The Little Go Book

by Karl Seguin

About This Book

License

The Little Go Book is licensed under the Attribution-NonCommercial-ShareAlike 4.0 International license. You should not have paid for this book.

You are free to copy, distribute, modify or display the book. However, I ask that you always attribute the book to me, Karl Seguin, and do not use it for commercial purposes.

You can see the full text of the license at:

http://creativecommons.org/licenses/by-nc-sa/4.0/

Latest Version

The latest source of this book is available at: http://github.com/karlseguin/the-little-go-book

Introduction

I've always had a love-hate relationship when it comes to learning new languages. On the one hand, languages are so fundamental to what we do, that even small changes can have measurable impact. That *aha* moment when something clicks can have a lasting effect on how you program and can redefine your expectations of other languages. On the downside, language design is fairly incremental. Learning new keywords, type system, coding style as well as new libraries, communities and paradigms is a lot of work that seems hard to justify. Compared to everything else we have to learn, new languages often feel like a poor investment of our time.

That said, we *have* to move forward. We *have* to be willing to take incremental steps because, again, languages are the foundation of what we do. Though the changes are often incremental, they tend to have a wide scope and they impact productivity, readability, performance, testability, dependency management, error handling, documentation, profiling, communities, standard libraries, and so on. Is there a positive way to say *death by a thousand cuts*?

That leaves us with an important question: **why Go?** For me, there are two compelling reasons. The first is that it's a relatively simple language with a relatively simple standard library. In a lot of ways, the incremental nature of Go is to simplify some of the complexity we've seen being added to languages over the last couple of decades. The other reason is that for many developers, it will complement your existing arsenal.

Go was built as a system language (e.g., operating systems, device drivers) and thus aimed at C and C++ developers. According to the Go team, and which is certainly true of me, application developers, not system developers, have become the primary Go users. Why? I can't speak authoritatively for system developers, but for those of us building websites, services, desktop applications and the like, it partially comes down to the emerging need for a class of systems that sit somewhere in between low-level system applications and higher-level applications.

Maybe it's a messaging, caching, computational-heavy data analysis, command line interface, logging or monitoring. I don't know what label to give it, but over the course of my career, as systems continue to grow in complexity and as concurrency frequently measures in the tens of thousands, there's clearly been a growing need for custom infrastructure-type systems. You *can* build such systems with Ruby or Python or something else (and many people do), but these types of systems can benefit from a more rigid type system and greater performance. Similarly, you *can* use Go to build websites (and many people do), but I still prefer, by a wide margin, the expressiveness of Node or Ruby for such systems.

There are other areas where Go excels. For example, there are no dependencies when running a compiled Go program. You don't have to worry if your users have Ruby or the JVM installed, and if so, what version. For this reason, Go is becoming increasingly popular as a language for command-line interface programs and other types of utility programs you need to distribute (e.g., a log collector).

Put plainly, learning Go is an efficient use of your time. You won't have to spend long hours learning or even mastering Go, and you'll end up with something practical from your effort.

A Note from the Author

I've hesitated writing this book for a couple reasons. The first is that Go's own documentation, in particular Effective Go, is solid.

The other is my discomfort at writing a book about a language. When I wrote The Little MongoDB Book, it was safe to assume most readers understood the basics of relational database and modeling. With The Little Redis Book, you could assume a familiarity with a key value store and take it from there.

As I think about the paragraphs and chapters that lay ahead, I know that I won't be able to make those same assumptions. How much time do you spend talking about interfaces knowing that for some, the concept will be new, while others won't need much more than *Go has interfaces*? Ultimately, I take comfort in knowing that you'll let me know if some parts are too shallow or others too detailed. Consider that the price of this book.

Getting Started

If you're looking to play a little with Go, you should check out the Go Playground which lets you run code online without having to install anything. This is also the most common way to share Go code when seeking help in Go's discussion forum and places like StackOverflow.

Installing Go is straightforward. You can install it from source, but I suggest you use one of the pre-compiled binaries. When you go to the download page, you'll see installers for various platforms. Let's avoid these and learn how to set up Go ourselves. As you'll see, it isn't hard.

Except for simple examples, Go is designed to work when your code is inside a workspace. The workspace is a folder composed of `bin`, `pkg` and `src` subfolders. You might be tempted to force Go to follow your own style - don't.

Normally, I put my projects inside of `~/code`. For example, `~/code/blog` contains my blog. For Go, my workspace is `~/code/go` and my Go-powered blog would be in `~/code/go/src/blog`. Since that's a lot to type, I use a symbolic link to make it accessible via `~/code/blog`:

```
ln -s ~/code/go/src/blog ~/code/blog
```

In short, create a `go` folder with a `src` subfolder wherever you expect to put your projects.

OSX / Linux

Download the `tar.gz` for your platform. For OSX, you'll most likely be interested in `go#.#.#.darwin-amd64-osx10.8.tar.gz`, where `#.#.#` is the latest version of Go.

Extract the file to `/usr/local` via `tar -C /usr/local -xzf go#.#.#.darwin-amd64-osx10.8.tar.gz`.

Set up two environment variables:

1. `GOPATH` points to your workspace, for me, that's `$HOME/code/go`.
2. We need to append Go's binary to our `PATH`.

You can set these up from a shell:

```
echo 'export GOPATH=$HOME/code/go' >> $HOME/.profile
echo 'export PATH=$PATH:/usr/local/go/bin' >> $HOME/.profile
```

You'll want to activate these variables. You can close and reopen your shell, or you can run `source $HOME/.profile`.

Type `go version` and you'll hopefully get an output that looks like `go version go1.3.3 darwin/amd64`.

Windows

Download the latest zip file. If you're on an x64 system, you'll want `go#.#.#.windows-amd64.zip`, where `#.#.#` is the latest version of Go.

Unzip it at a location of your choosing. `c:\Go` is a good choice.

Set up two environment variables:

1. `GOPATH` points to your workspace. That might be something like `c:\users\goku\work\go`.
2. Add `c:\Go\bin` to your `PATH` environment variable.

Environment variables can be set through the `Environment Variables` button on the `Advanced` tab of the `System` control panel. Some versions of Windows provide this control panel through the `Advanced System Settings` option inside the `System` control panel.

Open a command prompt and type `go version`. You'll hopefully get an output that looks like `go version go1.3.3 windows/amd64`.

Chapter 1 - The Basics

Go is a compiled, statically typed language with a C-like syntax and garbage collection. What does that mean?

Compilation

Compilation is the process of translating the source code that you write into a lower level language – either assembly (as is the case with Go), or some other intermediary language (as with Java and C#).

Compiled languages can be unpleasant to work with because compilation can be slow. It's hard to iterate quickly if you have to spend minutes or hours waiting for code to compile. Compilation speed is one of the major design goals of Go. This is good news for people working on large projects as well as those of us used to a quick feedback cycle offered by interpreted languages.

Compiled languages tend to run faster and the executable can be run without additional dependencies (at least, that's true for languages like C, C++ and Go which compile directly to assembly).

Static Typing

Being statically typed means that variables must be of a specific type (int, string, bool, []byte, etc.). This is either achieved by specifying the type when the variable is declared or, in many cases, letting the compiler infer the type (we'll look at examples shortly).

There's a lot more that can be said about static typing, but I believe it's something better understood by looking at code. If you're used to dynamically typed languages, you might find this cumbersome. You're not wrong, but there are advantages, especially when you pair static typing with compilation. The two are often conflated. It's true that when you have one, you normally have the other but it isn't a hard rule. With a rigid type system, a compiler is able to detect problems beyond mere syntactical mistakes as well as make further optimizations.

C-Like Syntax

Saying that a language has a C-like syntax means that if you're used to any other C-like languages such as C, C++, Java, JavaScript and C#, then you're going to find Go familiar – superficially, at least. For example, it means `&&` is used as a boolean AND, `==` is used to compare equality, `{` and `}` start and end a scope, and array indexes start at 0.

C-like syntax also tends to mean semi-colon terminated lines and parentheses around conditions. Go does away with both of these, though parentheses are still used to control precedence. For example, an `if` statement looks like this:

```
if name == "Leto" {
  print("the spice must flow")
}
```

And in more complicated cases, parentheses are still useful:

```
if (name == "Goku" && power > 9000) || (name == "gohan" && power < 4000)  {
  print("super Saiyan")
}
```

Beyond this, Go is much closer to C than C# or Java - not only in terms of syntax, but in terms of purpose. That's reflected in the terseness and simplicity of the language which will hopefully start to become obvious as you learn it.

Garbage Collected

Some variables, when created, have an easy-to-define life. A variable local to a function, for example, disappears when the function exits. In other cases, it isn't so obvious – at least to a compiler. For example, the lifetime of a variable returned by a function or referenced by other variables and objects can be tricky to determine. Without garbage collection, it's up to developers to free the memory associated with such variables at a point where the developer knows the variable isn't needed. How? In C, you'd literally `free(str);` the variable.

Languages with garbage collectors (e.g., Ruby, Python, Java, JavaScript, C#, Go) are able to keep track of these and free them when they're no longer used. Garbage collection adds overhead, but it also eliminates a number of devastating bugs.

Running Go Code

Let's start our journey by creating a simple program and learning how to compile and execute it. Open your favorite text editor and write the following code:

```
package main

func main() {
  println("it's over 9000!")
}
```

Save the file as **`main.go`**. For now, you can save it anywhere you want; we don't need to live inside Go's workspace for trivial examples.

Next, open a shell/command prompt and change the directory to where you saved the file. For me, that means typing `cd ~/code`.

Finally, run the program by entering:

```
go run main.go
```

If everything worked, you should see *it's over 9000!*.

But wait, what about the compilation step? **`go run`** is a handy command that compiles *and* runs your code. It uses a temporary directory to build the program, executes it and then cleans itself up. You can see the location of the temporary file by running:

```
go run --work main.go
```

To explicitly compile code, use `go build`:

```
go build main.go
```

This will generate an executable `main` which you can run. On Linux / OSX, don't forget that you need to prefix the executable with dot-slash, so you need to type `./main`.

While developing, you can use either `go run` or `go build`. When you deploy your code however, you'll want to deploy a binary via `go build` and execute that.

Main

Hopefully, the code that we just executed is understandable. We've created a function and printed out a string with the built-in `println` function. Did `go run` know what to execute because there was only a single choice? No. In Go, the entry point to a program has to be a function called `main` within a package `main`.

We'll talk more about packages in a later chapter. For now, while we focus on understanding the basics of Go, we'll always write our code within the `main` package.

If you want, you can alter the code and change the package name. Run the code via `go run` and you should get an error. Then, change the name back to `main` but use a different function name. You should see a different error message. Try making those same changes but use `go build` instead. Notice that the code compiles, there's just no entry point to run it. This is perfectly normal when you are, for example, building a library.

Imports

Go has a number of built-in functions, such as `println`, which can be used without reference. We can't get very far though, without making use of Go's standard library and eventually using third-party libraries. In Go, the `import` keyword is used to declare the packages that are used by the code in the file.

Let's change our program:

```
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) != 2 {
    os.Exit(1)
  }
  fmt.Println("It's over ", os.Args[1])
```

```
}
```

Which you can run via:

```
go run main.go 9000
```

We're now using two of Go's standard packages: `fmt` and `os`. We've also introduced another built-in function `len`. `len` returns the size of a string, or the number of values in a dictionary, or, as we see here, the number of elements in an array. If you're wondering why we expect 2 arguments, it's because the first argument – at index 0 – is always the path of the currently running executable. (Change the program to print it out and see for yourself.)

You've probably noticed we prefix the function name with the package, e.g., `fmt.Println`. This is different from many other languages. We'll learn more about packages in later chapters. For now, knowing how to import and use a package is a good start.

Go is strict about importing packages. It will not compile if you import a package but don't use it. Try to run the following:

```
package main

import (
  "fmt"
  "os"
)

func main() {
}
```

You should get two errors about `fmt` and `os` being imported and not used. Can this get annoying? Absolutely. Over time, you'll get used to it (it'll still be annoying though). Go is strict about this because unused imports can slow compilation; admittedly a problem most of us don't have to this degree.

Another thing to note is that Go's standard library is well documented. You can head over to http://golang.org/pkg/fmt/#Println to learn more about the `Println` function that we used. You can click on that section header and see the source code. Also, scroll to the top to learn more about Go's formatting capabilities.

If you're ever stuck without internet access, you can get the documentation running locally via:

```
godoc -http=:6060
```

and pointing your browser to `http://localhost:6060`

Variables and Declarations

It'd be nice to begin and end our look at variables by saying *you declare and assign to a variable by doing x = 4.* Unfortunately, things are more complicated in Go. We'll begin our conversation by looking at simple examples. Then,

in the next chapter, we'll expand this when we look at creating and using structures. Still, it'll probably take some time before you truly feel comfortable with it.

You might be thinking *Woah! What can be so complicated about this?* Let's start looking at some examples.

The most explicit way to deal with variable declaration and assignment in Go is also the most verbose:

```
package main

import (
  "fmt"
)

func main() {
  var power int
  power = 9000
  fmt.Printf("It's over %d\n", power)
}
```

Here, we declare a variable `power` of type `int`. By default, Go assigns a zero value to variables. Integers are assigned `0`, booleans `false`, strings `""` and so on. Next, we assign `9000` to our `power` variable. We can merge the first two lines:

```
var power int = 9000
```

Still, that's a lot of typing. Go has a handy short variable declaration operator, `:=`, which can infer the type:

```
power := 9000
```

This is handy, and it works just as well with functions:

```
func main() {
  power := getPower()
}

func getPower() int {
  return 9001
}
```

It's important that you remember that `:=` is used to declare the variable as well as assign a value to it. Why? Because a variable can't be declared twice (not in the same scope anyway). If you try to run the following, you'll get an error.

```
func main() {
  power := 9000
  fmt.Printf("It's over %d\n", power)

  // COMPILER ERROR:
  // no new variables on left side of :=
```

```
  power := 9001
  fmt.Printf("It's also over %d\n", power)
}
```

The compiler will complain with *no new variables on left side of :=*. This means that when we first declare a variable, we use `:=` but on subsequent assignment, we use the assignment operator `=`. This makes a lot of sense, but it can be tricky for your muscle memory to remember when to switch between the two.

If you read the error message closely, you'll notice that *variables* is plural. That's because Go lets you assign multiple variables (using either `=` or `:=`):

```
func main() {
  name, power := "Goku", 9000
  fmt.Printf("%s's power is over %d\n", name, power)
}
```

As long as one of the variables is new, `:=` can be used. Consider:

```
func main() {
  power := 1000
  fmt.Printf("default power is %d\n", power)

  name, power := "Goku", 9000
  fmt.Printf("%s's power is over %d\n", name, power)
}
```

Although `power` is being used twice with `:=`, the compiler won't complain the second time we use it, it'll see that the other variable, `name`, is a new variable and allow `:=`. However, you can't change the type of `power`. It was declared (implicitly) as an integer and thus, can only be assigned integers.

For now, the last thing to know is that, like imports, Go won't let you have unused variables. For example,

```
func main() {
  name, power := "Goku", 1000
  fmt.Printf("default power is %d\n", power)
}
```

won't compile because `name` is declared but not used. Like unused imports it'll cause some frustration, but overall I think it helps with code cleanliness and readability.

There's more to learn about declaration and assignments. For now, remember that you'll use `var NAME TYPE` when declaring a variable to its zero value, `NAME := VALUE` when declaring and assigning a value, and `NAME = VALUE` when assigning to a previously declared variable.

Function Declarations

This is a good time to point out that functions can return multiple values. Let's look at three functions: one with no return value, one with one return value, and one with two return values.

```
func log(message string) {
}

func add(a int, b int) int {
}

func power(name string) (int, bool) {
}
```

We'd use the last one like so:

```
value, exists := power("goku")
if exists == false {
  // handle this error case
}
```

Sometimes, you only care about one of the return values. In these cases, you assign the other values to _:

```
_, exists := power("goku")
if exists == false {
  // handle this error case
}
```

This is more than a convention. _, the blank identifier, is special in that the return value isn't actually assigned. This lets you use _ over and over again regardless of the returned type.

Finally, there's something else that you're likely to run into with function declarations. If parameters share the same type, we can use a shorter syntax:

```
func add(a, b int) int {

}
```

Being able to return multiple values is something you'll use often. You'll also frequently use _ to discard a value. Named return values and the slightly less verbose parameter declaration aren't that common. Still, you'll run into all of these sooner than later so it's important to know about them.

Before You Continue

We looked at a number of small individual pieces and it probably feels disjointed at this point. We'll slowly build larger examples and hopefully, the pieces will start to come together.

If you're coming from a dynamic language, the complexity around types and declarations might seem like a step backwards. I don't disagree with you. For some systems, dynamic languages are categorically more productive.

If you're coming from a statically typed language, you're probably feeling comfortable with Go. Inferred types and multiple return values are nice (though certainly not exclusive to Go). Hopefully as we learn more, you'll appreciate the clean and terse syntax.

Chapter 2 - Structures

Go isn't an object-oriented (OO) language like C++, Java, Ruby and C#. It doesn't have objects nor inheritance and thus, doesn't have the many concepts associated with OO such as polymorphism and overloading.

What Go does have are structures, which can be associated with methods. Go also supports a simple but effective form of composition. Overall, it results in simpler code, but there'll be occasions where you'll miss some of what OO has to offer. (It's worth pointing out that *composition over inheritance* is an old battle cry and Go is the first language I've used that takes a firm stand on the issue.)

Although Go doesn't do OO like you may be used to, you'll notice a lot of similarities between the definition of a structure and that of a class. A simple example is the following `Saiyan` structure:

```
type Saiyan struct {
  Name string
  Power int
}
```

We'll soon see how to add a method to this structure, much like you'd have methods as part of a class. Before we do that, we have to dive back into declarations.

Declarations and Initializations

When we first looked at variables and declarations, we looked only at built-in types, like integers and strings. Now that we're talking about structures, we need to expand that conversation to include pointers.

The simplest way to create a value of our structure is:

```
goku := Saiyan{
  Name: "Goku",
  Power: 9000,
}
```

Note: The trailing `,` in the above structure is required. Without it, the compiler will give an error. You'll appreciate the required consistency, especially if you've used a language or format that enforces the opposite.

We don't have to set all or even any of the fields. Both of these are valid:

```
goku := Saiyan{}

// or

goku := Saiyan{Name: "Goku"}
goku.Power = 9000
```

Just like unassigned variables have a zero value, so do fields.

Furthermore, you can skip the field name and rely on the order of the field declarations (though for the sake of clarity, you should only do this for structures with few fields):

```
goku := Saiyan{"Goku", 9000}
```

What all of the above examples do is declare a variable `goku` and assign a value to it.

Many times though, we don't want a variable that is directly associated with our value but rather, we want a variable that has a pointer to our value. A pointer is a memory address; it's the location of where to find the actual value. It's a level of indirection. Loosely, it's the difference between being at a house and having directions to the house.

Why do we want a pointer to the value, rather than the actual value? It comes down to the way Go passes arguments to a function: as copies. Knowing this, what does the following print?

```
func main() {
  goku := Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s Saiyan) {
  s.Power += 10000
}
```

The answer is 9000, not 19000. Why? Because `Super` made changes to a copy of our original `goku` value and thus, changes made in `Super` weren't reflected in the caller. To make this work as you probably expect, we need to pass a pointer to our value:

```
func main() {
  goku := &Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s *Saiyan) {
  s.Power += 10000
}
```

We made two changes. The first is that we used the `&` operator to get the address of our value (it's called the *address of* operator). Next, we changed the type of parameter `Super` expects. It used to expect a value of type `Saiyan` but now expects an address of type `*Saiyan`, where `*X` means *pointer to value of type X*. There's obviously some relation between the types `Saiyan` and `*Saiyan`, but they are two distinct types.

Note that we're still passing a copy of `goku's` value to `Super` it just so happens that `goku's` value has become an address. That copy is the same address as the original, which is what that indirection buys us. Think of it as copying the directions to a restaurant. What you have is a copy, but it still points to the same restaurant as the original.

We can prove that it's a copy by trying to change where it points to (not something you'd likely want to actually do):

```
func main() {
  goku := &Saiyan{"Goku", 9000}
  Super(goku)
  fmt.Println(goku.Power)
}

func Super(s *Saiyan) {
  s = &Saiyan{"Gohan", 1000}
}
```

The above, once again, prints 9000. This is how many languages behave, including Ruby, Python, Java and C#. Go, and to some degree C#, simply make the fact visible.

It should also be obvious that copying a pointer is going to be cheaper than copying a complex structure. On a 64-bit machine, a pointer is 64 bits large. If we have a structure with many fields, maybe even a large string or array, creating copies can be expensive. The real value of pointers though is that they let you share values. Do we want `Super` to alter a copy of `goku` or alter the shared `goku` value itself?

All this isn't to say that you'll always want a pointer. At the end of this chapter, after we've seen a bit more of what we can do with structures, we'll re-examine the pointer-versus-value question.

Functions on Structures

We can associate a method with a structure:

```
type Saiyan struct {
  Name string
  Power int
}

func (s *Saiyan) Super() {
  s.Power += 10000
}
```

In the above code, we say that the type `*Saiyan` is the **receiver** of the `Super` method. We call `Super` like so:

```
goku := &Saiyan{"Goku", 9001}
goku.Super()
fmt.Println(goku.Power) // will print 19001
```

Constructors

Structures don't have constructors. Instead, you create a function that returns an instance of the desired type (like a factory):

```
func NewSaiyan(name string, power int) *Saiyan {
  return &Saiyan{
    Name: name,
    Power: power,
  }
}
```

This pattern rubs a lot of developers the wrong way. On the one hand, it's a pretty slight syntactical change; on the other, it does feel a little less compartmentalized.

Our factory doesn't have to return a pointer; this is absolutely valid:

```
func NewSaiyan(name string, power int) Saiyan {
  return Saiyan{
    Name: name,
    Power: power,
  }
}
```

Fields of a Structure

In the example that we've seen so far, `Saiyan` has two fields `Name` and `Power` of types **`string`** and **`int`**, respectively. Fields can be of any type – including other structures and types that we haven't explored yet such as arrays, maps, interfaces and functions.

For example, we could expand our definition of `Saiyan`:

```
type Saiyan struct {
  Name string
  Power int
  Father *Saiyan
}
```

which we'd initialize via:

```
gohan := &Saiyan{
  Name: "Gohan",
  Power: 1000,
  Father: &Saiyan {
    Name: "Goku",
    Power: 9001,
    Father: nil,
  },
}
```

Composition

Go supports composition, which is the act of including one structure into another. In some languages, this is called a trait or a mixin. Languages that don't have an explicit composition mechanism can always do it the long way. In Java:

```
public class Person {
  private string name;

  public string getName() {
    return this.name;
  }
}

public class Saiyan {
  // Saiyan is said to have a person
  private Person person;

  // we forward the call to person
  public string getName() {
    return this.person.getName();
  }
  ...
}
```

This can get pretty tedious. Every method of `Person` needs to be duplicated in `Saiyan`. Go avoids this tediousness:

```
type Person struct {
  Name string
}

func (p *Person) Introduce() {
  fmt.Printf("Hi, I'm %s\n", p.Name)
}

type Saiyan struct {
  *Person
  Power int
}

// and to use it:
goku := &Saiyan{
  Person: &Person{"Goku"},
  Power: 9001,
}
goku.Introduce()
```

The `Saiyan` structure has a field of type `*Person`. Because we didn't give it an explicit field name, we can implicitly access the fields and functions of the composed type. However, the Go compiler *did* give it a field name, consider the perfectly valid:

```
goku := &Saiyan{
  Person: &Person{"Goku"},
}
fmt.Println(goku.Name)
fmt.Println(goku.Person.Name)
```

Both of the above will print "Goku".

Is composition better than inheritance? Many people think that it's a more robust way to share code. When using inheritance, your class is tightly coupled to your superclass and you end up focusing on hierarchy rather than behavior.

Overloading

While overloading isn't specific to structures, it's worth addressing. Simply, Go doesn't support overloading. For this reason, you'll see (and write) a lot of functions that look like `Load`, `LoadById`, `LoadByName` and so on.

However, because implicit composition is really just a compiler trick, we can "overwrite" the functions of a composed type. For example, our `Saiyan` structure can have its own `Introduce` function:

```
func (s *Saiyan) Introduce() {
  fmt.Printf("Hi, I'm %s. Ya!\n", s.Name)
}
```

The composed version is always available via `s.Person.Introduce()`.

Pointers versus Values

As you write Go code, it's natural to ask yourself *should this be a value, or a pointer to a value?* There are two pieces of good news. First, the answer is the same regardless of which of the following we're talking about:

- A local variable assignment
- Field in a structure
- Return value from a function
- Parameters to a function
- The receiver of a method

Secondly, if you aren't sure, use a pointer.

As we already saw, passing values is a great way to make data immutable (changes that a function makes to it won't be reflected in the calling code). Sometimes, this is the behavior that you'll want but more often, it won't be.

Even if you don't intend to change the data, consider the cost of creating a copy of large structures. Conversely, you might have small structures, say:

```
type Point struct {
  X int
  Y int
}
```

In such cases, the cost of copying the structure is probably offset by being able to access `X` and `Y` directly, without any indirection.

Again, these are all pretty subtle cases. Unless you're iterating over thousands or possibly tens of thousands of such points, you wouldn't notice a difference.

Before You Continue

From a practical point of view, this chapter introduced structures, how to make an instance of a structure a receiver of a function, and added pointers to our existing knowledge of Go's type system. The following chapters will build on what we know about structures as well as the inner workings that we've explored.

Chapter 3 - Maps, Arrays and Slices

So far we've seen a number of simple types and structures. It's now time to look at arrays, slices and maps.

Arrays

If you come from Python, Ruby, Perl, JavaScript or PHP (and more), you're probably used to programming with *dynamic arrays*. These are arrays that resize themselves as data is added to them. In Go, like many other languages, arrays are fixed. Declaring an array requires that we specify the size, and once the size is specified, it cannot grow:

```
var scores [10]int
scores[0] = 339
```

The above array can hold up to 10 scores using indexes `scores[0]` through `scores[9]`. Attempts to access an out of range index in the array will result in a compiler or runtime error.

We can initialize the array with values:

```
scores := [4]int{9001, 9333, 212, 33}
```

We can use `len` to get the length of the array. `range` can be used to iterate over it:

```
for index, value := range scores {

}
```

Arrays are efficient but rigid. We often don't know the number of elements we'll be dealing with upfront. For this, we turn to slices.

Slices

In Go, you rarely, if ever, use arrays directly. Instead, you use slices. A slice is a lightweight structure that wraps and represents a portion of an array. There are a few ways to create a slice, and we'll go over when to use which later on. The first is a slight variation on how we created an array:

```
scores := []int{1,4,293,4,9}
```

Unlike the array declaration, our slice isn't declared with a length within the square brackets. To understand how the two are different, let's see another way to create a slice, using `make`:

```
scores := make([]int, 10)
```

We use `make` instead of `new` because there's more to creating a slice than just allocating the memory (which is what `new` does). Specifically, we have to allocate the memory for the underlying array and also initialize the slice. In the above, we initialize a slice with a length of 10 and a capacity of 10. The length is the size of the slice, the capacity is the size of the underlying array. Using `make` we can specify the two separately:

```
scores := make([]int, 0, 10)
```

This creates a slice with a length of 0 but with a capacity of 10. (If you're paying attention, you'll note that `make` and `len` *are* overloaded. Go is a language that, to the frustration of some, makes use of features which aren't exposed for developers to use.)

To better understand the interplay between length and capacity, let's look at some examples:

```
func main() {
  scores := make([]int, 0, 10)
  scores[5] = 9033
  fmt.Println(scores)
}
```

Our first example crashes. Why? Because our slice has a length of 0. Yes, the underlying array has 10 elements, but we need to explicitly expand our slice in order to access those elements. One way to expand a slice is via `append`:

```
func main() {
  scores := make([]int, 0, 10)
  scores = append(scores, 5)
  fmt.Println(scores) // prints [5]
}
```

But that changes the intent of our original code. Appending to a slice of length 0 will set the first element. For whatever reason, our crashing code wanted to set the element at index 5. To do this, we can re-slice our slice:

```
func main() {
  scores := make([]int, 0, 10)
  scores = scores[0:6]
  scores[5] = 9033
  fmt.Println(scores)
}
```

How large can we resize a slice? Up to its capacity which, in this case, is 10. You might be thinking *this doesn't actually solve the fixed-length issue of arrays.* It turns out that `append` is pretty special. If the underlying array is full, it will create a new larger array and copy the values over (this is exactly how dynamic arrays work in PHP, Python, Ruby, JavaScript, …). This is why, in the example above that used `append`, we had to re-assign the value returned by `append` to our `scores` variable: `append` might have created a new value if the original had no more space.

If I told you that Go grew arrays with a 2x algorithm, can you guess what the following will output?

```
func main() {
  scores := make([]int, 0, 5)
  c := cap(scores)
  fmt.Println(c)

  for i := 0; i < 25; i++ {
```

```
    scores = append(scores, i)

    // if our capacity has changed,
    // Go had to grow our array to accommodate the new data
    if cap(scores) != c {
      c = cap(scores)
      fmt.Println(c)
    }
  }
}
```

The initial capacity of `scores` is 5. In order to hold 20 values, it'll have to be expanded 3 times with a capacity of 10, 20 and finally 40.

As a final example, consider:

```
func main() {
  scores := make([]int, 5)
  scores = append(scores, 9332)
  fmt.Println(scores)
}
```

Here, the output is going to be `[0, 0, 0, 0, 0, 9332]`. Maybe you thought it would be `[9332, 0, 0, 0, 0]`? To a human, that might seem logical. To a compiler, you're telling it to append a value to a slice that already holds 5 values.

Ultimately, there are four common ways to initialize a slice:

```
names := []string{"leto", "jessica", "paul"}
checks := make([]bool, 10)
var names []string
scores := make([]int, 0, 20)
```

When do you use which? The first one shouldn't need much of an explanation. You use this when you know the values that you want in the array ahead of time.

The second one is useful when you'll be writing into specific indexes of a slice. For example:

```
func extractPowers(saiyans []*Saiyans) []int {
  powers := make([]int, len(saiyans))
  for index, saiyan := range saiyans {
    powers[index] = saiyan.Power
  }
  return powers
}
```

The third version is a nil slice and is used in conjunction with **append**, when the number of elements is unknown.

The last version lets us specify an initial capacity; useful if we have a general idea of how many elements we'll need.

Even when you know the size, `append` can be used. It's largely a matter of preference:

```
func extractPowers(saiyans []*Saiyans) []int {
  powers := make([]int, 0, len(saiyans))
  for _, saiyan := range saiyans {
    powers = append(powers, saiyan.Power)
  }
  return powers
}
```

Slices as wrappers to arrays is a powerful concept. Many languages have the concept of slicing an array. Both JavaScript and Ruby arrays have a `slice` method. You can also get a slice in Ruby by using `[START..END]` or in Python via `[START:END]`. However, in these languages, a slice is actually a new array with the values of the original copied over. If we take Ruby, what's the output of the following?

```
scores = [1,2,3,4,5]
slice = scores[2..4]
slice[0] = 999
puts scores
```

The answer is `[1, 2, 3, 4, 5]`. That's because `slice` is a completely new array with copies of values. Now, consider the Go equivalent:

```
scores := []int{1,2,3,4,5}
slice := scores[2:4]
slice[0] = 999
fmt.Println(scores)
```

The output is `[1, 2, 999, 4, 5]`.

This changes how you code. For example, a number of functions take a position parameter. In JavaScript, if we want to find the first space in a string (yes, slices work on strings too!) after the first five characters, we'd write:

```
haystack = "the spice must flow";
console.log(haystack.indexOf(" ", 5));
```

In Go, we leverage slices:

```
strings.Index(haystack[5:], " ")
```

We can see from the above example, that `[X:]` is shorthand for *from X to the end* while `[:X]` is shorthand for *from the start to X*. Unlike other languages, Go doesn't support negative values. If we want all of the values of a slice except the last, we do:

```
scores := []int{1, 2, 3, 4, 5}
scores = scores[:len(scores)-1]
```

The above is the start of an efficient way to remove a value from an unsorted slice:

```
func main() {
  scores := []int{1, 2, 3, 4, 5}
  scores = removeAtIndex(scores, 2)
  fmt.Println(scores)
}

func removeAtIndex(source []int, index int) []int {
  lastIndex := len(source) - 1
  //swap the last value and the value we want to remove
  source[index], source[lastIndex] = source[lastIndex], source[index]
  return source[:lastIndex]
}
```

Finally, now that we know about slices, we can look at another commonly used built-in function: `copy`. `copy` is one of those functions that highlights how slices change the way we code. Normally, a method that copies values from one array to another has 5 parameters: `source`, `sourceStart`, `count`, `destination` and `destinationSource`. With slices, we only need two:

```
import (
  "fmt"
  "math/rand"
  "sort"
)

func main() {
  scores := make([]int, 100)
  for i := 0; i < 100; i++ {
    scores[i] = int(rand.Int31n(1000))
  }
  sort.Ints(scores)

  worst := make([]int, 5)
  copy(worst, scores[:5])
  fmt.Println(worst)
}
```

Take some time and play with the above code. Try variations. See what happens if you change copy to something like `copy(worst[2:4], scores[:5])`, or what if you try to copy more or less than `5` values into `worst`?

Maps

Maps in Go are what other languages call hashtables or dictionaries. They work as you expect: you define a key and value, and can get, set and delete values from it.

Maps, like slices, are created with the `make` function. Let's look at an example:

```
func main() {
  lookup := make(map[string]int)
  lookup["goku"] = 9001
  power, exists := lookup["vegeta"]

  // prints 0, false
  // 0 is the default value for an integer
  fmt.Println(power, exists)
}
```

To get the number of keys, we use `len`. To remove a value based on its key, we use `delete`:

```
// returns 1
total := len(lookup)

// has no return, can be called on a non-existing key
delete(lookup, "goku")
```

Maps grow dynamically. However, we can supply a second argument to `make` to set an initial size:

```
lookup := make(map[string]int, 100)
```

If you have some idea of how many keys your map will have, defining an initial size can help with performance.

When you need a map as a field of a structure, you define it as:

```
type Saiyan struct {
  Name string
  Friends map[string]*Saiyan
}
```

One way to initialize the above is via:

```
goku := &Saiyan{
  Name: "Goku",
  Friends: make(map[string]*Saiyan),
}
goku.Friends["krillin"] = ... //todo load or create Krillin
```

There's yet another way to declare and initialize values in Go. Like `make`, this approach is specific to maps and arrays. We can declare as a composite literal:

```
lookup := map[string]int{
  "goku": 9001,
  "gohan": 2044,
}
```

We can iterate over a map using a `for` loop combined with the `range` keyword:

```
for key, value := range lookup {
  ...
}
```

Iteration over maps isn't ordered. Each iteration over a lookup will return the key value pair in a random order.

Pointers versus Values

We finished Chapter 2 by looking at whether you should assign and pass pointers or values. We'll now have this same conversation with respect to array and map values. Which of these should you use?

```
a := make([]Saiyan, 10)
//or
b := make([]*Saiyan, 10)
```

Many developers think that passing `b` to, or returning it from, a function is going to be more efficient. However, what's being passed/returned is a copy of the slice, which itself is a reference. So with respect to passing/returning the slice itself, there's no difference.

Where you will see a difference is when you modify the values of a slice or map. At this point, the same logic that we saw in Chapter 2 applies. So the decision on whether to define an array of pointers versus an array of values comes down to how you use the individual values, not how you use the array or map itself.

Before You Continue

Arrays and maps in Go work much like they do in other languages. If you're used to dynamic arrays, there might be a small adjustment, but `append` should solve most of your discomfort. If we peek beyond the superficial syntax of arrays, we find slices. Slices are powerful and they have a surprisingly large impact on the clarity of your code.

There are edge cases that we haven't covered, but you're not likely to run into them. And, if you do, hopefully the foundation we've built here will let you understand what's going on.

Chapter 4 - Code Organization and Interfaces

It's now time to look at how to organize our code.

Packages

To keep more complicated libraries and systems organized, we need to learn about packages. In Go, package names follow the directory structure of your Go workspace. If we were building a shopping system, we'd probably start with a package name "shopping" and put our source files in `$GOPATH/src/shopping/`.

We don't want to put everything inside this folder though. For example, maybe we want to isolate some database logic inside its own folder. To achieve this, we create a subfolder at `$GOPATH/src/shopping/db`. The package name of the files within this subfolder is simply `db`, but to access it from another package, including the `shopping` package, we need to import `shopping/db`.

In other words, when you name a package, via the **`package`** keyword, you provide a single value, not a complete hierarchy (e.g., "shopping" or "db"). When you import a package, you specify the complete path.

Let's try it. Inside your Go workspace's `src` folder (which we set up in Getting Started of the Introduction), create a new folder called `shopping` and a subfolder within it called `db`.

Inside of `shopping/db`, create a file called `db.go` and add the following code:

```
package db

type Item struct {
  Price float64
}

func LoadItem(id int) *Item {
  return &Item{
    Price: 9.001,
  }
}
```

Notice that the name of the package is the same as the name of the folder. Also, obviously, we aren't actually accessing the database. We're just using this as an example to show how to organize code.

Now, create a file called `pricecheck.go` inside of the main `shopping` folder. Its content is:

```
package shopping

import (
  "shopping/db"
)
```

```
func PriceCheck(itemId int) (float64, bool) {
  item := db.LoadItem(itemId)
  if item == nil {
    return 0, false
  }
  return item.Price, true
}
```

It's tempting to think that importing `shopping/db` is somehow special because we're inside the `shopping` package/-folder already. In reality, you're importing `$GOPATH/src/shopping/db`, which means you could just as easily import `test/db` so long as you had a package named `db` inside of your workspace's `src/test` folder.

If you're building a package, you don't need anything more than what we've seen. To build an executable, you still need a **`main`**. The way I prefer to do this is to create a subfolder called **`main`** inside of `shopping` with a file called **`main.go`** and the following content:

```
package main

import (
  "shopping"
  "fmt"
)

func main() {
  fmt.Println(shopping.PriceCheck(4343))
}
```

You can now run your code by going into your `shopping` project and typing:

```
go run main/main.go
```

Cyclical Imports

As you start writing more complex systems, you're bound to run into cyclical imports. This happens when package A imports package B but package B imports package A (either directly or indirectly through another package). This is something the compiler won't allow.

Let's change our shopping structure to cause the error.

Move the `Item` definition from `shopping/db/db.go` into `shopping/pricecheck.go`. Your `pricecheck.go` file should now look like:

```
package shopping

import (
  "shopping/db"
```

```
)

type Item struct {
  Price float64
}

func PriceCheck(itemId int) (float64, bool) {
  item := db.LoadItem(itemId)
  if item == nil {
    return 0, false
  }
  return item.Price, true
}
```

If you try to run the code, you'll get a couple of errors from `db/db.go` about `Item` being undefined. This makes sense. `Item` no longer exists in the `db` package; it's been moved to the shopping package. We need to change `shopping/db/db.go` to:

```
package db

import (
  "shopping"
)

func LoadItem(id int) *shopping.Item {
  return &shopping.Item{
    Price: 9.001,
  }
}
```

Now when you try to run the code, you'll get a dreaded *import cycle not allowed* error. We solve this by introducing another package which contains shared structures. Your directory structure should look like:

```
$GOPATH/src
  - shopping
    pricecheck.go
    - db
      db.go
    - models
      item.go
    - main
      main.go
```

`pricecheck.go` will still import `shopping/db`, but `db.go` will now import `shopping/models` instead of `shopping`, thus breaking the cycle. You'll often need to share more than just `models`, so you might have other similar folder

named `utilities` and such. The important rule about these shared packages is that they shouldn't import anything from the `shopping` package or any sub-packages. In a few sections, we'll look at interfaces which can help us untangle these types of dependencies.

Visibility

Go uses a simple rule to define what types and functions are visible outside of a package. If the name of the type or function starts with an uppercase letter, it's visible. If it starts with a lowercase letter, it isn't.

This also applies to structure fields. If a structure field name starts with a lowercase letter, only code within the same package will be able to access them.

For example, if our `items.go` file had a function that looked like:

```
func NewItem() *Item {
  // ...
}
```

it could be called via `models.NewItem()`. But if the function was named `newItem`, we wouldn't be able to access it from a different package.

Go ahead and change the name of the various functions, types and fields from the `shopping` code. For example, if you rename the `Item's Price` field to `price`, you should get an error.

Package Management

The `go` command we've been using to `run` and `build` has a `get` subcommand which is used to fetch third-party libraries. `go get` supports various protocols but for this example, we'll be getting a library from Github, meaning, you'll need `git` installed on your computer.

Assuming you already have git installed, from a shell/command prompt, enter:

```
go get github.com/mattn/go-sqlite3
```

`go get` fetches the remote files and stores them in your workspace. Go ahead and check your `$GOPATH/src`. In addition to the `shopping` project that we created, you'll now see a `github.com` folder. Within, you'll see a `mattn` folder which contains a `go-sqlite3` folder.

We just talked about how to import packages that live in our workspace. To use our newly gotten `go-sqlite3` package, we'd import it like so:

```
import (
  "github.com/mattn/go-sqlite3"
)
```

I know this looks like a URL but in reality, it'll simply import the `go-sqlite3` package which it expects to find in `$GOPATH/src/github.com/mattn/go-sqlite3`.

Dependency Management

`go get` has a couple of other tricks up its sleeve. If we `go get` within a project, it'll scan all the files, looking for `imports` to third-party libraries and will download them. In a way, our own source code becomes a `Gemfile` or `package.json`.

If you call `go get -u` it'll update the packages (or you can update a specific package via `go get -u FULL_PACKAGE_NAME`).

Eventually, you might find `go get` inadequate. For one thing, there's no way to specify a revision, it always points to the master/head/trunk/default. This is an even larger problem if you have two projects needing different versions of the same library.

To solve this, you can use a third-party dependency management tool. They are still young, but two promising ones are goop and godep. A more complete list is available at the go-wiki.

Interfaces

Interfaces are types that define a contract but not an implementation. Here's an example:

```
type Logger interface {
  Log(message string)
}
```

You might be wondering what purpose this could possibly serve. Interfaces help decouple your code from specific implementations. For example, we might have various types of loggers:

```
type SqlLogger struct { ... }
type ConsoleLogger struct { ... }
type FileLogger struct { ... }
```

Yet by programming against the interface, rather than these concrete implementations, we can easily change (and test) which we use without any impact to our code.

How would you use one? Just like any other type, it could be a structure's field:

```
type Server struct {
  logger Logger
}
```

or a function parameter (or return value):

```
func process(logger Logger) {
  logger.Log("hello!")
}
```

In a language like C# or Java, we have to be explicit when a class implements an interface:

```
public class ConsoleLogger : Logger {
  public void Logger(message string) {
    Console.WriteLine(message)
  }
}
```

In Go, this happens implicitly. If your structure has a function name `Log` with a `string` parameter and no return value, then it can be used as a `Logger`. This cuts down on the verboseness of using interfaces:

```
type ConsoleLogger struct {}
func (l ConsoleLogger) Log(message string) {
  fmt.Println(message)
}
```

It also tends to promote small and focused interfaces. The standard library is full of interfaces. The `io` package has a handful of popular ones such as `io.Reader`, `io.Writer`, and `io.Closer`. If you write a function that expects a parameter that you'll only be calling `Close()` on, you absolutely should accept an `io.Closer` rather than whatever concrete type you're using.

Interfaces can also participate in composition. And, interfaces themselves can be composed of other interfaces. For example, `io.ReadCloser` is an interface composed of the `io.Reader` interface as well as the `io.Closer` interface.

Finally, interfaces are commonly used to avoid cyclical imports. Since they don't have implementations, they'll have limited dependencies.

Before You Continue

Ultimately, how you structure your code around Go's workspace is something that you'll only feel comfortable with after you've written a couple of non-trivial projects. What's most important for you to remember is the tight relationship between package names and your directory structure (not just within a project, but within the entire workspace).

The way Go handles visibility of types is straightforward and effective. It's also consistent. There are a few things we haven't looked at, such as constants and global variables but rest assured, their visibility is determined by the same naming rule.

Finally, if you're new to interfaces, it might take some time before you get a feel for them. However, the first time you see a function that expects something like `io.Reader`, you'll find yourself thanking the author for not demanding more than he or she needed.

Chapter 5 - Tidbits

In this chapter, we'll talk about a miscellany of Go's feature which didn't quite fit anywhere else.

Error Handling

Go's preferred way to deal with errors is through return values, not exceptions. Consider the `strconv.Atoi` function which takes a string and tries to convert it to an integer:

```
package main

import (
  "fmt"
  "os"
  "strconv"
)

func main() {
  if len(os.Args) != 2 {
    os.Exit(1)
  }

  n, err := strconv.Atoi(os.Args[1])
  if err != nil {
    fmt.Println("not a valid number")
  } else {
    fmt.Println(n)
  }
}
```

You can create your own error type; the only requirement is that it fulfills the contract of the built-in `error` interface, which is:

```
type error interface {
  Error() string
}
```

More commonly, we can create our own errors by importing the `errors` package and using it in the `New` function:

```
import (
  "errors"
)


func process(count int) error {
```

```
  if count < 1 {
    return errors.New("Invalid count")
  }
  ...
  return nil
}
```

There's a common pattern in Go's standard library of using error variables. For example, the `io` package has an `EOF` variable which is defined as:

```
var EOF = errors.New("EOF")
```

This is a package variable (it's defined outside of a function) which is publicly accessible (upper-case first letter). Various functions can return this error, say when we're reading from a file or STDIN. If it makes contextual sense, you should use this error, too. As consumers, we can use this singleton:

```
package main

import (
  "fmt"
  "io"
)

func main() {
  var input int
  _, err := fmt.Scan(&input)
  if err == io.EOF {
    fmt.Println("no more input!")
  }
}
```

As a final note, Go does have `panic` and `recover` functions. `panic` is like throwing an exception while `recover` is like `catch`; they are rarely used.

Defer

Even though Go has a garbage collector, some resources require that we explicitly release them. For example, we need to `Close()` files after we're done with them. This sort of code is always dangerous. For one thing, as we're writing a function, it's easy to forget to `Close` something that we declared 10 lines up. For another, a function might have multiple return points. Go's solution is the `defer` keyword:

```
package main

import (
  "fmt"
```

```
  "os"
)

func main() {
  file, err := os.Open("a_file_to_read")
  if err != nil {
    fmt.Println(err)
    return
  }
  defer file.Close()
  // read the file
}
```

If you try to run the above code, you'll probably get an error (the file doesn't exist). The point is to show how `defer` works. Whatever you `defer` will be executed after the method returns, even if it does so violently. This lets you release resources near where it's initialized and takes care of multiple return points.

go fmt

Most programs written in Go follow the same formatting rules, namely, a tab is used to indent and braces go on the same line as their statement.

I know, you have your own style and you want to stick to it. That's what I did for a long time, but I'm glad I eventually gave in. A big reason for this is the `go fmt` command. It's easy to use and authoritative (so no one argues over meaningless preferences).

When you're inside a project, you can apply the formatting rule to it and all sub-projects via:

```
go fmt ./...
```

Give it a try. It does more than indent your code; it also aligns field declarations and alphabetically orders imports.

Initialized If

Go supports a slightly modified if-statement, one where a value can be initiated prior to the condition being evaluated:

```
if x := 10; count > x {
  ...
}
```

That's a pretty silly example. More realistically, you might do something like:

```
if err := process(); err != nil {
  return err
}
```

Interestingly, while the values aren't available outside the if-statement, they are available inside any `else if` or `else`.

Empty Interface and Conversions

In most object-oriented languages, a built-in base class, often named `object`, is the superclass for all other classes. Go, having no inheritance, doesn't have such a superclass. What it does have is an empty interface with no methods: `interface{}`. Since every type implements all 0 of the empty interface's methods, and since interfaces are implicitly implemented, every type fulfills the contract of the empty interface.

If we wanted to, we could write an `add` function with the following signature:

```
func add(a interface{}, b interface{}) interface{} {
  ...
}
```

To convert a variable to a specific type, you use `.(TYPE)`:

```
return a.(int) + b.(int)
```

You also have access to a powerful type switch:

```
switch a.(type) {
  case int:
    fmt.Printf("a is now an int and equals %d\n", a)
  case bool, string:
    // ...
  default:
    // ...
}
```

You'll see and probably use the empty interface more than you might first expect. Admittedly, it won't result in clean code. Converting values back and forth is ugly and dangerous but sometimes, in a static language, it's the only choice.

Strings and Byte Arrays

Strings and byte arrays are closely related. We can easily convert one to the other:

```
stra := "the spice must flow"
byts := []byte(stra)
strb := string(byts)
```

In fact, this way of converting is common across various types as well. Some functions explicitly expect an `int32` or an `int64` or their unsigned counterparts. You might find yourself having to do things like:

```
int64(count)
```

Still, when it comes to bytes and strings, it's probably something you'll end up doing often. Do note that when you use `[]byte(X)` or `string(X)`, you're creating a copy of the data. This is necessary because strings are immutable.

Strings are made of `runes` which are unicode code points. If you take the length of a string, you might not get what you expect. The following prints 3:

```
fmt.Println(len("□"))
```

If you iterate over a string using `range`, you'll get runes, not bytes. Of course, when you turn a string into a `[]byte` you'll get the correct data.

Function Type

Functions are first-class types:

```
type Add func(a int, b int) int
```

which can then be used anywhere – as a field type, as a parameter, as a return value.

```
package main

import (
  "fmt"
)

type Add func(a int, b int) int

func main() {
  fmt.Println(process(func(a int, b int) int{
      return a + b
  }))
}

func process(adder Add) int {
  return adder(1, 2)
}
```

Using functions like this can help decouple code from specific implementations much like we achieve with interfaces.

Before You Continue

We looked at various aspects of programming with Go. Most notably, we saw how error handling behaves and how to release resources such as connections and open files. Many people dislike Go's approach to error handling. It can feel like a step backwards. Sometimes, I agree. Yet, I also find that it results in code that's easier to follow. `defer` is an

unusual but practical approach to resource management. In fact, it isn't tied to resource management only. You can use `defer` for any purpose, such as logging when a function exits.

Certainly, we haven't looked at all of the tidbits Go has to offer. But you should be feeling comfortable enough to tackle whatever you come across.

Chapter 6 - Concurrency

Go is often described as a concurrent-friendly language. The reason for this is that it provides a simple syntax over two powerful mechanisms: goroutines and channels.

Goroutines

A goroutine is similar to a thread, but it is scheduled by Go, not the OS. Code that runs in a goroutine can run concurrently with other code. Let's look at an example:

```
package main

import (
  "fmt"
  "time"
)

func main() {
  fmt.Println("start")
  go process()
  time.Sleep(time.Millisecond * 10) // this is bad, don't do this!
  fmt.Println("done")
}

func process() {
  fmt.Println("processing")
}
```

There are a few interesting things going on here, but the most important is how we start a goroutine. We simply use the `go` keyword followed by the function we want to execute. If we just want to run a bit of code, such as the above, we can use an anonymous function. Do note that anonymous functions aren't only used with goroutines, however.

```
go func() {
  fmt.Println("processing")
}()
```

Goroutines are easy to create and have little overhead. Multiple goroutines will end up running on the same underlying OS thread. This is often called an M:N threading model because we have M application threads (goroutines) running on N OS threads. The result is that a goroutine has a fraction of overhead (a few KB) than OS threads. On modern hardware, it's possible to have millions of goroutines.

Furthermore, the complexity of mapping and scheduling is hidden. We just say *this code should run concurrently* and let Go worry about making it happen.

If we go back to our example, you'll notice that we had to `Sleep` for a few milliseconds. That's because the main process exits before the goroutine gets a chance to execute (the process doesn't wait until all goroutines are finished before exiting). To solve this, we need to coordinate our code.

Synchronization

Creating goroutines is trivial, and they are so cheap that we can start many; however, concurrent code needs to be coordinated. To help with this problem, Go provides `channels`. Before we look at `channels`, I think it's important to understand a little bit about the basics of concurrent programming.

Writing concurrent code requires that you pay specific attention to where and how you read and write values. In some ways, it's like programming without a garbage collector – it requires that you think about your data from a new angle, always watchful for possible danger. Consider:

```
package main

import (
  "fmt"
  "time"
)

var counter = 0

func main() {
  for i := 0; i < 2; i++ {
    go incr()
  }
  time.Sleep(time.Millisecond * 10)
}

func incr() {
  counter++
  fmt.Println(counter)
}
```

What do you think the output will be?

If you think the output is `1, 2` you're both right and wrong. It's true that if you run the above code, you'll very likely get that output. However, the reality is that the behavior is undefined. Why? Because we potentially have multiple (two in this case) goroutines writing to the same variable, `counter`, at the same time. Or, just as bad, one goroutine would be reading `counter` while another writes to it.

Is that really a danger? Yes, absolutely. `counter++` might seem like a simple line of code, but it actually gets broken down into multiple assembly statements – the exact nature is dependent on the platform that you're running. It's true that, in this example, the most likely case is things will run just fine. However, another possible outcome would be

that they both see `counter` when its equal to `0` and you get an output of `1, 1`. There are worse possibilities, such as system crashes or accessing an arbitrary piece of data and incrementing it!

The only concurrent thing you can safely do to a variable is to read from it. You can have as many readers as you want, but writes need to be synchronized. There are various ways to do this, including using some truly atomic operations that rely on special CPU instructions. However, the most common approach is to use a mutex:

```
package main

import (
  "fmt"
  "time"
  "sync"
)

var (
  counter = 0
  lock sync.Mutex
)

func main() {
  for i := 0; i < 2; i++ {
    go incr()
  }
  time.Sleep(time.Millisecond * 10)
}

func incr() {
  lock.Lock()
  defer lock.Unlock()
  counter++
  fmt.Println(counter)
}
```

A mutex serializes access to the code under lock. The reason we simply define our lock as `lock sync.Mutex` is because the default value of a `sync.Mutex` is unlocked.

Seems simple enough? The example above is deceptive. There's a whole class of serious bugs that can arise when doing concurrent programming. First of all, it isn't always so obvious what code needs to be protected. While it might be tempting to use coarse locks (locks that cover a large amount of code), that undermines the very reason we're doing concurrent programming in the first place. We generally want fine locks; else, we end up with a ten-lane highway that suddenly turns into a one-lane road.

The other problem has to do with deadlocks. With a single lock, this isn't a problem, but if you're using two or more locks around the same code, it's dangerously easy to have situations where goroutineA holds lockA but needs access to lockB, while goroutineB holds lockB but needs access to lockA.

It actually *is* possible to deadlock with a single lock, if we forget to release it. This isn't as dangerous as a multi-lock deadlock (because those are *really* tough to spot), but just so you can see what happens, try running:

```
package main

import (
  "time"
  "sync"
)

var (
  lock sync.Mutex
)

func main() {
  go func() { lock.Lock() }()
  time.Sleep(time.Millisecond * 10)
  lock.Lock()
}
```

There's more to concurrent programming than what we've seen so far. For one thing, since we can have multiple reads at the same time, there's another common mutex called a read-write mutex. This exposes two locking functions: one to lock readers and one to lock writers. In Go, `sync.RWMutex` is such a lock. In addition to the `Lock` and `Unlock` methods of a `sync.Mutex`, it also exposes `RLock` and `RUnlock` methods; where `R` stands for *Read*. While read-write mutexes are commonly used, they place an additional burden on developers: we must now pay attention to not only when we're accessing data, but also how.

Furthermore, part of concurrent programming isn't so much about serializing access across the narrowest possible piece of code; it's also about coordinating multiple goroutines. For example, sleeping for 10 milliseconds isn't a particularly elegant solution. What if a goroutine takes more than 10 milliseconds? What if it takes less and we're just wasting cycles? Also, what if instead of just waiting for goroutines to finish, we want to tell one *hey, I have new data for you to process!*?

These are all things that are doable without `channels`. Certainly for simpler cases, I believe you **should** use primitives such as `sync.Mutex` and `sync.RWMutex`, but as we'll see in the next section, `channels` aim at making concurrent programming cleaner and less error-prone.

Channels

The challenge with concurrent programming stems from sharing data. If your goroutines share no data, you needn't worry about synchronizing them. That isn't an option for all systems, however. In fact, many systems are built with the exact opposite goal in mind: to share data across multiple requests. An in-memory cache or a database, are good examples of this. This is becoming an increasingly common reality.

Channels help make concurrent programming saner by taking shared data out of the picture. A channel is a communication pipe between goroutines which is used to pass data. In other words, a goroutine that has data can pass it to another goroutine via a channel. The result is that, at any point in time, only one goroutine has access to the data.

A channel, like everything else, has a type. This is the type of data that we'll be passing through our channel. For example, to create a channel which can be used to pass an integer around, we'd do:

```
c := make(chan int)
```

The type of this channel is `chan int`. Therefore, to pass this channel to a function, our signature looks like:

```
func worker(c chan int) { ... }
```

Channels support two operations: receiving and sending. We send to a channel by doing:

```
CHANNEL <- DATA
```

and receive from one by doing

```
VAR := <-CHANNEL
```

The arrow points in the direction that data flows. When sending, the data flows into the channel. When receiving, the data flows out of the channel.

The final thing to know before we look at our first example is that receiving and sending to and from a channel is blocking. That is, when we receive from a channel, execution of the goroutine won't continue until data is available. Similarly, when we send to a channel, execution won't continue until the data is received.

Consider a system with incoming data that we want to handle in separate goroutines. This is a common requirement. If we did our data-intensive processing on the goroutine which accepts the incoming data, we'd risk timing out clients. First, we'll write our worker. This could be a simple function, but I'll make it part of a structure since we haven't seen goroutines used like this before:

```
type Worker struct {
  id int
}

func (w Worker) process(c chan int) {
  for {
    data := <-c
    fmt.Printf("worker %d got %d\n", w.id, data)
  }
}
```

Our worker is simple. It waits until data is available then "processes" it. Dutifully, it does this in a loop, forever waiting for more data to process.

To use this, the first thing we'd do is start some workers:

```
c := make(chan int)
for i := 0; i < 4; i++ {
  worker := Worker{id: i}
  go worker.process(c)
}
```

And then we can give them some work:

```
for {
  c <- rand.Int()
  time.Sleep(time.Millisecond * 50)
}
```

Here's the complete code to make it run:

```
package main

import (
  "fmt"
  "time"
  "math/rand"
)

func main() {
  c := make(chan int)
  for i := 0; i < 5; i++ {
    worker := &Worker{id: i}
    go worker.process(c)
  }

  for {
    c <- rand.Int()
    time.Sleep(time.Millisecond * 50)
  }
}

type Worker struct {
  id int
}

func (w *Worker) process(c chan int) {
  for {
    data := <-c
    fmt.Printf("worker %d got %d\n", w.id, data)
  }
```

```
}
```

We don't know which worker is going to get what data. What we do know, what Go guarantees, is that the data we send to a channel will only be received by a single receiver.

Notice that the only shared state is the channel, which we can safely receive from and send to concurrently. Channels provide all of the synchronization code we need and also ensure that, at any given time, only one goroutine has access to a specific piece of data.

Buffered Channels

Given the above code, what happens if we have more data coming in than we can handle? You can simulate this by changing the worker to sleep after it has received data:

```
for {
  data := <-c
  fmt.Printf("worker %d got %d\n", w.id, data)
  time.Sleep(time.Millisecond * 500)
}
```

What's happening is that our main code, the one that accepts the user's incoming data (which we just simulated with a random number generator) is blocking as it sends to the channel because no receiver is available.

In cases where you need high guarantees that the data is being processed, you probably will want to start blocking the client. In other cases, you might be willing to loosen those guarantees. There are a few popular strategies to do this. The first is to buffer the data. If no worker is available, we want to temporarily store the data in some sort of queue. Channels have this buffering capability built-in. When we created our channel with `make`, we can give our channel a length:

```
c := make(chan int, 100)
```

You can make this change, but you'll notice that the processing is still choppy. Buffered channels don't add more capacity; they merely provide a queue for pending work and a good way to deal with a sudden spike. In our example, we're continuously pushing more data than our workers can handle.

Nevertheless, we can get a sense that the buffered channel is, in fact, buffering by looking at the channel's `len`:

```
for {
  c <- rand.Int()
  fmt.Println(len(c))
  time.Sleep(time.Millisecond * 50)
}
```

You can see that it grows and grows until it fills up, at which point sending to our channel start to block again.

Select

Even with buffering, there comes a point where we need to start dropping messages. We can't use up an infinite amount of memory hoping a worker frees up. For this, we use Go's `select`.

Syntactically, `select` looks a bit like a switch. With it, we can provide code for when the channel isn't available to send to. First, let's remove our channel's buffering so that we can clearly see how `select` works:

```
c := make(chan int)
```

Next, we change our `for` loop:

```
for {
  select {
  case c <- rand.Int():
    //optional code here
  default:
    //this can be left empty to silently drop the data
    fmt.Println("dropped")
  }
  time.Sleep(time.Millisecond * 50)
}
```

We're pushing out 20 messages per second, but our workers can only handle 10 per second; thus, half the messages get dropped.

This is only the start of what we can accomplish with `select`. A main purpose of select is to manage multiple channels. Given multiple channels, `select` will block until the first one becomes available. If no channel is available, `default` is executed if one is provided. A channel is randomly picked when multiple are available.

It's hard to come up with a simple example that demonstrates this behavior as it's a fairly advanced feature. The next section might help illustrate this though.

Timeout

We've looked at buffering messages as well as simply dropping them. Another popular option is to timeout. We're willing to block for some time, but not forever. This is also something easy to achieve in Go. Admittedly, the syntax might be hard to follow but it's such a neat and useful feature that I couldn't leave it out.

To block for a maximum amount of time, we can use the `time.After` function. Let's look at it then try to peek beyond the magic. To use this, our sender becomes:

```
for {
  select {
  case c <- rand.Int():
  case <-time.After(time.Millisecond * 100):
    fmt.Println("timed out")
```

```
    }
    time.Sleep(time.Millisecond * 50)
  }
```

`time.After` returns a channel, so we can `select` from it. The channel is written to after the specified time expires. That's it. There's nothing more magical than that. If you're curious, here's what an implementation of `after` could look like:

```
func after(d time.Duration) chan bool {
  c := make(chan bool)
  go func() {
    time.Sleep(d)
    c <- true
  }()
  return c
}
```

Back to our `select`, there are a couple of things to play with. First, what happens if you add the `default` case back? Can you guess? Try it. If you aren't sure what's going on, remember that `default` fires immediately if no channel is available.

Also, `time.After` is a channel of type `chan time.Time`. In the above example, we simply discard the value that was sent to the channel. If you want though, you can receive it:

```
case t := <-time.After(time.Millisecond * 100):
  fmt.Println("timed out at", t)
```

Pay close attention to our `select`. Notice that we're sending to `c` but receiving from `time.After`. `select` works the same regardless of whether we're receiving from, sending to, or any combination of channels:

- The first available channel is chosen.
- If multiple channels are available, one is randomly picked.
- If no channel is available, the default case is executed.
- If there's no default, select blocks.

Finally, it's common to see a `select` inside a `for`. Consider:

```
for {
  select {
  case data := <-c:
    fmt.Printf("worker %d got %d\n", w.id, data)
  case <-time.After(time.Millisecond * 10):
    fmt.Println("Break time")
    time.Sleep(time.Second)
  }
}
```

Before You Continue

If you're new to the world of concurrent programming, it might all seem rather overwhelming. It categorically demands considerably more attention and care. Go aims to make it easier.

Goroutines effectively abstract what's needed to run concurrent code. Channels help eliminate some serious bugs that can happen when data is shared by eliminating the sharing of data. This doesn't just eliminate bugs, but it changes how one approaches concurrent programming. You start to think about concurrency with respect to message passing, rather than dangerous areas of code.

Having said that, I still make extensive use of the various synchronization primitives found in the `sync` and `sync/atomic` packages. I think it's important to be comfortable with both. I encourage you to first focus on channels, but when you see a simple example that needs a short-lived lock, consider using a mutex or read-write mutex.

Conclusion

I recently heard Go described as a *boring* language. Boring because it's easy to learn, easy to write and, most importantly, easy to read. Perhaps, I did this reality a disservice. We *did* spend three chapters talking about types and how to declare variables after all.

If you have a background in a statically typed language, much of what we saw was probably, at best, a refresher. That Go makes pointers visible and that slices are thin wrappers around arrays probably isn't overwhelming to seasoned Java or C# developers.

If you've mostly been making use of dynamic languages, you might feel a little different. It *is* a fair bit to learn. Not least of which is the various syntax around declaration and initialization. Despite being a fan of Go, I find that for all the progress towards simplicity, there's something less than simple about it. Still, it comes down to some basic rules (like you can only declare variable once and `:=` does declare the variable) and fundamental understanding (like `new(X)` or `&X{}` only allocate memory, but slices, maps and channels require more initialization and thus, `make`).

Beyond this, Go gives us a simple but effective way to organize our code. Interfaces, return-based error handling, `defer` for resource management and a simple way to achieve composition.

Last but not least is the built-in support for concurrency. There's little to say about goroutines other than they're effective and simple (simple to use anyway). It's a good abstraction. Channels are more complicated. I always think it's important to understand basics before using high-level wrappers. I *do* think learning about concurrent programming without channels is useful. Still, channels are implemented in a way that, to me, doesn't feel quite like a simple abstraction. They are almost their own fundamental building block. I say this because they change how you write and think about concurrent programming. Given how hard concurrent programming can be, that is definitely a good thing.