the following contents are in the file *main.go* in the root directory:

```
package main

import (
    "fmt"

    "github.com/learning-go-book-2e/package_example/do-format"
    "github.com/learning-go-book-2e/package_example/math"
)

func main() {
    num := math.Double(2)
    output := format.Number(num)
    fmt.Println(output)
}
```

The first line of this file is familiar. All the programs before this chapter have put `package main` as the first line in the code. I'll talk more about what this means in a bit.

Next is the import section. It imports three packages. The first is `fmt`, which is in the standard library. You've done this in previous chapters. The next two imports refer to the packages within the program. You must specify an *import path* when importing from anywhere besides the standard library. The import path consists of the module path followed by the path to the package within the module. For example, the import `"github.com/learning-go-book-2e/package_example/math"` has the module path `github.com/learning-go-book-2e/package_example`, while `/math` is the path to the package within the module.

It is a compile-time error to import a package but not use any of the identifiers exported by the package. This ensures that the binary produced by the Go compiler includes only code that's actually used in the program.

You might come across obsolete documentation on the web that mentions relative path import paths. They do not work with modules. (And were a bad idea, anyway.)

When you run this program, you'll see the following output:

```
$ go build
$ ./package_example
The number is 4
```

The `main` function called the `Double` function in the `math` package by prefixing the function name with the package name. You've seen this in previous chapters when calling functions in the standard library. You also call the `Number` function in the `format` package. You might wonder where this `format` package came from, since the import says `github.com/learning-go-book-2e/package_example/do-format`.

Every Go file in a directory must have an identical package clause. (You'll see one tiny exception to this rule in "Testing Your Public API" on page 377.) You imported the `format` package with the import path *github.com/learning-go-book-2e/package_example/do-format*. That's because *the name of a package is determined by its package clause, not its import path*.

As a general rule, you should make the name of the package match the name of the directory that contains the package. It is hard to discover a package's name if it does not match the containing directory. However, in a few situations you'll use a different name for the package than for the directory.

The first is something you have been doing all along without realizing it. You declare a package to be a starting point for a Go application by using the special package name `main`. Since you cannot import the `main` package, this doesn't produce confusing import statements.

The other reasons for having a package name not match your directory name are less common. If your directory name contains a character that's not valid in a Go identifier, then you must choose a package name that's different from your directory name. In your case, `do-format` is not a valid identifier name, so it's replaced with `format`. It's better to avoid this by never creating a directory with a name that's not a valid identifier.

The final reason for creating a directory whose name doesn't match the package name is to support versioning using directories. I'll talk about this more in "Versioning Your Module" on page 252.

As was discussed in "Blocks" on page 67, package names in `import` statements are in the *file block*. If you use exported symbols from one package in two different files

in another package, you must import the first package in both files in the second package.

Naming Packages

Package names should be descriptive. Rather than have a package called `util`, create a package name that describes the functionality provided by the package. For example, say you have two helper functions: one to extract all names from a string and another to format names properly. Don't create two functions in a `util` package called `ExtractNames` and `FormatNames`. If you do, every time you use these functions, they will be referred to as `util.ExtractNames` and `util.FormatNames`, and that `util` package tells you nothing about what the functions do.

One option is to create one function called `Names` in a package called `extract` and a second function called `Names` in a package called `format`. It's OK for these two functions to have the same name, because they will always be disambiguated by their package names. The first will be referred to as `extract.Names` when imported, and the second will be referred to as `format.Names`.

An even better option is to think about parts of speech. A function or method does something, so it should be a verb or action word. A package would be a noun, a name for the kind of item that is created or modified by the functions in the package. By following these rules, you would create a package called `names` with two functions, `Extract` and `Format`. The first would then be referred to as `names.Extract` and the second would be `names.Format`.

You should also avoid repeating the name of the package in the names of functions and types within the package. Don't name your function `ExtractNames` when it is in the `names` package. The exception to this rule occurs when the name of the identifier is the same as the name of the package. For example, the package `sort` in the standard library has a function called `Sort`, and the `context` package defines the `Context` interface.

Overriding a Package's Name

Sometimes you'll find yourself importing two packages whose names collide. For example, the standard library includes two packages for generating random numbers; one is cryptographically secure (`crypto/rand`), and the other is not (`math/rand`). The regular generator is fine when you aren't generating random numbers for encryption, but you need to seed it with an unpredictable value. A common pattern is to seed a regular random number generator with a value from a cryptographic generator. In Go, both packages have the same name (`rand`). When that happens, you provide an alternate name for one package within the current file. Try out this code on The Go

Playground or in the *sample_code/package_name_override* directory in the Chapter 10 repository. First, look at the import section:

```
import (
    crand "crypto/rand"
    "encoding/binary"
    "fmt"
    "math/rand"
)
```

You import `crypto/rand` with the name `crand`. This overrides the name `rand` that's declared within the package. You then import `math/rand` normally. When you look at the `seedRand` function, you see that you access identifiers in `math/rand` with the `rand` prefix, and use the `crand` prefix with the `crypto/rand` package:

```
func seedRand() *rand.Rand {
    var b [8]byte
    _, err := crand.Read(b[:])
    if err != nil {
        panic("cannot seed with cryptographic random number generator")
    }
    r := rand.New(rand.NewSource(int64(binary.LittleEndian.Uint64(b[:]))))
    return r
}
```

You can use two other symbols as a package name. The package name `.` (dot) places all the exported identifiers in the imported package into the current package's namespace; you don't need a prefix to refer to them. This is discouraged because it makes your source code less clear. You can no longer tell whether something is defined in the current package or whether it was imported by simply looking at its name.

You can also use `_` (underscore) as the package name. You'll explore what this does when I talk about `init` in "Avoiding the init Function if Possible" on page 239.

As I discussed in "Shadowing Variables" on page 68, package names can be shadowed. Declaring variables, types, or functions with the same name as a package makes the package inaccessible within the block with that declaration. If this is unavoidable (for example, a newly imported package has a name that conflicts with an existing identifier), override the package's name to resolve the conflict.

Documenting Your Code with Go Doc Comments

An important part of creating a module for others to use is documenting it properly. Go has its own format for writing comments that are automatically converted into documentation. It's called *Go Doc* format, and it's very simple. Here are the rules:

- Place the comment directly before the item being documented, with no blank lines between the comment and the declaration of the item.
- Start each line of the comment with double slashes (//), followed by a space. While it's legal to use /* and */ to mark your comment block, it is idiomatic to use double slashes.
- The first word in the comment for a symbol (a function, type, constant, variable, or method) should be the name of the symbol. You can also use "A" or "An" before the symbol name to help make the comment text grammatically correct.
- Use a blank comment line (double slashes and a newline) to break your comment into multiple paragraphs.

As I'll talk about in "Using pkg.go.dev" on page 251, you can view public documentation online in HTML format. If you want to make your documents look a little snazzier, there are a couple of ways to format it:

- If you want your comment to contain some preformatted content (such as a table or source code), put an additional space after the double slashes to indent the lines with the content.
- If you want a header in your comment, put a `#` and a space after the double slashes. Unlike with Markdown, you cannot use multiple `#` characters to make different levels of headers.
- To make a link to another package (whether or not it is in the current module), put the package path within brackets (`[` and `]`).
- To link to an exported symbol, place its name in brackets. If the symbol is in another package, use `[pkgName.SymbolName]`.
- If you include a raw URL in your comment, it will be converted into a link.
- If you want to include text that links to a web page, put the text within brackets (`[` and `]`). At the end of the comment block, declare the mappings between your text and their URLs with the format `// [`*`TEXT`*`]: `*`URL`*. You'll see a sample of this in a moment.

Comments before the package declaration create package-level comments. If you have lengthy comments for the package (such as the extensive formatting documentation in the `fmt` package), the convention is to put the comments in a file in your package called *doc.go*.

Let's go through a well-commented file, starting with the package-level comment in Example 10-1.

Example 10-1. A package-level comment

```
// Package convert provides various utilities to
// make it easy to convert money from one currency to another.
package convert
```

Next, place a comment on an exported struct (see Example 10-2). Notice that it starts with the name of the struct.

Example 10-2. A struct comment

```
// Money represents the combination of an amount of money
// and the currency the money is in.
//
// The value is stored using a [github.com/shopspring/decimal.Decimal]
type Money struct {
    Value    decimal.Decimal
    Currency string
}
```

Finally, there is a comment on a function (see Example 10-3).

Example 10-3. A well-commented function

```
// Convert converts the value of one currency to another.
//
// It has two parameters: a Money instance with the value to convert,
// and a string that represents the currency to convert to. Convert returns
// the converted currency and any errors encountered from unknown or unconvertible
// currencies.
//
// If an error is returned, the Money instance is set to the zero value.
//
// Supported currencies are:
//   - USD - US Dollar
//   - CAD - Canadian Dollar
//   - EUR - Euro
//   - INR - Indian Rupee
//
// More information on exchange rates can be found at [Investopedia].
//
// [Investopedia]: https://www.investopedia.com/terms/e/exchangerate.asp
func Convert(from Money, to string) (Money, error) {
    // ...
}
```

Go includes a command-line tool called `go doc` that displays godocs. The command `go doc` *`PACKAGE_NAME`* displays the documentation for the specified package and a

list of the identifiers in the package. Use `go doc` *`PACKAGE_NAME.IDENTIFIER_NAME`* to display the documentation for a specific identifier in the package.

If you want to preview your documentation's HTML formatting before it is published on the web, use `pkgsite`. This is the same program that powers `pkg.go.dev` (which I will talk about later in the chapter). To install `pkgsite`, use this command:

```
$ go install golang.org/x/pkgsite/cmd/pkgsite@latest
```

(I will talk more about `go install` in "Adding Third-Party Tools with go install" on page 264).

To view your source code's comments rendered as HTML, go to the root of your module and run this:

```
$ pkgsite
```

Then go to *http://localhost:8080* to view your project and its source code.

You can find even more details about comments and potential pitfalls by reading through the official Go Doc Comments documentation.

Make sure you comment your code properly. At the very least, any exported identifier should have a comment. In "Using Code-Quality Scanners" on page 267, you will look at some third-party tools that can report missing comments on exported identifiers.

Using the internal Package

Sometimes you want to share a function, type, or constant among packages in your module, but you don't want to make it part of your API. Go supports this via the special `internal` package name.

When you create a package called `internal`, the exported identifiers in that package and its subpackages are accessible only to the direct parent package of `internal` and the sibling packages of `internal`. Let's look at an example to see how this works. You can find the code on GitHub. The directory tree is shown in Figure 10-1.

You've declared a simple function in the *internal.go* file in the `internal` package:

```
func Doubler(a int) int {
    return a * 2
}
```

You can access this function from *foo.go* in the `foo` package and from *sibling.go* in the `sibling` package.

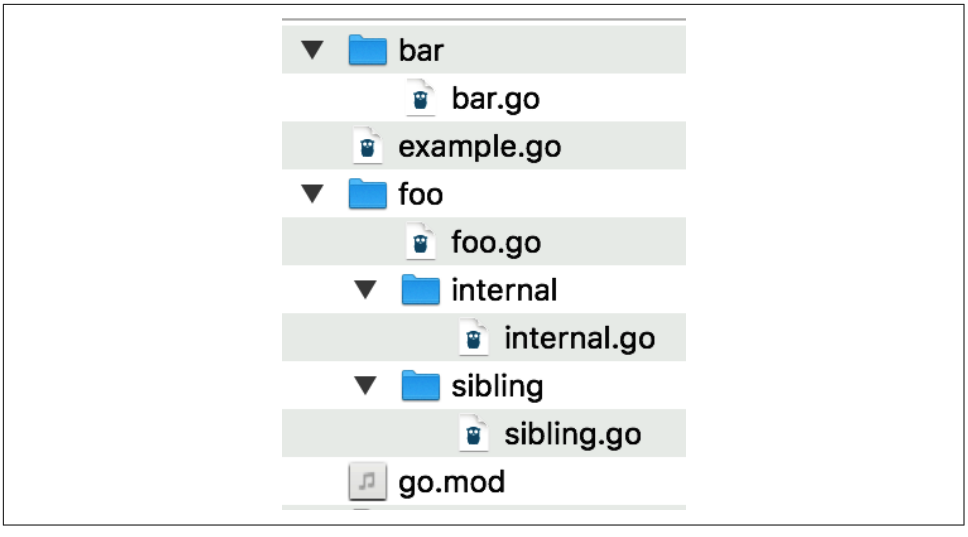

Figure 10-1. The file tree for `internal_package_example`

Be aware that attempting to use the internal function from *bar.go* in the `bar` package or from *example.go* in the root package results in a compilation error:

```
$ go build ./...
package github.com/learning-go-book-2e/internal_example
example.go:3:8: use of internal package
github.com/learning-go-book-2e/internal_example/foo/internal not allowed

package github.com/learning-go-book-2e/internal_example/bar
bar/bar.go:3:8: use of internal package
github.com/learning-go-book-2e/internal_example/foo/internal not allowed
```

Avoiding Circular Dependencies

Two of the goals of Go are a fast compiler and easy-to-understand source code. To support this, Go does not allow you to have a *circular dependency* between packages. If package A imports package B, directly or indirectly, package B cannot import package A, directly or indirectly.

Let's look at a quick example to explain the concept. You can find the code in the *sample_code/circular_dependency_example* directory in the Chapter 10 repository. There are two packages, `pet` and `person`. In *pet.go* in the `pet` package, you have this:

```
import "github.com/learning-go-book-2e/ch10/sample_code/
circular_dependency_example/person"

var owners = map[string]person.Person{
        "Bob":   {"Bob", 30, "Fluffy"},
```

```
        "Julia": {"Julia", 40, "Rex"},
}
```

While in *person.go* in the `person` package, you have this:

```
import "github.com/learning-go-book-2e/ch10/sample_code/
circular_dependency_example/pet"

var pets = map[string]pet.Pet{
        "Fluffy": {"Fluffy", "Cat", "Bob"},
        "Rex":    {"Rex", "Dog", "Julia"},
}
```

If you try to build this program, you'll get an error:

```
$ go build ./sample_code/circular_dependency_example
package github.com/learning-go-book-2e/ch10/sample_code/
    circular_dependency_example
        imports github.com/learning-go-book-2e/ch10/sample_code/
        circular_dependency_example/person
        imports github.com/learning-go-book-2e/ch10/sample_code/
        circular_dependency_example/pet
        imports github.com/learning-go-book-2e/ch10/sample_code/
        circular_dependency_example/person: import cycle not allowed
```

If you find yourself with a circular dependency, you have a few options. In some cases, this is caused by splitting up packages too finely. If two packages depend on each other, there's a good chance they should be merged into a single package. You can merge the `person` and `pet` packages into a single package, and that solves your problem.

If you have a good reason to keep your packages separated, it may be possible to move just the items that cause the circular dependency to one of the two packages or to a new package.

Organizing Your Module

There's no official way to structure the Go packages in your module, but several patterns have emerged over the years. They are guided by the principle that you should focus on making your code easy to understand and maintain.

When your module is small, keep all your code in a single package. As long as no other modules depend on your module, there is no harm in delaying organization.

As your module grows, you'll want to impose some order to make your code more readable. The first question to ask is what type of module you are creating. You can group modules into two broad categories: those that are intended as a single application and those that are primarily intended as libraries. If you are sure that your module is intended to be used only as an application, make the root of the project the `main` package. The code in the `main` package should be minimal; place all your logic

in an *internal* directory, and the code in the `main` function will simply invoke code within *internal*. This way, you can ensure that no one is going to create a module that depends on your application's implementation.

If you want your module to be used as a library, the root of your module should have a package name that matches the repository name. This makes sure that the import name matches the package name. To make this work, you must ensure that your repository name is a valid Go identifier. In particular, you cannot use a hyphen as a word separator in your repository name, because a hyphen is not a valid character in a package name.

It's not uncommon for library modules to have one or more applications included with them as utilities. In this case, create a directory called *cmd* at the root of your module. Within *cmd*, create one directory for each binary built from your module. For example, you might have a module that contains both a web application and a command-line tool that analyzes data in the web application's database. Use `main` as the package name within each of these directories.

For more detailed information, this blog post by Eli Bendersky provides good advice on how you should structure a simple Go module.

As projects get more complicated, you'll be tempted to break up your packages. Make sure to organize your code to limit the dependencies among packages. One common pattern is to organize your code by slices of functionality. For example, if you wrote a shopping site in Go, you might place all the code for customer management in one package and all the code for inventory management in another. This style limits the dependencies among packages, which makes it easier to later refactor a single web application into multiple microservices. This style is in contrast to the way that many Java applications are organized, with all the business logic in one package, all the database logic in another package, and the data transfer objects in a third.

When developing a library, take advantage of the `internal` package. If you create multiple packages within a module and they are outside an `internal` package, exporting a symbol so it can be used by another package in your module means it can also be used by *anyone* who imports your module. There's a principle in software engineering called Hyrum's law: "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody." Once something is part of your API, you have a responsibility to continue supporting it until you decide to make a new version that's not backward compatible. You will learn how to do this in "Updating to Incompatible Versions" on page 248. If you have some symbols that you want to share only within your module, put them in `internal`. If you change your mind, you can always move the package out of `internal` later.

For a good overview of Go project structure advice, watch Kat Zien's talk from GopherCon 2018, "How Do You Structure Your Go Apps".

The "golang-standards" GitHub repository claims to be the "standard" module layout. Russ Cox, the development lead for Go, has publicly stated that it is not endorsed by the Go team and that the structure it recommends is in fact an antipattern. Please do not cite this repository as a way to organize your code.

Gracefully Renaming and Reorganizing Your API

After using a module for a while, you might realize that its API is not ideal. You might want to rename some of the exported identifiers or move them to another package within your module. To avoid a backward-breaking change, don't remove the original identifiers; provide an alternate name instead.

This is easy with a function or method. You declare a function or method that calls the original. For a constant, simply declare a new constant with the same type and value, but a different name.

If you want to rename or move an exported type, you use an alias. Quite simply, an *alias* is a new name for a type. You saw in Chapter 7 how to use the `type` keyword to declare a new type based on an existing one. You also use the `type` keyword to declare an alias. Let's say you have a type called `Foo`:

```
type Foo struct {
    x int
    S string
}

func (f Foo) Hello() string {
    return "hello"
}

func (f Foo) goodbye() string {
    return "goodbye"
}
```

If you want to allow users to access `Foo` by the name `Bar`, all you need to do is this:

```
type Bar = Foo
```

To create an alias, use the `type` keyword, the name of the alias, an equals sign, and the name of the original type. The alias has the same fields and methods as the original type.

The alias can even be assigned to a variable of the original type without a type conversion:

```
func MakeBar() Bar {
    bar := Bar{
        x: 20,
        S: "Hello",
    }
    var f Foo = bar
    fmt.Println(f.Hello())
    return bar
}
```

One important point to remember: an alias is just another name for a type. If you want to add new methods or change the fields in an aliased struct, you must add them to the original type.

You can alias a type that's defined in the same package as the original type or in a different package. You can even alias a type from another module. There is one drawback to an alias in another package: you cannot use an alias to refer to the unexported methods and fields of the original type. This limitation makes sense, as aliases exist to allow a gradual change to a package's API, and the API consists only of the exported parts of the package. To work around this limitation, call code in the type's original package to manipulate unexported fields and methods.

Two kinds of exported identifiers can't have alternate names. The first is a package-level variable. The second is a field in a struct. Once you choose a name for an exported struct field, there's no way to create an alternate name.

Avoiding the init Function if Possible

When you read Go code, it is usually clear which methods and functions are called. One of the reasons Go doesn't have method overriding or function overloading is to make it easier to understand what code is running. However, there is a way to set up state in a package without explicitly calling anything: the `init` function. When you declare a function named `init` that takes no parameters and returns no values, it runs the first time the package is referenced by another package. Since `init` functions do not have any inputs or outputs, they can work only by side effect, interacting with package-level functions and variables.

The `init` function has another unique feature. Go allows you to declare multiple `init` functions in a single package, or even in a single file in a package. There's a documented order for running multiple `init` functions in a single package, but rather than remembering it, it's better to simply avoid them.

Some packages, like database drivers, use `init` functions to register the database driver. However, you don't use any of the identifiers in the package. As mentioned earlier, Go doesn't allow you to have unused imports. To work around this, Go allows *blank imports*, where the name assigned to an import is the underscore (_). Just as an underscore allows you to skip an unused return value from a function, a blank

import triggers the `init` function in a package but doesn't give you access to any of the exported identifiers in the package:

```
import (
    "database/sql"

    _ "github.com/lib/pq"
)
```

This pattern is considered obsolete because it's unclear that a registration operation is being performed. Go's compatibility guarantee for its standard library means that you are stuck using it to register database drivers and image formats, but if you have a registry pattern in your own code, register your plug-ins explicitly.

The primary use of `init` functions today is to initialize package-level variables that can't be configured in a single assignment. It's a bad idea to have mutable state at the top level of a package, since it makes it harder to understand how data flows through your application. That means that any package-level variables configured via `init` should be *effectively immutable*. While Go doesn't provide a way to enforce that their value does not change, you should make sure that your code does not change them. If you have package-level variables that need to be modified while your program is running, see if you can refactor your code to put that state into a struct that's initialized and returned by a function in the package.

The nonexplicit invocation of `init` functions means that you should document their behavior. For example, a package with an `init` function that loads files or accesses the network should call this out in a package-level comment so that security-conscious users of your code aren't surprised by unexpected I/O.

Working with Modules

You've seen how to work with packages within a single module, and now it's time to see how to integrate with third-party modules and the packages within them. After that, you'll learn about publishing and versioning your own modules and Go's centralized services: pkg.go.dev, the module proxy, and the checksum database.

Importing Third-Party Code

So far, you've imported packages from the standard library like `fmt`, `errors`, `os`, and `math`. Go uses the same import system to integrate packages from third parties. Unlike many other compiled languages, Go always builds applications from source code into a single binary file. This includes the source code of your module and the source code of all the modules on which your module depends. (The Go compiler is smart enough to not include unreferenced packages in the binary it produces.) Just as you saw when you imported a package from within your own module, when you

import a third-party package, you specify the location in the source code repository where the package is located.

Let's look at an example. I mentioned back in Chapter 2 that you should never use floating-point numbers when you need an exact representation of a decimal number. If you do need an exact representation, one good option is the `decimal` module from ShopSpring. You are also going to look at a simple formatting module that I've written for this book. Both of these modules are used in a small program in the money repository for the book. This program calculates the price of an item with the tax included and prints the output in a neat format.

The following code is in *main.go*:

```
package main

import (
    "fmt"
    "log"
    "os"

    "github.com/learning-go-book-2e/formatter"
    "github.com/shopspring/decimal"
)

func main() {
    if len(os.Args) < 3 {
        fmt.Println("Need two parameters: amount and percent")
        os.Exit(1)
    }
    amount, err := decimal.NewFromString(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    percent, err := decimal.NewFromString(os.Args[2])
    if err != nil {
        log.Fatal(err)
    }
    percent = percent.Div(decimal.NewFromInt(100))
    total := amount.Add(amount.Mul(percent)).Round(2)
    fmt.Println(formatter.Space(80, os.Args[1], os.Args[2],
                                total.StringFixed(2)))
}
```

The two imports `github.com/learning-go-book-2e/formatter` and `github.com/shopspring/decimal` specify third-party imports. Note that they include the location of the package in the repository. Once they're imported, you access the exported items in these packages just like any other imported package.

Before building the application, look at the *go.mod* file. Its contents should be as follows:

```
module github.com/learning-go-book-2e/money

go 1.20
```

If you try to do a build, you get the following message:

```
$ go build
main.go:8:2: no required module provides package
    github.com/learning-go-book-2e/formatter; to add it:
        go get github.com/learning-go-book-2e/formatter
main.go:9:2: no required module provides package
    github.com/shopspring/decimal; to add it:
        go get github.com/shopspring/decimal
```

As the errors indicate, you cannot build the program until you add references to the third-party modules to your *go.mod* file. The `go get` command downloads modules and updates the *go.mod* file. You have two options when using `go get`. The simplest option is to tell `go get` to scan your module's source code and add any modules that are found in `import` statements to *go.mod*:

```
$ go get ./...
go: downloading github.com/shopspring/decimal v1.3.1
go: downloading github.com/learning-go-book-2e/formatter
    v0.0.0-20220918024742-1835a89362c9
go: downloading github.com/fatih/color v1.13.0
go: downloading github.com/mattn/go-colorable v0.1.9
go: downloading github.com/mattn/go-isatty v0.0.14
go: downloading golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c
go: added github.com/fatih/color v1.13.0
go: added github.com/learning-go-book-2e/formatter
    v0.0.0-20220918024742-1835a89362c9
go: added github.com/mattn/go-colorable v0.1.9
go: added github.com/mattn/go-isatty v0.0.14
go: added github.com/shopspring/decimal v1.3.1
go: added golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c
```

Because the location of the package is in the source code, `go get` is able to get the package's module and download it. If you look in the *go.mod* file now, you'll see this:

```
module github.com/learning-go-book-2e/money

go 1.20

require (
    github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-1835a89362c9
    github.com/shopspring/decimal v1.3.1
)

require (
    github.com/fatih/color v1.13.0 // indirect
```

```
    github.com/mattn/go-colorable v0.1.9 // indirect
    github.com/mattn/go-isatty v0.0.14 // indirect
    golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c // indirect
)
```

The first `require` section of the *go.mod* file lists the modules that you've imported into your module. After the module name is a version number. In the case of the `formatter` module, it doesn't have a version tag, so Go makes up a *pseudoversion*.

You also see a second `require` directive section that has modules with an *indirect* comment. One of these modules (`github.com/fatih/color`) is directly used by `formatter`. It, in turn, depends on the other three modules in the second `require` directive section. All the modules used by all your module's dependencies (and your dependencies' dependencies, and so on) are included in the *go.mod* file for your module. The ones that are used only in dependencies are marked as indirect.

In addition to updating *go.mod*, a *go.sum* file is created. For each module in the dependency tree of your project, the *go.sum* file has one or two entries: one with the module, its version, and a hash of the module; the other with the hash of the *go.mod* file for the module. Here's what the *go.sum* file looks like:

```
github.com/fatih/color v1.13.0 h1:8LOYc1KYPPmyKMuN8QV2DNRWNbLo6LZ0iLs...
github.com/fatih/color v1.13.0/go.mod h1:kLAiJbzzSOZDVNGyDpeOxJ47H46q...
github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-1835a8...
github.com/learning-go-book-2e/formatter v0.0.0-20220918024742-1835a8...
github.com/mattn/go-colorable v0.1.9 h1:sqDoxXbdeALODt0DAeJCVp38ps9Zo...
github.com/mattn/go-colorable v0.1.9/go.mod h1:u6P/XSegPjTcexA+o6vUJr...
github.com/mattn/go-isatty v0.0.12/go.mod h1:cbi8OIDigv2wuxKPP5vlRcQ1...
github.com/mattn/go-isatty v0.0.14 h1:yVuAays6BHfxijgZPzw+3Zlu5yQgKGP...
github.com/mattn/go-isatty v0.0.14/go.mod h1:7GGIvUiUoEMVVmxf/4nioHXj...
github.com/shopspring/decimal v1.3.1 h1:2Usl1nmF/WZucqkFZhnfFYxxxu8LG...
github.com/shopspring/decimal v1.3.1/go.mod h1:DKyhrW/HYNuLGql+MJL6WC...
golang.org/x/sys v0.0.0-20200116001909-b77594299b42/go.mod h1:h1NjWce...
golang.org/x/sys v0.0.0-20200223170610-d5e6a3e2c0ae/go.mod h1:h1NjWce...
golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c h1:F1jZWGFhYfh0Ci...
golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c/go.mod h1:oPkhp1M...
```

You'll see what these hashes are used for in "Module Proxy Servers" on page 259. You might also notice that there are multiple versions of some of the dependencies. I'll talk about that in "Minimal Version Selection" on page 247.

Let's validate that your modules are now set up correctly. Run `go build` again, and then run the `money` binary and pass it some arguments:

```
$ go build
$ ./money 99.99 7.25
99.99           7.25                                            107.24
```

This sample program was checked in without *go.sum* and with an incomplete *go.mod*. This was done so you could see what happens when these files are populated. When committing your own modules to source control, always include up-to-date *go.mod* and *go.sum* files. Doing so specifies exactly what versions of your dependencies are being used. This enables *repeatable builds*; when anyone else (including your future self) builds this module, they will get the exact same binary.

As I mentioned, there's another way to use `go get`. Instead of telling it to scan your source code to discover modules, you can pass the module paths to `go get`. To see this work, roll back the changes to your *go.mod* file and remove the *go.sum* file. On Unix-like systems, the following commands will do this:

```
$ git restore go.mod
$ rm go.sum
```

Now, pass the module paths to `go get` directly:

```
$ go get github.com/learning-go-book-2e/formatter
go: added github.com/learning-go-book-2e/
    formatter v0.0.0-20200921021027-5abc380940ae
$ go get github.com/shopspring/decimal
go: added github.com/shopspring/decimal v1.3.1
```

Sharp-eyed readers might have noticed that when we used `go get` a second time, the `go: downloading` messages weren't displayed. The reason is that Go maintains a *module cache* on your local computer. Once a version of a module is downloaded, a copy is kept in the cache. Source code is pretty compact, and drives are pretty large, so this isn't usually a concern. However, if you want to delete the module cache, use the command `go clean -modcache`.

Take a look at the contents of *go.mod*, and they'll look a little different than before:

```
module github.com/learning-go-book-2e/money

go 1.20

require (
    github.com/fatih/color v1.13.0 // indirect
    github.com/learning-go-book-2e/
    formatter v0.0.0-20220918024742-1835a89362c9 // indirect
    github.com/mattn/go-colorable v0.1.9 // indirect
    github.com/mattn/go-isatty v0.0.14 // indirect
    github.com/shopspring/decimal v1.3.1 // indirect
    golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c // indirect
)
```

Notice that all the imports are marked as *indirect*, not just the ones that came from `formatter`. When you run `go get` and pass it a module name, it doesn't check your source code to see whether the module you specified is used within your main module. Just to be safe, it adds an indirect comment.

If you want to fix this label automatically, use the command `go mod tidy`. It scans your source code and synchronizes the *go.mod* and *go.sum* files with your module's source code, adding and removing module references. It also makes sure that the indirect comments are correct.

You might be wondering why you would want to bother using `go get` with a module name. The reason is that it allows you to update the version of an individual module.

Working with Versions

Let's see how Go's module system uses versions. I've written a simple module that you're going to use in another tax-collection program. In *main.go*, there are the following third-party imports:

```
"github.com/learning-go-book-2e/simpletax"
"github.com/shopspring/decimal"
```

As before, the sample program wasn't checked in with *go.mod*, and *go.sum* updated, so you could see what happens. When the program is built, you see the following:

```
$ go get ./...
go: downloading github.com/learning-go-book-2e/simpletax v1.1.0
go: added github.com/learning-go-book-2e/simpletax v1.1.0
go: added github.com/shopspring/decimal v1.3.1
$ go build
```

The *go.mod* file has been updated:

```
module github.com/learning-go-book-2e/region_tax

go 1.20

require (
    github.com/learning-go-book-2e/simpletax v1.1.0
    github.com/shopspring/decimal v1.3.1
)
```

There is also a *go.sum* file with hashes for your dependencies. Run the code and see whether it's working:

```
$ ./region_tax 99.99 12345
2022/09/19 22:04:38 unknown zip: 12345
```

That looks like an unexpected answer. The latest version of the module might have a bug. By default, Go picks the latest version of a dependency when you add it to your module. However, one of the things that makes versioning useful is that you can

specify an earlier version of a module. First, you can see what versions of the module are available with the `go list` command:

```
$ go list -m -versions github.com/learning-go-book-2e/simpletax
github.com/learning-go-book-2e/simpletax v1.0.0 v1.1.0
```

By default, the `go list` command lists the packages that are used in your module. The `-m` flag changes the output to list the modules instead, and the `-versions` flag changes `go list` to report on the available versions for the specified module. In this case, you see that there are two versions, v1.0.0 and v1.1.0. Let's downgrade to version v1.0.0 and see if that fixes your problem. You do that with the `go get` command:

```
$ go get github.com/learning-go-book-2e/simpletax@v1.0.0
go: downloading github.com/learning-go-book-2e/simpletax v1.0.0
go: downgraded github.com/learning-go-book-2e/simpletax v1.1.0 => v1.0.0
```

The `go get` command lets you work with modules, updating the versions of your dependencies.

Now if you look at *go.mod*, you'll see that the version has been changed:

```
module github.com/learning-go-book-2e/region_tax

go 1.20

require (
    github.com/learning-go-book-2e/simpletax v1.0.0
    github.com/shopspring/decimal v1.3.1
)
```

You also see in *go.sum* that it contains both versions of `simpletax`:

```
github.com/learning-go-book-2e/simpletax v1.0.0 h1:KZU8aXRCHkvgFmBWkV...
github.com/learning-go-book-2e/simpletax v1.0.0/go.mod h1:lR4YYZwbDTI...
github.com/learning-go-book-2e/simpletax v1.1.0 h1:sG83gscauX/b8yKKY9...
github.com/learning-go-book-2e/simpletax v1.1.0/go.mod h1:lR4YYZwbDTI...
```

This is fine; if you change a module's version or even remove a module from your module, an entry for it might still remain in *go.sum*. This doesn't cause problems.

When you build and run the code again, the bug is fixed:

```
$ go build
$ ./region_tax 99.99 12345
107.99
```

Semantic Versioning

Software has had version numbers from time immemorial, but there has been little consistency in what version numbers mean. The version numbers attached to Go modules follow the rules of *semantic versioning*, also known as *SemVer*. By requiring

semantic versioning for modules, Go makes its module management code simpler while ensuring that users of a module understand what a new release promises.

If you aren't familiar with SemVer, check out the full specification. The very short explanation is that semantic versioning divides a version number into three parts: the *major* version, the *minor* version, and the *patch* version, which are written as `major.minor.patch` and preceded by a `v`. The patch version number is incremented when fixing a bug; the minor version number is incremented (and the patch version is set back to 0) when a new, backward-compatible feature is added; and the major version number is incremented (and minor and patch are set back to 0) when making a change that breaks backward compatibility.

Minimal Version Selection

At some point, your module will depend on two or more modules that all depend on the same module. As often happens, these modules declare that they depend on different minor or patch versions of that module. How does Go resolve this?

The module system uses the principle of *minimal version selection*: you will always get the lowest version of a dependency that is declared to work in all the *go.mod* files across all your dependencies. Let's say that your module directly depends on modules A, B, and C. All three of these modules depend on module D. The *go.mod* file for module A declares that it depends on v1.1.0, module B declares that it depends on v1.2.0, and module C declares that it depends on v1.2.3. Go will import module D only once, and it will choose version v1.2.3, as that, in the words of the Go Modules Reference, is the minimum version that satisfies all requirements.

You can see this in action with your sample program from "Importing Third-Party Code" on page 240. The command `go mod graph` shows the dependency graph of your module and all its dependencies. Here are a few lines of its output:

```
github.com/learning-go-book-2e/money github.com/fatih/color@v1.13.0
github.com/learning-go-book-2e/money github.com/mattn/go-colorable@v0.1.9
github.com/learning-go-book-2e/money github.com/mattn/go-isatty@v0.0.14
github.com/fatih/color@v1.13.0 github.com/mattn/go-colorable@v0.1.9
github.com/fatih/color@v1.13.0 github.com/mattn/go-isatty@v0.0.14
github.com/mattn/go-colorable@v0.1.9 github.com/mattn/go-isatty@v0.0.12
```

Each line lists two modules, the first being the parent and the second being the dependency and its version. You'll notice the `github.com/fatih/color` module is declared to depend on version v0.0.14 of `github.com/mattn/go-isatty`, while `github.com/mattn/go-colorable` depends on v0.0.12. The Go compiler selects version v0.0.14 to use, because it is the minimal version that meets all requirements. This occurs even though, as of this writing, the latest version of `github.com/mattn/go-isatty` is v0.0.16. Your minimal version requirement is met with v0.0.14, so that's what is used.

This system isn't perfect. You might find that while module A works with version v1.1.0 of module D, it does not work with version v1.2.3. What do you do then? Go's answer is that you need to contact the module authors to fix their incompatibilities. The *import compatibility rule* says, "If an old package and a new package have the same import path, the new package must be backward compatible with the old package." All minor and patch versions of a module must be backward compatible. If they aren't, it's a bug. In our hypothetical example, either module D needs to be fixed because it broke backward compatibility, or module A needs to be fixed because it made a faulty assumption about the behavior of module D.

You might not find this answer satisfying, but it's honest. Some build systems, like npm, will include multiple versions of the same package. This can introduce its own set of bugs, especially when there is package-level state. It also increases the size of your application. In the end, some things are better solved by community than code.

Updating to Compatible Versions

What if you explicitly want to upgrade a dependency? Let's assume that after writing the initial program, there are three more versions of `simpletax`. The first fixes problems in the initial v1.1.0 release. Since it's a bug patch release with no new functionality, it would be released as v1.1.1. The second keeps the current functionality but also adds a new function. It would get the version number v1.2.0. Finally, the third fixes a bug that was found in version v1.2.0. It has the version number v1.2.1.

To upgrade to the bug patch release for the current minor version, use the command `go get -u=patch github.com/learning-go-book-2e/simpletax`. Since you had downgraded to v1.0.0, you would remain on that version, because there is no patch version with the same minor version.

Upgrade to version v1.1.0 by using `go get github.com/learning-go-book-2e/simpletax@v1.1.0` and then run `go get -u=patch github.com/learning-go-book-2e/simpletax`. This upgrades the version to v1.1.1.

Finally, use the command `go get -u github.com/learning-go-book-2e/simpletax` to get the most recent version of `simpletax`. That upgrades you to version v1.2.1.

Updating to Incompatible Versions

Let's go back to the program. You're expanding to Canada, and luckily, a version of the `simpletax` module handles both the US and Canada. However, this version has a slightly different API than the previous one, so its version is v2.0.0.

To handle incompatibility, Go modules follow the *semantic import versioning* rule. This rule has two parts:

- The major version of the module must be incremented.
- For all major versions besides 0 and 1, the path to the module must end in v*N*, where *N* is the major version.

The path changes because an import path uniquely identifies a package. By definition, incompatible versions of a package are not the same package. Using different paths means that you can import two incompatible versions of a package into different parts of your program, allowing you to upgrade gracefully.

Let's see how this changes the program. First, change the import of `simpletax` to this:

```
"github.com/learning-go-book-2e/simpletax/v2"
```

This changes your import to refer to the v2 module.

Next, change the code in `main` to the following:

```
func main() {
    amount, err := decimal.NewFromString(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    zip := os.Args[2]
    country := os.Args[3]
    percent, err := simpletax.ForCountryPostalCode(country, zip)
    if err != nil {
        log.Fatal(err)
    }
    total := amount.Add(amount.Mul(percent)).Round(2)
    fmt.Println(total)
}
```

The program is now reading a third parameter from the command line, which is the country code. The program also calls a different function in the `simpletax` package. When you run `go get ./...`, the dependency is automatically updated:

```
$ go get ./...
go: downloading github.com/learning-go-book-2e/simpletax/v2 v2.0.0
go: added github.com/learning-go-book-2e/simpletax/v2 v2.0.0
```

Build and run the program to see the new output:

```
$ go build
$ ./region_tax 99.99 M4B1B4 CA
112.99
$ ./region_tax 99.99 12345 US
107.99
```

When you look at the *go.mod* file, you'll see that the new version of `simpletax` is included:

```
module github.com/learning-go-book-2e/region_tax

go 1.20

require (
    github.com/learning-go-book-2e/simpletax v1.0.0
    github.com/learning-go-book-2e/simpletax/v2 v2.0.0
    github.com/shopspring/decimal v1.3.1
)
```

And *go.sum* has been updated as well:

```
github.com/learning-go-book-2e/simpletax v1.0.0 h1:KZU8aXRCHkvgFmBWkV...
github.com/learning-go-book-2e/simpletax v1.0.0/go.mod h1:lR4YYZwbDTI...
github.com/learning-go-book-2e/simpletax v1.1.0 h1:sG83gscauX/b8yKKY9...
github.com/learning-go-book-2e/simpletax v1.1.0/go.mod h1:lR4YYZwbDTI...
github.com/learning-go-book-2e/simpletax/v2 v2.0.0 h1:EUFWy1BBA2omgkm...
github.com/learning-go-book-2e/simpletax/v2 v2.0.0/go.mod h1:yGLh6ngH...
github.com/shopspring/decimal v1.3.1 h1:2Usl1nmF/WZucqkFZhnfFYxxxu8LG...
github.com/shopspring/decimal v1.3.1/go.mod h1:DKyhrW/HYNuLGql+MJL6WC...
```

The old versions of `simpletax` are still referenced, even though they are no longer used. Use `go mod tidy` to remove those unused versions. Then you'll see only v2.0.0 of `simpletax` referenced in *go.mod* and *go.sum*.

Vendoring

To ensure that a module always builds with identical dependencies, some organizations like to keep copies of their dependencies inside their module. This is known as *vendoring*. It's enabled by running the command `go mod vendor`. This creates a directory called *vendor* at the top level of your module that contains all your module's dependencies. These dependencies are used in place of the module cache stored on your computer.

If new dependencies are added to *go.mod* or versions of existing dependencies are upgraded with `go get`, you need to run `go mod vendor` again to update the *vendor* directory. If you forget to do this, `go build`, `go run`, and `go test` will display an error message and refuse to run.

Older Go dependency management systems required vendoring, but with the advent of Go modules and proxy servers (see "Module Proxy Servers" on page 259 for details), the practice is falling out of favor. One reason you might still want to vendor is that it can make building your code faster and more efficient when working with some CI/CD (continuous integration/continuous delivery) pipelines. If a pipeline's build servers are ephemeral, the module cache may not be preserved. Vendoring dependencies allows these pipelines to avoid making multiple network

calls to download dependencies every time a build is triggered. The downside is that it dramatically increases the size of your codebase in version control.

Using pkg.go.dev

While there isn't a single centralized repository of Go modules, there is a single service that gathers together documentation on Go modules. The Go team has created a site called *pkg.go.dev* that automatically indexes open source Go modules. For each module, the package index publishes the godocs, the license used, the *README*, the module's dependencies, and what open source modules depend on the module. You can see the info that *pkg.go.dev* has on your `simpletax` module in Figure 10-2.

Figure 10-2. Use pkg.go.dev to find and learn about third-party modules

Publishing Your Module

Making your module available to other people is as simple as putting it in a version control system. This is true whether you are releasing your module as open source on a public version control system like GitHub or a private one that's hosted by you or

within a company. Since Go programs build from source code and use a repository path to identify themselves, there's no need to explicitly upload your module to a central library repository, as you do for Maven Central or npm. Make sure you check in both your *go.mod* file and your *go.sum* file.

While most Go developers use Git for version control, Go also supports Subversion, Mercurial, Bazaar, and Fossil. By default, Git and Mercurial can be used for public repositories, and any of the supported version control systems can be used for private repositories. For details, check out the version control system documentation for Go modules.

When releasing an open source module, you should include a file named *LICENSE* in the root of your repository that specifies the open source license under which you are releasing your code. It's FOSS is a good resource for learning more about the various kinds of open source licenses.

Roughly speaking, you can divide open source licenses into two categories: permissive (which allows users of your code to keep their code private) and nonpermissive (which requires users of your code to make their code open source). While the license you choose is up to you, the Go community favors permissive licenses, such as BSD, MIT, and Apache. Since Go compiles third-party code directly into every application, the use of a nonpermissive license like the GPL would require people who use your code to release their code as open source as well. For many organizations, this is not acceptable.

One final note: do not write your own license. Few people will trust that it has been properly reviewed by a lawyer, and they can't tell what claims you are making on their module.

Versioning Your Module

Whether your module is public or private, you should properly version your module so that it works correctly with Go's module system. As long as you are adding functionality or patching bugs, the process is simple. Store your changes in your source code repository, then apply a tag that follows the semantic versioning rules I discussed in "Semantic Versioning" on page 246.

Go's semantic versioning supports the concept of *pre-releases*. Let's assume that the current version of your module is tagged `v1.3.4`. You are working on version 1.4.0, which is not quite done, but you want to try importing it into another module. What you should do is append a hyphen (-) to the end of your version tag, followed by an identifier for the pre-release build. In this case, use a tag like `v1.4.0-beta1` to indicate beta 1 of version 1.4.0 or `v1.4.0-rc2` to indicate release candidate 2. If you

want to depend on a pre-release candidate, you must specify its version explicitly in `go get`, as Go will not automatically select a pre-release version.

If you reach a point where you need to break backward compatibility, the process is more complicated. As you saw when importing version 2 of the `simpletax` module, a backward-breaking change requires a different import path. There are a few steps to take.

First you need to choose a way to store your new version. Go supports two ways for creating the different import paths:

- Create a subdirectory within your module named *vN*, where *N* is the major version of your module. For example, if you are creating version 2 of your module, call this directory *v2*. Copy your code into this subdirectory, including the *README* and *LICENSE* files.
- Create a branch in your version control system. You can put either the old code or the new code on the new branch. Name the branch *vN* if you are putting the new code on the branch, or *vN-1* if you are putting the old code there. For example, if you are creating version 2 of your module and want to put version 1 code on the branch, name the branch *v1*.

After you decide how to store your new code, you need to change the import path in the code in your subdirectory or branch. The module path in your *go.mod* file must end with */vN*, and all the imports within your module must use */vN* as well. Going through all your code can be tedious, but Marwan Sulaiman has created a tool that automates the work. Once the paths are fixed, go ahead and implement your changes.

Technically, you could just change *go.mod* and your import statements, tag your main branch with the latest version, and not bother with a subdirectory or versioned branch. However, this is not a good practice, as it makes it unclear where to find older major versions of your module.

When you are ready to publish your new code, place a tag on your repository that looks like v*N*.0.0. If you are using the subdirectory system or keeping the latest code on your main branch, tag the main branch. If you are placing your new code on a different branch, tag that branch instead.

You can find more details on updating your code to an incompatible version in the post "Go Modules: v2 and Beyond" on The Go Blog and in "Developing a Major Version Update" on the Go developer website.

Overriding Dependencies

Forks happen. While there's a bias against forking in the open source community, sometimes a module stops being maintained or you want to experiment with changes that aren't accepted by the module's owner. A `replace` directive redirects all references to a module across all your module's dependencies and replaces them with the specified fork of the module. It looks like this:

```
replace github.com/jonbodner/proteus => github.com/someone/my_proteus v1.0.0
```

The original module location is specified on the left side of the `=>` and the replacement on the right. The right side must have a version specified, but specifying a version is optional for the left side. If a version is specified on the left side, only that specific version will be replaced. If the version is not specified, any version of the original module will be replaced with the specific version of the fork.

A `replace` directive can also refer to a path on your local filesystem:

```
replace github.com/jonbodner/proteus => ../projects/proteus
```

With a local `replace` directive, the module version must be omitted from the right side.

Avoid using local `replace` directives. They provided a way to modify multiple modules simultaneously before the invention of Go workspaces, but now they are a potential source of broken modules. (I will cover workspaces shortly.) If you share your module via version control, a module with local references in `replace` directives will probably not build for anyone else, since you cannot guarantee that other people will have the replacement modules in the same locations on their drive.

You also might want to block a specific version of a module from being used. Perhaps it has a bug or is incompatible with your module. Go provides the `exclude` directive to prevent a specific version of a module from being used:

```
exclude github.com/jonbodner/proteus v0.10.1
```

When a module version is excluded, any mentions of that module version in any dependent module are ignored. If a module version is excluded and it's the only version of that module that's required in your module's dependencies, use `go get` to add an indirect import of a different version of the module to your module's *go.mod* file so that your module still compiles.

Retracting a Version of Your Module

Sooner or later, you will accidentally publish a version of your module that you don't want anyone to use. Perhaps it was released by accident before testing was complete. Maybe after its release, a critical vulnerability is discovered and no one should use it any more. No matter the reason, Go provides a way for you to indicate that certain versions of a module should be ignored. This is done by adding a `retract` directive to the *go.mod* file of your module. It consists of the word `retract` and the semantic version that should no longer be used. If a range of versions shouldn't be used, you can exclude all versions in that range by placing the upper and lower bounds within brackets, separated by a comma. While it's not required, you should include a comment after a version or version range to explain the reason for the retraction.

If you wish to retract multiple nonsequential versions, you can specify them with multiple `retract` directives. In the examples shown, version 1.5.0 is excluded, as are all versions from 1.7.0 to 1.8.5, inclusive:

```
retract v1.5.0 // not fully tested
retract [v1.7.0, v.1.8.5] // posts your cat photos to LinkedIn w/o permission
```

Adding a `retract` directive to *go.mod* requires you to create a new version of your module. If the new version contains only the retraction, you should retract it as well.

When a version is retracted, existing builds that specified the version will continue to work, but `go get` and `go mod tidy` will not upgrade to them. They will no longer appear as options when you use the `go list` command. If the most recent version of a module is retracted, it will no longer be matched with `@latest`; the highest unretracted version will match instead.

While `retract` can be confused with `exclude`, there's a very important difference. You use `retract` to prevent others from using specific versions of *your* module. An `exclude` directive blocks you from using versions of another module.

Using Workspaces to Modify Modules Simultaneously

There's one drawback to using your source code repository and its tags as a way to track your dependencies and their versions. If you want to make changes to two (or more) modules simultaneously, and you want to experiment with those changes across modules, you need a way to use a local copy of a module instead of the version in the source code repository.

You can find obsolete advice online to try to solve this issue with temporary `replace` directives in *go.mod* that point to local directories. Do not do this! It's too easy to forget to undo these changes before committing and pushing your code. Workspaces were introduced to avoid this antipattern.

Go uses *workspaces* to address this issue. A workspace allows you to have multiple modules downloaded to your computer, and references between those modules will automatically resolve to the local source code instead of the code hosted in your repository.

This section assumes that you have a GitHub account. If you don't, you can still follow along. I'm going to use the organization name `learning-go-book-2e`, but you should replace it with your GitHub account name or organization.

Let's start with two sample modules. Create a *my_workspace* directory and in that directory, create two more directories, *workspace_lib* and *workspace_app*. In the *workspace_lib* directory, run `go mod init github.com/learning-go-book-2e/workspace_lib`. Create a file called *lib.go* with the following content:

```
package workspace_lib

func AddNums(a, b int) int {
    return a + b
}
```

In the *workspace_app* directory, run `go mod init github.com/learning-go-book-2e/workspace_app`. Create a file called *app.go* with the following contents:

```
package main

import (
    "fmt"
    "github.com/learning-go-book-2e/workspace_lib"
)

func main() {
    fmt.Println(workspace_lib.AddNums(2, 3))
}
```

In previous sections, you used `go get ./...` to add `require` directives to *go.mod*. Let's see what happens if you try it here:

```
$ go get ./...
github.com/learning-go-book-2e/workspace_app imports
    github.com/learning-go-book-2e/workspace_lib: cannot find module
        providing package github.com/learning-go-book-2e/workspace_lib
```

Since *workspace_lib* hasn't been pushed to GitHub yet, you can't pull it in. If you try to run `go build`, you will get a similar error:

```
$ go build
app.go:5:2: no required module provides
    package github.com/learning-go-book-2e/workspace_lib; to add it:
        go get github.com/learning-go-book-2e/workspace_lib
```

Let's take advantage of workspaces to allow *workplace_app* to see the local copy of *workspace_lib*. Go to the *my_workspace* directory and run the following commands:

```
$ go work init ./workspace_app
$ go work use ./workspace_lib
```

This creates a *go.work* file in *my_workspace* with the following contents:

```
go 1.20

use (
    ./workspace_app
    ./workspace_lib
)
```

The *go.work* file is meant for your local computer only. Do not commit it to source control!

Now if you build *workspace_app*, everything works:

```
$ cd workspace_app
$ go build
$ ./workspace_app
5
```

Now that you are sure that *workspace_lib* does the right thing, it can be pushed to GitHub. In GitHub, create an empty public repository called `workspace_lib` and then run the following commands in the *workspace_lib* directory:

```
$ git init
$ git add .
$ git commit -m "first commit"
$ git remote add origin git@github.com:learning-go-book-2e/workspace_lib.git
$ git branch -M main
$ git push -u origin main
```

After running these commands, go to *https://github.com/learning-go-book-2e/workspace_lib/releases/new* (replacing "learning-go-book-2e" with your account or organization), and create a new release with the tag v0.1.0.

Now if you go back to the *workspace_app* directory and run `go get ./...`, the `require` directive is added, because there is a public module that can be downloaded:

```
$ go get ./...
go: downloading github.com/learning-go-book-2e/workspace_lib v0.1.0
go: added github.com/learning-go-book-2e/workspace_lib v0.1.0
$ cat go.mod
module github.com/learning-go-book-2e/workspace_app

go 1.20

require github.com/learning-go-book-2e/workspace_lib v0.1.0
```

You can validate that your code is working with the public module by setting the environment variable `GOWORK=off` and building your application:

```
$ rm workspace_app
$ GOWORK=off go build
$ ./workspace_app
5
```

Even though there is now a `require` directive referring to the public module, you can continue to make updates in our local workspace and they will be used instead. In *workspace_lib*, modify the *lib.go* file and add the following function:

```
func SubNums(a, b int) int {
    return a - b
}
```

In *workspace_app*, modify the *app.go* file and add the following line to the end of the `main` function:

```
    fmt.Println(workspace_lib.SubNums(2,3))
```

Now run `go build` and see it use the local module instead of the public one:

```
$ go build
$ ./workspace_app
5
-1
```

Once you have made the edits and want to release your software, you need to update the version information in your modules' *go.mod* files to refer to the updated code. This requires you to commit your modules to source control in dependency order:

1. Choose a modified module that has no dependencies on any of the modified modules in your workspace.
2. Commit this module to your source code repository.
3. Create a new version tag on the newly committed module in your source code repository.

4. Use `go get` to update the version specified in *go.mod* in the modules that depend on the newly committed module.
5. Repeat the first four steps until all modified modules are committed.

If you have to make changes to *workspace_lib* in the future and want to test them in *workspace_app* without pushing back to GitHub and creating lots of temporary versions, you can `git pull` the latest versions of the modules into your workspace again and make your updates.

Module Proxy Servers

Rather than relying on a single, central repository for libraries, Go uses a hybrid model. Every Go module is stored in a source code repository, like GitHub or GitLab. But by default, `go get` doesn't fetch code directly from source code repositories. Instead, it sends requests to a *proxy server* run by Google. When the proxy server receives the `go get` request, it checks its cache to see if there has been a request for this module version before. If so, it returns the cached information. If a module or a version of a module isn't cached on the proxy server, it downloads the module from the module's repository, stores a copy, and returns the module. This allows the proxy server to keep copies of every version of virtually all public Go modules.

In addition to the proxy server, Google also maintains a *checksum database*. It stores information on every version of every module cached by the proxy server. Just as the proxy server protects you from a module or a version of a module being removed from the internet, the checksum database protects you against modifications to a version of a module. This could be malicious (someone has hijacked a module and slipped in malicious code), or it could be inadvertent (a module maintainer fixes a bug or adds a new feature and reuses an existing version tag). In either case, you don't want to use a module version that has changed because you won't be building the same binary and don't know what the effects are on your application.

Every time you download a module via `go get` or `go mod tidy`, the Go tools calculate a hash for the module and contact the checksum database to compare the calculated hash to the hash stored for that module's version. If they don't match, the module isn't installed.

Specifying a Proxy Server

Some people object to sending requests for third-party libraries to Google. There are a few options:

- You can disable proxying entirely by setting the `GOPROXY` environment variable to `direct`. You'll download modules directly from their repositories, but if you

depend on a version that's removed from the repository, you won't be able to access it.

- You can run your own proxy server. Both Artifactory and Sonatype have Go proxy server support built into their enterprise repository products. The Athens Project provides an open source proxy server. Install one of these products on your network and then point `GOPROXY` to the URL.

Using Private Repositories

Most organizations keep their code in private repositories. If you want to use a private module in another Go module, you can't request it from Google's proxy server. Go will fall back to checking the private repository directly, but you might not want to leak the names of private servers and repositories to external services.

If you are using your own proxy server, or if you have disabled proxying, this isn't an issue. Running a private proxy server has some additional benefits. First, it speeds up downloading of third-party modules, as they are cached in your company's network. If accessing your private repositories requires authentication, using a private proxy server means that you don't have to worry about exposing authentication information in your CI/CD pipeline. The private proxy server is configured to authenticate to your private repositories (see the authentication configuration documentation for Athens), while the calls to the private proxy server are unauthenticated.

If you are using a public proxy server, you can set the `GOPRIVATE` environment variable to a comma-separated list of your private repositories. For example, if you set `GOPRIVATE` to:

```
GOPRIVATE=*.example.com,company.com/repo
```

any module stored in a repository that's located at any subdomain of *example.com* or at a URL that starts with *company.com/repo* will be downloaded directly.

Additional Details

The Go Team has a complete Go Modules Reference available online. In addition to the content in this chapter, the Module Reference also covers topics like using version control systems besides Git, the structure and API of the module cache, additional environment variables for controlling module lookup behavior, and the REST API for the module proxy and checksum database.

Exercises

1. Create a module in your own public repository. This module has a single function named `Add` with two `int` parameters and one `int` return value. This function adds the two parameters together and returns them. Make this version v1.0.0.
2. Add godoc comments to your module that describe the package and the `Add` function. Be sure to include a link to *https://www.mathsisfun.com/numbers/addition.html* in your `Add` function godoc. Make this version v1.0.1.
3. Change `Add` to make it generic. Import the `golang.org/x/exp/constraints` package. Combine the `Integer` and `Float` types in that package to create an interface called `Number`. Rewrite `Add` to take in two parameters of type `Number` and return a value of type `Number`. Version your module again. Because this is a backward-breaking change, this should be v2.0.0 of your module.

Wrapping Up

In this chapter, you've learned how to organize code and interact with the ecosystem of Go source code. You've seen how modules work, how to organize your code into packages, how to use third-party modules, and how to release modules of your own. In the next chapter, you're going to take a look at more of the development tools that are included with Go, learn about some essential third-party tools, and explore some techniques to give you better control over your build process.