100 Go Mistakes

How to Avoid Them

Teiva Harsanyi

MEAP

MANNING

MEAP Edition
Manning Early Access Program

100 Go Mistakes: How to Avoid Them

Version 1

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP of *100 Go Mistakes: How to Avoid Them*.

In 2019, I wrote a blog post called *The Top 10 Most Common Mistakes I've Seen in Go Projects*. This post went pretty popular, and I've received much good feedback. From this point onwards, it has started to seed into my mind that perhaps I could make a book about mistakes in Go.

Throughout 2020, I kept collecting ideas and feedback from colleagues, the internet, other books, and, because I have to be fair, mistakes of my own (yes, I made a certain number of mistakes described in this book). At the end of the year, I was convinced I had enough content to start writing about it.

This book collects 100 common mistakes made by Go developers. It tackles all the core aspects of the language: code organization, data and control structures, string, functions and methods, error management, concurrency, testing, optimizations, and production. Each mistake being described explains why this is considered a problem and the different solutions to avoid it.

To read this book, you should have at least a basic understanding of the Go syntax. Then, perhaps you've just started learning Go for a couple of months, or you've been working with it for years now. In both cases, I'm pretty convinced that you will learn from it. This book's goal is to help you become a proficient Go developer and help you in your career.

I would also encourage you to post any questions or comments you may have about the content in the liveBook discussion forum. We appreciate knowing where we can make improvements and increase your understanding of the material.

Thanks again for your interest.

Teiva Harsanyi

brief contents

1 Go: simple to learn but hard to master

This chapter covers

- Reminding us what makes Go an efficient, scalable, and productive language
- Exploring why Go is simple to learn but hard to master

Programming has evolved heavily during the past decades. Most modern systems aren't written anymore by a single person but by organizations consisting of multiple programmers, sometimes even thousands. Our code must be readable, expressive, and maintainable to guarantee a system's durability over the years. Meanwhile, in a fast-moving world, maximizing agility and reducing the time to market is critical for most organizations. Programming should also follow this trend and ensure that software engineers are as productive as possible when reading, writing, and maintaining code.

As a response to these challenges, Google has conceived the Go programming language in 2007. Since then, many organizations have adopted the language to support various use cases: API, automation, databases, CLI (Command Line Interfaces), etc. Go is today considered by many as the language of the cloud. One of Go's key factors of success was because it's regarded as a simple programming language. A newcomer can learn all the language's main features in less than a day. However, as we will see in this chapter, simple to learn doesn't necessarily mean easy to master.

This idea is the one that guided me to write a book to help developers making the most effective use of the Go programming language. Yet, one question: why should you read a book about common Go mistakes? Why not deepen your knowledge with an ordinary book that would dig into different topics? Neuroscientists proved that when we are facing mistakes are the best times for brain growth[1]. Haven't you already experience the process of learning from a mistake and

recall months or even years after the context related to it? As presented in *Learning from Errors* by Janet Metcalfe, this characteristic is because mistakes have a facilitative effect. The main idea is that we can remember not only the error but also the context surrounding the mistake. This is one of the reasons why learning from mistakes is so efficient.

Following these principles, this book will consist of 100 common mistakes made by Go developers in key areas of the language. Meanwhile, to strengthen the facilitative effect we mentioned, each mistake will, as far as possible, be accompanied by real-world examples. This book is not only about theory. Its primary goal is to help you in the path of mastering Go.

1.1 Go Outline

Let's refresh our minds about what makes Go a language so popular and efficient for modern systems.

1.1.1 Features

Feature-wise, Go has no type inheritance, no exceptions, no macros, no partial functions, no support for lazy evaluation or immutability, no operator overloading, no pattern matching, no implicit conversions, etc.

Why are these features missing in the language? The official Go FAQ gives us an insight:

> *Why does Go not have feature X? Your favorite feature may be missing because it doesn't fit, because it affects compilation speed or clarity of design, or because it would make the fundamental system model too difficult.*
>
> *– Go FAQ*

Therefore, counting the number of features of a programming language is probably not the main aspect that we should look at. At least, it's not what Go promotes.

Type inheritance is a good example. The problem with inheritance is that it can make the code path more complex to understand in large codebases. Indeed, it can become quickly complicated for developers to maintain a mental model if most of the interactions are based on inheritance. It has long been advised that programmers should favor composition over inheritance. Thereby, it wasn't included in the language. It's one of many examples where the Go designers intentionally preferred other aspects of the language rather than adding as many features as possible.

Another example is data structures. Go has only three standard data structure types:

- The array that can store a fixed amount of elements of the same type
- The slice, a dynamic version of an array that provides a more powerful and convenient way to store a collection of elements
- The map, the Go hashtable implementation to store a list of key/value

Hence, no linked list, no binary tree, no binary heap, etc. This lack of standard data structures might somewhat surprise newcomers. However, this choice was also made purposely by the Go designers. For example, most often, a slice is the most efficient way for a CPU to access a dynamic list of elements. It involves a predictable access pattern and relies on data locality, making it more efficient in most practical cases compared to a linked list. These three basic types can handle most of the use cases we'll encounter while developing in Go.

Stability is also an essential characteristic of Go. Even though Go receives frequent updates (performance improvements, security patches, etc.), it has remained a very stable language over the past years. Stability is an essential aspect of adopting a language at the scale of an organization. One may even consider it as the best *feature* of the language.

All in all, Go is not the language that contains the most features. Yet, everything is designed to consider all the aspects of a programming language and provide the best possible balance to developers.

1.1.2 Developer Productivity

Today, we build, we test, and we deploy faster than ever. Software programming had to follow this trend. Go is considered a productive language for developers. Let's understand why.

GO IS CONCISE

First of all, we should mention that Go is a concise language: it's based on 25 keywords only. If we compare to other languages, Java and Rust have more than 50, C++ has more than 100, etc.

One may argue whether Go applications are concise, for example, because of errors management. Yet, Go being a concise language means, in general, a shallow learning curve for newcomers. In Go, a developer can go through resources like tour.golang.org and learn Go rapidly.

GO IS EXPRESSIVE

We can also highlight that it's considered expressive. Expressiveness in a programming language means how natural and intuitive we can write and read code. As Robert C. Martin wrote in *Clean Code: A Handbook of Agile Software Craftsmanship*, the ratio of time spent reading versus writing is well over 10 to 1. Therefore, working with an expressive language, especially in large-scale organizations, is a critical characteristic. Also, a reduced number of ways to solve common problems makes large Go codebases, in general, easier to approach compared to other programming languages.

GO COMPILES QUICKLY

Another essential aspect of developer productivity is compilation time. As developers, is there anything more exasperating than having to wait for a build completion to execute unit tests, for example?

Targeting fast compilation has always been a conscious choice from the Go designers. Upfront, Go was designed to provide a model for software construction that simplifies dependency analysis and avoids much of the overhead of C-style include files and libraries. Hence, delivering to the developers fast compilation times.

In summary, Go is considered a productive language for developers for three main reasons: conciseness, expressiveness and efficiency. Yet, as you can imagine, productivity is not the only aspect to consider in a language. Let's see other aspects that make Go a language so popular.

1.1.3 Safety

Go is a statically typed language. Hence, type checking is made at compile-time instead of runtime. It ensures that the code we write is type-safe in most cases.

Furthermore, Go has a garbage collector to help developers in dealing with memory management. It's not the responsibility of the developer to manage memory directly. The garbage collector is responsible for tracking memory allocations and freeing up allocations when they are no longer needed. It adds a little overhead in terms of execution time. For that reason, Go is not intended to be used in real-time applications as it's generally impossible to make strict guarantees about the execution time. However, it's an assumed tradeoff as it reduces the development effort significantly and lowers the risks of application crashes or memory leaks.

Another aspect that may be scary for developers is pointers. A pointer is a variable that contains the address of another variable. Pointers are a core aspect of the Go language. However, pointers in Go are not complicated to handle because they are explicit (unlike references), and there is no such thing as pointer arithmetic. What is the reason? Again, to reduce the risks of writing unsafe applications.

Thanks to these features, Go is a very safe language to work with, which positively impacts Go applications' reliability in general.

1.1.4 Concurrency

In 2005, Herb Sutter, a prominent C++ expert, wrote a blog post called *The free lunch is over*. He mentioned that over the past 30 years, CPU designers had achieved significant improvements in three main areas:

- Clock speed

- Execution optimization
- Caching

Improving these three areas led to improving sequential applications' performance (non-parallel, single-threaded, single-process) throughout the years. However, according to Herb Sutter, it was time to stop expecting CPUs to get consistently faster. This assumption was validated in the past years. As shown in figure 1.1, beginning in about 2004, the speed improvement on a single-threaded execution ceased to be linear. Even worse, it tends to reach a ceiling.

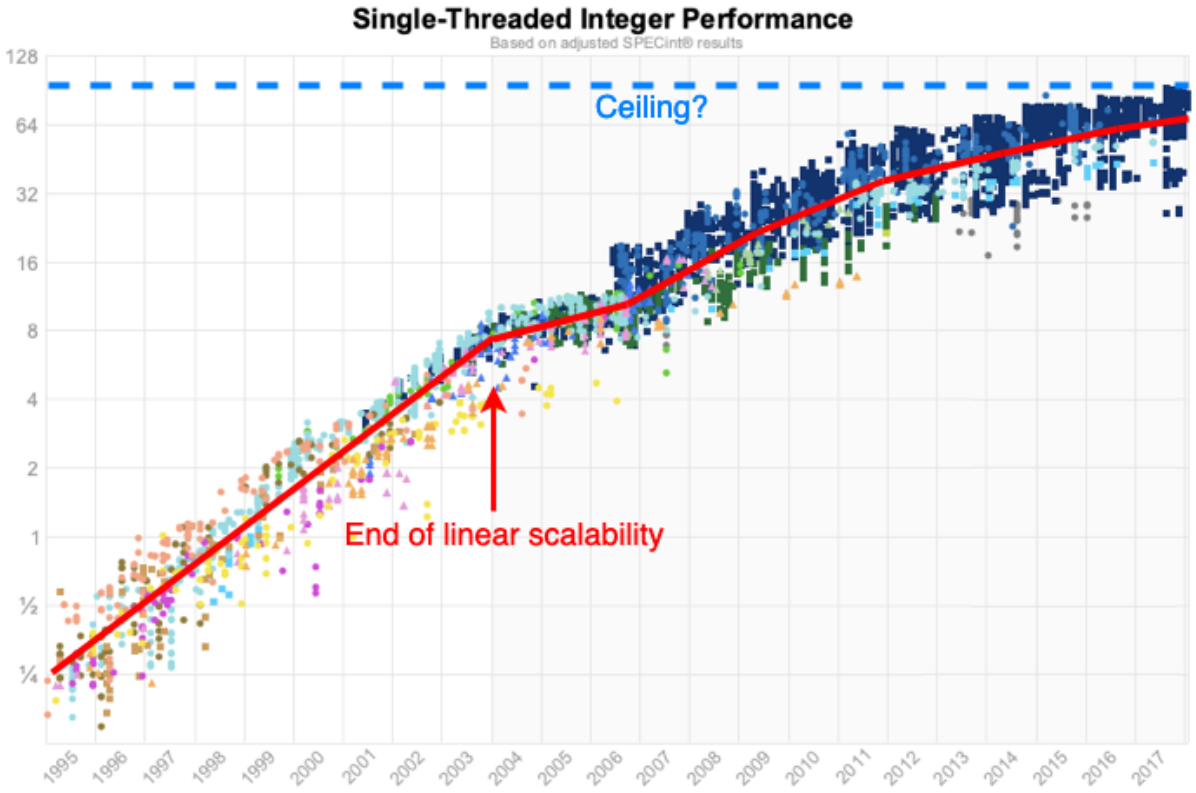

Figure 1.1 In 2004, we can notice that the scale of the single-threaded benchmark ceased to be linear.

Herb Sutter followed up in mentioning that it was the right time to shift the way we develop applications. In the meantime, CPU designers stopped focusing only on clock speed and optimizations. Instead, they began to consider other approaches, like multi-cores and hyperthreading (multiple logical cores on the same physical core). Instead of writing sequential applications and expecting CPUs always to get faster, concurrency would be the next major revolution for software developers.

The Go programming language was designed upfront with concurrency in mind. Its concurrency model is based on Communicating Sequential Processes (CSP). We'll take a look at this model in the next section.

CSP MODEL

The CSP model is a concurrency paradigm that relies on message passing. Instead of having to share memory, processes will communicate by exchanging messages through channels (a kind of mailbox) as shown in figure 1.2:

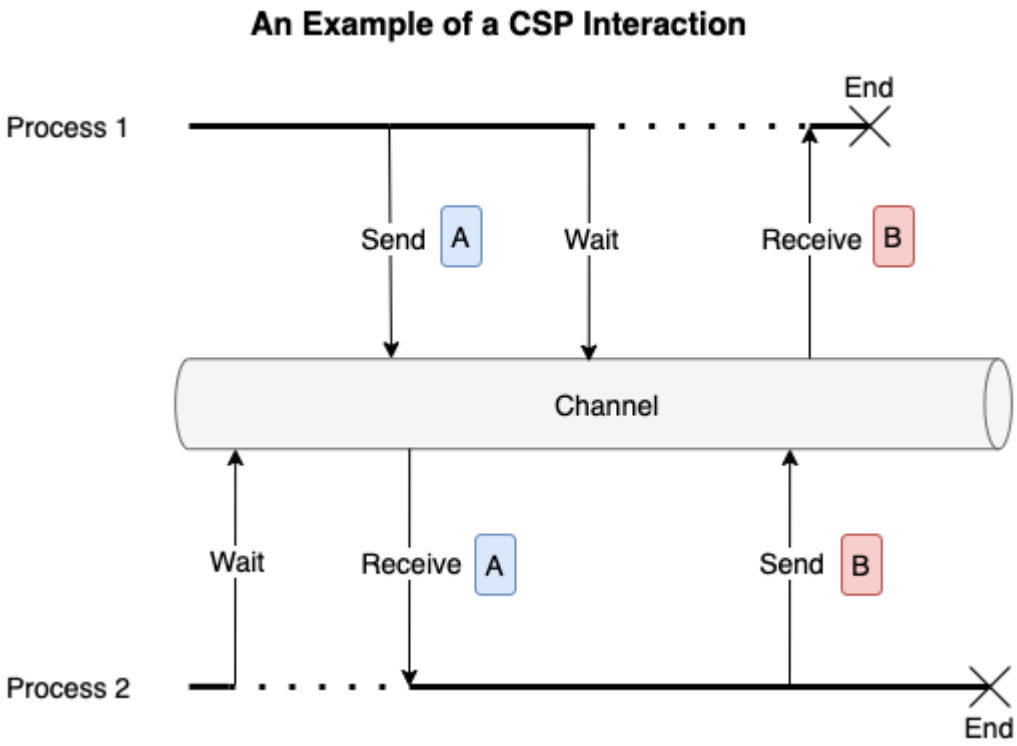

Figure 1.2 An interaction example with two sequential processes communicating through a channel using send and wait for operations.

In figure 1.2, we can see an interaction between two processes based on CSP. Each process is implemented sequentially. No callback to make the whole interaction more complicated to reason about. The first process sends a message `A` and, at some point, waits for a response. The second process waits for message `A`, performs some works, and sends a response as message `B`:

What is the rationale to promote message passing over memory sharing?

Today, all the CPUs have different caching levels to speed up access to the main memory (RAM). A variable shared across different threads may be duplicated multiple times. Therefore, sharing memory is an illusion provided by modern CPUs (we will delve into these concepts in the concurrency chapter).

Embracing message passing fits with the way modern CPUs are built, which, most of the time, has a significant impact on performance. Also, it makes complex interactions easier to reason about. We don't have to handle a complex chain of callbacks: everything is written sequentially.

Go implements the CSP model with two primitives: the goroutine and the channel.

A goroutine can be seen as a *lightweight* thread. Unlike threads that are scheduled by the OS, goroutines are scheduled by the Go runtime. One goroutine belongs to one thread at a time, and one thread can handle multiple goroutines, as we can see in figure 1.3:

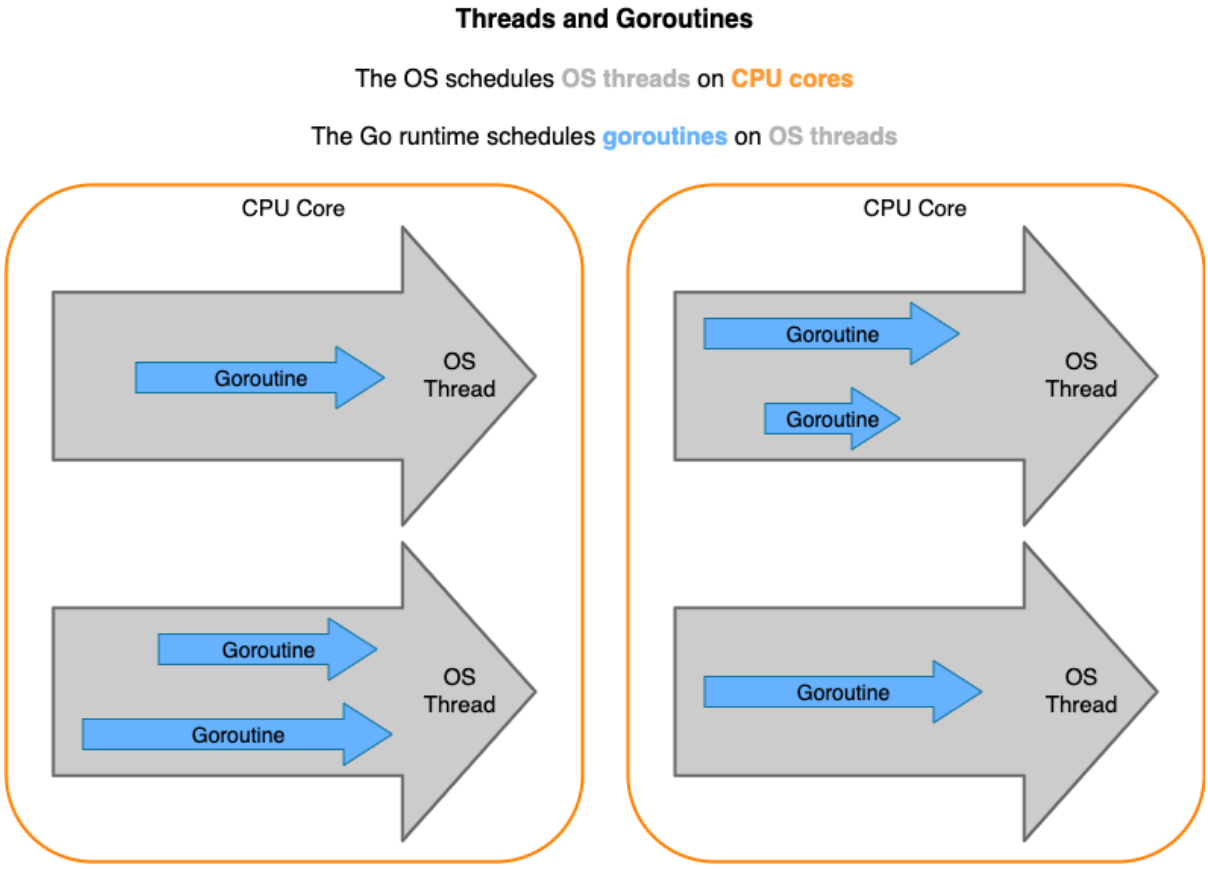

Figure 1.3 A thread is scheduled by the OS on a CPU core (logical processor). Each goroutine belongs to a Go thread. It's up to the Go runtime to schedule the goroutines accordingly and add or remove OS threads.

The OS is in charge of scheduling threads on the CPU cores. Meanwhile, the Go runtime determines the most appropriate number of Go threads depending on the workload and schedule the goroutines on these threads. Compared to a thread, the cost of creating a goroutine is cheaper in terms of startup time and memory (only 2KB of stack size). The context switching operation from one goroutine to another is also faster than context switching threads. Hence, it's not rare to see applications creating hundreds if not thousands of goroutines simultaneously.

On the other hand, a channel is the data structure that allows exchanging data between different goroutines. Each message sent to a channel is received by at most one goroutine. The only broadcast operation (1-to-N) is a channel closure that propagates an event received by multiple goroutines.

Having these primitives as part of the core language is an amazing feature. There is no need to rely on any external library. Developers can write concurrent code in a clean, expressive, and

standard way. It's still possible to use the shared memory approach using mutexes. Yet, in most cases, we should favor the message-passing approach, mainly because, as discussed, such an approach leverages how modern CPUs are built.

Message passing is a powerful concurrency approach, but it does not prevent data races. Fortunately, Go provides a powerful tool with the race detector.

We went through multiple aspects of the language showing how much Go is powerful and simple to learn. So why should you read a book about Go to extend your knowledge?

1.2 Simple Doesn't Mean Easy

There is a subtle difference between simple and easy. Simple applied to a technology means not complicated to learn or to understand. However, easy means that we would achieve everything without much effort. Go is simple to learn yet hard to master.

Let's take concurrency as an example. In 2019, a study focused on concurrency bugs was published: Understanding Real-World Concurrency Bugs in Go[2]. This study was the first systematic analysis on concurrency bugs, focused on six popular Go repositories: Docker, Kubernetes, etcd, CockroachDB, BoltDB, and gRPC.

There are many interesting takeaways from this study. In all these repositories, the authors have shown that despite being fostered in Go, the message passing approach was used less often than the shared memory approach. The study also highlighted that most of the blocking bugs were caused by an inaccurate usage of message passing, despite the belief that message passing is easier to handle and less error-prone.

What shall we conclude about this study? Should we be scared about using the message passing approach in our applications? Of course not. First, both paradigms, shared memory and message passing can live side by side. It also means that it's up to us, Go developers, to make some progress and thoroughly understand the implications of the message passing approach to avoid repeating the most common concurrency mistakes. However, it means that message passing, despite being easy to learn and use in theory, isn't easy to master in practice.

This idea, simple doesn't mean easy, can be generalized to many aspects of Go, not only concurrency; for example:

- When to use interfaces?
- When to use value or pointer receivers?
- How to deal efficiently with slices?
- How to handle error management cleanly and expressively?
- How to avoid memory leaks?
- How to write relevant tests and benchmarks?
- How to make an application production-ready?

To be a proficient Go developer, we should have a thorough understanding of many aspects of the language, which requires a lot of time, effort, and mistakes. This book aims to help you accelerate your way to become this proficient developer by collecting and presenting 100 common Go mistakes in various aspects of the language: basics, code organization, data and control structures, string, functions and methods, error management, concurrency, testing, optimizations, and production.

1.3 Summary

- Go is a modern programming language that puts a lot of effort into developer productivity, which is crucial for most companies today.
- Go is simple to learn but hard to master; this is why we need to deepen our knowledge if we want to make the most effective use of Go.
- Learning via mistakes and concrete examples is a powerful means.

2 Basics

This chapter covers

- Variable shadowing
- Logging side effects
- Comparing values
- Preventing common JSON mistakes
- Formatting network addresses
- Handling enums
- Using `defer` and understanding how arguments are handled
- Releasing resources
- Octal literal
- Integer overflow
- Using linters

In this chapter, we are going to cover various basic mistakes made by Go developers. For example, when it comes to implementing HTTP services in Go, we have to know how to prevent common JSON mistakes, format network addresses, and release HTTP body resources using `defer` efficiently. We will also see some basic mistakes that all developers may have already made, such as unintended variable shadowing, how to handle value comparisons in Go, or not using linters. We will delve into each mistake and explain when we can face such problems and the possible solutions to overcome them.

2.1 Unintended Variable Shadowing

The scope of a variable refers to its visibility. In other words, the part of the program where a name binding is valid. In Go, a variable name declared in a block may be redeclared in an inner block. This principle, called variable shadowing, is prone to common mistakes.

In the following example, we will see an occurence of a bug because of a shadowed variable. We will create an HTTP client in two different ways, depending on the value of a `tracing` boolean:

```
var client *http.Client   ❶
if tracing {
    client, err := createClientWithTracing()   ❷
    if err != nil {
        return err
    }
    log.Println(client)
} else {
    client, err := createDefaultClient()   ❸
    if err != nil {
        return err
    }
    log.Println(client)
}
// Use client
```

❶ Declare a `client` variable

❷ Create an HTTP client with tracing enabled, the `client` variable is shadowed in this block

❸ Create a default HTTP client, the `client` variable is also shadowed in this block

First, we declare a `client` variable. Then, in both inner blocks, we used the `:=` operator, also called the short variable declaration operator. This operator creates a new `client` variable with the same name as the original one; it doesn't assign a value to the initial `client` variable declared. As a result, following this example, the HTTP client will always be nil.

> **NOTE** **This code compiles because the inner `client` variables are used in the logging calls. If not, we would have compilation errors: `client declared and not used`.**

How can we ensure assigning a value to `client`? There are two different options.

The first option is to use temporary variables in the inner blocks this way:

```
var client *http.Client
if tracing {
    c, err := createClientWithTracing()   ❶
    if err != nil {
        return err
    }
    client = c   ❷
} else {
    c, err := createDefaultClient()
    if err != nil {
        return err
    }
    client = c
}
// Use client
```

❶ Create a `c` temporary variable

❷ Assign this temporary variable to `client`

The `c` variables live only inside the if/else block. Then, we assign these variables to `client`.

The second option is to use the assignment operator (=) in the inner blocks to assign the function results to the `client` variable directly. However, it requires creating an `error` variable as the assignment operator works only if a variable name has already been declared:

```
var client *http.Client
var err error   ❶
if tracing {
        client, err = createClientWithTracing()   ❷
        if err != nil {
                return err
        }
} else {
        client, err = createDefaultClient()
        if err != nil {
                return err
        }
}
```

❶ Declare an `err` variable

❷ Use the assignment operator to assign the `*http.Client` returned to the `client` variable directly

In this example, we also assign to `client` the result of the inner calls. Which option is the best? The first option is probably more convenient in most cases, but there is no strong directive.

Variable shadowing occurs when a variable name is redeclared in an inner block, and we've seen that this practice is prone to mistakes. Imposing a rule to forbid shadowed variables should be taken depending on the projects and the contexts. For example, sometimes, it can be pretty convenient to reuse an existing variable name, like `err` for errors. Yet, in general, we should remain cautious as we have seen that we can face a scenario where a code compiles, but an assignment may not be done to the variable we believe. Later in this chapter, we will also see how to detect shadowed variables, which may help us spot possible bugs.

In the next section, we will discuss possible side effects related to logging.

2.2 Ignoring Logging Side Effects

Logging is an important aspect of application monitoring. While using logging in Go or any language, it's essential to understand the potential side effects and impacts. Let's first refresh our minds about how logging works in Go; then, we'll see the reasons why we have to be cautious.

The Go standard library has a built-in `log` package providing most of the basic logging features.

It defines a `log.Logger` type with methods for formatting output. It also includes a predefined `log.Logger` accessible through helper functions:

- `log.Print` (not formatted), `log.Printf` (`f` for formatted), and `log.Println` (`ln` to indidicate a log followed by a new line)
- `log.Fatal`, `log.Fatalf`, and `log.Fatalln`
- `log.Panic`, `log.Panicf`, and `log.Panicln`

Here is a basic example using `log.Printf` to print to the standard logger a message with an argument:

Listing 2.1 Basic log example

```
package main

import "log"

func main() {
    c := count()
    log.Printf("count=%d", c)   ❶
}

func count() int {
    return 0
}
```

❶ Log the c variable

This code will print the following log message:

```
2020/12/20 12:34:38 count=0
```

Now, let's use logging in a concrete example. We will log a message using `log.Fatal` if the Go application was executed without arguments:

```
func main() {
    var mode string
    if len(os.Args) == 1 {   ❶
        log.Fatal("warning: mode is empty")   ❷
        mode = "default"
    } else {
        mode = os.Args[1]
    }
    log.Printf("using mode %s", mode)
    // ...
}
```

❶ Check if no arguments was provided

❷ Log a fatal message

If we run this example without arguments, it will print the following output:

```
2020/11/29 00:06:14 mode is empty

Process finished with exit code 1
```

As we can see, the application exited right after the `log.Fatal` call; it did not continue its execution. Looking at the implementation of `log.Fatal` function, we can notice that following the logging part, the function will call `os.Exit(1)`:

```
func Fatal(v ...interface{}) {
    std.Output(2, fmt.Sprint(v...))
    os.Exit(1)
}
```

Calling `os.Exit(1)` will cause the application to stop right away. It's the reason why our application stopped right after the call to `log.Fatal`.

We should also have in mind that `log.Panic` also lead to a side effect after logging the message:

```
func Panic(v ...interface{}) {
    s := fmt.Sprint(v...)
    std.Output(2, s)
    panic(s)
}
```

We will see in the error management chapter what is the exact impact of calling `panic`. In a nutshell, it's another side effect to consider as it will stop the current goroutine's normal execution.

When we use a logging library, be it the standard `log` package or an external logging library, we have to understand the logging functions' side effects clearly. It isn't considered bad practice to use a logging function with a side effect as long as we're aware of it. We should also remain careful while migrating from one library to another if the side effects are not identical.

In the next section, we will see how to work with enums and avoid some common mistakes.

2.2.1 Enum Validity

Another best practice to have in mind is when we need to check whether an enum value is valid, meaning within the range of the constants defined. Indeed, as enums in Go are constructed using type aliases, we can't implement restrictions. For example, what will happen if we receive the following JSON body?

```
{
  "id": 1236,
  "weekday": 100   ❶
}
```

❶ 100 is outside of the range of possible weekdays

Unmarshaling this payload will not cause any error. Indeed, as the `Weekday` type is an `int`, `100` is a valid value.

To check if the provided `weekday` field is valid, we can implement a custom check against its value (`7`). There is another way to declare an `end` constant as the latest enum value and implement the validity check function like so:

```
type Weekday uint32

const (
    Unknown Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
    end   ❶
)

func (w Weekday) isValid() bool {
    return w < end   ❷
}
```

❶ Unexported constant

❷ Check value against `end`

NOTE **The `end` constant should stay unexported as clients shouldn't use it.**

In this implementation, the validity check is implemented against the `end` constant. Even if new `Weekday` values are added, if we keep the `end` constant as the last one of the enum, the `isValid` method will always tell us whether the provided value is considered as a valid week day:

```
fmt.Println(Monday.isValid()) // true
fmt.Println(Weekday(2).isValid()) // true
fmt.Println(Weekday(100).isValid()) // false
```

In summary, if we have to handle an enum:

- We should always define a specific type.
- If we don't care about the enum values, we can use the constant generator `iota` to simplify the definition. Otherwise, for example, "this constant has to be equal to 3", then we shouldn't juggle with `iota` but explicit values.
- The unknown value should always be assigned to the zeroed value.
- To check the validity of an enum, we can do it with an unexported constant defined as the latest one.

Let's now jump into the next section. When it comes to comparing Go's values, what are the common mistakes to avoid as Go developers?

2.3 Comparing Values Incorrectly

Comparing values is a common operation in software development. Be it in a function to compare two objects or in testing to compare a value against an expectation, implementing comparisons is quite frequent. Our first instinct could be to use the == operator everywhere. Yet, as we will see in this section, this shouldn't always be the case. So when is it appropriate to use ==?

Let's start with a concrete example. We will create a basic `customer` struct and use == to compare two instances. What should be the output of this code?

```
type customer struct {
   id string
}

func main() {
   cust1 := customer{id: "x"}
   cust2 := customer{id: "x"}
   fmt.Println(cust1 == cust2)
}
```

Comparing these two `customer` structs is a valid operation in Go, and it will print `true`. Now, what happens if we make a slight modification to the `customer` struct to add a slice field:

```
type customer struct {
   id         string
   operations []float64   ❶
}

func main() {
   cust1 := customer{id: "x", operations: []float64{1.}}
   cust2 := customer{id: "x", operations: []float64{1.}}
   fmt.Println(cust1 == cust2)
}
```

❶ New field

We might expect this code to print `true` as well. However, it doesn't even compile:

```
invalid operation: cust1 == cust2 (struct containing []float64 cannot be compared)
```

The problem relates to how == and `!=` operators work. It's essential to understand how to use both to make sure we can make comparisons effectively. We can use both operators to compare two different types if the two types are *comparable*. Comparable types in Go include:

- Booleans: == and `!=` to compare whether two booleans are equals.
- Numerics: == and `!=` to compare whether two numerics are equals. These operations compile if both variables have the same type or converted to be of the same type.
- Strings: == and `!=` to compare whether two strings are equals. We can also use , >=, < and > operators to compare two strings based on their lexical order.
- Pointers: == and `!=` to compare whether two pointers point to the same value in memory

or if both are nil.

- Channels: == and `!=` to compare whether two channels were created by the same call to `make` or if both are nil.

We can also add to this lists structs and arrays if they are composed of comparable types only. So, no maps and no slices. In the first version, `customer` was composed of a single comparable type (a string), so the == comparison was valid. Conversely, in the second version, as `customer` contained a slice, it wasn't possible to use the == operator and lead to a compilation error.

We should also know the possible issues of using == and `!=` with `interface{}` types. What if we first cast them into `interface{}` instead of comparing two' customer' struct?

```
var cust1 interface{} = customer{id: "x", operations: []float64{1.}}
var cust2 interface{} = customer{id: "x", operations: []float64{1.}}
fmt.Println(cust1 == cust2)
```

This code compiles as both structs are cast into `interface{}` types. Yet, it leads to a runtime panic:

```
panic: runtime error: comparing uncomparable type main.customer
```

Having these behaviors in mind, what are the options if we have to compare two slices, two maps, or two structs containing non-comparable types? If we stick with the standard library, the other option is to use reflection and `reflect.DeepEqual`. This function reports whether two elements are *deeply equal*. The elements it accepts are basic types plus array, struct, slice, map, pointer, interface, and functions.

Let's rerun the first example, using `reflect.DeepEqual`:

```
cust1 := customer{id: "x", operations: []float64{1.}}
cust2 := customer{id: "x", operations: []float64{1.}}
fmt.Println(reflect.DeepEqual(cust1, cust2))
```

Here, even though the `customer` struct contains non-comparable types (slice), it will operate as expected and print `true`.

However, there are two main catches to keep in mind while using `reflect.DeepEqual`. We already mentioned the first one as this function makes the distinction between empty and nil collections. Let's compare two `customer` structs with one containing a nil slice whereas the second one an empty slice:

```
var cust1 interface{} = customer{id: "x"}   ❶
var cust2 interface{} = customer{id: "x", operations: []float64{}}   ❷
fmt.Printf("%t\n", reflect.DeepEqual(cust1, cust2))
```

❶ Nil slice

❷ Empty slice

This code prints `false` as `reflect.DeepEqual` considers empty and nil collections different. Is this a problem? Not necessarily. For example, if we want to compare the result of two unmarshaling operations, we may wish for this difference to be raised. It depends on our use case. Yet, it's worth having this behavior in mind to use `reflect.DeepEqual` effectively.

The other catch is something pretty standard in most of the languages. As this function uses reflection, it has a performance penalty. Doing a couple of benchmarks locally with structs of different sizes, on average `reflect.DeepEqual` is about 100 times slower than `==`.

In general, we should bear in mind that the `==` operator is pretty limited. For example, it doesn't work with slices and maps. Using `reflect.DeepEqual` is a solution in most cases, but there are a couple of catches, such as a performance penalty. Some other options are possible such as implementing a customer equals function/method or using external libraries such as `google/go-cmp` or `stretchr/testify` in the context of unit tests. One last thing to mention, we should also note that the standard library has some existing comparison methods. For example, if we want to compare two byte slices, we can use the `bytes.Compare` function. We should always ensure whether the standard library didn't already cover your use case before looking at external ones.

In the next section, we will discuss three common mistakes related to JSON handling.

2.4 JSON Handling Mistakes

Go has excellent support for JSON with the `encoding/json` package. This section will cover three common mistakes related to encoding (marshaling) and decoding (unmarshaling) JSON data.

2.4.1 Empty JSON

First, let's tackle a recurring problem: marshaling a type that generates an empty JSON. We will use the following `point` struct:

```
type point struct {
    x float32
    y float32
}
```

This struct represents a cartesian point with two fields: `x` and `y`. Let's create a new `point` and use the standard `json.Marshal` to encode it into a JSON output:

```
p := point{3., 2.5}
b, err := json.Marshal(p)   ❶
if err != nil {
    return err
}
fmt.Println(string(b))   ❷
```

❶ Marshal `p`

❷ `b` being an `[]byte` to make it human-readable we need to convert it in a string first.

Unfortunately, the output is empty:

```
{}
```

So what have we missed? Is it because we forgot to set the JSON tags on the struct? In Go, a struct tag is an annotation that appears after the field type. We can use tags to force the field name:

```
type point struct {
    x float32 `json:"x"`   ❶
    y float32 `json:"y"`   ❷
}
```

❶ Set the `x` JSON tag

❷ Set the `y` JSON tag

Yet, the output remains empty with this version: `{}`. Indeed, it isn't required to set the JSON struct tags to marshal/unmarshal JSON data. By default, the JSON field names will be the name of the struct fields.

Is it because the type is unexported? As we recall, a Go type is exported (public) if it starts with a capital. Let's rename the `point` struct into `Point`:

```
type Point struct {   ❶
    x float32
    y float32
}
```

❶ The struct is now exported

Yet, still empty: `{}`. So a struct doesn't have to be exported to be marshaled/unmarshaled. However, the problem we faced is because the **fields** inside the struct are unexported. Indeed, if we now try with this new `point` struct:

```
type point struct {
    X float32   ❶
    Y float32   ❷
}
```

❶ `X` field is now exported

❷ `Y` field is now exported

The result is not empty anymore:

```
{"X":3,"Y":2.5}
```

Therefore, to be marshaled/unmarshaled, the fields of a struct have to be exported.

One thing to note, what if we want to ignore some fields during the marshaling step? For example, we may want to ignore marshaling a password field. There are two options. First, as we have seen, we make these fields unexported. The second option if we are obliged to keep these fields exported because they are used by external packages, for example, is to use a specific JSON tag value. Indeed, using `-` makes an exported field ignored during marshaling/unmarshaling:

```
type Foo struct {
    A string
    b string   ❶
    C string `json:"-"`   ❷
}

f := Foo{
    A: "a",
    b: "b",
    C: "c",
}
b, err := json.Marshal(f)
if err != nil {
    return err
}
fmt.Println(string(b))
```

❶ The field is ignored because unexported

❷ The field is ignored because of the `-` JSON tag

As the `b` field is unexported and `C` is explicitly ignored, the marshaled struct will only contain `A`:

```
{"A":"a"}
```

So to be marshaled/unmarshaled, the JSON fields have to be exported. We don't have to export the type nor enforce using tags. We can also ignore fields by making them unexported or using a specific JSON tag.

In the next section, we will see a common JSON mistake occurring while dealing with anonymous fields.

2.4.2 Missing Fields

In Go, a struct field is called embedded if we declare it without a name. Embedded fields are used to *promote* the fields and methods of the embedded type, as we will see in the next example:

```
type Event struct {
    ID int
    time.Time   ❶
}
```

❶ Embedded field

`time.Time` is an embedded field as it's declared without a name. If we create an `Event` struct, we can access to the `time.Time` methods directly at the `Event` level:

```
event := Event{}
second := event.Second()   ❶
```

❶ If the `time.Time` was not embedded, for example with a specific `t` field name, we would have access to the method using `event.t.Second()`

The `Second` method is *promoted* as accessible via the `Event` struct directly. It's why embedded fields are mainly used with structs or interfaces, not basic types like `int` or `string`.

What are the possible impacts of embedded fields with JSON marshaling? Let's find it out in the following example. We will instantiate an `Event` and marshal it into JSON. What should be the output of this code?

```
event := Event{
    ID:   1234,
    Time: time.Now(),   ❶
}

b, err := json.Marshal(event)
if err != nil {
    return err
}

fmt.Printf("json: %s\n", string(b))
```

❶ The field name of an anonymous field during a struct instantiation is the name of the struct (`Time`)

We may expect this code to print the following:

```
{"ID":1234,"Time":"2021-04-19T21:15:08.381652+02:00"}
```

Instead, it will print this:

```
"2020-12-21T00:08:22.81013+01:00"
```

How can we explain this output? What happened to the `ID` field and the 1234 value? As this field is exported, it should have been marshaled. To understand this problem, we have to clarify two things.

First, if an embedded field type implements an interface, the struct containing the embedded field will also implement this interface. In a sense, it inherits from a capability. Let's take a look at the following example where we define one, implemented by a `Foo` struct and `Foo` being an embedded member in a `Bar` struct:

```
type Printer interface {   ❶
    print()
}

type Foo struct{}

func (f Foo) print() {   ❷
    fmt.Println("foo")
}

type Bar struct {
    Foo   ❸
}
```

❶ Define a `Printer` interface and a `print()` method

❷ Make `Foo` implementing `Printer`

❸ Add to `Bar` an embedded `Foo` type

Here, as `Foo` implements `Printer` and that `Foo` is an embedded member of `Bar`, `Bar` is a `Printer`. We can instantiate a `Bar` struct and call the `print` method without accessing its internal `Foo` member.

The second point to clarify is that we can override the default marshaling behavior by making a type implementing `json.Marshaler` interface. This interface contains a single `MarshalJSON` method:

```
type Marshaler interface {
    MarshalJSON() ([]byte, error)
}
```

Here is an example with a custom marshaling:

```
type foo struct{}   ❶

func (_ foo) MarshalJSON() ([]byte, error) {   ❷
    return []byte(`"foo"`), nil   ❸
}

func main() {
    b, err := json.Marshal(foo{})   ❹
    if err != nil {
        panic(err)
    }
    fmt.Println(string(b))
}
```

❶ Define the struct

❷ Implement the `MarshalJSON` method

❸ Return a static response

❹ `json.Marshal` will then rely on the custom `MarshalJSON` implementation

As we have overridden the JSON marshaling behavior, this code will print `foo`.

Having these two points clarified, let's get back to the problem with the `Event` struct. We have to know that `time.Time` **implements** the `json.Marshaler` interface. As `time.Time` is an embedded field of `Event`, it promotes its methods. Therefore, `Event` also implement `json.Marshaler`.

Consequently, when we pass an `Event` to `json.Marshal`, it will not use the default marshaling behavior but the one provided by `time.Time`. It's the reason why marshaling an `Event` leads ignored the `ID` field.

To fix this issue, there are two main possibilities.

First, we can make the `time.Time` field not embedded anymore like this:

```
type Event struct {
    ID   int
    Time time.Time   ❶
}
```

❶ `time.Time` is not an embedded type anymore

This way, if we marshal a version of this `Event` struct, it will print something like:

```
{"ID":1234,"Time":"2020-12-21T00:30:41.413417+01:00"}
```

If we want to keep or have to keep the `time.Time` field as embedded, the other option is to make `Event` implementing the `json.Marshaler` interface. Yet, this solution is more cumbersome and will require making sure that the `MarshalJSON` method is always up to date with the `Event` struct.

We should be careful with embedded fields. While promoting the fields and methods of an embedded field type can be sometimes convenient, it can also lead to subtle bugs as the parent struct may then implement interfaces implicitly. When using embedded fields, we should ensure an understanding of the possible side effects.

In the next section, we will see another common JSON mistake related to using `time.Time`.

2.4.3 JSON and Monotonic Clock

An OS handles two different clock types: wall and monotonic. Let's see in this section a possible impact while working with JSON and `time.Time` and learn why this clock distinction is essential to understand.

In the following example, we will keep considering an `Event` struct but containing a single `time.Time` field (not embedded):

```
type Event struct {
    Time time.Time
}
```

We will instantiate an `Event`, marshal it into JSON and unmarshal it into another struct. Then, we will compare both structs. Let's find out if the marshaling/unmarshaling process is always symmetric:

```
t := time.Now()  ❶
event1 := Event{  ❷
    Time: t,
}

b, err := json.Marshal(event1)  ❸
if err != nil {
    return err
}

var event2 Event
err = json.Unmarshal(b, &event2)  ❹
if err != nil {
    return err
}

isEquals := event1 == event2
```

❶ Get the current local time

❷ Instantiate an `Event` struct

❸ Marshal into JSON

❹ Unmarshal JSON

What should be the value of `isEquals`? It will be false, not true. How can we explain it?

First, let's print the content of `event1` and `event2`:

```
fmt.Println(event1.Time)  ❶
fmt.Println(event2.Time)  ❷
```

❶ *2021-01-10 17:13:08.852061 +0100 CET m=+0.000338660*

❷ *2021-01-10 17:13:08.852061 +0100 CET*

So we can notice that it prints two different contents. `event1` is similar to `event2` except for the `m=+0.000338660` part. What is the meaning of it?

We should know that operating systems provide two clock types. First, the wall clock is used to know the current time of the day. This clock is subject to potential variations. For example, if synchronized using NTP (Network Time Protocol), the clock can jump backward or forward in time. We shouldn't measure durations using the wall-clock as we may face strange behavior such as negative durations. It's the reason why operating systems provide a second clock type: monotonic clocks. The monotonic clock guarantees that the time will always move forward and will not be impacted by jumps in time. It can be affected by potential frequency adjustments (if the server, for example, detects that the local quartz is moving at a different pace than the NTP server) but never by jumps in time.

In Go, instead of splitting the two clocks in two different APIs, the `time.Time` may contain both a wall and monotonic time.

When we get the local time using `time.Now()`, it returns a `time.Time` with both times:

```
2021-01-10 17:13:08.852061 +0100 CET m=+0.000338660
---------------------------------- --------------
             Wall time               Monotonic time
```

Conversely, when we unmarshal the JSON, the `time.Time` field doesn't contain the monotonic time, only the wall time. Therefore, when we compared both structs, the result was false because of a monotonic time difference. It's also the reason why we noticed a difference while printing both structs.

How can we fix this problem? There are two main options.

When we used `==` operator to compare both `time.Time`, it compared all the struct fields, including the monotonic part. To avoid this, we can use the `Equal` method instead:

```
areTimesEqual := event1.Time.Equal(event2.Time)  ❶
```

❶ true

The `Equal` methods doesn't consider monotonic time. Therefore, `areTimesEqual` will be true. However, in this case, we only compared the `time.Time` fields, not the parent `Event` structs.

The second option is to keep the `==` to perform the comparison of the two structs but to strip away the monotonic time using the `Truncate` method with the 0 value:

```
t := time.Now()
event1 := Event{
    Time: t.Truncate(0),   ❶
}

b, err := json.Marshal(event1)
if err != nil {
    return err
}

var event2 Event
err = json.Unmarshal(b, &event2)
if err != nil {
    return err
}

isEquals := event1 == event2   ❷
```

❶ Strip away the monotonic time

❷ Performs the comparison using == operator

With this version, the two `time.Time` fields are now equal. Therefore, `isEquals` will be true.

In summary, the marshaling/unmarshaling process is not always symmetric, and we faced this case with a struct containing a `time.Time`. We should keep this principle in mind so that we don't write erroneous tests, for example.

In the next section, let's discuss how to not format network addresses in Go.

2.5 Formatting Network Addresses for IPv4 Solely

To create an HTTP server using the standard Go library, we have to provide a network address. For example, the first argument to the `http.ListenAndServe` function is the address we want to use. It is pretty frequent to see Go repositories using the `fmt.Sprintf` function to format the TCP address using the hostname and the port:

```
func createHTTPServer(host, port string, handler http.Handler) error {
    addr := fmt.Sprintf("%s:%s", host, port)   ❶
    return http.ListenAndServe(addr, handler)   ❷
}
```

❶ Format TCP address

❷ Use this address for the server to listen on

What we pass to `http.ListenAndServe` as a network address results from the `fmt.Sprintf` function call. This implementation is perfectly valid with IPv4 hosts. For example:

- `myhostname`
- `127.0.0.1`
- Empty string

Yet, what if we use an IPv6 host? If we run our function using the IPv6 version of localhost (`0000:0000:0000:0000:0000:0000:0000:0001`), it will generate the following error:

```
listen tcp: address 0000:0000:0000:0000:0000:0000:0000:0001:90:
  too many colons in address
```

Indeed, if we pass an IPv6 address to `http.ListenAndServe`, it has to be formatted with brackets this way: `[0000:0000:0000:0000:0000:0000:0000:0001]:80`. Should we revise our implementation to handle a different logic to format the address depending on the IP version used?

Fortunately, the `net` package provides a solution. Instead of using `fmt.Sprintf`, we can use `net.JoinHostPort` this way:

```
func createHTTPServer(host, port string, handler http.Handler) error {
    addr := net.JoinHostPort(host, port)
    return http.ListenAndServe(addr, handler)
}
```

`net.JoinHostPort` is compatible with both IPv4 and IPv6 hosts. If your company migrates to IPv6, you will not be required to modify how to create HTTP servers in every repository. It's the reason why we should rely on `net.JoinHostPort` instead of a manual operation to format a network address.

In the next section, let's see how to not handle enums in Go and the best practices.

2.6 Handling Enums Incorrectly

An enum is a data type consisting of a set of values. In Go, there is no `enum` keyword as such. However, the best practice to deal with a set of values is to use type aliasing and constants. Yet, we don't achieve the same level of safeness that we can get with other languages. It's the reason why we have to handle enums carefully. Let's go through a couple of best practices and common mistakes to avoid.

Here is an example to list the days of the week:

```
type Weekday int   ❶

const (
    Monday    Weekday = 0   ❷
    Tuesday   Weekday = 1
    Wednesday Weekday = 2
    Thursday  Weekday = 3
    Friday    Weekday = 4
    Saturday  Weekday = 5
    Sunday    Weekday = 6
)
```

❶ Declare a custom `Weekday` type

❷ Create a `Monday` constant as a `Weekday`

The benefit of creating a `Weekday` type is to enforce compile-time checks and improve readability. If we hadn't created a `Weekday` type, the following function's signature could be a little blurry for callers:

```
func GetCurrentWeekday() int {
    // ...
}
```

An `int` can hold any value, and a reader can't guess what will return this function without reading the documentation or the code. Instead, defining a `Weekday` type, it makes the function signature clearer:

```
func getCurrentWeekday() Weekday {
    // ...
}
```

In this case, we make the return type explicit.

The way we created the `Weekday` enum is ok. However, there is an idiomatic way to declare the enum constants in Go, using the constant generator `iota`.

NOTE **In this example, we could have also declared `Weekday` as an `uint32` to enforce positive values and ensure a 32-bit allocation per `Weekday` variable.**

2.6.1 *iota*

`iota` is used to create a sequence of related values without explicitly setting the values. It instructs the compiler to duplicate every constant expression until the block ends or an assignment is found.

Here is the `Weekday` version written using `iota`:

```
type Weekday int

const (
    Monday Weekday = iota   ❶
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
)
```

❶ Use `iota` to define an enum

The value of `iota` begins at zero and increments by one for each item. This version is identical

to the first version we wrote:

- `Monday` = 0
- `Tuesday` = 1
- `Wednesday` = 2
- etc.

Using `iota` allows us to avoid declaring constant values manually. Manually setting constants can be error-prone with large enums, for example. Furthermore, we don't have to repeat the `Weekday` type to all the constants: all the constants we defined are a `Weekday` type.

NOTE **We can also use `iota` in more complex expressions. Here is an example took from Effective Go to handle a `ByteSize` enum:**

```
type ByteSize float64

const (
    _           = iota   ❶
    KB ByteSize = 1 << (10 * iota)   ❷
    MB   ❸
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

❶ **Ignore the first value by assigning to blank identifier**

❷ **At this stage `iota` is equal to 1, hence `KB` will be set to 1 << (10 * 1)**

❸ **At this stage `iota` is equal to 2, it will repeat the previous expression, hence `MB` will be set to 1 << (10 * 2)**

Let's now see how to handle unknown values in Go enums.

2.6.2 Unknown Value

Now that we have understood or refreshed our minds on how to handle an enum in Go, let's consider the following example. We will implement an HTTP handler that unmarshals (decodes) a JSON request to a `Request` struct. This struct will contain a `Weekday` type in which we will add an `Unknown` value. Here is a first implementation:

```
type Weekday int   ❶

const (
    Monday Weekday = iota
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
    Unknown   ❷
)

type Request struct {   ❸
    ID      int     `json:"id"`
    Weekday Weekday `json:"weekday"`
}

func httpHandler(w http.ResponseWriter, r *http.Request) {   ❹
    bytes, err := readBody(r)   ❺
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    var request Request
    err = json.Unmarshal(bytes, &request)   ❻
    if err != nil {
        w.WriteHeader(http.StatusBadRequest)
        return
    }

    // Use request
}
```

❶ Reuse the `Weekday` enum we defined

❷ `Unknown` constant

❸ Declare a `Request` struct containing a `Weekday` field

❹ Implement an HTTP handler

❺ Read the body and returns a `[]byte`

❻ Unmarshal the JSON request

In this example, we created a `Request` struct that is unmarshaled from a JSON request. This code is perfectly valid Go-wise. For example, we can receive such a JSON body and unmarshal it correctly:

```
{
  "id": 1234,
  "weekday": 0
}
```

Here, the `Weekday` field would be equal to 0: `Monday`.

Now, what will happen if the JSON body doesn't contain the `weekday` field this way?

```
{
  "Id": 1235
}
```

Unmarshaling this body will not cause any error. Yet, the `weekday` field of the `Request` struct will be set to the zeroed value on an `int` type: 0. Thus, `Monday`, just like in the previous request.

How can we find a way to distinguish a request passing Monday from a request without a weekday? The problem is related to the way we declared the `Weekday` enum. Indeed, `Unknown` was the last constant of the enum. Hence its value is equal to 7.

To fix this issue, the best practice to handle an enum's unknown value is to set its value to 0 (the zeroed value of an int). Instead, we should have declared `Weekday` this way:

```
type Weekday int

const (
    Unknown Weekday = iota   ❶
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    Sunday
)
```

❶ `Unknown` is now equal to 0

If the JSON body's weekday value is empty, it will be unmarshaled as `Unknown`; which is what we need.

As a rule of thumb, an enum's unknown value should be set to the enum type's zeroed value. This way, we can make the distinction between explicit and missing values.

In the next section, we will discuss the usage of the `defer` keyword.

2.7 Not Using defer

In the following example, we will implement a function to copy a file from one destination to another. We will also manage the file descriptor closure as once a `*os.File` is opened for reading or writing, it must be closed using the `Close` method. Finally, at the end of the function, we will use the `Sync` method to flush the filesystem buffer to force writing on the disk, making the copy durable. Here is a possible implementation:

```
func CopyFile(srcName, dstName string) error {
    src, err := os.Open(srcName)  ❶
    if err != nil {
        return err
    }

    stat, err := src.Stat()
    if err != nil {
        src.Close()
        return err
    }

    if stat.IsDir() {  ❷
        src.Close()
        return fmt.Errorf("file %q is a directory", srcName)
    }

    dst, err := os.Create(dstName)  ❸
    if err != nil {
        src.Close()
        return err
    }

    _, err = io.Copy(dst, src)  ❹
    if err != nil {
        src.Close()
        dst.Close()
        return err
    }

    err = dst.Sync()  ❺
    src.Close()
    dst.Close()
    return err
}
```

❶ Open the source file

❷ Check if directory

❸ Create the destination file

❹ Copy the source to the destination

❺ Flush the filesystem buffer

NOTE **Closing an `*os.File` returns an error. However, the errors can be safely ignored in this example as we force flushing the filesystem buffer. Otherwise, we should at least log a message if an error occurs. We will see in the error management chapter how to gracefully handle errors in a deferred statement.**

This implementation works. We open the source file, check whether it's a directory, then handle the copy. However, we can notice some boilerplate code:

- `src.Close()` is repeated five times
- `dst.Close()` is repeated twice

Having to think about the possible paths where the source and destination files have to be closed

makes our code error-prone. Fortunately, Go provides a solution to this problem with the `defer` keyword as presented in figure 2.1:

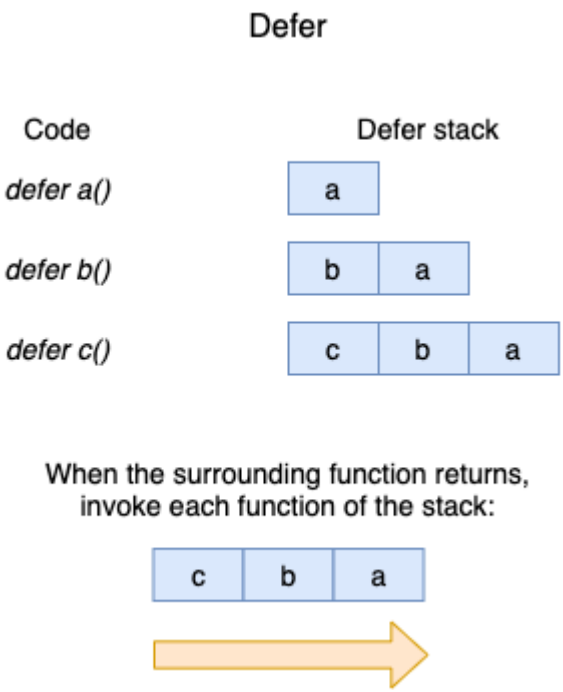

Figure 2.1 In the code, we call successively `defer a()`, `defer b()` and `defer c()`.

Calling `defer` schedules a function call when the surrounding function returns. The deferred function is guaranteed to be executed even if the surrounding function panics and terminates unexpectedly. Each deferred call is pushed onto a stack. When the surrounding function returns, the deferred calls are popped off the stack (Last-In-First-Out order). Here, it will call `c()`, then `b()`, and finally `a()`.

NOTE **A deferred call is not executed when the surrounding block exits but when the surrounding function returns:**

```
func main() {
    fmt.Print("a")
    if true {
        defer fmt.Print("b")
    }
    fmt.Print("c")
}
```

This code prints `acb`, not `abc`.

Let's come back to the `CopyFile` function and implement another version using the `defer` keyword:

```
func CopyFile(srcName, dstName string) error {
    src, err := os.Open(srcName)
    if err != nil {
        return err
    }
    defer src.Close()  ❶

    stat, err := src.Stat()
    if err != nil {
        return err
    }

    if stat.IsDir() {
        return fmt.Errorf("file %q is a directory", srcName)
    }

    dst, err := os.Create(dstName)
    if err != nil {
        return err
    }
    defer dst.Close()  ❷

    _, err = io.Copy(dst, src)
    if err != nil {
        return err
    }

    return dst.Sync()
}
```

❶ Defer the call to `src.Close()`

❷ Defer the call to `dst.Close()`

In this implementation, we removed the duplicated close calls thanks to `defer`. It makes the function more lightweight and easier to read. We don't have to recall closing `src` and `dst` at the end of each code path which is less error-prone.

A `defer` statement is often used with operations paired operations such as open/close, connect/disconnect, and lock/unlock to make sure all the resources are released in all cases. Here is another example with a `sync.Mutex`:

```
type Store struct {
    mutex sync.Mutex
    data  map[string]int
}

func (s *Store) Set(key string, value int) {
    s.mutex.Lock()  ❶
    defer s.mutex.Unlock()  ❷

    s.data[key] = value
}
```

❶ Mutex lock

❷ Unlock the mutex as a deferred statement

We lock the mutex using `s.mutex.Lock()` and call the unlock paired operation using `defer`

`s.mutex.Unlock()`.

NOTE **If we have to implement a pre and post-action such as a mutex lock/unlock that doesn't return any values, we can also implement it this way:**

we can also use `defer` this way:

```
func (s *Store) Set(key string, value int) {
    defer s.lockUnlock()()  ❶

    s.data[key] = value
}

func (s *Store) lockUnlock() func() {
    s.mutex.Lock()
    return func() {
        s.mutex.Unlock()
    }
}
```

❶ **Execute `s.lockUnlock()` right away but defer the execution of `s.lockUnlock()()`**

What we call as a deferred statement is not `s.lockUnlock()` but `s.lockUnlock()()`. Therefore, the `s.lockUnlock()` part is executed right away (`s.mutex.Lock`), but the closure returned will be executed as the deferred operation (`s.mutex.Unlock()`). It adds some syntactic sugar to handle pre/post actions in a function using a single line, which can sometimes be quite handy.

If using such a pattern, be also mindful that facing `s.lockUnlock()()` with two sets of parenthesis might be pretty confusing depending on the seniority of your team.

Let's also be aware of the possible implications while refactoring a code. For example, suppose we want to refactor a `main` function containing a deferred call into multiple functions. In that case, the `defer` statement will not be executed once the application completes but once the subfunction does:

```
// Before
func main() {
   consumer := createConsumer()
   defer consumer.Close()  ❶
   // ...
}

// After
func main() {
   consumer := handleConsumer()
   // ...
}

func handleConsumer() Consumer {
   consumer := createConsumer()
   defer consumer.Close()  ❷
   return consumer
}
```

❶ Executed once the application stops

❷ Executed once `handleConsumer` ends

Here, we introduced a bug by refactoring the creation of the `consumer` as `consumer.Close()` will be executed as soon as `handleConsumer` ends. It may look like a straightforward comment, but when we have to refactor a lot of code, it can sometimes be easy to miss the deferred statements.

Please also note that before Go 1.14, the deferred statement wasn't inlined. Inlining is a compiler optimization to substitute a function's body to save a function call's overhead. It was the reason why, in some projects where performance was a critical factor, the `defer` keyword was not always used. Yet, Since Go 1.14, deferred statements can be inlined (up to 15 inlined deferred calls per function).

In summary, `defer` allows to avoid boilerplate code and reducing the risks to forget actions such as resource closure, disconnection, mutex unlock, etc. In the next section, we will keep discussing `defer` and how arguments and receivers are evaluated.

2.8 Ignoring How defer Arguments and Receivers are Evaluated

We have seen that a `defer` statement defers a function's execution until the surrounding function returns. So far, we have only used `defer` to call functions without arguments. Yet, if a function with arguments is deferred, how are these arguments evaluated? We will delve into this question with two subsections to discuss arguments evaluation and using `defer` with a pointer or value receiver.

2.8.1 Arguments Evaluation

In the following example, we will implement a Uber-like application to find the best available driver for a passenger making a ride request. We will implement a `FindDrivers` function that receives a list of drivers, applies two filters, and returns a subset of drivers. Meanwhile, we will use two functions for monitoring purpose, `logStatus` and `incrementStatusCounter`, with one of these statuses:

- The first filter failed
- The second filter failed
- All the filters succeeded

To avoid repeating the calls to `logStatus` and `incrementStatusCounter`, we will use `defer`:

```
type Status int  ❶

const (
    StatusSuccess Status = iota
    StatusRadiusFilterError
    StatusActivityFilterError
)

func FindDrivers(drivers []Driver) ([]Driver, error) {
    var status Status
    defer logStatus(status)  ❷
    defer incrementStatusCounter(status)  ❸

    var err error
    drivers, err = applyRadiusFilter(drivers)
    if err != nil {
        status = StatusRadiusFilterError  ❹
        return nil, err
    }

    drivers, err = applyActivityFilter(drivers)
    if err != nil {
        status = StatusActivityFilterError  ❺
        return nil, err
    }

    status = StatusSuccess  ❻
    return drivers, nil
}
```

❶ Define a `Status` enum

❷ Defer the call to `logStatus`

❸ Defer the call to `incrementStatusCounter`

❹ Set status to radius filter error

❺ Set status to activity filter error

❻ Set status to success

First, we declare a `status` variable. This variable is passed to both `logStatus` and `incrementStatusCounter` functions. Throughout the function, depending on the possible

errors, we update the `status` variable.

If we give this function a try, regardless of what occurs, we will always call `logStatus` and `incrementStatusCounter` with the same status: `StatusSuccess` (0). How is it possible?

Something important to understand with `defer` is that the function's arguments are evaluated right away, not once the surrounding function returns. Why is it important?

In this example, we have called `logStatus(status)` and `incrementStatusCounter(status)` as deferred functions. Therefore, Go will schedule these calls with the current value of `status` at this stage. As `status` was initialized using `var status Status`, its current value is 0, hence `StatusSuccess`.

How can we solve this problem if we want to keep using `defer`? There are two main solutions.

The first solution is to pass to the deferred function a pointer. A pointer stores the memory address of another variable. Even though the pointer will be evaluated immediately, the variable it references may change throughout the function.

```
func FindDrivers(drivers []Driver) ([]Driver, error) {
    var status Status
    defer logStatus(&status)  ❶
    defer incrementStatusCounter(&status)  ❷

    var err error
    drivers, err = applyRadiusFilter(drivers)
    if err != nil {
        status = StatusRadiusFilterError
        return nil, err
    }

    drivers, err = applyActivityFilter(drivers)
    if err != nil {
        status = StatusActivityFilterError
        return nil, err
    }

    status = StatusSuccess
    return drivers, nil
}
```

❶ Call defer on `logStatus` with a `*Status` type

❷ Call defer on `incrementStatusCounter` with a `*Status` type

We modified `logStatus` and `incrementStatusCounter` to accept a `*Status`, so we also changed how we were calling these functions. The rest of the implementation remains the same. Because `status` is a pointer, when these two functions will be scheduled, it will be done with a pointer referencing the updated status.

However, as we said, this solution requires changing the signature of the two functions, which may not always be possible. There is a second possible solution: calling as a deferred statement a

closure. A closure is a function value that references variables from outside its body. For example:

```
func f() {
    s := "foo"
    go func() {
        fmt.Println(s)   ❶
    }()
}
```

❶ References the `s` variable outside of the function body

We have already mentioned that the arguments passed to a deferred function are evaluated right away. However, the variables referenced by the closure are evaluated during the closure execution (hence, when the surrounding function returns).

Here is an example to clarify how deferred closures are working:

```
func f() {
    i := 0
    j := 0
    defer func(i int) {   ❶
        fmt.Println(i, j)   ❷
    }(i)   ❸

    i++
    j++
}
```

❶ Call as a deferred function a closure that accepts an integer as an input

❷ `i` is the function input and `j` is an external variable

❸ Pass `i` to the closure (evaluated right away)

Here, the closure references two variables: `i` and `j`. `i` is passed as a function argument, so it's evaluated immediately. Conversely, `j` references a variable outside of the closure body, so it's evaluated when the closure is executed. If we run this example, it will print `0 1`.

Therefore, we can use a closure to implement a new version of `FindDrivers`:

```
func FindDrivers(drivers []Driver) ([]Driver, error) {
    var status Status
    defer func() {   ❶
        logStatus(status)   ❷
        incrementStatusCounter(status)   ❸
    }()   ❹

    var err error
    drivers, err = applyRadiusFilter(drivers)
    if err != nil {
        status = StatusRadiusFilterError
        return nil, err
    }

    drivers, err = applyActivityFilter(drivers)
    if err != nil {
        status = StatusActivityFilterError
        return nil, err
    }

    status = StatusSuccess
    return drivers, nil
}
```

❶ Call a closure as the deferred function

❷ Call `logStatus` within the closure by referencing the `status` variable

❸ Call `incrementStatusCounter` within the closure by referencing the `status` variable

❹ Empty arguments

We wrapped the calls to `logStatus` and `incrementStatusCounter` in a closure without arguments. This closure references the `status` variable, which is defined outside of its body. Therefore, we will call the two functions with the latest `status` value.

Now, what about using `defer` on a method with a pointer or value receiver? Let's see how things work.

2.8.2 Pointer and Value Receiver

While attaching a method to a receiver, the receiver itself can be a value or a pointer. As a short reminder, pointer receivers can modify the value to which the receiver points. Conversely, with a value receiver, a method operates on a copy of the original type.

When we use `defer` on a method, the same logic related to arguments evaluation applies. With a value receiver, the receiver is evaluated immediately:

```
func main() {
    s := Struct{id: "foo"}
    defer s.print()  ❶
    s.id = "bar"  ❷
}

type Struct struct {
    id string
}

func (s Struct) print() {
    fmt.Println(s.id)  ❸
}
```

❶ `s` is evaluated immediately

❷ Update `s.id` (not visible)

❸ foo

In this example, we call as a deferred statement the `print` method. This method has a value receiver, so `defer` will schedule the method's execution with a struct that contains an `id` field equal to `foo'. Therefore, this example prints `foo`.

Conversely, if the receiver is a pointer, the potential changes to the variable referenced by the pointer will be visible:

```
func main() {
    s := &Struct{id: "foo"}
    defer s.print()  ❶
    s.id = "bar"  ❷
}

type Struct struct {
    id string
}

func (s *Struct) print() {
    fmt.Println(s.id)  ❸
}
```

❶ `s` being a pointer, it is evaluated immediately but may reference another variable when the deferred method is executed

❷ Update `s.id` (visible)

❸ bar

The `s` pointer is also evaluated immediately while calling `defer`. However, this pointer references a struct that mutates before the surrounding function returns. Hence, this example prints `bar`.

In summary, we have to remind that calling `defer` on a function or method, the call's arguments are evaluated immediately. For a method, the receiver is also evaluated immediately. If we want to delay the evaluation, it can be done either using pointers or closures.

In the next section, we will keep using `defer` in resource closure as it's also a source of common mistakes in Go.

2.9 Not Closing Resources

It's frequent for developers to work with transient resources that must be closed at some point in the code to avoid leaks, for example, on disk or in memory. We have to know that all the structs implementing `io.Closer` have to be closed eventually.

Let's see one example where we will write a `getBody` function that will make an HTTP GET request and return the HTTP body response. Here is a first implementation:

```
func getBody(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }

    body, err := ioutil.ReadAll(resp.Body)   ❶
    if err != nil {
        return "", err
    }

    return string(body), nil
}
```

❶ Read `resp.Body` and get a body as a `[]byte`

We use `http.Get`, then we parse the response using `ioutil.ReadAll`. This function looks ok. At least, it correctly returns the HTTP response body. However, there is a resource leak. Let's understand where.

`resp` is an `*http.Response` type. It contains a `Body io.ReadCloser` field (`io.ReadCloser` containing both an `io.Reader` and an `io.Closer`). This body has to be closed if `http.Get` does not return an error; otherwise, it's a resource leak. It will keep some memory allocated, which is no longer needed but can't be collected by the GC and may prevent clients from reusing the TCP connection in the worst cases.

The most convenient way to deal with body closure is to handle it as a deferred statement this way:

```
func getBody(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }

    defer resp.Body.Close()  ❶

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }

    return string(body), nil
}
```

❶ If `http.Get` does not return an error, we will close the response body using `defer`

In this implementation, we properly handle the body resource closure as a deferred function that will be executed once `getBody` returns.

> **NOTE** **We should note that `resp.Body.Close()` returns an error. We will see in the error management chapter how to deal with errors in deferred function. For the time being, but also in the following examples, we will purposely ignore the error.**

We should note that the response body has to get closed regardless if we read it or not. For example, in the following function we just return the HTTP status code. However, the body must be closed as well:

```
func getStatusCode(url string) (int, error) {
    resp, err := http.Get(url)
    if err != nil {
        return 0, err
    }

    defer resp.Body.Close()  ❶

    return resp.StatusCode, nil
}
```

❶ Close response body even if we don't read it

We should also make sure to call the resource closure at the right moment. For example, if were deferring the call to `resp.Body.Close()` regardless of the error type:

```
func getStatusCode(url string) (int, error) {
    resp, err := http.Get(url)
    defer resp.Body.Close()  ❶
    if err != nil {
        return 0, err
    }

    return resp.StatusCode, nil
}
```

❶ At this stage, `resp` may be nil

As `resp` may be nil, this code may lead to an application panic:

```
panic: runtime error: invalid memory address or nil pointer dereference
```

One last thing to note related to HTTP request body closure. It's not that unfrequent to see implementations that closes the body if the response is not empty, not if the error is nil:

```
resp, err := http.Get(url)
if resp != nil {  ❶
    defer resp.Body.Close()  ❷
}

if err != nil {
    return "", err
}
```

❶ If response is not nil

❷ Close the response body as a deferred function

This implementation is not correct. It's based on the fact that in some conditions (e.g., redirection failure), `resp` and `err` will both be not nil. Yet, according to the official Go documentation:

> *On error, any Response can be ignored. A non-nil Response with a non-nil error only occurs when CheckRedirect fails, and even then, the returned Response.Body is already closed.*

Therefore, the `if resp != nil {}` check is not necessary. We should stick with the initial solution that closes the body in a deferred function only if there is no error.

NOTE **On server-side, while implementing an HTTP handler, it isn't required to close the request body as it's done automatically by the server.**

Closing a resource to avoid leaks is not solely related to HTTP body management. In general, all the struct implementing the `io.Closer` interface should be closed at some point. This interface contains a single `Close` method:

```
type Closer interface {
    Close() error
}
```

Let's see other cases where a resource has to be closed to avoid leaks.

2.9.1 `sql.Rows`

`sql.Rows` is a struct used as a result of an SQL query. As this struct implements `io.Closer`, it has to be closed. We can also handle the closure as a deferred function like this:

```
db, err := sql.Open("postgres", dataSourceName)   ❶
if err != nil {
    return err
}

rows, err := db.Query("SELECT * FROM MYTABLE")   ❷
if err != nil {
    return err
}

defer rows.Close()   ❸

// Use rows
```

❶ Create an SQL connection

❷ Perform an SQL query

❸ Close the rows

Following the `Query` call, if it doesn't return an error, we should close `rows` eventually.

2.9.2 `os.File`

`os.File` represents an open file descriptor. In the same way than `sql.Rows`, it has to be closed eventually:

```
f, err := os.Open("events.log")   ❶
if err != nil {
    return err
}

defer f.Close()   ❷

// Use file descriptor
```

❶ Open file

❷ Close the file descriptor

Again, we use `defer` to schedule the call to the `Close` method when the surrounding function returns.

> **NOTE** **Having a success while closing a writable `os.File` does not guarantee the file to be written on disk. Indeed, the write can still live in a buffer on the filesystem and not be flushed on disk. If durability is a critical factor, we should use the `Sync()` method.**

2.9.3 Compression implementations

A couple of compression implementations also have to be closed. Indeed, they create internal buffers that should also be released manually. For example: `gzip.Writer`.

```
var b bytes.Buffer   ❶
w := gzip.NewWriter(&b)   ❷

defer w.Close()   ❸
```

❶ Creates a buffer

❷ Creates a new gzip writer

❸ Close the `gzip.Writer`

The same logic applies for a `gzip.Reader`:

```
var b bytes.Buffer   ❶
r, err := gzip.NewReader(&b)   ❷
if err != nil {
    return nil, err
}

defer r.Close()   ❸
```

❶ Creates a buffer

❷ Creates a new gzip writer

❸ Close the `gzip.Writer`

To summarize this section, we've seen how important it is to close ephemeral resources to not lead to leaks. Ephemeral resources have to be closed at the right time and in specific situations. Sometimes, it's not always clear upfront about what has to be closed. We can only acquire this knowledge by a careful read of the API documentation and/or with experience. Yet, we should bear in mind that if a struct implements the `io.Closer` interface, we have to call the `Close` method eventually.

Let's now see a common confusion related to octal literals.

2.10 Creating Confusion with Octal Literals

What do you believe should be the output of the following code?

```
sum := 100 + 010
fmt.Println(sum)
```

At first sight, we may expect this code to print the result of 100 + 10: 110. Yet, it will print 108 instead. How is it possible?

In Go, we have to know that a number starting with 0 is considered an octal integer (base 8). 10 in base 8 equals 8 in base 10. Thus, the sum is equal to 100 + 8 = 108. It's an important property of integers to keep in mind to avoid confusion while reading existing code.

Octal integers are useful on different occasions. For instance, if we want to open a file using `os.OpenFile`. This function requires passing permission as an `uint32`. If we want to match the Linux permissions, we can pass an octal number for readability instead of a base 10 number:

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0644)
```

Also, something to note is that we can add an `o` character (the letter in lowercase) following the zero:

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0o644)
```

Using `0o` instead of only `0` doesn't change anything. Yet, it makes the use of an octal integer even more explicit readability-wise.

> **NOTE** **We can also use an uppercase `O` character instead of a lowercase `o`, but passing `0O644` can increase the confusion instead of being helpful.**

We should also note the other integer literals representation:

- Binary: `0b` or `0B` prefix. For example, `0b100` is equal to 4 in base 10.
- Hexadecimal: `0x` or `0X` prefix. For example, `0xF` is equal to 15 in base 10.
- Imaginary: `i` suffix. For example, `3i`.

Finally, for readability reasons, we can also use an underscore character as a separator. For example, we can write 1 billion this way: `1_000_000_000`. And we should also note that we can use the underscore operator with non-decimal integers (e.g. `0b00_00_01`).

In summary, Go handles binary, hexadecimal, imaginary, and octal numbers. Octal numbers start by 0. However, to improve readability and potential mistakes with future code readers, we should make octal numbers explicit using an `0o` prefix.

In the next section, we will dig into integers and discuss how overflows are handled in Go.

2.11 Neglecting Integer Overflows

In this chapter, we will delve into how integers overflows are managed. But first, let's remind us of a couple of things related to integers.

Go provides a total of ten integer types. There are four signed integer types:

- `int8`: 8 bits
- `int16`: 16 bits
- `int32`: 32 bits
- `int64`: 64 bits

Four unsigned integer types:

- * `uint8`: 8 bits
- * `uint16`: 16 bits
- * `uint32`: 32 bits
- * `uint64`: 64 bits

Furthermore, two integer types are the most commonly used: * `int` * `uint`

These two types have a size that depends on the system the application is executed: 32 bits on 32-bit systems or 64 bits on 64-bit systems (way more frequent nowadays).

Let's use an `int32` that we will initialize to its maximum value and increment it. What should be the behavior of this code?

```
var counter int32 = math.MaxInt32
counter++
fmt.Printf("counter=%d\n", counter)
```

This code compiles and runs without leading to any panic. However, the `counter++` statement will generate a so-called integer overflow:

```
counter=-2147483648
```

An integer overflow occurs when an arithmetic operation creates a value outside of the range that can be represented with a given number of bytes.

An `int32` is represented using 32 bits. Here is the binary representation of the maximum `int32` value (`math.MaxInt32`):

```
01111111111111111111111111111111
 |------31 bits set to 1-------|
```

As an `int32` is a signed integer, the bit on the left represents the integer's sign: 0 for positive, 1 for negative. If we increment this integer, there is no space left to represent the new value. Hence, it will lead to an integer overflow. Binary-wise, here is the new value:

```
10000000000000000000000000000000
 |------31 bits set to 0-------|
```

As we can notice, the bit sign is now equal to 1, meaning negative. This value is the smallest possible value for a signed integer represented with 32 bits.

> **NOTE** The smallest possible negative value is not `10000000000000000000000000000000`. Indeed, most systems rely on the two's complement operation to represent binary numbers. The main goal of this operation is to make x - x equals to zero regardless of x.

In Go, an integer overflow that can be detected at compile time will generate a compilation error. For example:

```
var counter int32 = math.MaxInt32 + 1
```

This code generates the following compilation error:

```
constant 2147483648 overflows int32
```

However, at runtime, an integer overflow is silent; it will not lead to an application panic. It is essential to have this behavior in mind. It may lead to sneaky bugs—for example, an integer increment or addition of positive integers that leads to a negative result.

Before delving into how to detect integer overflow with common operations, let's think about when we should worry about it. In most contexts, like handling a counter of requests or basic additions/multiplications, we shouldn't worry too much, or at least we should make sure to use the correct integer type. Yet, in some specific cases, such as memory-constrained projects using small integer types, having to deal with large numbers, or when doing conversions, we may want to check possible overflows. For example, the Ariane 5 launch failure is due to an overflow resulting from converting a 64-bit floating-point to a 16-bit signed integer.

2.11.1 Detect Integer Overflow During Increment

If we want to detect an integer overflow during an increment operation with a type based on a defined size (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`), it can be achieved by checking the value against the `math` constants. For example, with an `int32`:

```
func Inc32(counter int32) int32 {
    if counter == math.MaxInt32 {   ❶
        panic("int32 overflow")
    }
    return counter + 1
}
```

❶ Compare with `math.MaxInt32`

In this function, we check whether the input is already equal to `math.MaxInt32`.

However, if an integer type doesn't have a defined size (`int` or `uint`), we can't use any standard constants. Therefore, if we want to work with `int` or `uint`, we have to handle the logic

manually.

One possible implementation is to do it by checking the sign after the increment operation:

```
func IncInt(counter int) int {
    if counter <= 0 {
        return counter + 1
    }

    counter++   ❶
    if counter < 0 {   ❷
        panic("int overflow")
    }
    return counter
}
```

❶ Increment counter

❷ Check sign

We increment the counter; then we check whether the sign has changed. However, This implementation is not optimized, as it requires multiple checks. We can improve the execution time of this function using binary operators. Go provides the following bitwise binary operators:

- `&`: bitwise AND
- `|`: bitwise OR
- `^`: bitwise XOR
- `&^`: bitwise clear (AND NOT)
- `<<`: left shift
- `>>`: right shift

Let's first tackle unsigned integers. If we want to compute our own `maxUint` constant, we can use the bitwise XOR operator, which inverts every bit of an integer. This constant can be computed with `^uint(0)`. Here is the bit representation on a 32-bit system for readability concern:

```
uint(0):  00000000000000000000000000000000
^uint(0): 11111111111111111111111111111111
```

Using this constant, we can now implement a more optimized version like this:

```
const maxUint = ^uint(0)   ❶

func IncUint(counter uint) uint {
    if counter == maxUint {   ❷
        panic("uint overflow")
    }
    return counter + 1
}
```

❶ Create `maxUint` constant

❷ Compare against `maxUint` constant

Instead of multiple checks, we just check the provided value against our own `maxUint` constant.

Now, what about implementing the same logic for incrementing an `int` type? The value of `maxInt` is the same as `maxUint` except the bit on the left should be 0 instead of 1. Using the right shift operator, we can shift the bits of `maxUint` on the right with `int(maxUint >> 1)`:

```
maxUint:          11111111111111111111111111111111
int(maxUint >> 1): 01111111111111111111111111111111
```

Then, here is an optimized `IncInt` implementation:

```
const (
    maxUint = ^uint(0)
    maxInt  = int(maxUint >> 1)   ❶
)

func IncInt(counter int) int {
    if counter == maxInt {   ❷
        panic("integer overflow")
    }
    return counter + 1
}
```

❶ Create `maxInt` constant

❷ Compare against `maxInt` constant

Again, we check the provided value against our own `maxInt` constant.

We have seen how to check integer overflows following an increment operation. Now, what about an addition?

2.11.2 Detect Integer Overflow During Addition

With additions, we can do it by reusing the `maxInt` constant:

```
const (
    maxUint = ^uint(0)
    maxInt  = int(maxUint >> 1)
)

func addInt(a, b int) int {
    if a > maxInt-b {   ❶
        panic("int overflow")
    }

    return a + b
}
```

❶ Check if integer overflow will occur

`a` and `b` being the two operands; if `a` is greater than `maxInt - b`, it means the operation will lead to an integer overflow.

Now, let's see with the latest operation: multiplication.

2.11.3 Detect Integer Overflow During Addition

Multiplications are a bit more complex to handle. We have to perform checks against the minimal integer. Again, as there is no such constant in Go, let's create it. `minInt` can be built by inverting every bit of the previously `maxInt` constant defined, meaning with `^maxInt`:

```
maxInt:  01111111111111111111111111111111
^maxInt: 10000000000000000000000000000000
```

The implementation is the following:

```
const (
    maxUint = ^uint(0)
    maxInt  = int(maxUint >> 1)
    minInt  = ^maxInt   ❶
)

func multiplyInt(a, b int) int {
    result := a * b
    if a == 0 || b == 0 || a == 1 || b == 1 {   ❷
        return result
    }
    if a == minInt || b == minInt {   ❸
        panic("integer overflow")
    }
    if result/b != a {   ❹
        panic("integer overflow")
    }
    return result
}
```

❶ Create `minInt` constant

❷ Check if one of the operands is equal to zero or one

❸ Check if one of the operands is equal to `minInt`

❹ Check if the multiplication will lead to an integer overflow

Checking an integer overflow with a multiplication requires multiple steps. First, testing if one of the operands is equal to zero, one or `minInt`. Then we divide the multiplication result by `b`. If the result is not equal to the original factor (`a`), it means an integer overflow occurred.

In summary, an integer overflow is a silent operation in Go. If we want to check whether an operation leads to an overflow to avoid sneaky errors, we can use the utility functions described in this section. Let's also remind that Go provides a package to deal with large numbers: `math/big`. It might also be an option if an `int` isn't enough.

In the next section, we will discuss an important aspect of Go: linters.

2.12 Not Using Linters

A linter is an automatic tool in charge of analyzing our code and catch errors. The scope of this section is not to give an exhaustive list of the existing linters; otherwise, it would become deprecated pretty quickly. Yet, it's more to remind us that there are many great Go linters out there to cover various use cases, and we should rely on them.

For example, in a previous section, we discussed potential errors related to variable shadowing. Using `vet`, a standard linter from the Go toolset, and `shadow` we can detect shadowed variables:

Let's see a quick example with the following code:

Listing 2.2 Shadowed variable example

```
package main

import "fmt"

func main() {
    i := 0
    if true {
        i := 1   ❶
        fmt.Println(i)
    }
    fmt.Println(i)
}
```

❶ Shadowed variable

`vet` being included alongside the Go binary, let's first install `shadow`, link it with Go `vet`, and then run it on the previous example.

```
$ go install golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow   ❶
go: finding golang.org/x/tools latest
go: downloading golang.org/x/tools v0.0.0-20201218024724-ae774e9781d2
go: extracting golang.org/x/tools v0.0.0-20201218024724-ae774e9781d2
$ go vet -vettool=$(which shadow)   ❷
./main.go:8:3: declaration of "i" shadows declaration at line 6   ❸
```

❶ shadow installation

❷ Linking to Go vet using the `vettol` argument

❸ Go vet was able to detect the shadow variable

The Go `vet` linters could detect that variable `i` shadows an existing declaration.

In general, we should rely as much as possible on linters. Indeed, it provides Go developers a powerful and automated way to detect various problems in our code.

This section's goal is not to list all the available linters. However, if you're not a regular user of linters, here is a possible list that you may want to use daily:

- vet: standard Go analyzer
- golint: standard Go style mistakes checker
- kisielk/errcheck: errors checker
- fzipp/gocyclo: cyclomatic complexity analyzer
- goconst: repeated string constants analyzer

Besides linters, we should also use code formatters to fix code style:

- gofmt: standard Go code formatter
- goimports: standard Go imports formatter

Meanwhile, we can also use golangci-lint. `golangci-lint` is an linting tool that provides a facade on top of many useful linters and formatters. Also, it allows running the linters in parallel to improve analysis speed.

Linters and formatters are a powerful way to improve the quality and the consistency of our codebase. Let's make sure to use them.

2.13 Summary

- Avoiding shadowed variables can help keep us from making mistakes like referencing the wrong variable or confusing readers.
- Understanding the possible side effects of logging libraries is important to avoid unintended consequences.
- To compare types in Go, we can use the `==` and `!=` operators if two types are comparable: boolean, numerics, strings, pointers, channels, and structs composed of comparable types solely. Otherwise, we have to use `reflect.DeepEqual` and pay the price of reflection or use custom implementations/libraries.
- To marshal/unmarshal a struct using the standard JSON library, the fields have to be exported. It isn't required to set the JSON tags.
- We should remain careful about using embedded fields in Go structs. It may lead to sneaky bugs like the one we saw with an embedded `time.Time` field implementing the `json.Marshaler` interface, hence overriding the default marshaling behavior.
- To compare two `time.Time` structs, we have to recall that `time.Time` contains both a wall and monotonic clock and that the comparison using the `==` operator is done on both clocks.
- To format a network address compatible with IPv4 and IPv6, we should rely on the `net.JoinHostPort` function.
- Defining specific enum types, using `iota`, and assigning the unknown value to the enum's type zeroed value are best practices in Go.
- `defer` is a powerful and convenient solution to avoid boilerplate code and reduce the risks of forgetting about a closure action, for example.
- When using `defer`, we have to recall that the call's arguments (and the receiver if in the context of a method) are evaluated immediately.
- We should eventually close all structs implementing `io.Closer` to avoid possible leaks.
- When reading an existing code, we should bear in mind that integers starting by 0 are octal numbers. Also, to improve readability, we should make octal integers explicit by prefixing them with `0o`.
- As integer overflows are handled silently in Go, we can implement our own functions to catch them.
- We should rely on existing linters to catch important issues and possible bugs.

3 Code organization

This chapter covers

- Avoiding nested code
- Misusing init functions
- Forcing to use getters/setters
- Avoiding code pollution with overusing interfaces
- Designing interfaces
- Avoiding as much as possible the `interface{}` type
- Using options pattern
- Organizing a Go project
- Avoiding common packages
- Preventing package name collision
- Code documentation

Organizing a Go code in an idiomatic, clean, and expressive way is not an easy task. It requires experience and mistakes to understand all the best practices on organizing a project, dealing with packages, using interfaces in the best possible way, and more. In this chapter, we will delve into common mistakes related to code organization.

3.1 Writing Nested Code

A mental model is an internal representation of a system's behavior. While programming, we have to keep maintaining mental models, for example, about overall code interactions and function implementations. A code is qualified as readable based on multiple criteria such as naming, consistency, formatting, etc. A readable code will require less cognitive effort to maintain a mental model; hence it will be easier to maintain.

A critical aspect of readability is the number of nested levels. Let's do an exercise. Suppose that

you work on a new project and have to understand the following function `join` function:

```
func join(s1, s2 string, max int) (string, error) {
    if s1 == "" {
        return "", errors.New("s1 is empty")
    } else {
        if s2 == "" {
            return "", errors.New("s2 is empty")
        } else {
            concat, err := concatenate(s1, s2)   (1)
            if err != nil {
                return "", err
            } else {
                if len(concat) > max {
                    return concat[:max], nil
                } else {
                    return concat, nil
                }
            }
        }
    }
}

func concatenate(s1 string, s2 string) (string, error) {
    // ...
}
```

(1) We call a `concatenate` function that will perform some specific concatenation work but may return some errors

This function concatenates two strings and returns a substring if the length is greater than `max`. Meanwhile, it handles checks on `s1`, `s2` and if the call to `concatenate` returns an error. From an implementation perspective, this function is correct. However, building a mental model encompassing all the different cases is probably not a straightforward task. Why? Because of the number of nested levels.

Now, let's try doing this exercise again with the same function, implemented differently:

```
func join(s1, s2 string, max int) (string, error) {
    if s1 == "" {
        return "", errors.New("s1 is empty")
    }
    if s2 == "" {
        return "", errors.New("s2 is empty")
    }
    concat, err := concatenate(s1, s2)
    if err != nil {
        return "", err
    }
    if len(concat) > max {
        return concat[:max], nil
    }
    return concat, nil
}

func concatenate(s1 string, s2 string) (string, error) {
    // ...
}
```

You should have probably noticed that building a mental model of this new version requires less

cognitive load, despite doing the same job as before. Yet, here we maintain only two nested levels. As mentioned by Mat Ryier, the author of the Go Time podcast:

> *Align the happy path to the left; you should quickly be able to scan down one column to see the expected execution flow*

It was difficult to distinguish the expected execution flow in the first version because of the nested `if/else` statements. Conversely, the second version simply requires scanning down one column to see the expected execution flow and the second column to see how the edge cases are handled as shown in figure 3.1:

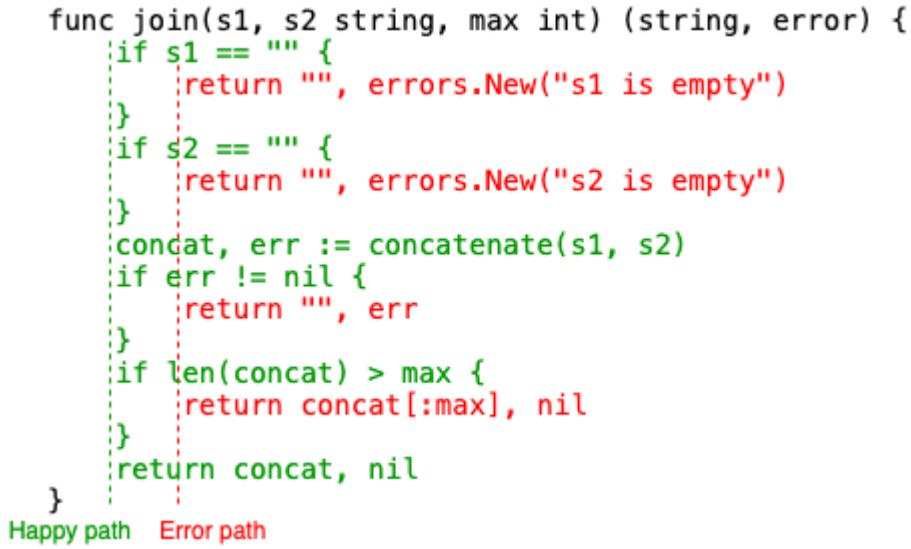

Figure 3.1 To understand the expected execution flow, we just have to scan the happy path column.

In general, the more nested levels a function requires, the more complex it is to read and understand. Let's see different applications of this rule to optimize our code for readability.

3.1.1 Condition Return

When an `if` block returns in all cases, we should omit the `else` block. We shouldn't write this:

```
if something() {
    return true
} else {
    // ...
}
```

But instead, omit the `else` block like this:

```
if something() {
    return true
}
// ...
```

With this new version, the code that was living previously in the `else` block is moved to the top-level, making it easier to read.

We can also follow this logic with an `else` block. Instead of an `if/else` where the `return` statement is in the `else`:

```
if s != "" {
    // ...
} else {
    return errors.New("empty string")
}
```

We should flip the `if` statement and return as early as possible:

```
if s == "" {   ❶
    return errors.New("empty string")
}
// ...
```

❶ Flip the `if` condition from `if s != ""` to `if s == ""`

In this new version, the code that was previously in the `if` block is again to the top level, again making it easier to read.

Now, let's apply this logic to errors.

3.1.2 Errors

The same logic applies to handle errors. To improve readability, the happy path should be aligned to the left. For example, we shouldn't write this:

```
value, err := something()
if err == nil {
    // ...
} else {
    return err
}
```

Instead, we should switch the `if` condition this way:

```
value, err := something()
if err != nil {
    return err
}
// ...
```

The new version is easier to read as it kept the happy path on the left edge and reduced the number of blocks.

Writing readable code is an important challenge for every developer. Striving in reducing the number of nested blocks, aligning the happy path on the left, and returning as early as possible are concrete means to improve a code's readability.

In the next section, we will discuss a common misuse in Go projects: init functions.

3.2 Misusing Init Functions

`init` functions are sometimes misused in Go applications. The potential consequences are poor error management or a code flow harder to understand. Let's refresh our minds about what is an init function. Then, we will see when it is or not recommended to use them.

3.2.1 Concepts

An init function is a function taking no arguments and returning no result (a `func()` function). When a package is initialized, all the constants and variables declarations in the package are evaluated. Then, the init functions are executed. Here is an example of a `main` package:

Listing 3.1 Package initialization

```
package main

import "fmt"

var a = func() int {
    fmt.Println("var")   ❶
    return 0
}()

func init() {
    fmt.Println("init")   ❷
}

func main() {
    fmt.Println("main")   ❸
}
```

❶ Executed first

❷ Executed second

❸ Executed last

Running this example would print the fo ut:

```
var
init
main
```

An init function is executed when a package is initialized. In the following example, we will define two packages, `main` and `redis` with `main` depending on `redis`:

Listing 3.2 main/main.go

```
package main

import (
    "fmt"

    "redis"
)

func init() {
    // ...
}

func main() {
    err := redis.Store("foo", "bar")   ❶
    // ...
}
```

❶ Dependency on the `redis` package

Listing 3.3 redis/redis.go

```
package redis

import ...

func init() {
    // ...
}

func Store(key, value string) error {
    // ...
}
```

As `main` depends on `redis', the `redis` package's init function will be executed first, followed by the init of the `main` package, then the `main` function itself, as shown in figure 3.2:

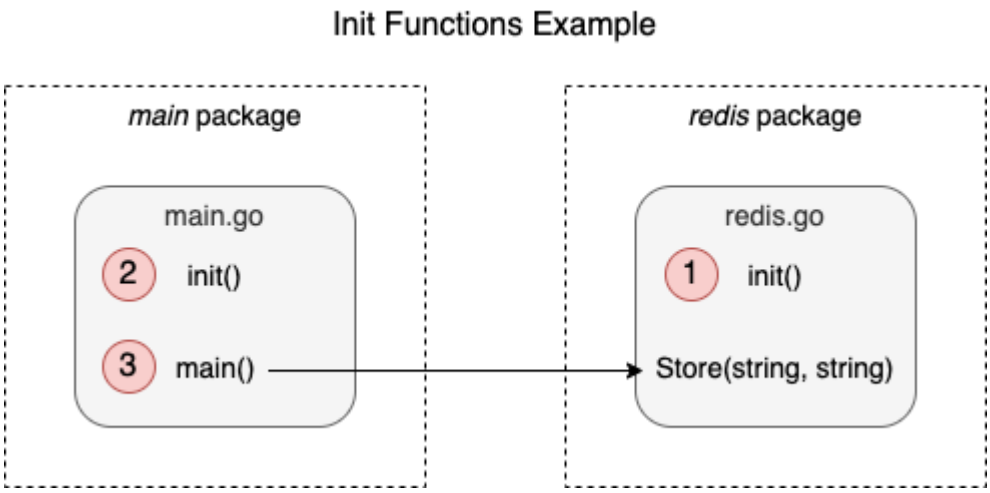

Figure 3.2 The init function of the `redis` package is executed first, then the init function of `main`, then the `main` function.

We can define multiple init functions per package. In this case, the execution order of the init

function inside a package is based on the source files' alphabetical order. For example, if a package contains an `a.go` and a `b.go` file and both have an init function, the `a.go` init function will be executed first. We shouldn't rely on the ordering of init functions within a package. Indeed, it can be dangerous as source files can be renamed, potentially impacting the execution order.

We can also define multiple init functions within the same source file. For example, this code is perfectly valid:

Listing 3.4 Multiple init functions per source file

```
package main

import "fmt"

func init() {   ❶
    fmt.Println("init 1")
}

func init() {   ❷
    fmt.Println("init 2")
}

func main() {
}
```

❶ First init function

❷ Second init function

The first init function executed is the first one we declared:

```
init 1
init 2
```

We can also use `init` functions for side effects. In the next example, we will define a `main` package that doesn't have a strong dependency on `foo` (e.g., no direct use of a public function). However, it requires the `foo` package to be initialized. We can do it using the _ operator this way:

Listing 3.5 main/main.go

```
package main

import (
    "fmt"

    _ "foo"   ❶
)

func main() {
    // ...
}
```

❶ Imports `foo` for side effects

In this case, the `foo` package will be initialized before `main`. Hence, the init functions of `foo` will be executed.

Another aspect of an `init` function is that it can't be invoked directly:

Listing 3.6 An invalid example of an init function invoked from another function

```
package main

func init() {}

func main() {
    init()  ❶
}
```

❶ Invalid reference

This code produces the following compilation error:

```
$ go build .
./main.go:6:2: undefined: init
```

Now that we refreshed our minds about how init functions work let's see when we should use or not use them.

3.2.2 When to Use Init Functions?

In the next example, we will create an SQL connection. We will use an init function and make the connection available as a global variable that we will later reuse.

```
var connection *sql.DB

func init() {
    dataSourceName := os.Getenv("MYSQL_DATA_SOURCE_NAME")   ❶
    c, err := sql.Open("mysql", dataSourceName)
    if err != nil {
        log.Panic(err)
    }
    err = connection.Ping()
    if err != nil {
        log.Panic(err)
    }
    connection = c   ❷
}
```

❶ Environement variable

❷ Assign the DB connection to the global `connection` variable

What should we think about our implementation? Let's describe three main downsides.

First, error management in an init function is somewhat limited. Indeed, an init function does not return any error, so we can decide to panic if something goes wrong. If a panic occurs in an init

function, it's impossible to recover from it, and the application will be stopped. In our example, it might be ok to stop the application anyway if creating the connection is an absolute necessity. However, it shouldn't be necessarily up to the package itself to decide whether to stop the application. Perhaps, a caller might have preferred to implement a retry or use a fallback mechanism. Packing in an init function prevents client packages from implementing their error handling logic.

Another important downside is related to testing. If we add tests to this file, the init function will be executed before running the test cases, which isn't necessarily what we want. For example, we may have wanted to add unit tests on a mapping function that doesn't require this connection to be created. Therefore, it would complicate to way to write unit tests.

The last downside is that our approach to create the connection requires to use a global variable. Global variables have some severe drawbacks, for example:

- They can be altered by any functions within the package
- It can also make unit tests more complicated as a function that depends on a shared global state is not pure

In most cases, we should favor encapsulating a variable rather than keeping it global.

These are the main downsides related to init functions. However, shall we avoid them at all costs? Not really. There are still use cases where init functions can be pretty useful. For example, the official Go blog (which is implemented in Go) uses init function to set up the static HTTP configuration:

```
func init() {
    redirect := func(w http.ResponseWriter, r *http.Request) {
        http.Redirect(w, r, "/", http.StatusFound)
    }
    http.HandleFunc("/blog", redirect)
    http.HandleFunc("/blog/", redirect)

    static := http.FileServer(http.Dir("static"))
    http.Handle("/favicon.ico", static)
    http.Handle("/fonts.css", static)
    http.Handle("/fonts/", static)

    http.Handle("/lib/godoc/", http.StripPrefix("/lib/godoc/",
            http.HandlerFunc(staticHandler)))
}
```

In this example, the init function cannot fail (`http.HandleFunc` can panic, but only if the handler is nil, which is not the case here), there is no need to create any global variable, and it will not impact possible unit tests. Therefore, this example is a good one where init functions can be useful.

In summary, we have seen that init functions may lead to some difficulties:

- Error management is limited

- It may complicate how to implement tests (e.g., external dependency to be set up, which may not be necessary for the scope of unit tests)
- If the initialization requires to set a state, it has to be done through global variables

We should be cautious with `init` functions. They can be helpful in some situations, such as defining static configuration; in most cases, we should handle initializations as ad-hoc functions, making the code flow more explicit.

3.3 Always Using Getters and Setters

In programming, data encapsulation refers to hiding the values or state of an object. Getters and setters are means to enable encapsulation by providing exported methods on top of unexported object fields.

In Go, there is no automatic support, as we can see in some languages. It is also neither considered mandatory nor idiomatic to use getters and setters to access struct fields. For example, the standard library implements structs where some fields are accessible directly, such as the `time.Timer` struct:

```
timer := time.NewTimer(time.Second)
<-timer.C  ❶
```

❶ `C` is a `chan Time` field

Because it's exported, we could even modify it directly like this:

```
timer := time.NewTimer(time.Second)
timer.C = make(chan time.Time)  ❶
<-timer.C
```

❶ Modify the `C` field of `timer`

Of course, we shouldn't do it. However, this example illustrates that even the standard library does not enforce using getters and/or setters.

On the other hand, we shouldn't forbid getters and setters. There are cases where they can be considered as appropriate. For example, a `Customer` struct holding a balance field. We may favor exposing a behavior through a getter and a setter method instead of exposing the field directly. If we do, the naming convention in Go we should follow with a field called `balance` is the following:

- The getter method should be named `Balance`
- The setter method should be named `SetBalance`

```
currentBalance := customer.Balance()   ❶
if currentBalance < 0 {
    customer.SetBalance(0)   ❷
}
```

❶ Getter

❷ Setter

In summary, We shouldn't overwhelm our code with getters and setters methods on structs if they don't bring any value. We should be pragmatic and strive to find the right balance between efficiency and following idioms that are sometimes considered indisputable in other languages. Let's bear in mind that Go is a unique language and was designed to be simple. Being an effective Go programmer also means trying to find the most straightforward possible solutions and designs.

3.4 Interface Pollution

Interface pollution is about overwhelming our code with unnecessary abstractions making it harder to understand and evolve. It's a common mistake made by developers coming from another language with different habits. Before delving into the topic, let's refresh our minds about what an interface is in Go. Then, we will see when it's considered appropriate to use interfaces and when it's not.

3.4.1 Go Interface Concepts

An interface provides a way to specify the behavior of an object. Interfaces are used to create common abstractions that can be implemented by multiple objects. What makes Go's interface so different is that they are satisfied implicitly. There is no explicit keyword like `implements` for example, to mark that an object X implements interface Y.

To understand what makes interfaces so powerful, we will dig into two popular ones from the standard library: `io.Reader` and `io.Writer`.

The `io` package provides abstractions for I/O primitives. Among these abstractions, `io.Reader` relates to reading data from a data source and `io.Writer` to writing data to a target as represented in figure 3.3:

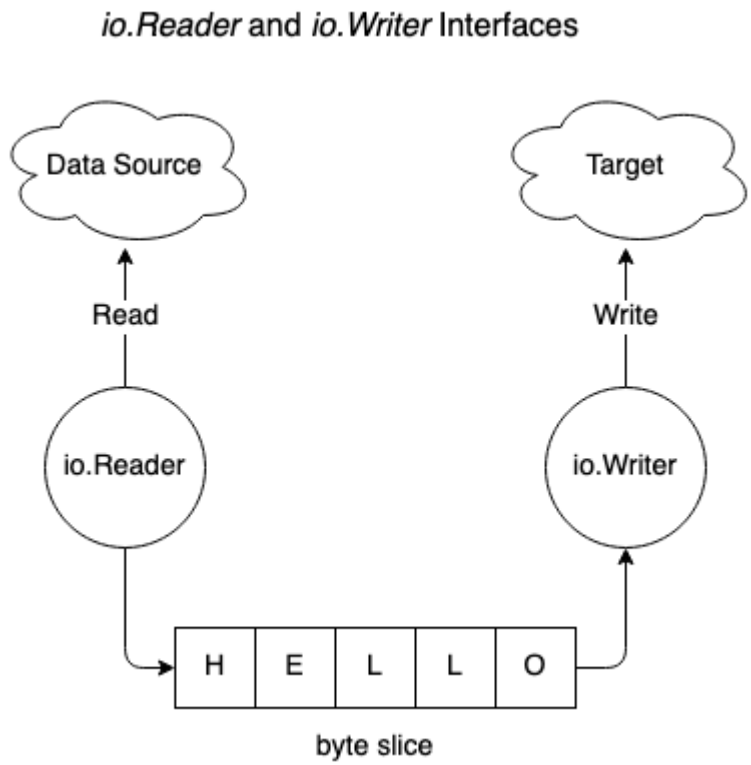

Figure 3.3 `io.Reader` reads from a data source and fills a byte slice, whereas `io.Writer` writes to a target from a byte slice.

The `io.Reader` contains a single `Read` method:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Custom implementations of the `io.Reader` interface should accept a slice of bytes, fill it with its data and return either the number of bytes read or an error.

Let's implement `io.Reader` to see how interfaces are implemented with a dummy `stringReader` struct.

```
type stringReader struct {
    s string
}

func (r stringReader) Read(p []byte) (n int, err error) {   ❶
    copy(p, r.s)
    return len(r.s), nil
}

func main() {
    reader := stringReader{s: "foo"}
    p := make([]byte, 3)
    n, _ := reader.Read(p)
    fmt.Println(n, string(p))   ❷
}
```

❶ Implementation of the `Read` method

```
❷ 3 foo
```

`stringReader` is an `io.Reader` as it contains a `Read` method with the same signature. As we mentioned, implementing an interface is done implicitly.

On the other hand, `io.Writer` defines a single method: `Write`.

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Custom implementations of `io.Writer` should write to a target the data coming from a slice and return either the number of bytes written or an error.

Therefore, both interfaces provide fundamental abstractions:

- `io.Reader` to read data from a source
- `io.Writer` to write data to a target

What is the rationale for having these two interfaces in the language? What is the point of creating these abstractions?

Let's assume we have to implement a function that should copy one file content to another. We could decide to create a specific function that would take as input two `*os.File`. Or, we can decide to create a more generic function using `io.Reader` and `io.Writer` abstractions:

```
func copySourceToDest(source io.Reader, dest io.Writer) error {
    // ...
}
```

This function would work with `*os.File` parameters (as `*os.File` implements both `io.Reader` and `io.Writer`) but also with many other types from the standard library as both abstractions are implemented by multiple Go types. Also regarding testing, instead of having to create temporary files which can be a little cumbersome, we could pass as a reader a `*strings.Reader` created from a string and a `*bytes.Buffer` as a writer this way:

```
func TestCopySourceToDest(t *testing.T) {
    const input = "foo"
    source := strings.NewReader(input)   ❶
    dest := bytes.NewBuffer(make([]byte, len(input)))   ❷

    err := copySourceToDest(source, dest)   ❸
    if err != nil {
        t.FailNow()
    }

    got := dest.String()
    if got != input {
        t.Errorf("expected: %s, got: %s", input, got)
    }
}
```

1. Create an `io.Reader`
2. Create an `io.Writer`
3. Call `copySourceToDest` from a `*strings.Reader` and a `*bytes.Buffer`

Therefore, having a generic function that accepts abstractions instead of concrete type simplifies writing unit tests.

Moreover, while designing interfaces, something to keep in mind is its granularity (how many methods does the interface contain). A known proverb in Go relates to how big an interface should be:

The bigger the interface, the weaker the abstraction

Indeed, adding methods to an interface can decrease its benefits in terms of reusability. `io.Reader` and `io.Writer` are powerful abstractions because they cannot get simpler. We should keep this principle in mind when designing interfaces.

In summary, interfaces can lead to creating powerful abstractions. Abstractions can help in many ways. For example, code decoupling, improving functions reusability, and facilitating unit tests. However, just like many software engineering areas, abusing a concept can lead to drawbacks.

3.4.2 When to Use Interfaces

When should we create interfaces in Go? Let's delve into different use cases where interfaces are considered to bring some value.

COMMON BEHAVIOR

The first option to use interfaces is when multiple objects implement the same behavior. Let's see this use case with a concrete example.

We will implement a function that receives a list of customers, applies multiple filters, and returns the customers' remaining list. Let's first define the three filters:

- `FilterByAge` to filter by age
- `FilterByCity` to filter by city
- `FilterByCount` to return a maximum number of customers (e.g., no more than 100 customers)

```
type FilterByAge struct{ minAge int }

func (f FilterByAge) filter(customers []Customer) ([]Customer, error)    ❶
{
    res := make([]Customer, 0)
    for _, customer := range customers {
        if customer.age < 0 {
            return nil, errors.New("negative age")
        }
        if customer.age >= f.minAge {
            res = append(res, customer)
        }
    }
    return res, nil
}

type FilterByCity struct{ city string }

func (f FilterByCity) filter(customers []Customer) ([]Customer, error)   ❷
{
    res := make([]Customer, 0)
    for _, customer := range customers {
        if customer.city == f.city {
            res = append(res, customer)
        }
    }
    return res, nil
}

type FilterByCount struct{ max int }

func (f FilterByCount) filter(customers []Customer) ([]Customer, error)   ❸
{
    if len(customers) < f.max {
        return customers, nil
    }
    return customers[:f.max], nil
}
```

❶ Filter by age implementation

❷ Filter by city implementation

❸ Filter with a maximum count

Now we will implement an `applyFilters` method that apply the three filters and return final list of customers:

```
type filterApplication struct {
    filterByAge    FilterByAge
    filterByCity   FilterByCity
    filterByCount  FilterByCount
}

// Init filterApplication

func (f filterApplication) applyFilters(customers []Customer) (
    []Customer, error) {
    res, err := f.filterByAge.filter(customers)   ❶
    if err != nil {
        return nil, err
    }

    res, err = f.filterByCity.filter(customers)   ❷
    if err != nil {
        return nil, err
    }

    res, err = f.filterByCount.filter(customers)   ❸
    if err != nil {
        return nil, err
    }

    return res, nil
}
```

❶ Apply first filter

❷ Apply second filter

❸ Apply third filter

This implementation works, but we can notice some boilerplate code as all the filters implement the same behavior: `filter([]Customer) ([]Customer, error)`. We could refactor our implementation using an interface this way:

```
type Filter interface {
    filter(customers []Customer) (result []Customer, err error)
}

type filterApplication struct {
    filters []Filter
}

// Init filterApplication

func (f filterApplication) applyFilters(customers []Customer) (
    []Customer, error) {
    for _, filter := range f.filters {   ❶
        res, err := filter.filter(customers)   ❷
        if err != nil {
            return nil, err
        }
        customers = res
    }

    return customers, nil
}
```

❶ Iterate over all the filters

❷ Apply each filter

In this example, creating a `Filter` abstraction using an interface helped us in reducing boilerplate code. We can extend our code by adding more filters to the `filterApplication` struct, the `applyFilters` method will remain the same.

NOTE **In the previous example, as the filtering abstraction contains a single method, it could also have been implemented using a closure. A closure is a function that depends on the value of one or more variables declared outside this function. We could have created a `func([]Customer) ([]Customer, error)` type and implement the filters like this:**

```
type filter func(customers []Customer) ([]Customer, error)   ❶

func filterByAge(minAge int) filter {   ❷
    return func(customers []Customer) ([]Customer, error) {   ❸
        res := make([]Customer, 0)
        for _, customer := range customers {
            if customer.Age < 0 {
                return nil, errors.New("negative age")
            }
            if customer.Age >= minAge {   ❹
                res = append(res, customer)
            }
        }
        return res, nil
    }
}
```

❶ **Define a common `func([]Customer) ([]Customer, error)` instead of an interface**

❷ **Create a function with the `minAge` passed as an argument and return a `filter` type**

❸ **Return a closure**

❹ **Use the `minAge` variable that is declared outside of the scope of the current function**

Using closures instead of interfaces can be interesting if we want to abstract a single function.

UNIT TESTING

Another important use case to use interfaces is to simplify writing unit tests. In a nutshell, when our code has some external dependencies, it can be convenient to wrap them into interfaces.

Let's extend our previous example. We will implement a method to grant a promotion to some customers. The logic will we be following:

- Retrieve all the customers from a database
- Apply the filters we have defined in the previous section
- Update the database to grant to the filtered customers the promotion

Let's implement the first version of this method. The struct will contain a `mysql.Store` struct that contains the methods to interact with the database:

```
type customerPromotion struct {
    filter filterApplication  ❶
    storer mysql.Store  ❷
}

func (c customerPromotion) setPromotion() error {
    customers, err := c.storer.GetAllCustomers()  ❸
    if err != nil {
        return err
    }

    filteredCustomers, err := c.filter.applyFilters(customers)  ❹
    if err != nil {
        return err
    }

    customerIDs := getCustomerIDs(filteredCustomers)
    return c.storer.SetPromotionToCustomers(customerIDs)  ❺
}
```

❶ Filters structure

❷ Structure to interact with the databse

❸ Get all customers

❹ Apply filters

❺ Update database

Now a question arises: how should we test this function? One first option would be to create a test that spins up a MySQL instance as a prerequisite. However, this test wouldn't be a unit test. We should consider unit tests as tests that run within a single process quickly and deterministically. So perhaps this is not the best option.

I'm not arguing whether the database interaction shouldn't be tested. We may want to implement integration tests to verify the SQL request, for example. However, these tests should be an addition to the `setPromotion` tests. So how can we implement unit tests for this method? It requires replacing an explicit dependency (here the integration to MySQL) with an interface this way:

```
type Storer interface {   ❶
    GetAllCustomers() (customers []Customer, err error)
    SetPromotionToCustomers(customerIDs []string) error
}

type customerPromotion struct {
    filter filterApplication
    storer Storer   ❷
}

func (c customerPromotion) setPromotion() error {
    customers, err := c.storer.GetAllCustomers()   ❸
    if err != nil {
        return err
    }

    filteredCustomers, err := c.filter.applyFilters(customers)
    if err != nil {
        return err
    }

    customerIDs := getCustomerIDs(filteredCustomers)
    return c.storer.SetPromotionToCustomers(customerIDs)   ❹
}
```

❶ Create an interface that contains the two methods required in `setPromotion`

❷ Reference the `Storer` interface instead of a concrete implementation

❸ Use the `Storer` interface

❹ Use the `Storer` interface

By creating an interface, we have decoupled our code from the concrete implementation. We can write unit tests using a test double of the `Storer` interface. A test double is a concept popularized by Martin Fowler. There are three main test doubles:

- Stub: an object that holds predefined data and uses it to reply to calls made during tests
- Mock: an extension of a stub object where we are also interested in registering the calls the object receives to perform further assertions
- Fake: working implementation but different from the production one (e.g., hashmap)

Using a test double, we can write unit tests for the `setPromotion` without relying on an external database. It's only possible because we wrapped our external dependency into an interface.

We have seen two main use cases to create interfaces:

- Creating abstraction of a shared behavior
- Replacing external dependencies with an interface to facilitate writing unit tests

NOTE **This list is not intended to be exhaustive. It contains directions to understand when it's helpful to use interfaces, but the more cases we could add, the more dependent on the actual context there would be.**

In the next section, we will see when it's not considered appropriate to use interfaces.

3.4.3 When to Not Use Interfaces

It is pretty common to see interfaces being overused in Go projects. Perhaps your background is C++ or Java, and you find it pretty natural to create interfaces before concrete types. However, this is not how things should work in Go.

As we mention, interfaces allow us to create abstractions, and abstractions **should be discovered, not created**. Said differently, we shouldn't start by creating abstractions in our code if there is no direct reason for that. We should strive to create interfaces when we need them, not when we foresee that we will need them.

What is the problem if we overuse interfaces? First, we should note that there is a performance impact of calling a method through an interface. It requires a lookup in a hashtable to find the concrete type it's pointing to. Yet, this is not the main problem as there are not many contexts in which this overhead should be forbidden, but it's still worth mentioning. The main issue is that it makes the code flow more complex. Adding a useless level of indirection doesn't bring any value; it just creates a useless abstraction making the code more complex to reason about.

In most cases, if we define an interface with only one implementation (we don't count test doubles here) and this implementation doesn't require any external dependency, we may ask ourselves whether this interface is worth it. If it's to simplify unit testing by creating a test double, why can't we call the concrete implementation directly? What's the rationale for it? There might be exceptions, for example, if the struct is complex to set up because of some state that we prefer to abstract it. Yet, in this particular scenario, using an interface is probably not required.

In summary, we should be cautious when creating abstractions in our code. Again, abstractions should be discovered, not created. It's pretty frequent for software developers to overengineer our code by trying to guess what would be the perfect level of abstraction based on what we think we might need later on. This process should be avoided as we would pollute our code with unnecessary abstractions making it more complex to reason about and to understand.

In the next section, we will discuss a common mistake related to interfaces: creating interfaces on the producer-side.

3.5 Interface on Producer-Side

We should now have a good understanding about what's an interface and that it should be used sparingly only in some specific cases. Yet, where should an interface live?

It's pretty common to see interfaces created on the producer-side, alongside concrete

implementations. This design is perhaps a habit from developers having once again a C++ or a Java background. However, in Go, this is, in most cases, not what we should do.

Let's discuss the following example. We implement a package to store and retrieve customer data. We also decide that the clients will access our package through the following interface:

```
type Storer interface {
    StoreCustomer(customer Customer) error
    GetCustomer(id string) error
    UpdateCustomer(customer Customer) error
    GetAllCustomers() ([]Customer, error)
    GetCustomersWithoutContract() ([]Customer, error)
    GetCustomersWithNegativeBalance() ([]Customer, error)
}
```

We might think that we have some excellent reasons to do it. Perhaps it's a good way to decouple the code client code from the actual implementation? Perhaps we do it to help clients creating test doubles? However, this is not a best practice.

The main reason is similar to what we described in the previous section: abstractions should be discovered, not created. It's not up to the producer to force a given abstraction for all the clients. Instead, it's up to the client to decide whether he needs some form of abstraction and then decide what's the best abstraction level for his needs.

In the previous example, perhaps one client will require using all the six methods. But maybe another client only needs to use `GetAllCustomers` and doesn't need the complete `Storer` to create a test double implementing all the methods. The best approach would have been to expose the producer-side concrete types and let the clients decide whether to use interfaces and the best granularity.

For the sake of completeness, let's mention that this approach, interfaces on the producer-side, is sometimes used in the standard library. For example, the `encoding` package defines interfaces implemented by other sub-packages such as `encoding/json` or `encoding/binary`. So is this implementation an anti-pattern? Of course not. In this particular case, the abstractions defined in `encoding` package are used across the standard library, and the language designers knew the benefit of creating these abstractions upfront.

In summary, interfaces shouldn't be created on the producer-side without a solid argument against the following question: will this abstraction be helpful for my clients?

In the next section, we will discuss a common mistake related to the `interface{}` type.

3.6 interface{} Says Nothing

In Go, an interface type that specifies zero methods is known as the empty interface. An empty interface can hold any value types:

```
func main() {
    var i interface{}

    i = 42   ❶
    i = "foo"   ❷
    i = struct {   ❸
        s string
    }{
        s: "bar",
    }
    i = f   ❹

    _ = i   ❺
}

func f() {}
```

❶ An int

❷ A string

❸ A struct

❹ A function

❺ Assignment so that the example compiles

In assigning a value to an `interface{}` type, we lose all type info which requires a type assertion to get anything useful out of the `i` variable.

Let's see a concrete example. We will implement a `Store` struct and the skeleton of two methods: `Get` and `Set`. These methods will be used to store different struct types: `Customer` and `Contract`:

```
package store

type Customer struct{
    // Some fields
}
type Contract struct{
    // Some fields
}

type Store struct{}

func (s *Store) Get(id string) (interface{}, error) {   ❶
    // ...
}

func (s *Store) Set(id string, v interface{}) error {   ❷
    // ...
}
```

❶ Return an empty interface

❷ Accept an empty interface

Though there is nothing wrong with `Store` compilation-wise, we should take a minute to think about the exposed methods' semantic. As we accept and return empty interfaces, the methods

lack expressiveness. If a developer has to use the `Store` struct, he will probably have to dig into the documentation or read the code itself to understand how to use these methods. Hence, accepting or returning an empty interface does not convey meaningful information.

Also, as there is no safeguard at compile-time, there is nothing that prevents a caller from calling these methods with whatever data type:

```
s := store.Store{}
s.Set("foo", 42)
```

By using empty interfaces, we lose some of the benefits of a statically-typed language.

Instead, we should avoid as much as possible empty interface types and make our signatures explicit. We should rather write it this way:

```
type Store struct{}

func (s *Store) GetContract(id string) (Contract, error) {
    // ...
}

func (s *Store) SetContract(id string, contract Contract) error {
    // ...
}

func (s *Store) GetCustomer(id string) (Customer, error) {
    // ...
}

func (s *Store) SetCustomer(id string, customer Customer) error {
    // ...
}
```

In this version, the methods are expressive. It reduces the risk of incomprehension for a consumer of this package. Also, we can create minimal abstractions using an interface. For example, if a consumer is interested only in the methods related to contracts, he can create an interface containing these two methods solely:

```
type Storer interface {
    GetContract(id string) (store.Contract, error)
    SetContract(id string, contract store.Contract) error
}
```

So why do we have an `interface{}` type in the language? What are the cases where empty interfaces are useful? Let's take a look at the standard library. We can notice multiple uses of empty interfaces such as `json.Marshal`, for example, to marshal an input value:

```
func Marshal(v interface{}) ([]byte, error) {
    // ...
}
```

Another example in the `sql` package that uses empty interfaces to pass variadic arguments while executing a query:

```
func (c *Conn) QueryContext(ctx context.Context, query string, args ...interface{}) (*Rows, error) {
    // ...
}
```

In summary, empty interfaces can be useful if there is a genuine need for accepting or returning any possible type. For example:

- Marshaling any type
- Accepting any type for formatting

In general, we should avoid over-generalizing at all costs the code we write. Perhaps, a little bit of duplicated code can occasionally be better if it improves other aspects such as code expressiveness.

Now that we have seen the drawbacks of using `interface{}` types, let's discuss in the next section common patterns to deal with options configuration.

3.7 Not Using the Functional Options Pattern

This section will go through a common use case to present how to make an API convenient and friendly to accept options configuration. We will delve into different options to finally present a popular solution in Go: the functional options pattern.

Let's say we have to design a library that exposes a function to create an HTTP server. This function would accept different inputs: an address and a port. The skeleton of the function would be the following:

```
func NewServer(addr string, port int) (*http.Server, error) {
    // ...
}
```

Consumers start to use this function, and everyone is happy. Yet, at some point in time, consumers begin to complain that this function is somewhat limited and lacks other parameters (e.g., write timeout, connection context). However, we start to notice that if we add new function parameters, it will break the compatibility, forcing the consumers to modify the way they call `NewServer`.

In the meantime, we would like to extend the logic related to port management this way, like presented in figure 3.4:

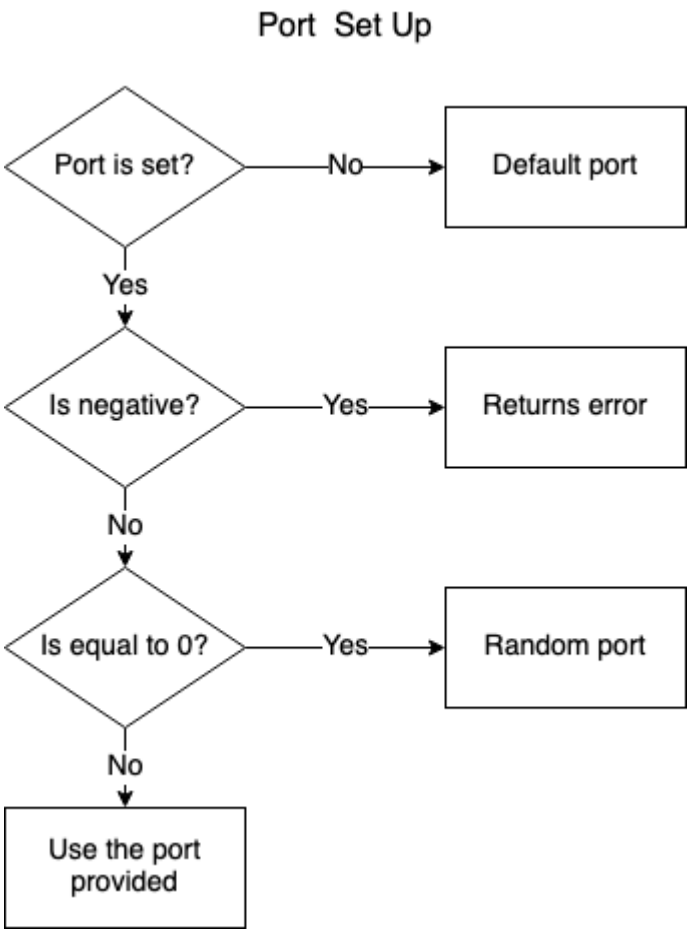

Figure 3.4 Logic related to the port option.

- If the port is not set, it uses the default one
- If the port is negative, it returns an error
- If the port is equal to 0, it uses a random port
- Otherwise, it uses the port provided by the client

How could we implement this function in an API-friendly way? Let's see all the different options.

3.7.1 Config Struct

The first possible approach is to use a config struct to handle the different options. We can split the parameters into two different categories: mandatory and optional. The mandatory parameters could live as function parameters, whereas the optional could be handled in the config struct:

```
type Config struct {
    Port        int
}

func NewServer(addr string, cfg Config) {
}
```

This solution fixes the compatibility issue. If we add new options, it will not break on the

client-side. However, this approach does not solve our requirement related to port management. Indeed, we should bear in mind that if a struct field is not provided, it will be initialized to its zero value:

- 0 for an integer
- 0.0 for a floating-point type
- "" for a string
- nil for slices, maps, channels, pointers, interfaces, and functions

Therefore, in the following example both structs are equal:

```
c1 := httplib.Config{
    Port: 0,   ❶
}
c2 := httplib.Config{
      ❷
}
```

❶ `Port` is initialized to 0

❷ `Port` is missing so initialized to 0

In our case, we need to find a way to distinguish a port purposely set to 0 and a missing port. Perhaps one option might be to handle all the parameters of the config struct as pointers this way:

```
type Config struct {
    Port        *int
}
```

This option would work but has a couple of downsides. First, it's not handy for clients to provide an integer pointer. Clients have to create a variable and then pass a pointer this way:

```
port := 0
config := httplib.Config{
    Port: &port,   ❶
}
```

❶ Provide integer pointer

It's not a showstopper as such, but the overall API becomes less convenient to use. The second downside is that a client using our library with the default configuration will have to pass an empty struct this way:

```
httplib.NewServer("localhost", httplib.Config{})
```

This code does not look great. Readers will have to understand what is the meaning of this magic struct.

Another option would be to use the classic builder pattern as presented in the next section.

3.7.2 Builder Pattern

The builder pattern provides a flexible solution to various object creation problems. Let's see how it can help us in designing a friendly API that tackles all our requirements, including port management:

```
type Config struct {   ❶
    Port int
}

type ConfigBuilder struct {   ❷
    Port *int
}

func (b *ConfigBuilder) Port(port int) {   ❸
    b.Port = &port
}

func (b *ConfigBuilder) Build() (Config, error) {   ❹
    cfg := Config{}

    if b.Port == nil {   ❺
        cfg.Port = defaultHTTPPort
    } else {
        if *b.Port == 0 {
            cfg.Port = randomPort()
        } else if *b.Port < 0 {
            return Config{}, errors.New("port should be positive")
        } else {
            cfg.Port = *b.Port
        }
    }

    return cfg, nil
}

func NewServer(addr string, config Config) (*http.Server, error) {
    // ...
}
```

❶ Config struct

❷ Config builder struct that contains an optional port

❸ Public method to set up the port

❹ Build method to create the config struct

❺ Main logic related to port management

And this is how a client would use our builder-based API (we assume that we have put our code in an `httplib` package):

```
builder := httplib.ConfigBuilder{}   ❶
builder.Port(8080)   ❷
cfg, err := builder.Build()   ❸
if err != nil {
    return err
}

server, err := httplib.NewServer("localhost", cfg)   ❹
if err != nil {
    return err
}
```

❶ Creates a builder config

❷ Set the port

❸ Build the config struct

❹ Pass the config struct

This approach makes the port management handier. It's not required to pass an integer pointer as the `Port` method accepts an integer. However, it still needs to pass a config struct that might be empty if a client requires the default configuration.

> **NOTE** **There are different variations on this approach. One variation for example, might be for `NewServer` to accept a `ConfigBuilder` struct and build the config internally. However, it does not fix the problem of being required to pass a config object in every case.**

Another downside in some situations is related to error management. In programming languages where exceptions can be thrown, the builder methods such as `Port` can raise exceptions if the input is invalid. In Go, we cannot make the builder methods returning an error. Indeed, the builder pattern is supposed to be called in a composite way (e.g. `builder.Port(8080).Timeout(time.Second).Certificate(cert)`). We don't want to let the client having to check the error every time. Therefore, the validation is delayed in the `Build` method. In some scenarios, this may not be expressive for clients.

Let's now see another pattern called the functional options pattern that relies on variadic arguments.

3.7.3 Functional Options Pattern

The latest approach we will delve into is the functional options pattern. Though there are different implementations with minor variations, the main idea remains the same as presented in figure 3.5:

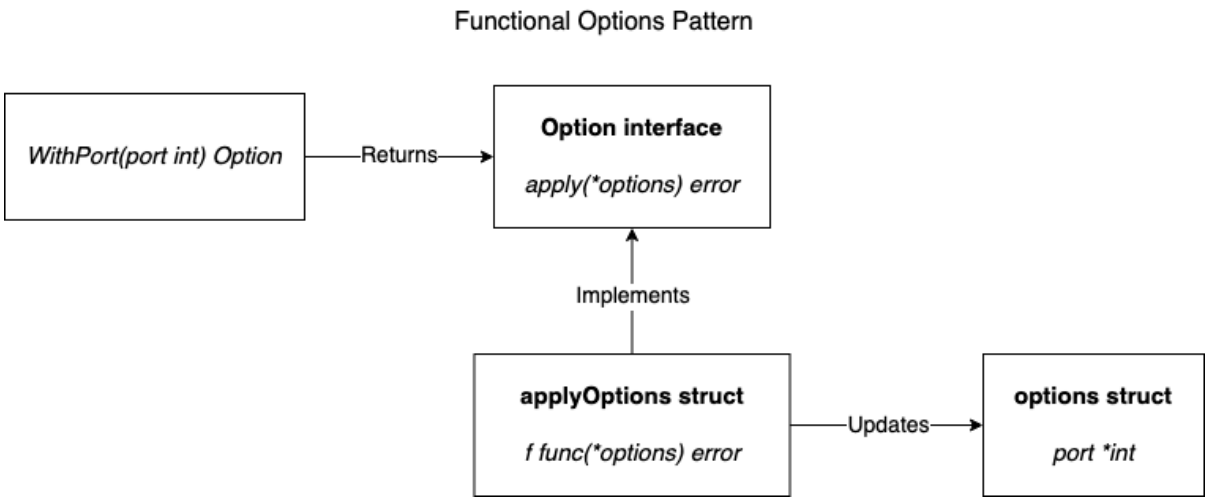

Figure 3.5 The `WithPort` option updates the final `options` struct.

Each option (such as `WithPort`) returns an implementation of the `Option` interface that will update the `options` struct, holding the final configuration. This struct is now private:

```
type options struct {
    port *int
}
```

In the meantime, we have to create a public `Option` interface and a private `applyOptions` that implements this interface:

```
type Option interface {
    apply(*options) error   ❶
}

type applyOptions struct {
    f func(*options) error   ❷
}

func (ao *applyOptions) apply(opts *options) error {
    return ao.f(opts)   ❸
}
```

❶ The `Option` interface is composed of a single `apply` method

❷ The `f` field is a function reference that will hold the logic on how to update the config struct

❸ `applyOptions` implements the `apply` method by calling the internal `f` function

The whole logic lives in the internal `f` function field. This `f` field is created by the public method used by clients to configure options. For example, the `WithPort` function:

```
func WithPort(port int) Option {
    return &applyOptions{
        f: func(options *options) error {   ❶
            if port < 0 {
                return errors.New("port should be positive")
            }
            options.port = &port
            return nil
        },
    }
}
```

❶ Initialize `f` by providing the logic how to validate inputs and update the config struct

Each config field requires to create a public method (that starts with the `With` prefix for conventions) containing a similar logic: how to validate inputs if needed and how to update the config struct.

Now, let's delve into the last part on the provider side, how to use these options:

```
func NewServer(addr string, opts ...Option) (*http.Server, error) {   ❶
    var options options   ❷
    for _, opt := range opts {   ❸
        err := opt.apply(&options)   ❹
        if err != nil {
            return nil, err
        }
    }

    // At this stage, the options struct is built and contains the config
    // Therefore, we can implement our logic related to port configuration
    var port int
    if options.port == nil {
        port = defaultHTTPPort
    } else {
        if *options.port == 0 {
            port = randomPort()
        } else {
            port = *options.port
        }
    }

    // ...
}
```

❶ Accept variadic `Option` arguments

❷ Create an empty `options` struct

❸ Iterate over all the input options

❹ Apply each option which will result in modifying the common `options` struct

As `NewServer` accepts variadic `Option` arguments, a client can now consume this API by passing zero to multiple options following the address:

```
// No options
server, err := httplib.NewServer("localhost")

// Multiple options
server, err := httplib.NewServer("localhost", httplib.WithPort(8080),
httplib.WithTimeout(time.Second))
```

Thanks to variadic arguments, if a client needs the default configuration, he doesn't have to provide an empty struct as we have seen in the previous options:

```
server, err := httplib.NewServer("localhost")
```

This is the functional options pattern. It provides a handy and API-friendly way to handle options. Though the builder pattern can be a valid option, it has some small downsides that tend to make the functional options pattern the idiomatic way to deal with this problem in Go. This pattern is used in many different libraries, such as gRPC.

In the next project, we will delve into a common mistake: Go projects organization.

3.8 Project Misorganization

Organizing a Go project is not an easy task. It's fairly frequent to find projects with a structure that lacks coherency. This section will first delve into a standard for structuring a project and then discuss a couple of best practices showing how to improve the way we can organize a project.

3.8.1 Project Structure

There is no strong convention brought by the language maintainer about how to structure a project in Go. However, one standard has emerged as a de facto one: golang-standards/project-layout.

If your project is small enough (only a couple of files), or if your company has already created its standard, it's probably not worth using or migrating to project-layout. Otherwise, it might worth considering adopting project-layout. Let's delve into this layout and see what the main folders are:

- `/cmd`: Main source files. The `main.go` of a `foo` application should live in `/cmd/foo/main.go`.
- `/internal`: Private code that we don't want others importing in their applications or libraries.
- `/pkg`: Public code that we want to expose to others.
- `/test`: Additional external tests and test data. Unit tests in Go live in the same package as the source files. However, tests such as public API or integration tests should live in `/test`.
- `/configs`: Configuration files.
- `/docs`: Design and user documents.
- `/examples`: Examples for our application and/or public library.

- `/api`: API contract files (e.g., Swagger, Protocol Buffers, etc.).
- `/web`: Web application-specific asset (static files, etc.).
- `/build`: Packaging and CI files.
- `/scripts`: Scripts for analysis, installation, etc.
- `/vendor`: Application dependencies (e.g., Go modules dependencies).

As you have noticed, there isn't a `/src` folder, just like in some other languages. `/src` being too generic, this layout favors using folders such as `/cmd`, `/internal` or `/pkg`.

Following this standard can be an excellent option to have some form of consistency across an organization. Instead of reinventing the wheel over the projects, following a standard may be a good option.

We mentioned that the main logic would live in `/internal` or `/pkg`. The next section will discuss how to organize our main logic.

3.8.2 Package Organization

In Go, there is no concept of sub-packages. However, we can decide to organize packages within subdirectories. If we take a look at the standard library, the `net` folder is organized this way:

```
/net
    /http
        client.go
        ...
    /smtp
        auth.go
        ...
    addrselect.go
    ...
```

We can notice that `net` acts both as a package and a folder that contains other packages. Yet, `http` does not inherit from `net` or have some specific access to the `net` package. Elements inside of `http` can only see exported `net` elements.

Therefore, subdirectories' main benefits are to keep packages in a place where they live with a high cohesion. Here `http` depends on `net` (for example, through `net.Conn`), but this is not mandatory. There are different schools of thought about the best way to use directories: shall we organize them by feature or by layer? It depends on what you prefer, and clearly, this is not the scope of this section. However, we should keep in mind that subdirectories can help in how we organize our code.

Regarding packages, there are multiple best practices that we should follow.

First, we should avoid premature packaging as it might cause us to overcomplicate a project. Sometimes, it's better to keep a simple organization and make our project evolve when we understand what it contains rather than forcing ourselves to have the perfect structure upfront.

Granularity is another thing to consider. We should avoid having dozens of nano package containing only one or two files. If we do, it's because we have probably missed some logical connections across these packages, making our project harder to understand for readers. Conversely, we should also avoid huge packages that dilute the meaning of a package name.

Package naming should also be taken with care. As all developers already experience it, naming is hard. We should name a package after what they provide, not what they contain to help clients understand a Go project. Also, for the same reason, naming should be meaningful. Therefore, a package name should be short, concise, and expressive. By convention, packages should be lower case, single-word names.

Regarding what to export, the rule is pretty straightforward. We should minimize as much as possible what should be exported so that it reduces the coupling between packages and keep unnecessary exported elements hidden. If we are unsure whether to export an element or not, we should default to not exporting it. Later, if we discover that we need to export it, we can adjust our code then. Let's also keep in mind some specific cases such as making fields exported to that a struct can be unmarshalled with `encoding/json`.

These are some standard best practices related to package organization. In the next section, we are going to tackle common packages.

3.9 Creating Utility Packages

We have just discussed standard practices related to package organization. This section will discuss a common bad practice: creating shared packages such as `utils`, `common` or `base`. We will see what the problems are with such an approach.

Let's see an example inspired by the official Go blog. It's about implementing a set data structure. A map where the value is ignored. The idiomatic way to do it in Go is to keep a `map[string]struct{}` as a `struct{}` type conveys that we are not interested in the type itself and we can pass a `struct{}{}` which has a zero-length. Let's expose two methods in a `util` package:

```
package util

func NewStringSet(...string) map[string]struct{} { ... }   ❶
func SortStringSet(map[string]struct{}) []string { ... }   ❷
```

❶ Creates a string set

❷ Return a sorted list of keys

A client of this package will then consume this package like this:

```
set := util.NewStringSet("c", "a", "b")
fmt.Println(util.SortStringSet(set))
```

The problem here is that `util` is meaningless. We even call it `common`, `shared` or `base`, it remains a meaningless name that doesn't provide any value.

Instead of a utility package, we should create an expressive package name such as `stringset`, for example:

```
package stringset

func New(...string) map[string]bool { ... }
func Sort(map[string]bool) []string { ... }
```

We removed the suffix as `NewStringSet` and `SortStringSet` respectively became simply `New` and `Sort`. On the client-side, it will now look like this:

```
set := stringset.New("c", "a", "b")
fmt.Println(stringset.Sort(set))
```

We could even go a step further. Instead of exposing utility functions, we can even create a specific type and expose `Sort` as a method this way:

```
package stringset

type Set map[string]bool
func New(...string) Set { ... }
func (s Set) Sort() []string { ... }
```

This change would make the client even simpler. There would only one reference to `stringset`:

```
set := stringset.New("c", "a", "b")
fmt.Println(set.Sort())
```

With this small refactoring, we got rid of a meaningless package name to expose an expressive API.

As mentioned by Dave Cheney, it's fairly frequent to find utility packages to handle common facilities. For example, common types between a client and a server. Instead of separate packages with one on them being shared, it's better to reduce the number of packages and combine them into a single package.

Naming a package is a critical piece of application design, and we should be cautious about it, but as a general rule of thumb, we should avoid utility packages.

In the next section, we will keep discussing packages with package collision.

3.10 Ignoring Package Name Collisions

Package collision occurs when a variable name collides with an existing package name. Let's see a concrete example with a library exposing a Redis client:

```
package redis

type Client struct { ... }

func NewClient() (*Client, error) { ... }

func (c *Client) Get(key string) (string, error) { ... }
```

Now let's jump on the client side. In Go, this code is perfectly valid:

```
redis := redis.NewClient()  ❶
v, err := redis.Get("foo")  ❷
```

❶ Call `NewClient` from the `redis` package

❷ Use the `redis` variable

The `redis` variable name collides with the `redis` package. Even though this is allowed, it should be avoided. Indeed, throughout the scope of the `redis` variable, the `redis` package is not accessible anymore. Also, suppose a qualifier references both a variable and a package name throughout a function. In that case, it might be ambiguous for a code reader to know what does a qualifier refers to. What are the options to avoid such a collision?

The first option is to find a different variable name. For example:

```
redisClient := redis.NewClient()
v, err := redisClient.Get("foo")
```

This is probably the most straightforward approach. However, if, for some reason, we prefer to keep naming our variable `redis`, we can play with package imports.

Using package imports, we can either use an import alias to change the qualifier to reference the `redis` package. For example:

```
import redisapi "mylib/redis"  ❶

// ...

redis := redisapi.NewClient()  ❷
v, err := redis.Get("foo")
```

❶ Create an alias to the `redis` package

❷ Access to the `redis` package via the `redisapi` alias

Here, we used the `redisapi` import alias to reference the `redis` package so that we can keep naming our variable `redis`.

The other option while dealing with package imports is to use dot imports. With dot imports, all the public elements of a package can be accessed without a qualifier:

```
import . "mylib/redis"  ❶

// ...

redis := NewClient()  ❷
v, err := redis.Get("foo")
```

❶ Use dot imports to import the `redis` package

❷ Reference the `redis` package without any qualifier

However, this approach might be somewhat confusing as it's not frequently used (except on rare occasions such as testing frameworks).

NOTE **We should also avoid naming collisions between a variable and a built-in function:**

```
copy := copyFile(src, dst)  ❶
```

❶ **The `copy` variable collides with the `copy` built-in function**

In this case, the `copy` built-in function wouldn't be accessible anymore as long as the `copy` variable lives.

In summary, we should prevent variable name collisions. If we face a collision, we should either find another meaningful name or, in the case of a package deal with package imports.

In the next section, we will discuss common mistakes related to code documentation.

3.11 Missing Code Documentation

Documentation is an important aspect of a developer's life. It simplifies how clients will use a library and how a code reader will understand our code. In Go, the rule to follow is the following: every exported element must be documented. Be it a structure, an interface, a function, etc. If it's exported it has to be documented.

Let's write the first example without any documentation:

```
package example

type Customer struct{}

type ClientAPI interface{}

const Pi = 3.14

var ErrorCount = 0

type Unit int8

func GetBalance() {}

func (c Customer) GetID() {}
```

Running golint (a standard Go style mistakes checker that we presented in the previous chapter) on this example will produce the following errors:

```
$ golint ./...
examples/example.go:3:6: exported type Customer should have comment or
    be unexported
examples/example.go:5:6: exported type ClientAPI should have comment or
    be unexported
examples/example.go:7:7: exported const Pi should have comment or
    be unexported
examples/example.go:9:5: exported var ErrorCount should have comment
    or be unexported
examples/example.go:11:6: exported type Unit should have comment
    or be unexported
examples/example.go:13:1: exported function GetBalance should have
    comment or be unexported
examples/example.go:15:1: exported method Customer.GetID should have
    comment or be unexported
```

So how should we document exported elements? The Go convention is to make the comments starting with the name of the public element. For example, a `Foo` interface comment must start with `// Foo`. Let's change our code to comply with this convention:

```
package example

// Customer is a customer representation.
type Customer struct{}

// ClientAPI is an interface to interact with a client.
type ClientAPI interface{}

// Pi is the mathematics constant.
const Pi = 3.14

// ErrorCount is a shared counter on the errors.
var ErrorCount = 0

// Unit is a specific type based on an int8.
type Unit int8

// GetBalance gets the balance.
func GetBalance() {}

// GetID gets the customer identifier.
func (c Customer) GetID() {}
```

Although it may seem somewhat redundant sometimes, the Go convention for each comment is to make it a full sentence that ends with punctuation.

We should also note that it is possible to deprecate a public element using the `// Deprecated:` comment this way:

```
// GetBalance gets the balance.
// Deprecated: Use GetLatestBalance instead.
func GetBalance() {}
```

Then, if a developer uses the `GetBalance` function, his IDE should raise a warning.

Another rule that is not mandatory but should be followed regarding documentation is to document each package. The comment must start by `// Package` and be followed by the package name this way:

```
// Package example provides a useful list of Go examples.
// Lorem ipsum dolor sit amet, consectetur adipiscing elit,
// sed do eiusmod tempor incididunt ut labore et dolore magna
// aliqua. Ut enim ad minim veniam, quis nostrud exercitation
// ullamco laboris nisi ut aliquip ex ea commodo consequat.
package example
```

The first line of a package comment should be concise as it will appear in the package list:

Name	Synopsis
archive	
tar	Package tar implements access to tar archives.
zip	Package zip provides support for reading and writing ZIP archives.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
builtin	Package builtin provides documentation for Go's predeclared identifiers.
bytes	Package bytes implements functions for the manipulation of byte slices.

Figure 3.6 An example of the generated Go standard library.

Then, we can provide all the information we need in the following lines.

Commenting a package can be done in any Go files of this package; there is no rule. In general, we should try to put it in a relevant file with the same name of the package or a specific file such as `doc.go`. Yet, there is no rule about which file should contain the package comment.

One last thing to mention regarding package documentation is that the comments not adjacent to the declaration are omitted. For example, the following copyright comment will not be visible in the produced documentation:

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.
 (1)
package http
```

(1) Empty line, therefore the above comment will not be included in the documentation

In summary, we should keep in mind that every exported element has to be documented. Running golint can automate detecting the elements of our code that misses related documentation.

3.12 Summary

- Avoid nested levels and keep the happy path aligned on the left to make building a mental model of the code easier.
- Limit the use of init functions because they have limited error management, which may complicate handling a state and implementing unit tests. If needed, initializations should be handled as ad-hoc functions.
- Forcing to use getters and setters is not idiomatic in Go. Being pragmatic and finding the right balance between efficiency and blindly following certain idioms should be the way to go.
- Abstractions should be discovered, not created. Therefore, we shouldn't create interfaces based on what we think we might need later on, neither create interfaces on the producer-side in most cases.
- Only use `interface{}` if you need to accept or return any possible type. Otherwise, an empty interface does not provide meaningful information and can lead to compile-time issues by allowing a caller to call methods with any data type.
- Using the functional options pattern is a handy and friendly way to handle options when exposing an API.
- Following a standard layout such as golang-standards/project-layout can be an excellent way to start structuring Go projects, especially if you are looking for existing conventions to standardize your project.
- Naming is a critical piece of application design. Creating packages such `common`, `util`, or `shared` doesn't bring much value for the reader. We should refactor such packages into meaningful and specific packages.
- To avoid naming collisions between variables and packages, use unique names for each. If this is not feasible, we can use an import alias to change the qualifier to differentiate the package name from the variable name or even think about a better name.
- When we create exported elements, be it internally in a repository or a public library, they have to be documented. It's an industry best practice that we should strive to follow.

Notes

1. J. S. Moser et al., “Mind Your Errors: Evidence for a Neural Mechanism Linking Growth Mind-set to Adaptive Posterror Adjustments,” Psychological Science 22/12 (2011)

2. Understanding Real-World Concurrency Bugs in Go by Tengfei Tu, Xiaoyu Liu, Linhai Song and Yiying Zhang