SAMPLE CHAPTER

GO IN ACTION

William Kennedy
WITH Brian Ketelsen
Erik St. Martin
FOREWORD BY Steve Francia

MANNING

Go in Action

by William Kennedy
with Brian Ketelsen
and Erik St. Martin

Chapter 2

brief contents

1 ■ Introducing Go 1
2 ■ Go quick-start 9
3 ■ Packaging and tooling 39
4 ■ Arrays, slices, and maps 57
5 ■ Go's type system 88
6 ■ Concurrency 128
7 ■ Concurrency patterns 158
8 ■ Standard library 184
9 ■ Testing and benchmarking 211

2 Go quick-start

In this chapter

- Reviewing a comprehensive Go program
- Declaring types, variables, functions, and methods
- Launching and synchronizing goroutines
- Writing generic code using interfaces
- Handling errors as normal program logic

Go has its own elegance and programming idioms that make the language productive and fun to code in. The language designers set out to create a language that would let them be productive without losing access to the lower-level programming constructs they needed. This balance is achieved through a minimized set of keywords, built-in functions, and syntax. Go also provides a comprehensive standard library. The standard library provides all the core packages programmers need to build real-world web- and network-based programs.

To see this in action, we'll review a complete Go program that implements functionality that can be found in many Go programs being developed today. The program pulls different data feeds from the web and compares the content against a search term. The content that matches is then displayed in the terminal window.

The program reads text files, makes web calls, and decodes both XML and JSON into struct type values, and it does all of this using Go concurrency to make things fast.

You can download and review the code in your favorite editor by navigating to the book repository for this chapter:

```
https://github.com/goinaction/code/tree/master/chapter2/sample
```

Don't feel that you need to understand everything you read and review in this chapter the first, second, or even the third time. Though many of the programming concepts you know today can be applied when learning Go, Go also has its unique idioms and style. If you can liberate yourself from your current programming language and look at Go with a fresh set of eyes and a clear mind, you'll find it easier to understand and appreciate, and you'll see Go's elegance.

2.1 Program architecture

Before we dive into the code, let's review the architecture behind the program (shown in figure 2.1) and see how searching all the different feeds is accomplished.

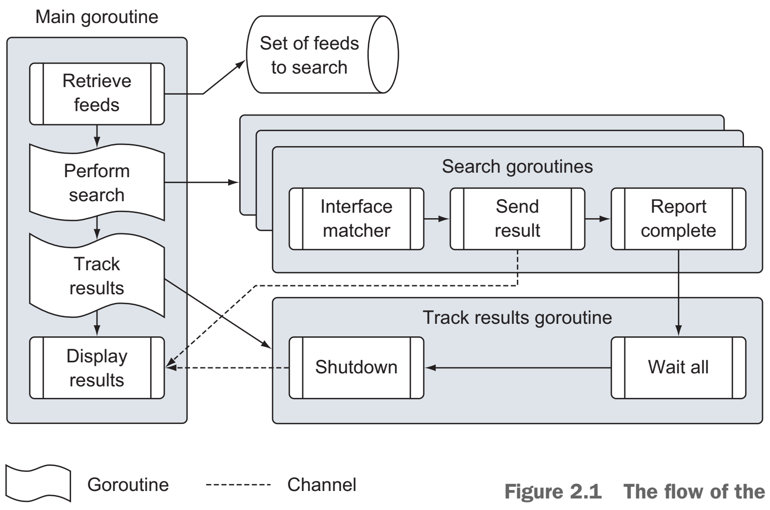

Figure 2.1 The flow of the program architecture

The program is broken into several distinct steps that run across many different goroutines. We'll explore the code as it flows from the main goroutine into the searching and tracking goroutines and then back to the main goroutine. To start, here's the structure of the project.

Listing 2.1 Project structure for the application

```
cd $GOPATH/src/github.com/goinaction/code/chapter2

- sample
    - data
        data.json   -- Contains a list of data feeds
    - matchers
```

```
        rss.go      -- Matcher for searching rss feeds
    - search
        default.go  -- Default matcher for searching data
        feed.go     -- Support for reading the json data file
        match.go    -- Interface support for using different matchers
        search.go   -- Main program logic for performing search
    main.go         -- Programs entry point
```

The code is organized into these four folders, which are listed in alphabetical order. The data folder contains a JSON document of data feeds the program will retrieve and process to match the search term. The matchers folder contains the code for the different types of feeds the program supports. Currently the program only supports one matcher that processes RSS type feeds. The search folder contains the business logic for using the different matchers to search content. Finally we have the parent folder, sample, which contains the main.go code file, which is the entry point for the program.

Now that you've seen where all the code for the program is, you can begin to explore and understand how the program works. Let's start with the entry point for the program.

2.2 Main package

The program's entry point can be found in the main.go code file. Even though there are only 21 lines of code, there are a few things going on that we have to mention.

Listing 2.2 main.go

```
01 package main
02
03 import (
04    "log"
05    "os"
06
07   _ "github.com/goinaction/code/chapter2/sample/matchers"
08    "github.com/goinaction/code/chapter2/sample/search"
09 )
10
11 // init is called prior to main.
12 func init() {
13     // Change the device for logging to stdout.
14     log.SetOutput(os.Stdout)
15 }
16
17 // main is the entry point for the program.
18 func main() {
19     // Perform the search for the specified term.
20     search.Run("president")
21 }
```

Every Go program that produces an executable has two distinct features. One of those features can be found on line 18. There you can see the function `main` declared. For the build tools to produce an executable, the function `main` must be declared, and it becomes the entry point for the program. The second feature can be found on line 01 of program.

Listing 2.3 main.go: line 01

```
01 package main
```

You can see the function `main` is located in a package called `main`. If your `main` function doesn't exist in package `main`, the build tools won't produce an executable.

Every code file in Go belongs to a package, and main.go is no exception. We'll go into much more detail about packages in chapter 3, because packages are an important feature of Go. For now, understand that packages define a unit of compiled code and their names help provide a level of indirection to the identifiers that are declared inside of them, just like a namespace. This makes it possible to distinguish identifiers that are declared with exactly the same name in the different packages you import.

Now turn your attention to lines 03 through 09 of the main.go code file, which declares imports.

Listing 2.4 main.go: lines 03–09

```
03 import (
04    "log"
05    "os"
06
07   _ "github.com/goinaction/code/chapter2/sample/matchers"
08    "github.com/goinaction/code/chapter2/sample/search"
09 )
```

Imports are just that: they import code and give you access to identifiers such as types, functions, constants, and interfaces. In our case, the code in the main.go code file can now reference the `Run` function from the `search` package, thanks to the import on line 08. On lines 04 and 05, we import code from the standard library for the `log` and `os` packages.

All code files in a folder must use the same package name, and it's common practice to name the package after the folder. As stated before, a package defines a unit of compiled code, and each unit of code represents a package. If you quickly look back at listing 2.1, you'll see how we have a folder in this project called `search` that matches the import path on line 08.

You may have noticed that on line 07 we import the `matchers` package and use the blank identifier before listing out the import path.

Listing 2.5 main.go: line 07

```
07    _ "github.com/goinaction/code/chapter2/sample/matchers"
```

This is a technique in Go to allow initialization from a package to occur, even if you don't directly use any identifiers from the package. To make your programs more readable, the Go compiler won't let you declare a package to be imported if it's not used. The blank identifier allows the compiler to accept the import and call any `init` functions that can be found in the different code files within that package. For our program, this is required because the rss.go code file in the `matchers` package contains an `init` function to register the RSS matcher for use. We'll come back to how all this works later.

The main.go code file also has an `init` function that's declared on lines 12 through 15.

Listing 2.6 main.go: lines 11–15

```
11 // init is called prior to main.
12 func init() {
13     // Change the device for logging to stdout.
14     log.SetOutput(os.Stdout)
15 }
```

All `init` functions in any code file that are part of the program will get called before the `main` function. This `init` function sets the logger from the standard library to write to the `stdout` device. By default, the logger is set to write to the `stderr` device. In chapter 7 we'll talk more about the `log` package and other important packages from the standard library.

Finally, let's look at the one statement that the `main` function performs on line 20.

Listing 2.7 main.go: lines 19–20

```
19     // Perform the search for the specified term.
20     search.Run("president")
```

Here you see a call to the `Run` function that belongs to the `search` package. This function contains the core business logic for the program, which requires a string for the search term. Once the `Run` function returns, the program will terminate.

Now we can look at the code that belongs to the `search` package.

2.3 Search package

The `search` package contains the framework and business logic for the program. The package is organized into four different code files, each with a unique responsibility. As we continue to follow the logic of the program, we'll explore each of these different code files.

Let's briefly talk about what a matcher is, since the entire program revolves around the execution of matchers. A matcher in our program is a value that contains specific intelligence for processing a feed type. In our program we have two matchers. The framework implements a default matcher that has no intelligence, and in the `matchers` package we have an implementation of an RSS matcher. The RSS matcher knows how to get, read, and search RSS feeds. Later on we could extend the program to use matchers that could read JSON documents or CSV files. We'll talk more about how to implement matchers later.

2.3.1 search.go

Following are the first nine lines of code that can be found inside the search.go code file. This is the code file where the `Run` function is located.

Listing 2.8 search/search.go: lines 01–09

```
01 package search
02
03 import (
04     "log"
05     "sync"
06 )
07
08 // A map of registered matchers for searching.
09 var matchers = make(map[string]Matcher)
```

As you'll see, each code file will contain the keyword `package` at the top with a name for the package. Each code file in the search folder will contain `search` for the package name. The lines from 03 through 06 import the `log` and `sync` packages from the standard library.

When you import code from the standard library, you only need to reference the name of the package, unlike when you import code from outside of the standard library. The compiler will always look for the packages you import at the locations referenced by the `GOROOT` and `GOPATH` environment variables.

Listing 2.9 `GOROOT` and `GOPATH` environmental variables

```
GOROOT="/Users/me/go"
GOPATH="/Users/me/spaces/go/projects"
```

The `log` package provides support for logging messages to the `stdout`, `stderr`, or even custom devices. The `sync` package provides support for synchronizing goroutines, which is required by our program. On line 09 you'll see our first variable declaration.

Listing 2.10 search/search.go: lines 08–09

```
08 // A map of registered matchers for searching.
09 var matchers = make(map[string]Matcher)
```

This variable is located outside the scope of any function and so is considered a package-level variable. The variable is declared using the keyword `var` and is declared as a `map` of `Matcher` type values with a key of type `string`. The declaration for the

`Matcher` type can be found in the match.go code file, and we'll describe the purpose of this type later. There's another important aspect of this variable declaration: the name of the variable `matchers` starts with a lowercase letter.

In Go, identifiers are either exported or unexported from a package. An exported identifier can be directly accessed by code in other packages when the respective package is imported. These identifiers start with a capital letter. Unexported identifiers start with a lowercase letter and can't be directly accessed by code in other packages. But just because an identifier is unexported, it doesn't mean other packages can't indirectly access these identifiers. As an example, a function can return a value of an unexported type and this value is accessible by any calling function, even if the calling function has been declared in a different package.

This variable declaration also contains an initialization of the variable via the assignment operator and a special built-in function called `make`.

Listing 2.11 Making a map

```
make(map[string]Matcher)
```

A `map` is a reference type that you're required to `make` in Go. If you don't make the `map` first and assign it to your variable, you'll receive errors when you try to use the `map` variable. This is because the zero value for a `map` variable is `nil`. In chapter 4 we'll go into greater detail about maps.

In Go, all variables are initialized to their zero value. For numeric types, that value is `0`; for strings it's an empty string; for Booleans it's `false`; and for pointers, the zero value is `nil`. When it comes to reference types, there are underlying data structures that are initialized to their zero values. But variables declared as a reference type set to their zero value will return the value of `nil`.

Now let's walk through the `Run` function that's called by the `main` function, which you saw earlier.

Listing 2.12 search/search.go: lines 11–57

```
11 // Run performs the search logic.
12 func Run(searchTerm string) {
13     // Retrieve the list of feeds to search through.
14     feeds, err := RetrieveFeeds()
15     if err != nil {
16         log.Fatal(err)
17     }
18
19     // Create a unbuffered channel to receive match results.
20     results := make(chan *Result)
21
22     // Setup a wait group so we can process all the feeds.
23     var waitGroup sync.WaitGroup
24
25     // Set the number of goroutines we need to wait for while
26     // they process the individual feeds.
27     waitGroup.Add(len(feeds))
28
29     // Launch a goroutine for each feed to find the results.
30     for _, feed := range feeds {
31         // Retrieve a matcher for the search.
32         matcher, exists := matchers[feed.Type]
33         if !exists {
34             matcher = matchers["default"]
35         }
36
37         // Launch the goroutine to perform the search.
38         go func(matcher Matcher, feed *Feed) {
39             Match(matcher, feed, searchTerm, results)
40             waitGroup.Done()
41         }(matcher, feed)
42     }
43
44     // Launch a goroutine to monitor when all the work is done.
45     go func() {
46         // Wait for everything to be processed.
47         waitGroup.Wait()
48
49         // Close the channel to signal to the Display
50         // function that we can exit the program.
51         close(results)
52     }()
53
54     // Start displaying results as they are available and
55     // return after the final result is displayed.
56     Display(results)
57 }
```

The Run function contains the main control logic for the program. It's a good representation of how Go programs can be structured to handle the launching and synchronization of goroutines that run concurrently. Let's walk through the logic section by section, and then explore the other code files that lend their support.

Let's review how the Run function is declared.

Listing 2.13 search/search.go: lines 11–12

```
11 // Run performs the search logic.
12 func Run(searchTerm string) {
```

To declare a function in Go, use the keyword `func` followed by the function name, any parameters, and then any return values. In the case of `Run`, you have a single parameter called `searchTerm` of type `string`. The term the program will search against is passed into the `Run` function, and if you look at the `main` function again, you can see that exchange.

Listing 2.14 main.go: lines 17–21

```
17 // main is the entry point for the program.
18 func main() {
19     // Perform the search for the specified term.
20     search.Run("president")
21 }
```

The first thing that the `Run` function does is retrieve a list of data feeds. These feeds are used to pull content from the internet that is then matched against the specified search term.

Listing 2.15 search/search.go: lines 13–17

```
13     // Retrieve the list of feeds to search through.
14     feeds, err := RetrieveFeeds()
15     if err != nil {
16         log.Fatal(err)
17     }
```

There are a few important concepts here that we need to go through. You can see on line 14 that we make a function call to the function `RetrieveFeeds`. This function belongs to the `search` package and returns two values. The first return value is a slice of `Feed` type values. A slice is a reference type that implements a dynamic array. You use slices in Go to work with lists of data. Chapter 4 goes into greater detail about slices.

The second return value is an error. On line 15, the error value is evaluated for errors, and if an error did occur, the function `Fatal` from the `log` package is called. The `Fatal` function accepts an error value and will log to the terminal window before terminating the program.

Though not unique to Go, you can see that our functions can have multiple return values. It's common to declare functions that return a value and an error value just like the `RetrieveFeeds` function. If an error occurs, never trust the other values being returned from the function. They should always be ignored, or else you run the risk of the code generating more errors or panics.

Let's take a closer look at how the values being returned from the function are being assigned to variables.

Listing 2.16 search/search.go: lines 13–14

```
13     // Retrieve the list of feeds to search through.
14     feeds, err := RetrieveFeeds()
```

Here you see the use of the short variable declaration operator (`:=`). This operator is used to both declare and initialize variables at the same time. The type of each value being returned is used by the compiler to determine the type for each variable, respectively. The short variable declaration operator is just a shortcut to streamline your code and make the code more readable. The variable it declares is no different than any other variable you may declare when using the keyword `var`.

Now that we have our list of data feeds, we can move on to the next line of code.

Listing 2.17 search/search.go: lines 19–20

```
19     // Create a unbuffered channel to receive match results.
20     results := make(chan *Result)
```

On line 20, we use the built-in function `make` to create an unbuffered channel. We use the short variable declaration operator to declare and initialize the channel variable with the call to `make`. A good rule of thumb when declaring variables is to use the keyword `var` when declaring variables that will be initialized to their zero value, and to use the short variable declaration operator when you're providing extra initialization or making a function call.

Channels are also a reference type in Go like maps and slices, but channels implement a queue of typed values that are used to communicate data between goroutines. Channels provide inherent synchronization mechanisms to make communication safe. In chapter 6 we'll go into more details about channels and goroutines.

The next two lines of code are used later to prevent the program from terminating before all the search processing is complete.

Listing 2.18 search/search.go: lines 22–27

```
22     // Setup a wait group so we can process all the feeds.
23     var waitGroup sync.WaitGroup
24
25     // Set the number of goroutines we need to wait for while
26     // they process the individual feeds.
27     waitGroup.Add(len(feeds))
```

In Go, once the `main` function returns, the program terminates. Any goroutines that were launched and are still running at this time will also be terminated by the Go runtime. When you write concurrent programs, it's best to cleanly terminate any goroutines that were launched prior to letting the `main` function return. Writing programs that can cleanly start and shut down helps reduce bugs and prevents resources from corruption.

Our program is using a `WaitGroup` from the `sync` package to track all the goroutines we're going to launch. A `WaitGroup` is a great way to track when a goroutine is finished performing its work. A `WaitGroup` is a counting semaphore, and we'll use it to count off goroutines as they finish their work.

On line 23 we declare a variable of type `WaitGroup` from the `sync` package. Then on line 27 we set the value of the `WaitGroup` variable to match the number of goroutines we're going to launch. As you'll soon see, we'll process each feed concurrently with its own goroutine. As each goroutine completes its work, it will decrement the count of the `WaitGroup` variable, and once the variable gets to zero, we'll know all the work is done.

Next let's look at the code that launches these goroutines for each feed.

Listing 2.19 search/search.go: lines 29–42

```
29    // Launch a goroutine for each feed to find the results.
30    for _, feed := range feeds {
31      // Retrieve a matcher for the search.
32      matcher, exists := matchers[feed.Type]
33      if !exists {
34        matcher = matchers["default"]
35      }
36
37      // Launch the goroutine to perform the search.
38      go func(matcher Matcher, feed *Feed) {
39        Match(matcher, feed, searchTerm, results)
40        waitGroup.Done()
41      }(matcher, feed)
42    }
```

The code for lines 30 through 42 iterate through the list of data feeds we retrieved earlier and launch a goroutine for each one. To iterate over the slice of feeds, we use the keywords `for range`. The keyword `range` can be used with arrays, strings, slices, maps, and channels. When we use `for range` to iterate over a slice, we get two values back on each iteration. The first is the index position of the element we're iterating over, and the second is a copy of the value in that element.

If you look closer at the `for range` statement on line 30, you'll see the use of the blank identifier again.

Listing 2.20 search/search.go: lines 29–30

```
29    // Launch a goroutine for each feed to find the results.
30    for _, feed := range feeds {
```

This is the second time you see the blank identifier being used. You first saw it in main.go when we imported the `matchers` package. Now it's being used as a substitution for the variable that would be assigned to the index value for the range call. When you have a function that returns multiple values, and you don't have a need for one, you can use the blank identifier to ignore those values. In our case with this range, we won't be using the index value, so the blank identifier allows us to ignore it.

The first thing we do in the loop is check the map for a `Matcher` value that can be used to process a feed of the specific feed type.

Listing 2.21 search/search.go: lines 31–35

```
31      // Retrieve a matcher for the search.
32      matcher, exists := matchers[feed.Type]
```

```
33          if !exists {
34              matcher = matchers["default"]
35          }
```

We haven't talked about how this map gets its values yet. You'll see later on how the program initializes itself and populates this map. On line 32 we check the map for a key that matches the feed type. When looking up a key in a map, you have two options: you can assign a single variable or two variables for the lookup call. The first variable is always the value returned for the key lookup, and the second value, if specified, is a Boolean flag that reports whether the key exists or not. When a key doesn't exist, the map will return the zero value for the type of value being stored in the map. When the key does exist, the map will return a copy of the value for that key.

On line 33 we check whether the key was located in the map, and if it's not, we assign the default matcher to be used. This allows the program to function without causing any issues or interruption for feeds that the program currently doesn't support. Then we launch a goroutine to perform the search.

Listing 2.22 search/search.go: lines 37–41

```
37          // Launch the goroutine to perform the search.
38          go func(matcher Matcher, feed *Feed) {
39              Match(matcher, feed, searchTerm, results)
40              waitGroup.Done()
41          }(matcher, feed)
```

In chapter 6 we'll go into more detail about goroutines, but for now a *goroutine* is a function that's launched to run independently from other functions in the program. Use the keyword `go` to launch and schedule goroutines to run concurrently. On line 38 we use the keyword `go` to launch an anonymous function as a goroutine. An *anonymous function* is a function that's declared without a name. In our `for range` loop, we launch an anonymous function as a goroutine for each feed. This allows each feed to be processed independently in a concurrent fashion.

Anonymous functions can take parameters, which we declare for this anonymous function. On line 38 we declare the anonymous function to accept a value of type `Matcher` and the address of a value of type `Feed`. This means the variable `feed` is a *pointer variable*. Pointer variables are great for sharing variables between functions. They allow functions to access and change the state of a variable that was declared within the scope of a different function and possibly a different goroutine.

On line 41 the values of the `matcher` and `feed` variables are being passed into the anonymous function. In Go, all variables are passed by value. Since the value of a pointer variable is the address to the memory being pointed to, passing pointer variables between functions is still considered a pass by value.

On lines 39 and 40 you see the work each goroutine is performing.

Listing 2.23 search/search.go: lines 39–40

```
39      Match(matcher, feed, searchTerm, results)
40      waitGroup.Done()
```

The first thing the goroutine does is call a function called `Match`, which can be found in the match.go code file. The `Match` function takes a value of type `Matcher`, a pointer to a value of type `Feed`, the search term, and the channel where the results are written to. We'll look at the internals of this function later, but for now it's enough to know that `Match` will search the feed and output matches to the `results` channel.

Once the function call to `Match` completes, we execute the code on line 40, which is to decrement the `WaitGroup` count. Once every goroutine finishes calling the `Match` function and the `Done` method, the program will know every feed has been processed. There's something else interesting about the method call to `Done`: the `WaitGroup` value was never passed into the anonymous function as a parameter, yet the anonymous function has access to it.

Go supports closures and you're seeing this in action. In fact, the `searchTerm` and `results` variables are also being accessed by the anonymous function via closures. Thanks to closures, the function can access those variables directly without the need to pass them in as parameters. The anonymous function isn't given a copy of these variables; it has direct access to the same variables declared in the scope of the outer function. This is the reason why we don't use closures for the `matcher` and `feed` variables.

Listing 2.24 search/search.go: lines 29–32

```
29  // Launch a goroutine for each feed to find the results.
30  for _, feed := range feeds {
31    // Retrieve a matcher for the search.
32    matcher, exists := matchers[feed.Type]
```

The values of the `feed` and `matcher` variables are changing with each iteration of the loop, as you can see on lines 30 and 32. If we used closures for these variables, as the values of these variables changed in the outer function, those changes would be reflected in the anonymous function. All the goroutines would be sharing the same variables as the outer function thanks to closures. Unless we passed these values in as function parameters, most of the goroutines would end up processing the same feed using the same matcher—most likely the last one in the `feeds` slice.

With all the search goroutines working, sending results on the `results` channel and decrementing the `waitGroup` counter, we need a way to display those results and keep the `main` function alive until all the processing is done.

Listing 2.25 search/search.go: lines 44–57

```
44  // Launch a goroutine to monitor when all the work is done.
45  go func() {
46    // Wait for everything to be processed.
47          waitGroup.Wait()
48
49          // Close the channel to signal to the Display
50          // function that we can exit the program.
51          close(results)
52      }()
53
54      // Start displaying results as they are available and
55      // return after the final result is displayed.
56      Display(results)
57 }
```

The code between lines 45 and 56 is tricky to explain until we dive deeper into some of the other code in the `search` package. For now let's describe what we see and come back to it later to understand the mechanics. On lines 45 through 52 we launch yet another anonymous function as a goroutine. This anonymous function takes no parameters and uses closures to access both the `WaitGroup` and `results` variables. This goroutine calls the method `Wait` on the `WaitGroup` value, which is causing the goroutine to block until the count for the `WaitGroup` hits zero. Once that happens, the goroutine calls the built-in function `close` on the channel, which as you'll see causes the program to terminate.

The final piece of code in the `Run` function is on line 56. This is a call to the `Display` function, which can be found in the match.go code file. Once this function returns, the program terminates. This doesn't happen until all the results in the channel are processed.

2.3.2 *feed.go*

Now that you've seen the `Run` function, let's look at the code behind the function call to `RetrieveFeeds` on line 14 of the search.go code file. This function reads the data.json file and returns the slice of data feeds. These feeds drive the content that will be searched by the different matchers. Here are the first eight lines of code that can be found inside the feed.go code file.

Listing 2.26 feed.go: lines 01–08

```
01 package search
02
03 import (
04     "encoding/json"
05     "os"
06 )
07
08 const dataFile = "data/data.json"
```

This code file exists in the search folder, and on line 01 the code file is declared to be in package `search`. You can see that on lines 03 through 06 we import two packages from the standard library. The `json` package provides support for encoding and decoding JSON and the `os` package provides support for accessing operating system functionality like reading files.

You may have noticed that to import the `json` package, we needed to specify a path that includes the encoding folder. Regardless of the path we specify, the name of the package is `json`. The physical location of the package from within the standard library doesn't change this fact. As we access functionality from the `json` package, we'll use just the name `json`.

On line 08 we declare a constant named `dataFile`, which is assigned a string that specifies the relative path to the data file on disk. Since the Go compiler can deduce the type from the value on the right side of the assignment operator, specifying the type when declaring the constant is unnecessary. We also use a lowercase letter for the name of the constant, which means this constant is unexported and can only be directly accessed by code within the `search` package.

Next let's look at a portion of the data.json data file.

Listing 2.27 data.json

```
[
    {
        "site" : "npr",
        "link" : "http://www.npr.org/rss/rss.php?id=1001",
        "type" : "rss"
    },
    {
        "site" : "cnn",
        "link" : "http://rss.cnn.com/rss/cnn_world.rss",
        "type" : "rss"
    },
    {
        "site" : "foxnews",
        "link" : "http://feeds.foxnews.com/foxnews/world?format=xml",
        "type" : "rss"
    },
    {
        "site" : "nbcnews",
        "link" : "http://feeds.nbcnews.com/feeds/topstories",
        "type" : "rss"
    }
]
```

The actual data file contains more than four data feeds, but listing 2.27 shows a valid version of the data file. The data file contains an array of JSON documents. Each document in the data file provides a name of the site we're getting the data from, a link to the data, and the type of data we expect to receive.

These documents need to be decoded into a slice of struct types so we can use this data in our program. Let's look at the struct type that will be used to decode this data file.

Listing 2.28 feed.go: lines 10–15

```
10 // Feed contains information we need to process a feed.
11 type Feed struct {
12     Name string `json:"site"`
13     URI  string `json:"link"`
14     Type string `json:"type"`
15 }
```

On lines 11 through 15 we declare a struct type named `Feed`, which is an exported type. This type is declared with three fields, each of which are strings that match the fields for each document in the data file. If you look at each field declaration, tags have been included to provide the metadata that the JSON decoding function needs to create the slice of `Feed` type values. Each tag maps a field name in the struct type to a field name in the document.

Now we can review the `RetrieveFeeds` function that we called on line 14 in the search.go code file. This is the function that reads the data file and decodes every document into a slice of `Feed` type values.

Listing 2.29 feed.go: lines 17–36

```
17 // RetrieveFeeds reads and unmarshals the feed data file.
18 func RetrieveFeeds() ([]*Feed, error) {
19     // Open the file.
20     file, err := os.Open(dataFile)
21     if err != nil {
22         return nil, err
23     }
24
25     // Schedule the file to be closed once
26     // the function returns.
27     defer file.Close()
28
29     // Decode the file into a slice of pointers
30     // to Feed values.
31     var feeds []*Feed
32     err = json.NewDecoder(file).Decode(&feeds)
33
34     // We don't need to check for errors, the caller can do this.
35     return feeds, err
36 }
```

Let's start with the declaration of the function on line 18. The function takes no parameters and returns two values. The first return value is a slice of pointers to `Feed` type values. The second return value is an error value that reports back if the function call was successful. As you'll continue to see, returning error values is common practice in this code example and throughout the standard library.

Now let's look at lines 20 through 23, where we use the `os` package to open the data file. The call to the `Open` method takes the relative path to our data file and returns two values. The first return value is a pointer to a value of type `File`, and the second return value is an error to check if the call to `Open` was successful. Immediately on line 21 we check the error value and return the error if we did have a problem opening the file.

If we're successful in opening the file, we then move to line 27. Here you see the use of the keyword defer.

Listing 2.30 feed.go: lines 25–27

```
25     // Schedule the file to be closed once
26     // the function returns.
27     defer file.Close()
```

The keyword defer is used to schedule a function call to be executed right after a function returns. It's our responsibility to close the file once we're done with it. By using the keyword defer to schedule the call to the close method, we can guarantee that the method will be called. This will happen even if the function panics and terminates unexpectedly. The keyword defer lets us write this statement close to where the opening of the file occurs, which helps with readability and reducing bugs.

Now we can review the final lines of code in the function. Let's look at lines 31 through 35.

Listing 2.31 feed.go: lines 29–36

```
29     // Decode the file into a slice of pointers
30     // to Feed values.
31     var feeds []*Feed
32     err = json.NewDecoder(file).Decode(&feeds)
33
34     // We don't need to check for errors, the caller can do this.
35     return feeds, err
36 }
```

On line 31 we declare a nil slice named feeds that contains pointers to Feed type values. Then on line 32 we make a call to the Decode method off the value returned by the NewDecoder function from the json package. The NewDecoder function takes the file handle we created from the method call to Open and returns a pointer to a value of type Decoder. From that value we call the Decode method, passing the address to the slice. The Decode method then decodes the data file and populates our slice with a set of Feed type values.

The Decode method can accept any type of value thanks to its declaration.

Listing 2.32 Using the empty interface

```
func (dec *Decoder) Decode(v interface{}) error
```

The parameter for the Decode method accepts a value of type interface{}. This is a special type in Go and works with the reflection support that can be found in the reflect package. In chapter 9 we'll go into more detail about reflection and how this method works.

The last line of code on line 35 returns the slice and error values back to the calling function. In this case there's no need for the function to check the error value after the call to Decode. The function is complete and the calling function can check the error value and determine what to do next.

Now it's time to see how the search code supports different types of feed implementations by reviewing the matcher code.

2.3.3 match.go/default.go

The match.go code file contains the support for creating different types of matchers that can be used by the search Run function. Let's go back and look at the code from the Run function that executes the search using the different types of matchers.

Listing 2.33 search/search.go : lines 29 - 42

```
29     // Launch a goroutine for each feed to find the results.
30     for _, feed := range feeds {
31         // Retrieve a matcher for the search.
32         matcher, exists := matchers[feed.Type]
33         if !exists {
34             matcher = matchers["default"]
35         }
36
37         // Launch the goroutine to perform the search.
38         go func(matcher Matcher, feed *Feed) {
39             Match(matcher, feed, searchTerm, results)
40             waitGroup.Done()
41         }(matcher, feed)
42     }
```

The code on line 32 looks up a matcher value based on the feed type; that value is then used to process a search against that specific feed. Then on line 38 through 41, a goroutine is launched for that matcher and feed value. The key to making this code work is the ability of this framework code to use an interface type to capture and call into the specific implementation for each matcher value. This allows the code to handle different types of matcher values in a consistent and generic way. Let's look at the code in match.go and see how we're able to implement this functionality.

Here are the first 17 lines of code for match.go.

Listing 2.34 search/match.go: lines 01–17

```
01 package search
02
03 import (
04     "log"
05 )
06
07 // Result contains the result of a search.
08 type Result struct {
09     Field   string
10     Content string
11 }
12
```

```
13 // Matcher defines the behavior required by types that want
14 // to implement a new search type.
15 type Matcher interface {
16     Search(feed *Feed, searchTerm string) ([]*Result, error)
17 }
```

Let's jump to lines 15 through 17 and look at the declaration of the interface type named `Matcher`. Up until now we've only been declaring struct types, but here you see code that's declaring an `interface` type. We'll get into a lot more detail about interfaces in chapter 5, but for now know that interfaces declare behavior that's required to be implemented by struct or named types to satisfy the interface. The behavior of an interface is defined by the methods that are declared within the interface type.

In the case of the `Matcher` interface, there's only one method declared, `Search`, which takes a pointer to a value of type `Feed` and a search term of type `string`. The method also returns two values: a slice of pointers to values of type `Result` and an error value. The `Result` type is declared on lines 08 through 11.

You follow a naming convention in Go when naming interfaces. If the interface type contains only one method, the name of the interface ends with the *er* suffix. This is the exact case for our interface, so the name of the interface is `Matcher`. When multiple methods are declared within an interface type, the name of the interface should relate to its general behavior.

For a user-defined type to implement an interface, the type in question needs to implement all the methods that are declared within that interface type. Let's switch to the default.go code file and see how the default matcher implements the `Matcher` interface.

Listing 2.35 search/default.go: lines 01–15

```
01 package search
02
03 // defaultMatcher implements the default matcher.
04 type defaultMatcher struct{}
05
06 // init registers the default matcher with the program.
07 func init() {
08     var matcher defaultMatcher
09     Register("default", matcher)
10 }
11
12 // Search implements the behavior for the default matcher.
13 func (m defaultMatcher) Search(feed *Feed, searchTerm string)
                                                    ([]*Result, error) {
14     return nil, nil
15 }
```

On line 04 we declare a struct type named `defaultMatcher` using an empty struct. An empty struct allocates zero bytes when values of this type are created. They're great when you need a type but not any state. For the default matcher, we don't need to maintain any state; we only need to implement the interface.

On lines 13 through 15 you see the implementation of the `Matcher` interface by the `defaultMatcher` type. The implementation of the interface method `Search` just returns `nil` for both return values. Other implementations, such as the implementation for the RSS matcher, will implement the specific business rules for processing searches in their version of this method.

The declaration of the `Search` method is declared with a value receiver of type `defaultMatcher`.

Listing 2.36 search/default.go: line 13

```
13 func (m defaultMatcher) Search
```

The use of a receiver with any function declaration declares a method that's bound to the specified receiver type. In our case, the declaration of the `Search` method is now bound to values of type `defaultMatcher`. This means we can call the method `Search` from values and pointers of type `defaultMatcher`. Whether we use a value or pointer of the receiver type to make the method call, the compiler will reference or dereference the value if necessary to support the call.

Listing 2.37 Example of method calls

```
// Method declared with a value receiver of type defaultMatcher
func (m defaultMatcher) Search(feed *Feed, searchTerm string)

// Declare a pointer of type defaultMatch
dm := new(defaultMatch)

// The compiler will dereference the dm pointer to make the call
dm.Search(feed, "test")


// Method declared with a pointer receiver of type defaultMatcher
func (m *defaultMatcher) Search(feed *Feed, searchTerm string)

// Declare a value of type defaultMatch
var dm defaultMatch

// The compiler will reference the dm value to make the call
dm.Search(feed, "test")
```

It's best practice to declare methods using pointer receivers, since many of the methods you implement need to manipulate the state of the value being used to make the method call. In the case of the `defaultMatcher` type, we want to use a value receiver because creating values of type `defaultMatcher` result in values of zero allocation. Using a pointer makes no sense since there's no state to be manipulated.

Unlike when you call methods directly from values and pointers, when you call a method via an interface type value, the rules are different. Methods declared with pointer receivers can only be called by interface type values that contain pointers. Methods declared with value receivers can be called by interface type values that contain both values and pointers.

Listing 2.38 Example of interface method call restrictions

```
// Method declared with a pointer receiver of type defaultMatcher
func (m *defaultMatcher) Search(feed *Feed, searchTerm string)

// Call the method via an interface type value
var dm defaultMatcher
var matcher Matcher = dm      // Assign value to interface type
matcher.Search(feed, "test") // Call interface method with value

> go build
cannot use dm (type defaultMatcher) as type Matcher in assignment


// Method declared with a value receiver of type defaultMatcher
func (m defaultMatcher) Search(feed *Feed, searchTerm string)

// Call the method via an interface type value
var dm defaultMatcher
var matcher Matcher = &dm     // Assign pointer to interface type
matcher.Search(feed, "test") // Call interface method with pointer

> go build
Build Successful
```

There's nothing else that the `defaultMatcher` type needs to do to implement the interface. From this point forward, values and pointers of type `defaultMatcher` satisfy the interface and can be used as values of type `Matcher`. That's the key to making this work. Values and pointers of type `defaultMatcher` are now also values of type `Matcher` and can be assigned or passed to functions accepting values of type `Matcher`.

Let's look at the implementation of the `Match` function declared in the match.go code file. This is the function called by the `Run` function on line 39 in the search.go code file.

Listing 2.39 search/match.go: lines 19–33

```
19 // Match is launched as a goroutine for each individual feed to run
20 // searches concurrently.
21 func Match(matcher Matcher, feed *Feed, searchTerm string,
                                              results chan<- *Result) {
22     // Perform the search against the specified matcher.
23     searchResults, err := matcher.Search(feed, searchTerm)
24     if err != nil {
25         log.Println(err)
26         return
27     }
28
29     // Write the results to the channel.
30     for _, result := range searchResults {
31         results <- result
32     }
33 }
```

This is the function that performs the actual search using values or pointers that implement the `Matcher` interface. This function accepts values of type `Matcher` as the first parameter. Only values or pointers that implement the `Matcher` interface will be accepted for this parameter. Since the `defaultMatcher` type now implements the interface declared with a value receiver, values or pointers of type `defaultMatcher` can be passed into this function.

On line 23, the `Search` method is called from the `Matcher` type value that was passed into the function. Here the specific implementation of the `Search` method for the value assigned to the `Matcher` variable is executed. Once the `Search` method returns, the error value on line 24 is checked for an error. If there's an error, the function writes the error to the log and returns. If the search doesn't return an error and there are results, the results are written to the channel so that they can be picked up by the main function that's listening on that channel.

The final piece of code in match.go is the `Display` function that's called by the `main` function on line 56. This is the function preventing the program from terminating until all the results from the search goroutines are received and logged.

Listing 2.40 search/match.go: lines 35–43

```
// Display writes results to the terminal window as they
// are received by the individual goroutines.
func Display(results chan *Result) {
    // The channel blocks until a result is written to the channel.
    // Once the channel is closed the for loop terminates.
    for result := range results {
        fmt.Printf("%s:\n%s\n\n", result.Field, result.Content)
    }
}
```

A bit of channel magic allows this function to process all of the results before returning. It's based on how channels and the keyword `range` behaves when a channel is closed. Let's briefly look at the code in the `Run` function again that closes the `results` channel and calls the `Display` function.

Listing 2.41 search/search.go: lines 44–57

```
44     // Launch a goroutine to monitor when all the work is done.
45     go func() {
46         // Wait for everything to be processed.
47         waitGroup.Wait()
48
49         // Close the channel to signal to the Display
50         // function that we can exit the program.
51         close(results)
52     }()
53
54     // Start displaying results as they are available and
55     // return after the final result is displayed.
56     Display(results)
57 }
```

The goroutine on lines 45 through 52 waits on the `waitGroup` for all the search goroutines to call the `Done` method. Once the last search goroutine calls `Done`, the `Wait` method returns, and then the code on line 51 closes the `results` channel. Once the channel is closed, the goroutine terminates and is no more.

You saw on lines 30 through 32 in the match.go code file where the search results were being written to the channel.

Listing 2.42 search/match.go: lines 29–32

```
29     // Write the results to the channel.
30     for _, result := range searchResults {
31         results <- result
32     }
```

If we look back at the `for range` loop on lines 40 through 42 of the match.go code file, we can connect the writing of the results, the closing of the channel, and the processing of results all together.

Listing 2.43 search/match.go: lines 38–42

```
38     // The channel blocks until a result is written to the channel.
39     // Once the channel is closed the for loop terminates.
40     for result := range results {
41         log.Printf("%s:\n%s\n\n", result.Field, result.Content)
42     }
```

The `for range` loop on line 40 of the match.go code file will block until a result is written to the channel. As each search goroutine writes its results to the channel (as you see on line 31 of the code file match.go), the `for range` loop wakes up and is given those results. The results are then immediately written to the log. It seems this `for range` loop is stuck in an endless loop, but it isn't. Once the channel is closed on line 51 of the search.go code file, the `for range` loop is terminated and the `Display` function returns.

Before we look at the implementation of the RSS matcher, let's review how the different matchers are initialized when the program starts. To see this we need to look back at lines 07 through 10 of the default.go code file.

Listing 2.44 search/default.go: lines 06–10

```
06 // init registers the default matcher with the program.
07 func init() {
08     var matcher defaultMatcher
09     Register("default", matcher)
10 }
```

The default.go code file has a special function declared called `init`. You saw this function also declared in the main.go code file, and we talked about how all the `init` functions in the program would be called before the `main` function begins. Let's look at the imports again from the main.go code file.

Listing 2.45 main.go: lines 07–08

```
07   _ "github.com/goinaction/code/chapter2/sample/matchers"
08     "github.com/goinaction/code/chapter2/sample/search"
```

The import to the search package on line 08 allows the compiler to find the `init` function in the default.go code file. Once the compiler sees the `init` function, it's scheduled to be called prior to the `main` function being called.

The `init` function in the default.go code file is performing a special task. It's creating a value of the `defaultMatcher` type and passing that value to the `Register` function that can be found in the search.go code file.

Listing 2.46 search/search.go: lines 59–67

```
// Register is called to register a matcher for use by the program.
func Register(feedType string, matcher Matcher) {
    if _, exists := matchers[feedType]; exists {
        log.Fatalln(feedType, "Matcher already registered")
    }

    log.Println("Register", feedType, "matcher")
    matchers[feedType] = matcher
}
```

This function is responsible for adding the `Matcher` value to the map of registered matchers. All of this registration needs to happen before the `main` function gets called. Using `init` functions is a great way to accomplish this type of initialized registration.

2.4 RSS matcher

The last piece of code to review is the implementation of the RSS matcher. Everything we've reviewed up to now was to allow the implementation of different matcher types to run and search content within the program's framework. The structure of the RSS matcher is similar to the structure of the default matcher. It's the implementation of the interface method `Search` that's different and in the end gives each matcher its uniqueness.

The RSS document in listing 2.47 shows you a sample of what we expect to receive when we use any link in the data feed that's typed as an RSS feed.

Listing 2.47 Expected RSS feed document

```
<rss xmlns:npr="http://www.npr.org/rss/" xmlns:nprml="http://api
    <channel>
        <title>News</title>
        <link>...</link>
        <description>...</description>
        <language>en</language>
        <copyright>Copyright 2014 NPR - For Personal Use
        <image>...</image>
        <item>
            <title>
                Putin Says He'll Respect Ukraine Vote But U.S.
            </title>
            <description>
                The White House and State Department have called on the
            </description>
```

If you take any link from listing 2.47 and put it in a browser, you'll be able to see a complete view of the expected RSS document. The implementation of the RSS matcher pulls down these RSS documents, searches the title and description fields for the search term, and sends the results over the `results` channel. Let's start by looking at the first 12 lines of code for the rss.go code file.

Listing 2.48 matchers/rss.go: lines 01–12

```
01 package matchers
02
03 import (
04     "encoding/xml"
05     "errors"
06     "fmt"
07     "log"
08     "net/http"
09     "regexp"
10
11     "github.com/goinaction/code/chapter2/sample/search"
12 )
```

As with every code file, we start on line 01 with the name of the package. This code file can be found in a folder called `matchers`, so the package name is `matchers`. Next we have six imports from the standard library and one import to the `search` package. Again, we have some packages from the standard library being imported from subfolders within the standard library, such as `xml` and `http`. Just like with the `json` package, the name of the last folder in the path represents the name of the package.

There are four struct types that are used to decode the RSS document, so we can use the document data in our program.

Listing 2.49 matchers/rss.go: lines 14–58

```
14 type (
15     // item defines the fields associated with the item tag
16     // in the rss document.
17     item struct {
18         XMLName     xml.Name `xml:"item"`
19         PubDate     string   `xml:"pubDate"`
20         Title       string   `xml:"title"`
21         Description string   `xml:"description"`
22          Link        string   `xml:"link"`
23          GUID        string   `xml:"guid"`
24          GeoRssPoint string   `xml:"georss:point"`
25      }
26
27      // image defines the fields associated with the image tag
28      // in the rss document.
29      image struct {
30          XMLName xml.Name `xml:"image"`
31          URL     string   `xml:"url"`
32          Title   string   `xml:"title"`
33          Link    string   `xml:"link"`
34      }
35
36      // channel defines the fields associated with the channel tag
37      // in the rss document.
38      channel struct {
39          XMLName        xml.Name `xml:"channel"`
40          Title          string   `xml:"title"`
41          Description    string   `xml:"description"`
42          Link           string   `xml:"link"`
43          PubDate        string   `xml:"pubDate"`
44          LastBuildDate  string   `xml:"lastBuildDate"`
45          TTL            string   `xml:"ttl"`
46          Language       string   `xml:"language"`
47          ManagingEditor string   `xml:"managingEditor"`
48          WebMaster      string   `xml:"webMaster"`
49          Image          image    `xml:"image"`
50          Item           []item   `xml:"item"`
51      }
52
53      // rssDocument defines the fields associated with the rss document
54      rssDocument struct {
55          XMLName xml.Name `xml:"rss"`
56          Channel channel  `xml:"channel"`
57      }
58 )
```

If you match these structures to the RSS document from any of the feed links, you'll see how everything correlates. Decoding XML is identical to how we decoded JSON in the feed.go code file. Next we can look at the declaration of the `rssMatcher` type.

Listing 2.50 matchers/rss.go: lines 60–61

```
60 // rssMatcher implements the Matcher interface.
61 type rssMatcher struct{}
```

Again, this looks just like how we declared the `defaultMatcher` type. We use an empty struct since we don't need to maintain any state; we just implement the `Matcher` interface. Next we have the implementation of the matcher `init` function.

Listing 2.51 matchers/rss.go: lines 63–67

```go
// init registers the matcher with the program.
func init() {
    var matcher rssMatcher
    search.Register("rss", matcher)
}
```

Just like you saw with the default matcher, the `init` function registers a value of the `rssMatcher` type with the program for use. Let's look at the import in the main.go code file once more.

Listing 2.52 main.go: lines 07–08

```go
07    _ "github.com/goinaction/code/chapter2/sample/matchers"
08     "github.com/goinaction/code/chapter2/sample/search"
```

The code in the main.go code file doesn't directly use any identifiers from the `matchers` package. Yet we need the compiler to schedule the call to the `init` function in the rss.go code file. On line 07 we accomplish this by using the blank identifier as the alias name for the import. This allows the compiler to not produce an error for declaring the import and to locate the `init` function. With all of the imports, types, and initialization set, let's look at the two remaining methods that support the implementation of the `Matcher` interface.

Listing 2.53 matchers/rss.go: lines 114–140

```go
// retrieve performs a HTTP Get request for the rss feed and decodes
func (m rssMatcher) retrieve(feed *search.Feed)
                                                    (*rssDocument, error) {
    if feed.URI == "" {
        return nil, errors.New("No rss feed URI provided")
    }

    // Retrieve the rss feed document from the web.
    resp, err := http.Get(feed.URI)
    if err != nil {
        return nil, err
    }

    // Close the response once we return from the function.
    defer resp.Body.Close()

    // Check the status code for a 200 so we know we have received a
    // proper response.
    if resp.StatusCode != 200 {
        return nil, fmt.Errorf("HTTP Response Error %d\n",
                                                       resp.StatusCode)
    }

    // Decode the rss feed document into our struct type.
136     // We don't need to check for errors, the caller can do this.
137     var document rssDocument
138     err = xml.NewDecoder(resp.Body).Decode(&document)
139     return &document, err
140 }
```

The unexported method `retrieve` performs the logic for pulling the RSS document from the web for each individual feed link. On line 121 you can see the use of the `Get` method from the `http` package. In chapter 8 we'll explore this package more, but for now Go makes it really easy to make web requests using the `http` package. When the `Get` method returns, we'll get back a pointer to a value of type `Response`. After checking for errors, we need to schedule the call to the `Close` method, which we do on line 127.

On line 131 we check the `StatusCode` field of the `Response` value to verify we received a `200`. Anything other than `200` must be handled as an error and we do just that. If the value isn't `200`, we then return a custom error using the `Errorf` function from the `fmt` package. The last three lines of code are similar to how we decoded the JSON data file. This time we use the `xml` package and call the same function named `NewDecoder`, which returns a pointer to a `Decoder` value. With the pointer, we call the `Decode` method passing the address of the local variable named `document` of type `rssDocument`. Then the address to the `rssDocument` type value and the error from the `Decode` method call are returned.

The final method to look at implements the `Matcher` interface.

Listing 2.54 matchers/rss.go: lines 69–112

```
69 // Search looks at the document for the specified search term.
70 func (m rssMatcher) Search(feed *search.Feed, searchTerm string)
                                          ([]*search.Result, error) {
71     var results []*search.Result
72
73     log.Printf("Search Feed Type[%s] Site[%s] For Uri[%s]\n",
                                     feed.Type, feed.Name, feed.URI)
74
75     // Retrieve the data to search.
76     document, err := m.retrieve(feed)
77     if err != nil {
78         return nil, err
79     }
80
81     for _, channelItem := range document.Channel.Item {
82         // Check the title for the search term.
83         matched, err := regexp.MatchString(searchTerm,
                                                   channelItem.Title)
84         if err != nil {
85             return nil, err
86         }
87
88         // If we found a match save the result.
89         if matched {
90            results = append(results, &search.Result{
91                Field:   "Title",
92                Content: channelItem.Title,
 93                 })
 94             }
 95
 96             // Check the description for the search term.
 97             matched, err = regexp.MatchString(searchTerm,
                                                   channelItem.Description)
 98             if err != nil {
 99                 return nil, err
100             }
101
102             // If we found a match save the result.
103             if matched {
104                 results = append(results, &search.Result{
105                     Field:   "Description",
106                     Content: channelItem.Description,
107                 })
108             }
109         }
110
111         return results, nil
112 }
```

We start on line 71 with the declaration of the `results` variable, which will be used to store and return any results that may be found.

Listing 2.55 matchers/rss.go: line 71

```
71      var results []*search.Result
```

We use the keyword `var` and declare a `nil` slice of pointers to `Result` type values. The declaration of the `Result` type can be found again on line 08 of the match.go code file. Next on line 76 we make a web call using the `retrieve` method we just reviewed.

Listing 2.56 matchers/rss.go: lines 75–79

```
75      // Retrieve the data to search.
76      document, err := m.retrieve(feed)
77      if err != nil {
78          return nil, err
79      }
```

The call to the `retrieve` method returns a pointer to a value of type `rssDocument` and an error value. Then, as you've seen throughout the code, we check the error value for errors and return if there was an error. If no error exists, we then iterate through the results performing the match of the search term against the title and description of the retrieved RSS document.

Listing 2.57 matchers/rss.go: lines 81–86

```
81      for _, channelItem := range document.Channel.Item {
82          // Check the title for the search term.
83          matched, err := regexp.MatchString(searchTerm,
                                                       channelItem.Title)
```

```
84          if err != nil {
85              return nil, err
86          }
```

Since the value of `document.Channel.Item` is a slice of `item` type values, we use a `for range` loop on line 81 to iterate through all the items. On line 83 we use the `MatchString` function from the `regexp` package to match the search term against the content in the `Title` field of the `channelItem` value. Then we check for errors on line 84. If there are no errors, we move to lines 89 through 94 to check the results of the match.

Listing 2.58 matchers/rss.go: lines 88–94

```
88          // If we found a match save the result.
89          if matched {
90             results = append(results, &search.Result{
91                 Field:   "Title",
92                 Content: channelItem.Title,
93             })
94          }
```

If the value of `matched` is `true` after the call to the `MatchString` method, we use the built-in function `append` to add the search results to the `results` slice. The built-in function `append` will grow the length and capacity of the slice as it needs to. You'll learn more about the built-in function `append` in chapter 4. The first parameter to `append` is the value of the slice you want to append to, and the second parameter is the value you want to append. In our case, we use a struct literal to declare and initialize a value of type `Result`, and then we use the ampersand (`&`) operator to get the address of this new value, which is stored in the slice.

After the title is checked for matches, lines 97 through 108 perform the same logic again for the description field. Finally, on line 111, the method returns the results to the calling function.

2.5 Summary

- Every code file belongs to a package, and that package name should be the same as the folder the code file exists in.
- Go provides several ways to declare and initialize variables. If the value of a variable isn't explicitly initialized, the compiler will initialize the variable to its zero value.
- Pointers are a way of sharing data across functions and goroutines.
- Concurrency and synchronization are accomplished by launching goroutines and using channels.
- Go provides built-in functions to support using Go's internal data structures.
- The standard library contains many packages that will let you do some powerful things.
- Interfaces in Go allow you to write generic code and frameworks.

PROGRAMMING

GO IN ACTION

Kennedy • Ketelsen • Martin

Application development can be tricky enough even when you aren't dealing with complex systems programming problems like web-scale concurrency and real-time performance. While it's possible to solve these common issues with additional tools and frameworks, Go handles them right out of the box, making for a more natural and productive coding experience. Developed at Google, Go powers nimble startups as well as big enterprises—companies that rely on high-performing services in their infrastructure.

Go in Action is for any intermediate-level developer who has experience with other programming languages and wants a jump-start in learning Go or a more thorough understanding of the language and its internals. This book provides an intensive, comprehensive, and idiomatic view of Go. It focuses on the specification and implementation of the language, including topics like language syntax, Go's type system, concurrency, channels, and testing.

What's Inside

- Language specification and implementation
- Go's type system
- Internals of Go's data structures
- Testing and benchmarking

This book assumes you're a working developer proficient with another language like Java, Ruby, Python, C#, or C++.

William Kennedy is a seasoned software developer and author of the blog GoingGo.Net. **Brian Ketelsen** and **Erik St. Martin** are the organizers of GopherCon and coauthors of the Go-based Skynet framework

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/go-in-action

"A concise and comprehensive guide to exploring, learning, and using Go."
—From the Foreword by Steven Francia Creator of Hugo

"This authoritative book is a one-stop shop for anyone just starting out with Go."
—Sam Zaydel, RackTop Systems

"Brilliantly written; a comprehensive introduction to Go. Highly recommended."
—Adam McKay, SUEZ

"This book makes the uncommon parts of Go understandable and digestible."
—Alex Vidal HipChat at Atlassian

Free eBook
SEE INSERT

ISBN 13: 978-1-61729-178-4
ISBN 10: 1-61729-178-1
54499
9 781617 291784

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]