The Art of Go - Basics

Introduction to Programming in Golang - Beginner to Intermediate

Harry Yoon

Version 2.0.1, 2023-03-15

Copyright

The Art of Go - Basics:
Introduction to Programming in Golang

Published: May 2021

Harry Yoon
San Diego, California

Preface

Learn programming for fun.

Go is one of the most popular programming languages. It is primarily used for Web backend and server side programming. But it is finding broader uses in other areas as well. Go is an interesting language. It is much simpler than most other modern programming languages. It is easier to learn.

Go almost has a "retro" feel to it. It does not support an object oriented programming style well. It does not support a functional programming style. It is almost like the good ol' "C". But it is easier to use. It is safer to use. It is more fun to use.

If you are just starting with programming, then Go is the perfect language to learn programming with. If you are a seasoned developer, and looking to expand your horizon, then Go is the perfect language to pick up as your next programming language.

The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate is written for a broad audience. It starts from the absolute basics and moves on to more advanced topics.

There are a lot of books and other resources that teach programming. They are more or less the same. ***The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate*** takes a rather unique approach. For one thing, it emphasizes "reading" before "writing". Readers are encouraged to read the book through, from beginning to end.

This book, for example, does not start with instructions on how to install the compiler tools, etc., which probably 99% of the introductory programming books do. It is an important topic, but it is easy to learn. You can easily figure it out (from other resources), when needed.

There are two kinds of knowledge. In fact, a whole spectrum between the two extremes. First, there is this "quick knowledge" for the lack of better words. Suppose that you have bought one of those "furniture kits" from IKEA, those kinds that are "assembly required". Knowing how to assemble the furniture is very important. But, it is not something you have to consciously "learn". Nobody studies the furniture assembly instructions before they buy an IKEA furniture. When you need it, you learn it (e.g., from the instruction that comes with the furniture). And then you forget it. That's a disposable knowledge.

On the other hand, there is a type of knowledge that requires some "understanding". Why does the Moon not "fall" to the earth while all apples from an apple tree do? It requires "understanding" to answer this kind of questions.

The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate tries to provide this type of knowledge when it comes to programming and programming in Go. This book emphasizes "concepts" over details.

Knowledge is largely about familiarity. The "deep knowledge" (the sort that requires understanding) is no different. Over time, with enough exposure to related facts and examples, you will feel like you know them, you understand them.

Throughout this book, we will introduce certain terms or concepts without precisely defining them first and then we will elaborate on them later in the book. You do not have to learn, understand, or memorize, everything on your first encounter. This book teaches programming in Go *through repetitions.*

There are a lot of programming "technicians". Hope that the readers of this book strive to become "artists". Artists in programming.

Good luck!

Table of Contents

Introduction

> Premature optimization is the root of all evil.
>
> — Donald Knuth

Learning a programming language is not much different from learning a foreign language. You learn from examples, primarily by listening and reading.

The speaking and writing abilities follow, more or less in sync with your listening and reading abilities.

Programming languages are not meant to be spoken, but the same principle applies. You learn from examples, primarily by reading well-written code.

The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate is organized into a series of small *lessons.*

Each lesson teaches basic concepts of programming, and programming in Go, in particular, by going through carefully-designed sample code.

All lessons are more or less "self-contained". But it will be best if you go through them sequentially, especially if you are new to programming.

The lessons gradually progress from basic topics to more advanced subjects. You can advance at your own pace. If you are just starting out, you can take your time.

If you have some programming experience (e.g., in other languages), then the book may seem to start too slow. You can skip, or skim through, some of the earlier lessons.

The book covers a lot of subjects, but it is not meant to be a comprehensive introduction to the Go programming language. This is not an academic textbook.

This is not a reference.

The Art of Go - Basics covers the following topics, among other things:

- The basic structure of a Go program.
- Basic constructs of the Go language such as expressions and statements.
- Primitive types, slices, maps, and functions.
- Custom types, in particular, structs and interfaces, and methods.
- Goroutines and channels.

The examples used in this book are, although small and elementary, all inspired by the real world use cases, to varying degrees. Readers are encouraged to try to "read" the code samples first, not just the text in the book, *before* diving into the main part of each lesson.

Much of the code may not make much sense at the first reading. But, that's how we learn a new language. That's how we learn a new skill.

There is a Chinese saying, which can be translated into something like this:

> If you read a book a hundred times, then the meaning of its content will eventually seem obvious to you.

The lessons are organized into four parts: First Steps, Moving Forward, Having Fun, and Final Projects.

This division is somewhat arbitrary, but the first part, **Part I: First Steps**, is mostly focused on general programming. If you are coming from a different programming language, especially C-style languages, then you will feel comfortable with lessons in this part. As stated, you can probably just skim over much of this part.

The second part, **Part II: Moving Forward**, focuses on some Go-specific topics like

Go `struct`s and `interface`s. We will also go over some basic examples of Web programming in Go as well in this part.

In **Part III: Having Fun**, we will cover a few independent topics, ranging from file system-related programming to Web scraping. We introduce "goroutines" in this part, which is often omitted in introductory books on Go.

Finally, in **Part IV: Final Projects**, we will work on a few complete projects, from beginning to end. We will design and implement a card game, "Go Fish", as a command line game. The readers are encouraged to tackle all the problems in this part. They are excellent projects for beginning programmers.

Exercises are optional. They may require knowledge on some subjects that we do not thoroughly cover in this book.

Regarding the sample programs, one thing to note is that we do not include comments in the code. Comments will mostly add clutter in books like this. The content of the book serves as code comments. And, more.

In practice, writing good comments, and documentations, is a very important skill to learn.

As stated, Go is a language "easy" to learn, and "easy" to start programming with. But, it has some quirks as well. The Go's language grammar is relatively simple, but there are a few exceptions to the general rules, and there are some "gotchas".

It will take time and effort to become really proficient in Go. Hope you find this book helpful in your journey into the programming world, in Go.

Let's get started!

Part I: First Steps

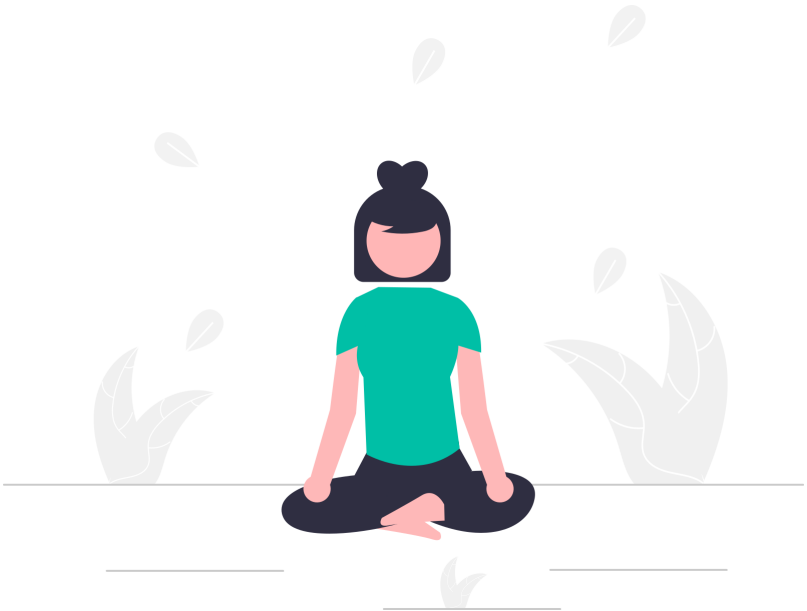
A journey of a thousand miles begins with a single step.

The code examples in this book are meant to be *read.* You do not have to type them on computer to get "hands-on" experience.

Lesson 1. The Simplest Go Program

1.1. Agenda

We will review our first Go program together.



1.2. Code Reading

Here's a small Go program.

smallest-program/main.go

```
1 package main
2
3 func main() {}
```

The label indicates that the source code is copied from a file *main.go* under a folder named *smallest-program*.

The line numbers, printed on the left-hand side, are not part of the program. They are included for easy reference.

1.2.1. Explanation

This is the simplest Go program that compiles and runs, in a single file (named "main.go" in this example). *It does nothing.*

You can run this program as follows on command line.

```
go run main.go
```

1.2.2. Keywords

This simple program includes two Go language "keywords".

A keyword in a programming language is a word or identifier that has special meaning to the language and the language tools such as compilers. Programmers cannot use language keywords for other purposes, e.g., as variable names, etc.

Go comprises 25 keywords. This example program uses the following two:

- `package`: The package line is a declaration indicating that this source file belongs to a certain package, *main* in this example.
- `func`: The keyword `func` declares a function and introduces its definition to the program. *main()* is a special function, and the func declaration has a slightly different semantics.

The book gradually introduces important concepts through repetitions. You do not have to try to understand everything in your first encounter, especially if you are new to programming. Remember, knowledge is mostly about familiarity.

1.2.3. Grammar

A Go program is written in one or more source files. Each source file must start with the `package` line. This is called a "package declaration".

A package is a basic unit of organization in the Go language. In many programming languages, a file is typically a basic unit. In Go, however, it is the package, which can include one or more files.

An executable Go program has to include a special package "main", as in this example.

A go program comprises a set of packages. Each package can include a certain top level declarations, and most importantly, function definitions. Functions are essential components of Go programs. We will get back to the topic of *functions* in later lessons.

Empty lines, such as the second line of this example code, are ignored by the compiler.

1.2.4. Deep Dive

In some program languages, an empty program is a perfectly valid one. In Go, however, an empty program (or, an empty source code file) does not exist.

Every runnable program has to include at least one (and no more than one) main function, which should be part of the `main` package.

If you run the above program, then you'll see the following output:

```
```

That is, *nothing*. The fact that the program runs and does not produce an output does not necessarily mean that the program has done nothing. But, in this case, the program itself does literally nothing.

A Go program can be clearly more complex than this simple example. A program can include many lines of code and it can include a large number of packages and source files, etc. But, there is one commonality across all different Go programs, from the simple to the most complex.

Each Go program starts running from the `main` function. The definition of a function, e.g., a set of statements to be run, is to be included inside a pair of curly braces, `{...}`.

In this example, the main function just happens to be empty. We can clearly see that there is nothing between the opening and closing curly braces in the source code. This means that we are telling the computer to do nothing. And, it will do nothing, if it runs successfully (other than the basic work done by the operating system to load the program into memory, etc.).

1.3. Summary

We introduced the important concepts of packages and functions in this lesson. They will be discussed in more detail in later lessons.

1.4. Questions

1. What is the first line of a Go program source file?
2. What is a `main()` function?

Author's Note

Development Environment Setup

We are not going to discuss how to install Go tools in this book. That is a "trivial" (albeit important) task, which you can figure out using various resources (e.g., Web search).

Having a Go development environment on your computer is not a prerequisite to reading this book.

If you haven't installed the Go tools, and if you would like to do so, then there is a quick instruction on the golang website: golang.org/doc/install.

You can also use the online Go Playground [https://play.golang.org/] to try out simple code.

Most of the sample code in Part I that fits, or can fit, into a single source file can be run on the Playground.

There are many different types of resources that can help you learn programming in Go. ***The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate*** is designed to be *read.* This book is not meant to be used as a stand-alone be-all and end-all book.

What is Programming?

This section is for readers who have never done programming before. You can skip this section if you have some (hands-on) experience with programming.

If you are completely new to coding, then it can be rather daunting. You may not be sure what we are doing here.

A program is essentially a series of instructions that a computer is to read and execute. The problem is that computers do not understand human languages like English.

We use specially created programming languages, like Go, to write programs. Computers, however, do not understand any of these programming languages either. We will have to go through a certain process to convert a program written in one of these "high-level" programming languages to an instruction set in the language that a particular computer understands, which is essentially in `0`'s and `1`'s.

This process, or at least the most important part of this process, is known as compilation. The details of this process varies from language/runtime to language/runtime.

Now, focusing mostly on the Go programming language, the Go compiler toolchain converts a "program" (e.g., a set of source code files, as we will discuss in more detail in the coming lessons) into an executable for a particular computer architecture.

By default, the executable is generated for the architecture of the computer where the build is done. But, you can also "cross build", that is, you can build executables for different target architectures.

This is one of the most interesting things about Go. Other high level programming languages comparable to Go such as Java or Python compile the source program

into "byte code" or "intermediate code", not directly to the machine code. These byte code run on a virtual machine, or an intermediate language runtime or interpreter.

Go does not have such virtual machines or runtimes. Go code is always compiled to an executable of a target architecture(s).

The "Go runtime" is embedded/linked into each executable/binary. The Go runtime includes a "garbage collector".

When a program runs, it uses/consumes various host computer resources such as memory. Some of the allocated memory may have to be cleaned, while the program is running, because they are no longer used. Otherwise, the program may run out of usable memory space. This is especially important for long running programs. The Go runtime periodically checks the memory and cleans up the space. This is called *garbage collection.*

To sum, a programmer creates a program, a set of source files, which is written in the Go programming language. The programmer then compiles/builds, or converts the source program into a binary/executable for a target computer architecture. The executable, or "the program", generated this way can be distributed and "installed" on other computers with the same architecture.

"Programming" may broadly refer to this whole process of creating an executable(s). Or, more typically, programming refers to a process of writing these source code, which are to perform a certain desired task, when compiled and built into a machine code.

A program may include various instructions (or, "statements"). It may include conditional statements and repetitions, or "loops", among other things.

Programmers use various tools to do programming. For example, they may use text editors or IDEs (integrated development environment). They also need to use certain tools, e.g., "compilers", that convert the given set of source files into a binary. When programming in Go, we use a suite of command line tools, "go", that

provide such functionalities. For example, we use *go build* to compile/build a Go source program.

We will look into these concepts in more detail throughout this book.

Lesson 2. Hello World 1

2.1. Agenda

We will review a few simple Go programs which do basic input and output in this and the following few lessons.

2.2. Code Reading

Here's a simple "hello world" program in Go:

hello-world-1/main.go

```
1 package main
2
3 func main() {
4     println("hello world!")
5 }
```

Throughout this book, the label of a source code snippet indicates that the example code is taken from a certain file in a certain

> folder (on the author's computer). The source file names, and the file paths, are largely irrelevant to the execution of a Go program.

2.2.1. Explanation

This program is not much more complicated than the one from Lesson 1. The code includes essentially one more line (excluding braces), `println(…)`. The `println()` function prints out its argument to the console output.

We can run this program as follows:

```
go run main.go
```

It produces the following output:

```
hello world!
```

2.2.2. Keywords

This "hello world" example code in Go includes the following two keywords:

- `package`: The `package` keyword declares a package that this source file belongs to, *main* in this particular example.
- `func`: The `func` keyword generally declares a function and introduces its name and definition into the program. The `main()` function is special in a Go program, and any executable program should include one and only one `main()` function.

> As stated, there will be (deliberately) a fair amount of repetitions across different lessons. You can skip any part of the book (not

> just the keyword sections) which you are already familiar with. As for the keywords, there is an appendix that includes all Go keywords, [appendix-section-go-keywords], at the end of the book.

2.2.3. Built-in Functions

`println()` (with a lowercase `p`) is a "built-in" function.

It takes one or more string arguments (e.g., `"hello world!"`), and it prints out the arguments to *stdout* (e.g., the console or terminal output). A newline is automatically added after the argument is printed (as the suffix `ln` indicates).

The Go programming language includes a few built-in functions which you can use in your programs without having to refer to any particular libraries.

Builtin functions may be considered "more important" or "more essential" in writing a program. The `println()` function is, however, an exception. There are "better" functions in other libraries that do what `println()` does, and more. This built-in function may be deprecated or removed in the future releases of Go (although unlikely).

This is the only lesson in which we use `println()` in this book.

2.2.4. Grammar

A Go program is organized into one or more packages. A `package` in Go is a fundamental building block, which plays an essential role in many different aspects of the language.

A package can be written in one or more source code files. Each source file must start with the package declaration (excluding white spaces and comments):

```
package <package_name>
```

This statement means that this source file (in which this source code is written) belongs to the named package.

In this example, the package name happens to be a special word "main", indicating that this source code file is part of the special *main* package of this program. Any executable Go program must include one, and only one, main package.

A package can include zero or more function definitions, among other things. A runnable Go program must include a special `main()` function as part of the main package. When the program is executed, the operating system invokes the `main()` function of the Go program as an entry point. The main function has the following syntax:

```
func main() {
    // A series of statements goes here
}
```

The middle line is a program comment, which is included here for illustration. Comments are (mostly) ignored by the compiler. If you are new to programming, the important part of this syntax is the keyword `func`, `main`, `()`, `{`, and `}`, in this particular order.

The body of a function, from right after the opening bracket “{” to just before the closing bracket “}”, can include any number of statements. A "statement" in a program is an instruction to the computer as to what needs to be done.

The main function of the example program above includes one statement (one more than the first example),

```
println("hello world!")
```

It is a "function call". This statement is an instruction to "call a function", the builtin `println()` function in this example, with an argument `"hello world!"`.

The text `"hello world!"` (including the opening and closing double quotes `""`) is an example of a "string literal".

As stated, we will briefly introduce certain concepts or topics in earlier lessons and elaborate on them later in the subsequent lessons. Many of the explanations given in this book, especially those in the earlier lessons, are, by necessity, incomplete.

2.2.5. Deep Dive

In some programming languages, it requires only one line of code to print "hello world" (or something similar) to the console.

In Go, it requires 4 or 5 lines (depending on how you define a "line"). Different programming languages have different tradeoffs in their language designs.

A Go program starts when the system calls the main function. The Go program executes the statements in a function from top to bottom, more or less. (Not all of them may end up being executed, however.) When a function has no more statements to run, the function returns. When the main function of a program has no more statements to run, the program exits.

In this example, the program has only one function, the `main()` function, and the function includes only one statement `println("hello world!")`, which is a function call.

When the `println()` function does its job (i.e., printing "hello world!" to the

standard output), it returns to the caller, which is the `main()` function. Since there is no more statement after `println()`, the program terminates.

Author's Note

Why Hello World?

Doing "Hello World" is now almost a rite of passage for beginning programmers.

So, why do we do it? One of the important roles which this type of simple programs play is, in fact, to verify your development environment setup.

Suppose that you have just installed the go tools from the golang.org website. How do you know that they "work"? How do you know that your installation was successful? A quick way to test the dev env setup, including the build tools, is to test the build system using a simple program.

That's where the Hello-World program comes in. When you install the Go tools (if you haven't already done so), try to build and run your hello world program, and make sure that it compiles and runs.

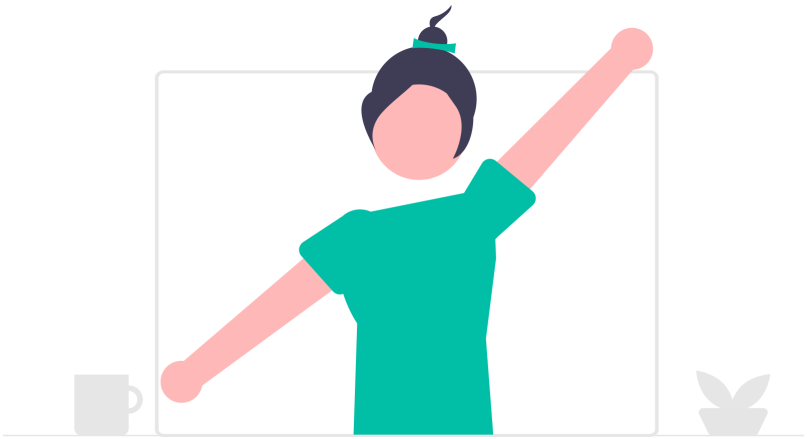
This is like a chicken-and-egg problem. ☺

If you don't have the tools, you cannot create a program. If you don't have a program, you cannot test the tools.

Lesson 3. Hello World 2

3.1. Agenda

We will review another simple Go program in this lesson.



3.2. Code Reading

Here's a slightly more complicated version of the "hello world" program.

hello-world-2/main.go

```
1 package main
2
3 import "fmt"
4
5 const name string = "Joe"
6
7 func main() {
8     var greeting string = "Hello"
```

```
 9     fmt.Println(greeting + " " + name)
10 }
```

This source code file has the same name "main.go", but it is stored in a different directory, "hello-world-2".

The file name "main.go" has no special significance. A source file that includes the main function of the program is typically, but not always, named "main.go".

3.2.1. Explanation

This example program includes a bit more components, and a few more lines of code. You can run the program as before:

```
go run main.go
```

It produces the following output:

```
Hello Joe
```

3.2.2. Keywords

This program includes three new keywords:

- `import`: `import` declares that the source file depends on the functionality of the imported package, which is specified by an identifier for finding and accessing the package.
- `const`: `const` declares a list of constant names. The const declaration binds those names to the values of a list of constant expressions. The number of identifiers must be equal to the number of expressions.

- `var`: `var` declares one or more variables. The var declaration binds the given identifiers to those variables, and gives each a type and an initial value.

3.2.3. Grammar

In Go, `packages` serve as basic components for code sharing. If you know how to find a package anywhere on the Internet, and if you have permission to access the source code of the package, then you can use it in your program, or more precisely in your package.

In many programming languages, code reuse is often based on special constructs like "libraries" (e.g., C/C++, ...) or "packages" (e.g., Node.js, Python, DotNet, ... not to be confused with Go `package`s), or just simple archive files like "jars" or "wars" (e.g., Java).

In Go, no special "packaging" is needed. If you know how to access the source code of a package, then you can use it.

Likewise, if anybody knows where you keep your source code (of a certain package), and if they have a permission to do so, then they can use your code (of that package).

We will discuss this code sharing aspect of `package`s further throughout the book, but one thing to note here is that you cannot share your `main` packages. The sole purpose of the `main` package, along with the `main` function, in a program is to make the program executable. The `main` packages cannot be shared with other Go programs.

Go comes with a set of special packages known as the "standard library" (just like any other programming language). The "fmt" package is one of them. And, the `import "fmt"` statement of the example code lets you use the fmt package in your code.

The Go language specification does not specify, or dictate, *how* exactly you specify

the location of somebody else's `package`s.

In the case of the standard libraries, you simply use the name of the package in the `import` declaration. The go compiler knows where to find them.

```
import "<package_name>"
```

The pair of double quotes around the package name is part of the syntax.

Once you "import" an package into your program, you can use the imported package just like it is a part of your own program. (We will cover the access control aspect of a `package` in later lessons.)

The `import` statement introduces the names, such as those of functions or other declarations, in the package into your program so that you can use them.

Line 9 of the "hello-world-2" example shows how to use `Println()` function from the "fmt" package. You just use the imported package name as a prefix with a dot, ".".

```
fmt.Println("Hello!")
```

The `fmt` package's `Println()` (with a capital `P`) is very similar to the builtin `println()` function. It prints out its string arguments to `stdout`.

In the example code, the argument happens to be an "expression", `greeting + " " + name`.

An expression is a fancy term for a "value" (as the compiler sees it) or anything that evaluates to a value. `"Hello!"` (a string literal) is a value, and hence it is an expression. A number `5` is an expression as well as `2 + 3` since it evaluates to `5`, which is a value.

In this example, the argument of the `Println()` function is a string concatenation (denoted by "+"), which evaluates to a value, i.e., another string. For example, `"hello" + "world"` is `"helloworld"`.

The expression, `greeting + " " + name`, includes two other Go programming constructs. Namely, "constants" and "variables".

In this case, the name `greeting` is a variable, as declared by the `var` keyword in line 8.

```
var greeting string = "Hello"
```

And, the name, `name`, is a constant, as declared by the `const` keyword in line 5.

```
const name string = "Joe"
```

Both declarations have more or less the same syntactic structure.

There is an equal sign `=` in the middle. On its left hand side, there is the keyword `var` or `const` followed by a name, or an "identifier", and a "type", `string` in this case. Note that, in some languages, the type comes before the identifier, and in other languages, the order is reversed.

On the right hand side of `=`, there is an expression, string literals in both cases.

In this particular example, `const` and `var` declarations are placed in different places, one inside a function (the `main()` function in this case) and the other outside a function. But, that is just incidental.

Both `const` and `var` declarations can be used within a function or outside. Their placement affects the constant/variable's "scope". "Scoping" is a big topic, and we will cover scoping throughout this book.

The difference between `const` and `var` is that `const` can declare names which do not change during the execution of a program. In fact, the value of a `const` name should be known at compile time. The right hand side of the `const` declaration should be a "constant expression".

On the other hand, the value of a `var` name *can* change. One can "assign" a different value, or it can change as a result of other operations.

Go is a statically typed language. All `const`s and `var`s have specific types, known to the compiler, at compile time (there are exceptions), and their types do not change during the execution of a program.

Clearly, `type` is a very big, and important, subject, and we will have to defer its full coverage to later lessons.

Just to give a quick explanation, however, a `type` defines what a const/var is, how to interpret its values in memory, and what kind of values the const/var is allowed to have, among other things.

Not to repeat the same points, but whatever doesn't make sense to you at this point, you can just ignore and move on. You can always come back later, if necessary.

3.2.4. APIs

Being familiar with common libraries, especially the standard libraries, is an important part of becoming a proficient programmer in a given programming language.

Although it is not a main focus of this book, we will try to touch as many standard library functions and types as possible, and we will document some of them in these special sections, "APIs".

In the example "hello-world-2", we use package "fmt", and one of its exported functions, `Println()`.

- Package fmt [https://golang.org/pkg/fmt/]: Package `fmt` implements formatted I/O with functions analogous to C's printf and scanf. The format "verbs" are derived from C's but are simpler.
  - func Println [https://golang.org/pkg/fmt/#Println]: `Println` formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended.

This information is taken from the official Go documentation pages.

3.2.5. Deep Dive

The example code, however simple it may be, illustrates one of the most important structures in a Go program.

A Go source file follows this structure:

1. `package` declaration (line 1),
2. one or more `import` declarations (line 3), if needed, and
3. "the rest" (lines 5~10).

Every Go source file has to follow this structure, *in this particular order*.

We have not fully explained what you can put in "the rest" part. But, we have seen some examples like function definitions. You cannot put arbitrary statements like `fmt.Println(…)` here. Many of the general kind statements are only allowed inside a function definition.

Other types of statements that can be put at the top-level, or in the "package scope",

include `const` and `var` declarations, as demonstrated by the example code.

As indicated, both kinds of declarations could have been put in the package-level scope. Or, both inside the `main()` function ("function scope").

A name can be used only within a valid scope, that is, (typically) within a block where the name is introduced, including its inner blocks, if any. In case of a const/var, its scope starts from the point where the name is first introduced and it ends at the point where the enclosing block ends. The "package scope" is a little bit different from other scopes (e.g., functions, blocks), but essentially the same rules apply. Any name that is included outside a function in a source file is in the package scope (which includes all source files within the same package).

As an example, the following code will be valid. The const `name` can be used within the `main()` function, or outside.

```
const name string = "Joe"
func main() { /* ... */ }
func anotherFunction() {
        fmt.Println("Again, my name is", name)
}
```

On the other hand, the variable `greeting` is in the main function's function scope. After the closing bracket `}` of the function, it cannot be used. In fact, this variable cannot be used before the declaration even within the same block. In this small example, there is nothing else before the declaration (line 8), but even if there were any statements, they could not have used `greeting`. In case of variables, the "scope" starts from the line where the variable is declared.

All C-style programming languages have more or less the same scoping rules.

In Javascript, if you have used Javascript before, one can declare

> variables in two different ways, using `var` and `let`. (Or, one can just declare variables globally, which is not generally recommended.) Variables declared with `let` follows the same scoping rule, whereas those declared with `var` have slightly different scopes. They can be used even before their declarations. Their values are `undefined` if they are accessed before their declarations.
>
> In Go, trying to use a const/var before its declaration will cause a compile time error.

Note also that we could have used either `const` or `var` for both `name` and `greeting`, in this particular example.

As a program gets larger and more complex, the choice of `const` vs `var`, and their scope, will be important. As a general rule, it is a good practice to prefer constants to variables whenever it makes sense. It is also a good practice to declare consts/vars within the smallest possible scope.

In this particular example of "hello-world-2", both names should have been `const` and both should have been declared within the `main()` function before we use them (e.g. before `fmt.Println()`).

But, choice of `const` and `var` also conveys certain information (to the compiler as well as to the human readers).

For example, suppose that your name happens to be "Joe". This fact will not change even if the program grows bigger and if the program ends up including more functions, etc. Putting the `name` name in a package scope as `const` would make more sense in that case.

On the other hand, suppose that your intention was trying out different greeting phrases within the program. In such a case, it would make sense to make `greeting` a variable. We could assign a different value to it, if needed.

For example,

```
var greeting string = "Hello"          ①
fmt.Println(greeting + " Joe!")        ②
greeting = "Hi"                        ③
fmt.Println(greeting + " Joe!")        ④
```

Many of the code snippets we use in this book may not be a complete program, as in this example. This should be obvious from the context or from other cues. For instance, this code snippet does not follow the general structure of a Go source file, and hence it cannot be a complete program, or even a package.

① is the now-familiar `var` declaration.

② the `Println()` will print out *Hello Joe!*

③ is an "assignment" statement. The assignment changes the value of the variable.

④ the output will now be *Hi Joe!*.

The first line of the code snippet declares a variable, `greeting`, and it initializes its value to `"Hello"`. One can also just declare a variable first and assign a value later in two different statements.

```
var greeting string
greeting = "Hello"
```

If a `var` is declared without an explicit initial value, then the type's "default value", or *"zero value"*, is automatically assigned. In the case of the `string` type, the default initial value is an empty string (`""`).

When a const/var is declared with an initial value, as in our lesson example, the

compiler may be able to easily infer the type of the constant/variable. In that case, the type specification in a const/const declaration may be omitted.

For example,

```
var greeting = "Hello"
```

The type of the variable `greeting` can be easily inferred, say, based on the type of the expression/value on the right hand side, i.e., a string literal in this example. Hence the variable `greeting` is of type `string`. This is called "type inference", and the type is normally omitted in const/var declarations unless necessary.

In the example program, the `fmt.Println()` function takes a single argument, which is a string concatenation of three strings. We could have assigned the result of the concatenation to a new variable and used that result as an argument to `Println()`.

For example,

```
var helloGreeting = greeting + " " + name
fmt.Println(helloGreeting)
```

There is really no difference. It's very likely that the compiler will generate more or less the same code.

In simple cases like this, it's just a matter of taste. As a program grows, there might be other requirements/constraints to consider and one style may be preferred to the other.

In this particular example, the variable `greeting`, which is `var` not `const`, may be reused for this purpose. For example,

```
greeting += " " + name
fmt.Println(greeting)
```

The `+=` operator is a shorthand for

```
greeting = greeting + " " + name
```

In the assignment statement in Go, the right hand side is always computed first before the assignment. Hence, when the right hand side expression is evaluated, the `greeting` variable holds the value `"Hello"`. After the assignment, however, `greeting` becomes `"Hello Joe"`, the concatenation of all three strings, including the "old" `greeting`.

Most functions take a fixed number of arguments, including zero. For example, the `main()` function takes no arguments. You cannot pass an argument of any kind.

The `fmt.Println()` function (as well as the builtin `println()`) takes a variable number of arguments. From zero to as many as you would like.

Another way to print out the desired text to the console, therefore, is something like this:

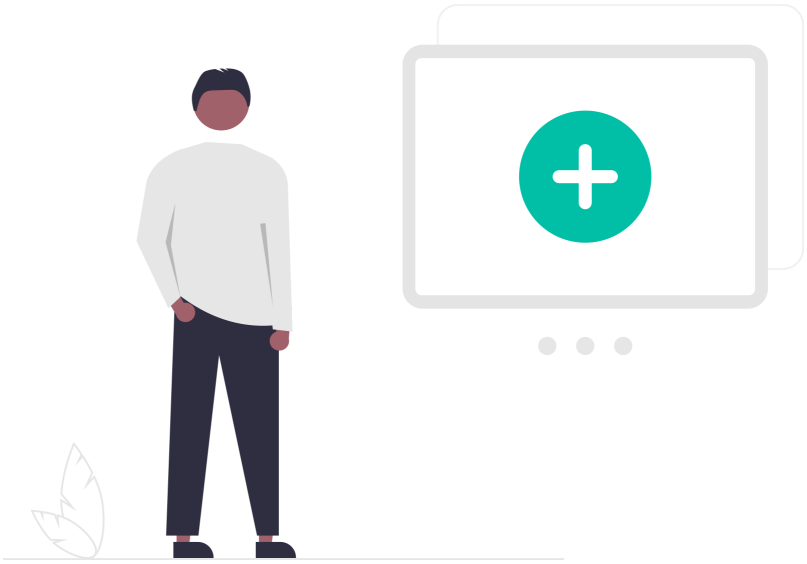
```
fmt.Println(greeting, name)
```

A space `" "` is automatically added between the arguments in case of `fmt.Println()`, and hence the output will be the same as the original, that is, *Hello Joe*.

Lesson 4. Hello World 3

4.1. Agenda

One more "hello world" program in Go.



4.2. Code Reading

Now we are moving beyond a simple "Hello World" program. (Well, just a little bit.) Let's take a look at the following sample code.

hello-world-3/main.go

```
1 package main
2
3 import (
4     "fmt"
```

```
    "os"
)

func main() {
    var greeting = "Hello"
    var name string

    if len(os.Args) > 1 {
        name = os.Args[1]
    } else {
        name = "you"
    }
    greeting += " " + name

    fmt.Println(greeting)
}
```

4.2.1. Explanation

This program takes an (optional) argument. If you run the program in the usual way,

```
go run main.go
```

It produces the following output:

```
Hello you
```

If you run the program as follows, with an extra text "Joe" in the command line,

```
go run main.go Joe
```

It produces a different output:

```
Hello Joe
```

4.2.2. Keywords

This program includes two more new keywords which we have not discussed yet.

- `if`: The `if` keyword is used to create a conditional statement. An `if` statement has a Boolean expression and one or two branches of execution. If the expression evaluates to true, then the `if` branch is executed. Otherwise, if the `else` branch is present, then that branch is executed.
- `else`: The `else` keyword is used to define an (optional) branch, which is executed when the `if` expression evaluates to false.

4.2.3. Built-in Functions

`len()` is a builtin function, which takes one argument. It returns the length of its argument based on its type.

As we will discuss shortly, in the case of an argument with a "slice" type (which happens to be the type of `os.Args`), `len()` returns the number of elements in the slice. If the argument is `nil`, then the length is zero.

All builtin functions are listed in the Appendix, [appendix-section-builtin-functions], at the end of the book.

4.2.4. Grammar

This example program introduces a few new constructs of the Go programming language.

The keywords `if` and `else` are used for a conditional statement. `if` is followed by a Boolean expression (i.e., an expression that evaluates to a `bool` value, `true` or `false`) and a "block" (from the opening bracket `{` to the closing bracket `}`).

If the Boolean expression (`len(os.Args) > 1` in this case) is evaluated to `true`, the statements in this block is executed, if any.

An optional `else` can be used, as in this example. The statements in this block is executed if the value of the Boolean expression is `false`.

For example, in the following example,

```
if false {
    fmt.Println("hello")
    fmt.Println("joe")
}
```

None of the strings will be printed. Neither "hello" nor "joe".

Go's `if-else` statements are similar, or equivalent, to those of other C-style languages.

One thing to note is that Go has a particular set of formatting rules. Note that the Boolean expression is not enclosed in parentheses (`()`) as in most other languages. The brackets are required even if there is only one statement in the block. The opening bracket (`{`) must be in the same line as the end of the Boolean expression. The `else` keyword, if present, must be in the same line as its opening bracket and `if's closing bracket, as shown in the example code of "hello-world-3".

This formatting rules are not limited to `if-else` statements. We have not mentioned it yet, but functions have to follow certain formatting rules as well. By the way, where are the semicolons (`;`)?

These rules can be confusing, especially to programmers coming from other C-style language background. Not just the specific rules but the very fact that these rules exist and that they are tightly integrated into the language (or, more precisely, into the compiler tool chain). We will get back to this topic later in the book, but for now, we learn by examples.

All operating systems allow passing some kind of arguments to a starting program. These are generally known as the "command line arguments". The C programming language started using a convention where the command line arguments are passed in to the `main()` function as the function's parameters.

The exact function signature is not important, but C's `main()` function accepts the command line arguments (say, multiple arguments separated by space) as a list of values or, an "array". Most C's descendant languages follow this convention, with minor variations.

Go does it slightly differently. Instead of using the `main()` function's arguments, Go stores the command line arguments into global variables when the program starts up.

This is where `os.Args` comes in. `os` is a standard library package, and it defines a package-level variable `Args` (just like we do with `var` or `const` in our programs).

`os.Args` is of a type "slice" of `strings`. We will discuss the slice type in more detail in later lessons, but for now it suffices to say that a single variable `os.Args` slice can store a list of values, strings in this case.

By the C convention, the first element of `os.Args`, `os.Args[0]` using the index notation (0-based), is the name of the program (used to start the program).

Go is a compiled language. The source code (from one or more packages, each of which can comprise one or more source files) is first compiled into a "binary" (in the languages that computers of a particular architecture understand). Then you can run the generated binary/executable (just like any other programs, or "apps", on your system).

When we do *go run main.go*, it is a shortcut provided by the `go` tool. In fact, this command does a two-step task: First, it compiles *main.go* into a binary (and stores it in a temporary location), and then executes the binary as if the program is being run from the current directory.

Normally, the "build" step is done by the *go build* command.

```
go build main.go
```

Running this command, when successful, generates an executable program, or a binary. If you do *ls -l*, then you can see an output like this:

```
total 1904
drwxrwxr-x  2 harry harry    4096 Apr 14 10:04 ./
drwxrwxr-x 35 harry harry    4096 Apr  5 17:57 ../
-rwxrwxr-x  1 harry harry 1933164 Apr 14 10:04 main*
-rw-rw-r--  1 harry harry     216 Apr 14 08:22 main.go
```

This particular output is taken from a Unix/Linux shell (in particular, BASH), but the explanation given here, and throughout this book, is not specific to a particular platform.

As you can see, there is an executable program named *main*, which was generated by the `go build` command. The name of the program *main* was taken from the file name "main.go". If you use a different file name for a source file that includes the

`main()` function, then go-build will use that name as the executable name by default.

For example, if we change the name "main.go" to "hello.go", the executable will be named *hello.*

```
go build hello.go
```

If you run the program as follows (e.g., in Bash shell):

```
./hello
```

It produces the same output as that from *go run hello.go*:

```
Hello you
```

Now, if you pass a command line argument, say, "Joe":

```
./hello Joe
```

It produces the same output as that from *go run hello.go Joe*:

```
Hello Joe
```

What happens if we run this program with more than one arguments? Say, how about *./hello Joe and Jill*? It will produce the same output, `Hello Joe`.

The `os.Args` would be a 4 element slice in this case, `{"./hello", "Joe",`

`"and", "Jill"}`, but our program, *hello* or *main*, ignores most of them and only uses the second element `os.Args[1]`. (Again, arrays and slices are zero-based, meaning that the first element has index 0, as in most C-style languages.)

The sample code of "hello-world-3" includes a builtin function `len()`. The `len()` function can take an argument of different types. In this example, the argument is a slice type, and it returns the number of elements in the slice.

If we run *./main*, then `len(os.Args)` will return 1 (the name of the executable is always the first element, `"./main"` in this case). If we run *./main Joe*, then it will return 2. In case of *./hello Joe and Jill*, `len(os.Args)` will be 4.

4.2.5. APIs

- Package os [https://golang.org/pkg/os/]: Package `os` provides a platform-independent interface to operating system functionality. The design is Unix-like, although the error handling is Go-like; failing calls return values of type `error` rather than error numbers. The `os` interface is intended to be uniform across all operating systems.
    - var Args [https://golang.org/pkg/os/#pkg-variables]: `Args` hold the command-line arguments, starting with the program name.

The API information is taken from the official Go documentation pages.

4.2.6. Deep Dive

You can import more than one packages into your program. This example code of "hello-world-3" imports "fmt" and "os".

According to the Go language specification, you can have multiple import declarations:

```
import "fmt"
import "os"
```

The Go formatter does not like this, however. It is considered a good practice to use a single import declaration with multiple packages.

```
import (
    "fmt"
    "os"
)
```

Note the syntax. It uses parentheses `()` to include one or more packages within a single import declaration.

The two are equivalent (as far as the compiler is concerned), but you will most likely only use the latter syntax.

Each package import spec should be listed in a separate line. You can use this form of `import` (with parentheses) even if one package is imported. As stated, in case of the standard packages, simply the package name is used to specify a package to include.

This code in "main.go" follows the general structure of a Go source file, as it must: the `package` declaration first, the `import` declaration(s), and then the rest, `main() {}` in this case.

Although this is how (a source file of) a Go program is structured, `imports` are generally declared while the program's main part is being written. For example, in line 12 you end up using the "os" package, and hence you will `import` "os" at this point. (Most IDEs can automatically take care of imports without you having to explicitly declare them.)

"IDE" stands for integrated development environment. An IDE is essentially a programmer's editor, in which you can write, test, and debug a program. If you run into the terms you are not familiar with in this book, then the best strategy is that you ignore them, but remember that you have run into something you don't know, and then just continue with reading. You can always do a Web search later when you have a chance.

One other important thing to note about `import` is that the declaration is "file-scoped". This is one of the few places where a source file plays a role. Most of other things that are in the file level are all package-scoped.

The import declarations have effects only within the file in which they are declared. If you need to use the same imported names (e.g., `os.Args`) from a different file, even in the same package, then you will need to use the same import declaration again in that file.

The `main()` function declares a couple of variables, `greeting` and `name`. Note the difference. Both are of type `string`, but `greeting` is explicitly initialized (with a string literal), and hence the type specification is omitted. It is of type `string` and the compiler can easily infer that.

On the other hand, `name` does not have an explicit initial value, and hence its type needs to be explicitly specified. The `name`'s initial value is an empty string `""` because a string type variable's default "zero value" is `""`.

In line 12, the `os.Args` variable (made available via the `import`) is checked, and, if its length is bigger than 1 (this example program does not care how long it is as long as it's bigger than 1), then it uses the second element (which is guaranteed to exist since `len(os.Args) > 1`) as `name` (line 13). If its length is not bigger than 1 (or, if it is less than 1 or equal to 1), then the command line argument `os.Args` is ignored. The `name` variable is assigned a generic "you" in this case (line 15).

It should be noted that which branch of the `if` will be executed cannot be known at this point, just by looking at the source code. The compiler does not know it either.

Which branch will be executed will be determined only at "run time".

Depending on the result of the conditional statement, `greeting` may be *Hello you* or something else, like *Hello Jill*, if the first command line argument (after the program name) happens to be "Jill". This is shown in line 17. The value of the `greeting` variable changes through string concatenation (with itself).

Then, the program prints out the result in line 19 using `fmt.Println()`, and the program terminates.

Note the `if-else` statement (including the variable declaration for `name`) in the program:

```
var name string
if len(os.Args) > 1 {
    name = os.Args[1]
} else {
    name = "you"
}
```

This could have been written as follows:

```
var name string = "you"
if len(os.Args) > 1 {
    name = os.Args[1]
}
```

They are equivalent. Depending on whether a user provides an argument or not, their behavior is exactly the same. Would you prefer one style over the other?

Why?

So, how does a user of this program know that she should provide a command line argument to get the behavior she desires. How does she know how this program works?

One way is to read the source code and understand what it does. Clearly, that is not practical for all but the simplest programs, however. Besides, in many cases, the users will not have access to the program source code.

This is where the "documentation" comes in. All programs (intended to be used by other users) should be documented. All source code that can be shared with other programmers should be documented. All exported names, variables and functions, etc., of a Go package should be documented. Many of the internal implementations should be documented, if necessary, for other programmers who may end up having to read your source code.

We will discuss more about documentation and comments, primarily with regards to the source code, later in the book.

All functions "return" (to whoever has called them). Even the special main function returns (to the operating system or the runtime).

When a function has no more thing to do (e.g., since it has reached the end of the function after executing the last statement, e.g., `fmt.Println()` in this example), it automatically returns.

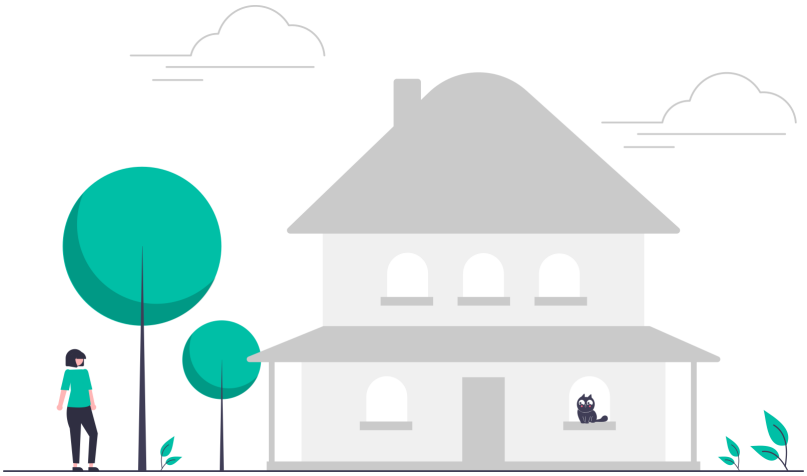
You can explicitly add a `return` statement at the end of a function, but that should not be normally needed, as in this example, unless a function happens to return a value(s). More on this later.

When the `main()` function of a program returns, the program terminates.

Lesson 5. Hello World 4

5.1. Agenda

One last "advanced" version of the Hello World program. ☺



5.2. Code Reading

We will deal with simple input handling in a Go program.

hello-world-4/main.go

```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "os"
7     "strings"
```

```
 8 )
 9
10 func main() {
11     fmt.Println("What is your name?")
12
13     reader := bufio.NewReader(os.Stdin)
14     name, _ := reader.ReadString('\n')
15     name = strings.TrimSuffix(name, "\n")
16     name = strings.Title(name)
17
18     fmt.Printf("Hello %s!\n", name)
19 }
```

Readers are encouraged to read the example source code, from top to bottom, even if it may not make much sense. You will recognize the things that you already know, and the things that you do not. That is a very important part of the learning process.

5.2.1. Explanation

The program asks the user for his/her name, and it uses the name to personalize its greeting.

Let's try running the program:

```
go run main.go
```

It prints out the following "question":

```
What is your name?
```

If you "answer" it with, say, "joe", then it prints out the following personalized greeting to the terminal.

```
Hello Joe!
```

The whole "conversation" may look like this:

```
What is your name?
jill                    ①
Hello Jill!
```

① is the input your provided on the terminal. You type, say, "jill", and press Enter (a newline). Then, the program reads your input and does what it is instructed to do next.

5.2.2. Grammar

This sample code is mostly based on what we have already covered in this book.

There is a 'package' declaration (line 1). There is an `import` declaration (lines 3~8). There is this `main()` function for the program (lines 10~19). There are function calls, that is, calling the functions from the standard library packages (line 11, lines 13~16, and line 18).

The empty lines are ignored by the compiler. They are often added to increase the readability (lines 2, 9, 12, and 17). Sometimes the Go formatter (from the standard Go tool chain) will insert empty lines to separate important parts, if you haven't already done so (lines 2 and 9). The Go formatter does not like more than one consecutive empty lines.

The statements in lines 13 and 14 use a new syntax, known as the "short variable declarations". Instead of the normal equal sign (`=`), these statements use `:=`.

This is equivalent to a combination of a variable declaration (with `var`) and an assignment of an initial value to the variable. This shorthand notation can be rather convenient in many circumstances.

One thing to note is that this syntax can only be used for "local variables", the variables defined within a function or other blocks. You cannot use the short variable declaration outside a function, that is, in the package-level scope.

Lines 13 and 14 are equivalent to the following:

```
var reader = bufio.NewReader(os.Stdin)
var name, _ = reader.ReadString('\n')
```

There is really not much difference. But, most experienced Go programmers prefer the new syntax. You will therefore see more of the short variable declarations in other people's code.

There are a few things to note, however. In this new syntax, you cannot specify a type of a variable (unlike the `var` declaration). And, related to this, you have to provide an initial value, which the compiler will use to infer its type, among other things.

There are some gotchas as well if you are not too careful, due to the variable scoping and "shadowing" rules, as we will discuss later.

Line 14 has an interesting syntax, if you are new to the Go programming language. Unlike many C-style languages, a function in Go can return more than one value. The left hand side of line 14 looks strange, `name, _`. The comma in the middle indicates that we are expecting that the function on the right hand side `reader.ReadString('\n')` return two values (not 1, not 3).

The first one of the two return values will be assigned to the newly declared variable `name`.

The underscore “_” in the second position is a predefined identifier, or a "name", that signals to the compiler that even though we receive two values from the function we will ignore the second one. `_` is known as the "blank identifier".

In Go, you cannot declare a variable that is not used in a program. Hence, the blank identifier `_` is needed in this case.

By the way, how do you know that this particular function `reader.ReadString()` returns two values?

One word. *Documentation.*

One of the interesting things about the short variable declaration (`:=`) is that there should be at least one new variable on the left hand side. Not every variable has to be new. Just at least one. The `name` variable in our example satisfies this requirement.

We are calling 6 different functions from 4 different (standard) packages in this small example. Note that their names are all capitalized, `Println()` and `Printf()` from the `fmt` package, `NewReader()` from `buffio`, `ReadString()` on a reader object of a type `*bufio.Reader`, which is returned from the `NewReader()` call, and `TrimSuffix()` and `Title()` from the `strings` package.

In the Go programming language, any variables, constants, or functions (or, types, which we have not introduced yet) of a package that are capitalized are "exported". That means, anyone who knows how to find, and who has access to, the package can use these variables, functions, etc.

On the other hand, none of the names that are not capitalized is exported. These variables, functions, etc. can only be used within the package where they are declared/defined.

Just to be clear, this is not a "convention". It is part of the Go language grammar.

By convention, Go programmers generally use *PascalCase* for exported names and *camelCase* for non-exported names, which satisfy the requirements. (Both PascalCase and camelCase capitalize each word in a name. In PascalCase, the first word is capitalized whereas the first letter in a name in the camelCase style has a lowercase.)

In line 15, a string literal `"\n"` is used as one of the arguments to `strings.TrimSuffix()`. `"\n"` is a one letter, or character, string. The character happens to be a newline. Character representation like this `\n` is known as an "escape character". Although there appears to be two characters (`\` and `n`), it really represents one character to the compiler, which could have been unrepresentable otherwise.

A pair of double quotes (`"` and `"`) is used to represent a string literal, as in the example of line 15. Line 14, however, includes a newline escape character in single quotes (opening `'` and closing `'`). `'\n'`, or `'a'`, is a byte literal (or, more precisely, a rune literal). It is a value of type `byte` (or `rune`).

`byte` is one of Go's builtin or primitive types, in particular a numeric type. (It represents a number.) Go has a number of primitive types. Here's a list of integer types.

int8

8 bit signed integer

int16

16 bit signed integer

int32

32 bit signed integer

int64

64 bit signed integer

int

signed integer (architecture-dependent, typically 32 or 64 bit)

uint8

8 bit unsigned integer

uint16

16 bit unsigned integer

uint32

32 bit unsigned integer

uint64

64 bit unsigned integer

uint

unsigned integer (architecture-dependent, typically 32 or 64 bit)

In addition, there are a few more integer numeric types. `byte` is an alias type for `uint8` and `rune` is an alias type for `int32`. A `rune` is more like a character. A rune can represent a Unicode character.

String is one of the more difficult types to deal with in Go. In fact, in any programming languages. In Go, the `string` type has a dual nature. A string can be viewed as a series of `byte`s or as a series of `rune`s.

In fact, a string, or the source code itself, in Go is encoded in UTF-8. A `rune` is a code point in UTF-8 encoding, which uses 1 to 4 bytes to represent a Unicode character.

Every character in ASCII (English) fits into a byte. It is often more convenient to use bytes than runes. The function `reader.ReadString()` of line 14 takes a byte argument to be used as a delimiter. You can view a byte in this context as a

character. In fact, in C and other similar languages, `'a'`, for instance, is a literal of type `char`.

This example code also includes other important concepts such as "pointers" and "methods", etc. in the Go programming language. They will be explained in later lessons. Knowing those concepts is not required to understand this code.

5.2.3. APIs

We have seen `fmt.Println()` before. In addition, this program includes `fmt.Printf()`.

- func Printf [https://golang.org/pkg/fmt/#Printf]: `Printf` formats according to a format specifier and writes to standard output. It returns the number of bytes written and any write error encountered.

In our example, `fmt.Printf("Hello %s!\n", name)`, the first argument `"Hello %s!\n"` is a format specifier. The `%s` is a placeholder, known as a "verb", which will be replaced by a subsequent argument, `name` in our example, when it is printed to the output. The `s` in `%s` comes from `string`. To print an integer number, you use `%d` (presumably from decimal or digit).

The number of the placeholders and the number of the arguments after the format specifier should match. For example,

```
func main() {
    count := 99
    food := "Hamburgers"
    fmt.Printf("I ordered %d %s.\n", count, food)
}
```

The output will be something like this:

```
I ordered 99 Hamburgers.
```

`fmt.Printf()`, and its related functions, is one of the most frequently used functions when you are learning the Go language (and beyond).

Other functions/methods included in this example are as follows:

- Package bufio [https://golang.org/pkg/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.
  - func NewReader [https://golang.org/pkg/bufio/#NewReader]: `NewReader` returns a new Reader whose buffer has the default size.
  - type Reader [https://golang.org/pkg/bufio/#Reader]: `Reader` implements buffering for an io.Reader object.
    - func (*Reader) ReadString [https://golang.org/pkg/bufio/#Reader.ReadString]: `ReadString` reads until the first occurrence of the delim argument in the input, returning a string containing the data up to and including the delimiter. If ReadString encounters an error before finding a delimiter, it returns the data read before the error and the error itself (often io.EOF).
- Package strings [https://golang.org/pkg/strings/]: Package strings implements simple functions to manipulate UTF-8 encoded strings.
  - func TrimSuffix [https://golang.org/pkg/strings/#TrimSuffix]: `TrimSuffix` returns the first string argument without the provided trailing suffix string, the second argument.
  - func Title [https://golang.org/pkg/strings/#Title]: `Title` returns a copy of the string argument with all Unicode letters that begin words mapped to their Unicode title case.

5.2.4. Deep Dive

This program of "hello-world-4" is not much more complicated than the previous one. But, it uses more library functions and other constructs.

This example code reads an input from the console, or "stdin", rather than getting it from the command line argument.

In line 11, it first prints out a question as a prompt to the user. He/she knows at this point that an input is expected, in particular, the user's name. The program then waits for the user input.

There are a number of ways that a Go program can process user input. This program uses an API `bufio.Reader.ReadString()`. In order to use `ReadString()`, an object of type `bufio.Reader` must be instantiated. That is what a package-level function `NewReader()` does from the `bufio` package in line 13. It creates an instance of type `bufio.Reader` and it returns its pointer. We will defer the topic of pointers to later lessons. But, for now you can ignore the difference between value types and pointer types.

It should be noted that the `NewReader()` function takes an argument of type `io.Reader` from the standard `io` package (which need not be explicitly imported in our program). In this example, we pass the predefined variable `os.Stdin` in the `os` package to `NewReader()`. The type of `os.Stdin` is `io.Reader`.

The created object is then assigned to a new variable `reader` in line 13.

The type `bufio.Reader` has a number of functions associated with it. These functions are often called "methods" rather than just functions. And, they are invoked with a slightly different syntax.

We call a package level function with the package name prefix, for example, as in `fmt.Println()`. `fmt` is a package name. `Println()` is a package level function defined in the `fmt` package.

In the case of methods, we use the variable name as a prefix. In `reader.ReadString()` of line 14, `reader` is the name of the variable (of type `bufio.Reader`), as defined in the previous line. `ReadString()` is a function, or a *method,* associated with type `bufio.Reader`.

`bufio.Reader.ReadString()` takes an argument of type `byte`, and returns a string which it has read when it encounters the given byte.

`byte` is another builtin type of the Go language. It literally represents a byte (8 bit). As stated, we can view it as a character in certain contexts.

The statement of line 14, therefore, reads a line of text from `os.Stdin` (a line terminates at a newline, by definition), and it stores its read value (a string) into a variable, `name`.

When `ReadString()` encounters an error while reading the input, it returns the error message as a second return value. For this simple example program, we ignore the error. Hence, the blank identifier (_).

The `strings` package include various helper functions to make it more convenient to deal with strings. `strings.TrimSuffix()` of line 15 removes the trailing newlines from the input, if any. `strings.Title()` of line 16 capitalize the value of `name`, if needed.

Notice the pattern. In each of these operations, the old `name` is being replaced by a new `name`. Since we do not need the old values after transformations, we can reuse the same variable.

Now, the last statement of the `main()` function prints out the personalized greeting to the console, ending with a newline (`\n`).

5.3. Summary

In this lesson and the preceding three lessons, we covered some basics of handling input and output in a Go program.

This kind of user interaction is often known as "command line interface", or *CLI* for short.

Although there are still a lot that we need to learn, these "hello-world" lessons introduce some of the most important and essential concepts of the Go programming language so that you can start programming in Go on your own.

5.4. Questions

1. What is the general structure of a Go program source file?
2. What is the role of a `main()` function in a Go program?
3. How do you use exported variables from another package in your program?
4. How do you use a function imported from another package?

5.5. Exercises

1. Write a version of Hello World program with the following requirements:
    - If the program is run with a command line argument, use it as a name as in "hello-world-3".
    - If a full name is provided ("first_name last_name"), then use the full name (including the space).
    - If the program starts without a command line argument, ask for the user's name as in "hello-world-4".
    - Print "Now <your name> is a Hello World programmer!" to the terminal.

To clarify, if you run your program this way,

```
./my-program John Smith
```

The output should be something like this:

```
Now John Smith is a Hello World programmer!
```

All exercises in this book are optional. In fact, it is best to skip exercises in your first reading, especially, if you are new to programming.

How to Use This Book

The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate is written for broad audience, from absolute beginners to more seasoned developers with experience in other programming languages who want to get a quick taste of Go.

This first part, `First Steps`, in particular, is primarily for beginners who are new to programming, or at least new to programming in Golang. Depending on your experience, you may find some of the lessons too easy or trivial. Or, too slow or boring. You can skip, or skim through, certain portions of the book.

Each lesson starts with a sample code, in the "code reading" or "code review" sections. Some lessons include multiple such sections. Learning a new programming language is not much different from learning a second/foreign language We will primarily emphasize "reading comprehension" skills in this book.

Through gradual exposure to many sample programs, from simple to more complex, you will get familiar with the Go language, without even touching a keyboard.

Here's a good way to use this book. *For each lesson, read the sample program(s) first*. If you understand what the overall program does, and know what each part of the program means syntactically, then you can skip the lesson. Go to the next lesson.

If you are new to programming, however, some parts of the book might be rather difficult or more or less incomprehensible. That is perfectly all right. Read the book through. Then, come back to the beginning, and read the book again. This time, test your knowledge by reading the sample code first. Do you understand what the program does? If so, then you can skip this particular lesson and go to the next one. If not, that's all right. Read the lesson again.

Eventually, you will end up "understanding" all sample code in this book regardless

of where you started in the first place.

Then, this book will have served its purpose.

Lesson 6. Simple Arithmetic

6.1. Agenda

We will learn how to do simple calculations in Go in this lesson.

If you are familiar with basics of programming (in any language), and if you find any of the lessons too slow moving, then you can skip them.

6.2. Code Reading

This sample code illustrates various operations using primitive types.

simple-arithmetic/main.go (lines 1~19)

```
1 package main
2
```

```
 3 import "fmt"
 4
 5 func main() {
 6     str := "go" + "lang"
 7     fmt.Printf("go + lang = %s\n", str)
 8
 9     sum := 1 + 1
10     fmt.Printf("1 + 1 = %d\n", sum)
11
12     diff := int16(5) - int16(2)
13     fmt.Printf("5 - 2 = %d\n", diff)
14
15     prod := 1.0 * 5.0
16     fmt.Printf("1.0 * 5.0 = %f\n", prod)
17
18     div := 8.0 / 3.0
19     fmt.Printf("8.0 / 3.0 = %.4f\n", div)
```

simple-arithmetic/main.go (lines 21~39)

```
21     numer := 7
22     denom := 2
23     quotient, remainder := numer/denom, numer%denom
24     fmt.Printf("%d / %d = %d\n", numer, denom, quotient)
25     fmt.Printf("%d %% %d = %d\n", numer, denom, remainder)
26
27     boolAnd := true && false
28     boolOr := true || false
29     fmt.Printf("t && f = %t; t || f = %t\n", boolAnd, boolOr)
30     fmt.Printf("t || f = %[2]t; t && f = %[1]t; t || f = %[2]t\n",
   boolAnd, boolOr)
31
32     var b1 byte = 0b10  // 00000010
33     var b2 byte = 0b110 // 00000110
```

```
34      bitAnd := b1 & b2
35      bitOr := b1 | b2
36      bitShift := b2 << 2
37      fmt.Printf("b1 & b2 = %08b; b1 | b2 = %08b; b2 << 2 = %08b\n",
38          bitAnd, bitOr, bitShift)
39  }
```

6.3. Explanation

The `main()` function of the program includes a series of statements which perform some basic operations. It prints out the results in various formats using `fmt.Printf()`.

If you run the program as before:

```
go run main.go
```

It produces the following output:

```
go + lang = golang
1 + 1 = 2
5 - 2 = 3
1.0 * 5.0 = 5.000000
8.0 / 3.0 = 2.6667
7 / 2 = 3
7 % 2 = 1
t && f = false; t || f = true
t || f = true; t && f = false; t || f = true
b1 & b2 = 00000010; b1 | b2 = 00000110; b2 << 2 = 00011000
```

6.4. Keywords

The sample code includes three keywords, which we already covered in the previous two lessons.

- `package`: The `package` line should be the first (non-empty) line in any source file.
- `import`: If you use functionalities from other packages, they need to be `import`ed.
- `func`: The `func` keyword is used to declare/define a new function.

`main` is not a Go language keyword although it has a special meaning. In particular, `func main()` is special in Go. Every executable program should include one and only one `main()` function in the `main` package.

As stated, you can find the keyword summary in the appendix, [appendix-section-go-keywords].

6.5. Grammar

A computer is a machine that "computes". At the most fundamental level, computers deal with numbers. Binary numbers.

Everything the computer stores and processes is `0`'s and `1`'s.

So, what does, say, `00110011` mean?

Suppose that we have a series of 0's and 1's in a particular location in memory, for instance. What do these numbers mean?

This is where the "type" comes in. Based on the type of the value at a certain memory location, we can interpret what those numbers mean. If a byte in memory

has a value `01000001` and if its type is `byte`, then it is `A` (or, a number 65). If the byte is a part of a 32 bit integer type value, then it could mean a very different thing. If this byte is a part of a value of a string type, then it could mean `A` or something completely different.

"Types" are of a fundamental importance in programming, especially for "statically typed" languages like Go.

We deal with a number of different primitive types in this sample code.

- String type: Lines 6~7
- Integer types: Lines 9~13, 21~25, 32~38
- Floating point number types: Lines 18~19
- Boolean type: Lines 27~30

As discussed in the previous lesson, Hello World 4, there are a number of different integer types, from `uint8` and `int8` to `uint64` and `int64` as well as machine-dependent `uint` and `int`.

When an integer literal, e.g., a whole number like `125`, is used as an initial value in a variable declaration (without an explicit type specification), the variable is inferred as `int`.

In most cases, without any special requirements, you can just use the `int` type for integral numbers.

Go has two builtin types for floating point numbers.

float32

32 bit floating point number

float64

64 bit floating point number

If a floating number literal, e.g., a number with a decimal point like `12.3`, is used as an initial value in a variable declaration (without an explicit type specification), then the variable is inferred as `float64`.

In an assignment, the types of the left hand side (e.g., a variable) and the right hand side (e.g., an expression) must match. In fact, their types must be identical.

If their types are "compatible" in some way, then a value can be "converted" to a desired type. We will cover the topic of type conversion, or casting, in later lessons.

In a multiple assignment (e.g., line 23), the types of each of the corresponding pair should be identical.

When you need to print variables of different types using `fmt.Printf()`, different kinds of "verbs" are used. For string, it's `%s`. For integers, it's `%d`. For floating point numbers, it's `%f`. For Boolean values, it's `%t`, and for bytes, it's `%b`, and so forth.

There are other rules governing the formatting specifiers of `fmt.Printf()` and the related functions. We will point them out when we encounter them throughout this book.

Go supports C/C++ style comments. `/* ... */` is a multi-line comment. Anything between `/*` and `*/` is ignored by the compiler. `// ...` is a single line comment. Anything after `//` in a line is ignored.

6.6. Deep Dive

The rules of basic operations in Go, such as addition and multiplication, are essentially identical to those of almost all of the C-style languages.

If you have been programming in any of these languages, then there is really not much new to learn in Go, as far as these basic operations go.

One thing to note is that all operands in an expression must have the same type. In

some C-style languages, including C, some implicit "widening" casting is allowed. For example, you can assign a value of `int32` to a variable of type `int64` because any value of type `int32` can be represented by type `int64`.

In Go, there is no implicit conversion. If the types are "compatible" and one or the other's type conversion is considered safe in certain operations, then they still need to be explicitly converted into one type or another to make their types match.

Line 6 of the sample code declares a variable `str` of type string (since the right hand side is an expression involving strings which will evaluate to string). The right hand side is a string concatenation and its value will be "golang". To print the value of `str`, we use the formatting verb `%s` (line 7).

In line 9, since the right hand side is an expression with integer literals, the new variable `sum` is of type `int`. To print the value of `sum`, we use the formatting verb `%d` (line 10).

In line 12, however, since the right hand side is an expression that evaluates to a value of type `int16`, the new variable `diff` is of type `int16`. Here, the integer literals are given specific types (via `type_name()` syntax).

One can also give a particular type to the variable in the declaration. For example,

```
var diff int16 = 5 - 2
```

In this example, `diff` is explicitly declared to be `int16`. Hence, the integer numbers on the right hand side are also assumed to be of type `int16`. To print the value of `diff`, we use the same formatting verb `%d` (line 13).

Line 15 of the example code introduces a floating point number operation. On the right hand side, two numbers are multiplied. Since there is no precise type information from this float number expression, the type of the newly declared variable `prod` will be `float64`, as stated earlier. To print the value of a floating

point number, we use a formatting verb `%f` (line 16).

The statement of line 18 follows the same pattern. The type of `div` is again `float64`. In this case, the evaluated value of the expression on the right hand side is `2.666666666666...`.

For output purposes, we can round off the number at a certain precision. `fmt.Printf()` of line 19 uses a formatting verb `%.4f`. This prints the number down to 4 decimals below zero.

One thing to note is that floating point number calculations on a computer have finite precisions. This is generally true, not just in cases like this example. The float numbers are only approximately correct, and one has to be mindful in floating number calculations.

Lines from 21 to 25 show other integer expressions, namely integer division and modulo operations.

In math, an integer division can produce a float number result, as in *3/2 = 1.5*. In Go, an integer division produces an integer number with the same type as their operands.

Dividing an integer `7` with an integer `2` is an integer `3`, as can be seen from the output of line 24. The modulo operation produces a remainder of an integer division, which is an integer `1` in this particular example.

In Go, one can declare or initialize multiple variables in one statement. For example, in line 23, both `quotient` and `remainder` are declared and initialized in one line. The same holds true with assignment.

```
quotient, remainder := numer/denom, numer%denom
```

In this particular example, this statement is equivalent to two separate statements.

```
quotient := numer/denom
remainder := numer%denom
```

But, in general, this may or may not hold true. Initialization/assignment generally depends on the order of the evaluation. In this two-statement example, `numer/denom` is evaluated first, and then the value is assigned to `quotient`. Next `numer%denom` is evaluated, and its result is assigned to `remainder`.

In multiple variable assignment, all expressions on the right hand side are evaluated first before any assignment. For example, in line 23, both expressions `numer/denom` and `numer%denom` are evaluated before each of their values are assigned to `quotient` and `remainder`, respectively.

In `Printf()` like functions, the format specifier includes a character `%`. Formatting verbs start with this character, and they have special meanings in this context. If you need to print the `%` character itself to the output, it needs to be escaped. Line 25 shows an example. The character `%` is escaped as `%%` in the formatting string.

Statements in lines from 27 to 30 demonstrate Boolean operations. Namely, Boolean AND `&&` (line 27) and Boolean OR `||` (line 28). Boolean operations in Go can be done only with Boolean type variables and expressions. As stated, there is no implicit type conversion in Go, and an integer value, for example, cannot be used as an operand of a Boolean operation.

As the output of `fmt.Printf()` in line 29 shows, `true && false` is `false` whereas `true || false` is `true`.

For completeness, `true && true` is `true` and `true || true` is `true`. `false && false` is `false` and `false || false` is `false`.

The statement of line 30 shows another interesting aspect of the `Printf()`-like functions. So far, we have used the same number of formatting verbs and their value arguments. They match one to one, from left to right, in the order provided.

One can use a square bracket notation to designate a particular argument. For example, `%[1]t` in the example refers to the first value argument (of type `bool`) whereas `%[2]t` refers to the second value argument, which happens to be `boolOr` in this example.

Note that, in this particular example, there are 3 verbs whereas there are 2 value arguments. This would not have been possible with positional matching only.

The final segment of the `main()` function (after the last empty line) illustrates bitwise operations.

First, numbers, or numeric literals, can be represented with decimal numbers or numbers with a different base. In particular, Go supports binary, octal, and hexadecimal number literals. Hexadecimal numbers start with `0x`. For example, `0x1` is 1. and `0x10` is 16 in decimal representation. Octal numbers start with `0` followed by another number, and Binary numbers start with `0b`.

The statements in lines 32 and 33 declare two byte variables, `b1` and `b2`, and initialize them with two numbers, 2 (`0b10`) and 6 (`0b110`), respectively. This could have been written as follows with multiple variable assignment syntax:

```
var b1, b2 byte = 0b10, 0b110
```

Since the values on the right hand side are constant expressions, this statement is equivalent to the the two statements in the example (lines 32 and 33).

The comments in those lines show the bit patterns of each number (since we are going to demonstrate the bitwise operations). Since leading zeros are ignored after the integer base prefix (that is, `0x10` is the same as `0x00010` as far as Go programs are concerned), we could have written it as follows:

```
var b1 byte = 0b00000010
```

```
var b2 byte = 0b00000110
```

Or, even

```
var b1 byte = 0b_0000_0010
var b2 byte = 0b_0000_0110
```

Underscores in integer literals are ignored. It's not entirely obvious from this simple example, but adding underscores in numbers can increase readability and reduce a chance of inadvertent errors.

In this particular example, note that a set of 4 digits in a binary number corresponds to 1 digit in a hexadecimal number.

Lines 34 and 35 demonstrate bitwise AND and bitwise OR operations, respectively. In bitwise AND or OR operations, corresponding bits of the two numbers, or bytes, are operated on independently of the other bits. `1 & 1` results in `1`. Likewise, `1 & 0` is `0`, `0 & 1` is `0`, and `0 & 0` is `0`. Also, `1 | 1` is `1`, `1 | 0` is `1`, `0 | 1` is `1`, and `0 | 0` is `0`.

The statement of line 36 demonstrates a (left) bit shift operation. It shifts each bit of the given byte (the first operand) to the left by the second operand. That is, `0b_1110_0010 << 1` becomes `0b_1100_0100`. Note that it does not "wrap around". Rather, it fills the right-most positions with zeros.

The same holds true with the right bit shift operator `>>`. It just moves bits to the right (with no wrapping).

The resulting bytes are printed with special formatting, `%08t`. This means that the argument type is a byte (and treats it as a byte rather than a number) and use 8 spaces. And, if the string representation occupies less than 8 spaces, then fill the empty spaces with `0`s.

You may think, by now, that there are so many rules in formatting. But you don't have to memorize them all. You can always look them up when needed in the API documentations. You will get more familiar, and they will seem more natural, over time.

One last thing to note from this example is that a statement can be written in more than one lines (as in lines 37 and 38). It seems obvious, but it is not. In Go, splitting a statement into multiple lines has to follow certain (formatting) rules.

On the flip side, in Go, more than one statement is not allowed in a single line, unlike in many other C-style programming languages such as C++, C#, Java, etc. More on this later.

6.7. Summary

We learned how to do basic operations in Go. There are integer and float operations. Boolean operations. And, bitwise operations.

We also learned how to write these values of different types to the console using `fmt.Printf()`.

Author's Note

Who is this book for?

Learning programming is hard. Learning a new language is hard. It's often frustrating. It's sometimes discouraging. ...

This book is written primarily for beginners who have tried, and tried, and given up because it was just too hard.

As stated, Go is probably one of the best languages to learn programming

with. In many ways, Go is a much better choice for beginning programmers than Javascript or Python, or PHP or Ruby.

This book is also written with more experienced programmers in mind who want to learn the Go programming language.

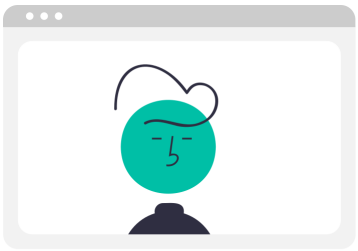
There are a lot of resources, but none really explains well what Go really is. Its apparent similarity to other C-style languages like C/C++, Java, C#, … can be deceiving. If you want to really use Go, then you really have to learn it as a *new* language. Learning just Go syntax will not do. You'll need deeper understanding.

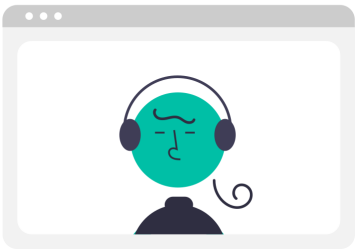
Hope you can find some interesting ideas in this book that will help you become a better Go programmer.

Lesson 7. A Tale of Two Numbers

7.1. Agenda

We will go over a series of small programs to illustrate various essential features of the Go programming language.





7.2. Code Reading - Sum of Two Numbers

two-numbers-1/main.go

```go
package main

import (
    "fmt"
)

func main() {
    num1, num2 := 10, 999
    sum := sum(num1, num2)
    fmt.Println("Sum is", sum)
}
```

```
12
13 func sum(x, y int) int {
14     sum := x + y
15     return sum
16 }
```

7.2.1. Explanation

This program adds two numbers, `10` and `999` and prints out its result.

```
Sum is 1009
```

7.2.2. Keywords

We are now familiar with keywords, `package`, `import`, and `func`. The example code uses one more keyword, `return`.

- `return`: A `return` statement is used in a function to terminate its execution. A `return` statement can optionally provide one or more result values.

7.2.3. Deep Dive

For the first time (in this book), we write our own (non-main) function.

The `func` keyword is used to declare a new function.

```
func sum(x, y int) int { /* ... */ }
```

The function definition (e.g., a series of statements) is included within the pair of curly brackets.

This function `sum()`, unlike the `main()` function, takes two arguments `x` and `y`, and it return a value of type `int`. That is what the "function signature", `func(int, int) int`, indicates. And, its implementation must be compatible with that declaration.

As with `const` and `var` declarations, the types of the arguments are given after the arguments. In this case, both `x` and `y` are `int`, and `x, y int` is a shorthand for `x int, y int`. Whenever there are multiple consecutive arguments in the argument list with the same type, all of them can be denoted with a single type name.

This example code follows the general structure of a Go program source file. The `package` declaration, the `import` statements (if needed), and then "the rest". The rest happens to include two functions in this example, `main()` and `sum()`. The order is largely irrelevant. We could put `sum()` first and `main()` at the end. As far as the Go compiler is concerned, there is little difference.

In some programming languages, most notably C and C++, we have to introduce the name before we can use it. This is called "forward declaration".

In this example, as written, we use the `sum()` function in `main()` before it is declared. In Go, this is perfectly all right.

The `sum()` function could have been put into a different file. Like this:

two-numbers-1/sum.go

```
1 package main
2
3 func sum(x, y int) int {
4     sum := x + y
5     return sum
6 }
```

And, the `main()` function, or any other function in the `main` package, can use the

`sum()` function. There is little difference whether you put function or other declarations in the same file or in different files. As stated, a `package` is the basic unit, not a source file, in Go. (There are some exceptions, however.)

We could have even put `sum()` into a different package. We will cover multi-package programs in later lessons.

The `sum()` function's "body" includes two statements.

```
sum := x + y
return sum
```

The first statement does the addition operation of the given arguments `x` and `y`, and it stores the computed value into a local variable `sum`. Then, in the second statement, the value of `sum` is returned to the caller (the `main()` function, in our example), as we promise in the function signature. For this, the `return` keyword is used.

The `return` statement can just return to the caller (with no arguments), or it can return one or more values (or, "references", which will be discussed further later in this lesson, as well as throughout this book).

In this example, it returns one value, the value of `sum`. It should be noted that we state that the function "returns the value of `sum`", not it "returns `sum`".

Go is a "block-scoped" language. We have not really discussed the important topic of "scoping", but in this example, the lifetime of the variable `sum` is limited to the function block. Or, more precisely, from the time when `sum` comes into existence (the first statement) to the time when the block ends, or the function `sum()` ends, with the closing `}`.

When the `sum()` function returns (the value of) `sum`, the runtime makes a copy of the value of `sum`. And the copied value is used in the context of the caller. In our

example,

```
sum := sum(num1, num2)
```

The value of the right hand side expression `sum(num1, num2)` is the copied value of `sum` (which does not exist outside the `sum()` function). This copied value is used as an initial value for a new `sum` variable, which is a local variable in the `main()` function.

Syntactically, `main()`'s `sum` has nothing to do with the `sum` variable of `sum()`.

Using the same names in different places may sound like a bad idea, which can cause confusion. But, it is natural to do so in many cases. There is no reason to use `var sum1`, `var sum2`, `func addition()` etc., as long as they do not "conflict" with each other. Using the same names in the same scope, or in the nested scopes, can have unintended consequences. One needs to be more careful in those circumstances.

One thing to note is that the `sum` local variable of the `sum()` function is mostly unnecessary, in this example. We could have done away with it:

```
func sum(x, y int) int {
    return x + y
}
```

These two implementations are mostly equivalent. Modern compilers will most likely generate the same machine code from these two different function definitions.

Some people prefer the shorter version, for example, because it's slightly more

concise. Some people prefer the original version, for example, because it's slightly more readable. (That is, because the variable is called a "sum" although the function name "sum()" is a dead giveaway in this particular example.)

In this simplest example, there is really no difference. But, in general, it should be noted that there is always a tradeoff between "simple" and "verbose". There is no absolute rule to prefer one style over the other.

Finally, the program writes the result to the console using the `fmt.Println()` function. As stated, this function can take an arbitrary number of arguments, and it prints out all of them, separated by space.

7.3. Code Reading - Bigger of Two Numbers

two-numbers-2/main.go

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      num1, num2 := 10, 999
9      max := bigger(num1, num2)
10     fmt.Println("Max is", max)
11 }
12
13 func bigger(x, y int) int {
14     if x > y {
15         return x
16     }
17     return y
```

```
18 }
```

7.3.1. Explanation

This program finds the bigger of the two numbers, `10` and `999` and prints out the result.

```
Max is 999
```

7.3.2. Deep Dive

We define a new function `bigger()` using the `func` keyword.

```
func bigger(x, y int) int { /* ... */ }
```

`bigger()` has the exact same function signature as the `sum()` function from the previous example, `func(int, int) int`. It accepts two `int` arguments and returns an `int` value.

The function names or the argument names have no real significance. We could have defined this function as follows:

```
func biggerOfTwo(n1, n2 int) int { /* ... */ }
```

It has the same signature.

This new program "two-numbers-2/main.go" has exactly the same structure as the previous example.

It starts with the package declaration, and then the import statement. "The rest" follows next, which happens to comprise two functions as before. As stated, the order of these functions has no real significance to the execution of the program.

Every Go program starts by executing the first statement of the `main()` function in the `main` package. Other functions are invoked as long as the statements in the `main()` function call those functions, either directly or indirectly.

We sometimes use phrases like a "call chain" or "call stack", in various contexts, to emphasize this aspect of program execution.

A function calls another function. This function calls another function. etc. An invoked function returns to the caller. This caller function in turn returns to its own caller. etc. It all starts from `main()` and ends at `main()` (in a normal program execution).

Go is a function-based programming language. Just like C. (Not to be confused with a functional programming language, however. Go does not support a functional programming style.)

The `bigger()` function looks like this:

```
func bigger(x, y int) int {
    if x > y {
        return x
    }
    return y
}
```

We introduced the `if` conditional statement in an earlier lesson. The if statement in this particular example comprises a Boolean expression (`x > y`), and an `if` block `{ return x }`, but not an `else` block.

If the expression `x > y` holds true for the given two numbers, it returns the value of the first of those two numbers. Otherwise, the `if` block is not executed, and it goes past the closing `}` of the `if` statement.

In this example, that happens to be another `return` statement, `return y`. It returns the value of the second argument, which happens to be equal to the first or bigger.

At first sight, there seems to be an asymmetry. If `x == y`, then it returns `y`, not `x`.

But, that's just because of the way the program is written. We are not actually returning "x" or "y". We are returning the value of `x`, or the value of `y`. When the value of `x` is equal to that of `y`, the statements `return x` and `return y` do exactly the same. They return a value that happens to be the same as `x` or `y` under that condition (`x == y`).

We could have used `x >= y`:

```
if x >= y {
    return x
}
return y
```

There will be no difference in the way this program behaves.

These operators, `>`, `>=`, and `==`, are known as comparison operators. The result of the comparison is a Boolean value. They compare the two given operands and determine the result based on the operator used.

`>` is a "bigger than" operator. If the value of the first operand is bigger than that of the second, it returns true. Otherwise, it returns false. `>=` represents "bigger than or equal to". `==` compares equality. `!=` is the opposite of `==`. It returns true if the values of the two operands are different. It returns false otherwise. `<=` is, in general, the opposite of `>`. It returns true if the value of the first argument is smaller than, or

equal to, that of the second. Otherwise, it returns false. `<` is a "less than" operator.

One thing to note is that, in this particular example, the `if` statement could have been written as follows.

```
if x > y {
    return x
} else {
    return y
}
```

There is semantically no difference between this and the original code in the sample.

Without formally defining these terms, "syntax" has something to do with "forms" and "semantics" is related to "meanings".

Go does not have a "ternary operator" (i.e., `?:`). This `if-else` statement could have been written in one line with a ternary operator.

In effect, the `bigger()` function returns the bigger value between the two arguments. If the two values are the same, then it just returns that value.

The `max` var in the `main()` function is then initialized with this value:

```
max := bigger(num1, num2)
```

At the risk of stating the obvious, values are "copied". Once the returned value from `bigger()` is assigned to the `max` var in `main()`, `max` has the same value. But, otherwise, the value of `max` has no memory, so to speak, as to where it came from.

Values are copied.

In the last line of the `main()` function, the `max` value is printed to the terminal, and the program terminates (because there is no more statement in the `main()` function).

7.4. Code Reading - Difference of Two Numbers

two-numbers-3/main.go

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      num1, num2 := 10, 999
9      d := diff(num1, num2)
10     fmt.Println("Difference is", d)
11 }
12
13 func diff(x, y int) int {
14     if x > y {
15         return x - y
16     } else {
17         return y - x
18     }
19 }
```

7.4.1. Explanation

This program computes the difference between two numbers, `10` and `999` and prints out its result.

```
Difference is 989
```

7.4.2. Deep Dive

Let's define the "difference" between two numbers as the value of a gap between two numbers. That is, the difference between 10 and 50 is 40, and the difference between 50 and 10 is 40.

It is not dependent on the order of the operands, unlike subtraction, for instance.

Based on this definition, we can easily implement a "difference" function:

```
func diff(x, y int) int {
    if x > y {
        return x - y
    } else {
        return y - x
    }
}
```

This function has the same signature as the previous two example functions, that is, `func(int, int) int`. It takes two `int` arguments and returns an `int` value.

If the value of the first argument `x` is bigger than that of the second `y`, the function returns the value of `x - y`, which will be non-negative.

Likewise, if the value of `x` is not bigger than that of `y`, the function returns the value

of `y - x` (through the `else` block), which will be again non-negative.

As in the previous sample function `bigger()`, the apparent asymmetry is incidental. There is no difference whether we use `x > y` or `x >= y` for the conditional expression. When `x == y`, the value of `x - y` is the same as that of `y - x`, namely, `0`.

Note that the Go standard library has a similar function in the `math` package, func math.Abs [https://golang.org/pkg/math/#Abs], which takes two `float64` arguments and returns a `float64` value.

When the `diff()` function returns, it return a value equal to `x - y` or `y - x` depending on their relative size. The value is then copied to a local variable `d` in the `main()` function.

```
d := diff(num1, num2)
```

As stated, although the `diff()` function is declared below 'main()', the `main()` function can still use `diff()` at this point. This is a general rule. Forward declaration is not needed in Go.

The value of `d` is then written to the console using the `fmt.Println()` function, and the `main()` function returns.

Before moving on to the next example, let's take a look at the `if` statement in the `diff()` function's body.

As with the previous example, `bigger()`, it could have been written one of two ways. An alternative would be:

```
func diff(x, y int) int {
    if x > y {
```

```
        return x - y
    }
    return y - x
}
```

In this simple example, these two implementations are equivalent. This is a rather special case. It is a very small function with not much complicated logic, and the `if` branch `returns`.

Now, the question is, which "form" is better, in general? One choice is `if bool {} else {}`:

```
if x > y {
    // Do something
} else {
    // Do something else
}
```

And, the other is `if bool {}; (the rest)`:

```
if x > y {
    // Do something
}
// Do the rest
```

This is really beyond the scope of this book, whose focus is teaching the Go language syntax, but it is important to think about this type of issues when you start learning programming.

Why would you prefer one style over the other? In what situations?

Note that the `if` branch and the `else` branch is "symmetric" in the first case. It can even be written as follows:

```
if x <= y {
    // Do something else
} else {
    // Do something
}
```

The `if` and `else` branches are expected to do more or less the same kind of tasks.

On the other hand, in the `if bool {}; (the rest)` form, the task to be done in "the rest" part might have intrinsically different characteristics than the task to be done in the `if` branch.

7.5. Code Reading - Average of Two Numbers

two-numbers-4/main.go

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      num1, num2 := 10, 999
9      avg := average(num1, num2)
10     fmt.Println("Average is", avg)
11 }
12
```

```
13 func average(x, y int) (avg float32) {
14     avg = float32(x+y) / 2.0
15     return
16 }
```

7.5.1. Explanation

This program computes the average of the two numbers, `10` and `999` and prints out its result.

```
Average is 504.5
```

7.5.2. Deep Dive

Although this code looks rather similar to the previous examples, there are a few differences worth noting.

As before, the `func` keyword is used to define a new function `average()`:

```
func average(x, y int) (avg float32) {
    avg = float32(x+y) / 2.0
    return
}
```

This function's signature is `func(int, int) float32`. Since the average computation may yield a non-integer number even if the two operands are `int`s, the function returns a floating point number.

```
avg = float32(x+y) / 2.0
```

In Go, all operands in an operation must have the same type. `x + y` evaluates to an `int`, and hence it has to be explicitly cast to the desired type, `float32(x+y)`.

The floating point literal `2.0` has no fixed type. Since the type of the left hand side of the division is `float32`, it is treated as `float32`. A mere `2` would have required an explicit casting since none of the integer number types is compatible with `float32`.

Now the computed value of type `float32` is assigned to the var `avg`.

The `average()` function uses a "named return value". Instead of declaring only the return type `float32`, it gives a name `avg` in the function declaration, `(avg float32)`. This variable can be used in a function body.

Then, the return statement does not have to explicitly specify the variable name. Just `return` suffices (instead of `return avg`) since it is already known that it is the value of `avg` that the function is returning.

As before, the function return value is copied to a new variable `avg` in the `main()` function. And, it is written to the terminal.

```
avg := average(num1, num2)
fmt.Println("Average is", avg)
```

This is just an example program for illustration, but if the program's primary purpose was to print out the computed value, we could do away with the local variable `avg`.

```
fmt.Println("Average is", average(num1, num2))
```

As we stated before, this is largely a personal preference in many cases.

7.6. Code Reading - Swap Two Numbers 1

two-numbers-5/main.go

```
 1 package main
 2
 3 import (
 4     "fmt"
 5 )
 6
 7 func main() {
 8     num1, num2 := 10, 999
 9     a, b := swap(num1, num2)
10     fmt.Println("Original:", num1, num2)
11     fmt.Println("Swapped:", a, b)
12 }
13
14 func swap(x, y int) (int, int) {
15     return y, x
16 }
```

7.6.1. Explanation

This program swaps two given numbers, `10` and `999` and prints out its result.

```
Original: 10 999
Swapped: 999 10
```

7.6.2. Deep Dive

This is another simple example to demonstrate features of Go functions. This

function takes two `int` arguments and returns them in the opposite order.

```
func swap(x, y int) (int, int) {
    return y, x
}
```

In fact, this function is so simple that we could have just used it inline. That is, instead of

```
a, b := swap(num1, num2)
```

We could have just done

```
a, b := num2, num1
```

That is exactly what the `swap()` function does. We have seen this before. It is called "multiple variable assignment" (although this example is really multiple variable *initialization*).

This is equivalent to

```
var a, b int = num2, num1
```

It is also equivalent to

```
var a, b int
a, b = num2, num1
```

Now, that is a multiple variable assignment.

As stated, the expressions on the right hand side of a multiple variable assignment statement are evaluated first. They happen to be `num2` and `num1`, in this case. Then the values on the right hand side are assigned to the variables on the left hand side, in the corresponding order. That is, something like `a = num2` and `b = num1`.

Without using the local variables `a` and `b`, the pair `num1` and `num2` can be swapped as follows:

```
num2, num1 = num1, num2
```

Again, it should be remembered that the right hand side is evaluated first *before* the assignment.

Now let's get back to our original sample code, in particular, the `swap()` function. It is equivalent to the following:

```
func swap(x, y int) (int, int) {
  x, y = y, x
    return x, y
}
```

Now let's go through this function in some detail.

First, this function returns more than one value. Two to be exact. We have seen a function like this. For example, `bufio.Reader.ReadString()` returns two values, the second of which is of an error type. This is one of the unique features of the Go programming language.

When the arguments of a function are passed in, as in this example, their values are copied. The caller, the `main()` function in this example, calls `swap()` with `num1` and

`num2`. They are copied to `x` and `y`, respectively.

Conceptually, this is almost like this:

```
x, y = num1, num2
```

Clearly, this is not a valid statement since `x` and `y` are local variables of `swap()` whereas `num1` and `num2` are variables locally declared in `main()`.

As stated, the names are not that significant. The `swap()` function could have used `num1` and `num2` as argument names instead of `x` and `y`.

Then, the values of the variables, `x` and `y`, are swapped in `x, y = y, x`. We have seen this before, but it is worth repeating. Statements like this may look rather strange to you if you have not seen much Go code.

An important thing to remember is that the `y` and `x` on the right side are values. More generally, expressions. On the other hand, the `x` and `y` on the left hand side are variables. It is an assignment.

So, the statement `x, y = y, x` assigns the (old) value of `y` to the variable `x`, and it assigns the old value of `x`, not the newly assigned value, to the variable `y`. Hence the values are swapped.

The `swap()` function then returns the (swapped) values of the variables `x` and `y` to the caller. Again, the values are copied, and `a` and `b` of the `main()` function now, in effect, have the values of the swapped values of `num1` and `num2`.

Since the argument values are copied when the `swap()` function is called, the original values of `num1` and `num2` in `main()` remain intact. When we print out both pairs of `num1` and `num2` and of `a` and `b`,

```
fmt.Println("Original:", num1, num2)
fmt.Println("Swapped:", a, b)
```

Their values are switched. The function does what it is supposed to do, namely, swapping the argument values.

7.7. Code Reading - Swap Two Numbers 2

two-numbers-6/main.go

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      num1, num2 := 10, 999
9      swap(&num1, &num2)
10     fmt.Println("Swapped:", num1, num2)
11 }
12
13 func swap(p, q *int) {
14     *p, *q = *q, *p
15 }
```

7.7.1. Explanation

This program does exactly the same thing as the previous example. It swaps two numbers, `10` and `999` in this case, and prints out the swapped pair.

```
Swapped: 999 10
```

7.7.2. Deep Dive

Depending on your previous experience, this might be one of the most esoteric examples (so far).

Go has a "pointer" type. It is based on the original C's pointer. But there are some subtle, and fundamental, differences.

We will not discuss the "unsafe" features of the Go programming language in this book.

A C's pointer refers to a location in memory, or an address, of a value. The pointers in Go are indeed based on the memory addresses. But, that aspect of pointers are not of much use, at least in the "safe Go". Go does not even allow "pointer arithmetic". (It should be noted that the Go runtime uses garbage collection.)

Instead, in this book, we will present an alternative explanation of Go pointers. We will first introduce some basic concepts in the remaining part of this lesson.

So far in this book, we have only dealt with "values". As stated, values are copied, e.g., when they are passed to a function. Variables of values follow what they call " value semantics".

Everything, every action, that we have seen so far follows the value semantics. In the context of the Go programming languages, the value semantics simply means "copy" semantics. Values are copied.

Not surprisingly, there is another way to deal with values in programming. The C language's pointers more or less behave in this way. It is called "reference semantics".

In many programming languages, all types but a few are reference types, which behave according to the reference semantics.

For example, in Java, all custom types are reference types. The primitive types are the only exceptions, which follow the good old value semantics.

We will look at pointers in Go from this perspective. Go pointers follow the reference semantics. Go pointers are "references".

Going back to the example code, the `swap()` function in this example is defined this way.

```
func swap(p, q *int) {
    *p, *q = *q, *p
}
```

The function's signature is `func(*int, *int)`. It does not return any value.

`*int` represents an `int` pointer type. For a value type, there is a corresponding pointer type. For a value type `T`, there is a pointer type `*T`, which follows the reference semantics.

For a reference, or a variable of a reference type, `p`, `*p` is the value associated with the reference. The `*` in this context is called a "dereference operator".

Now,

```
*p, *q = *q, *p
```

This statement is very much like `x, y = y, x` in the previous example.

On the right hand side, there are two values. `*q` is the value associated with the

pointer `q`. `*p` is the value associated with the pointer `p`. `*q` and `*p` roughly correspond to the `y` and `x` of the previous `swap()` function, respectively.

The `*p` and `*q` on the left hand side, however, are syntactically different. `*X` on the left hand side of an assignment, for instance, means that we are assigning a value to the associated value of the pointer variable `X`, say, rather than assigning the value to the (pointer) variable.

In many cases, we are more interested in the underlying value of a pointer rather than the pointer itself. A pointer, or a reference, is just a wrapper, or a handle, which provides the reference semantics for the associated value.

Now the result of this two value assignment is that the values of the underlying variables end up being swapped.

After calling the swap function,

```
swap(&num1, &num2)
```

The values of `num1` and `num2` are swapped. Note that there are no return values. Calling this `swap()` with references, or pointers, directly changes the associated values of those pointers.

Notice the syntax. The `&` is a reference operator. (It is also known as an "address of" operator in C.) `&num1` is an `int` pointer since `num1` is an `int` type. Likewise, `&num2` is an `int` pointer.

The new `swap()` function of this example takes two `int` pointers. Hence, `swap(&num1, &num2)` is syntactically correct.

This can be written as follows to make it easier to see the types:

```
var p1, p2 *int = &num1, &num2
```

```
swap(p1, p2)
```

Not to make things more confusing, but the passed-in pointer arguments to a function are in fact "copied". The pointer `&num1` of `main()`, for instance, is different from `p` of `swap()`. But they "point" to *the same* underlying value. That is how the pointer arguments to a function can change the content of the values which the pointers are holding, or pointing to.

If you look at the values of `num1` and `num2` at this point in the `main()` function, after calling `swap()`, their values will have been swapped, as can be verified by the `fmt.Println()` statement in the next line.

7.8. Summary

We covered various important aspects of functions in Go in this lesson.

A function can be declared with the `func` keyword. The order or place of a function within a package is not important. A Go function can take zero or more arguments and return zero or more values.

We also introduced pointer types in Go.

Pointers in Go are more closely related to references in garbage-collected languages such as Java and C# rather than to pointers in C and C++. Go's pointers follow reference semantics.

We will often use the terms pointers and references interchangeably in this book.

7.9. Questions

1. What is a function signature?
2. What does a `return` statement do?

3. How do you return multiple values from a function?
4. What is a pointer in the Go programming language?

Author's Note

Who is This Book Not for?

Learning new things, or new skills, requires patience, and perseverance.

If you are only interested in getting some quick knowledge on the Go programming language, then this book may not be for you. There are a lot of books, and other resources, which claim that they can make you a world-class programmer in 7 days, or in 24 hours, or even in 5 hours or less.

This book does not make such promises. This book does not make promises that you will get a job as a Go programmer and make tons of money, after finishing all the lessons in this book.

If you view learning programming, learning a new programming language, as something that you'll have to do to advance in your career, to become rich, etc., then this book may not be for you.

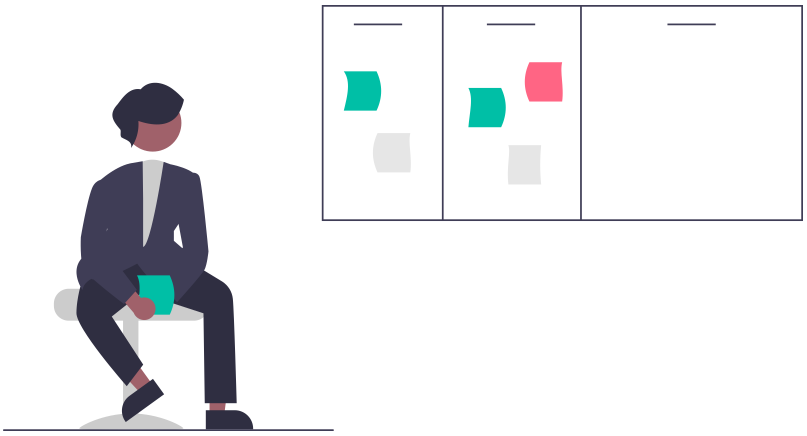
Learning programming can be fun.

Programming can be fun. And, useful. Not just as a career choice. But as a pastime, as a hobby. Just as an endeavor to satisfy your intellectual curiosities.

Lesson 8. Multiplication Table

8.1. Agenda

We will cover in this lesson basics of `slice`s and `for`-loops in Go.



8.2. Code Reading

This program is a little bit longer than those we have seen so far. It includes a few new concepts.

multiplication-table/main.go (lines 1~13)

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 const low int = 2
```

```
 8 const high int = 9
 9
10 func main() {
11     fmt.Println("Multiplication Table:")
12     printMultiplicationTable()
13 }
```

multiplication-table/main.go (lines 15~41)

```
15 func printMultiplicationTable() {
16     axis := make([]int, high-low+1)
17     for i := 0; i < high-low+1; i++ {
18         axis[i] = low + i
19     }
20
21     fmt.Print("        ")
22     for _, v := range axis {
23         fmt.Printf("%4d", v)
24     }
25     fmt.Println("")
26     fmt.Print("       -")
27     for range axis {
28         fmt.Printf("%4s", "--")
29     }
30     fmt.Println("")
31
32     for _, l := range axis {
33         fmt.Printf("%4d", l)
34         fmt.Printf("%4c", '|')
35         for _, r := range axis {
36             m := l * r
37             fmt.Printf("%4d", m)
38         }
39         fmt.Println("")
```

```
40      }
41  }
```

8.2.1. Explanation

We all learned the multiplication table by heart. The code generates a multiplication table within a given integer range.

If you build and run the program from command line, it produces the following output:

```
         2   3   4   5   6   7   8   9    ①
     -  --  --  --  --  --  --  --  --    ②
 2   |   4   6   8  10  12  14  16  18    ③
 3   |   6   9  12  15  18  21  24  27
 4   |   8  12  16  20  24  28  32  36
 5   |  10  15  20  25  30  35  40  45
 6   |  12  18  24  30  36  42  48  54
 7   |  14  21  28  35  42  49  56  63
 8   |  16  24  32  40  48  56  64  72
 9   |  18  27  36  45  54  63  72  81
```

① "Header".

② Divider between the header and the body.

③ Actual multiplication values start from this line.

8.2.2. Keywords

This program includes a few keywords that we have already seen and that we are more or less familiar with by now, `package`, `import`, `func`, and `const`.

It also includes the following two new keywords:

- `for`: The `for` keyword is used to create a compound statement that specifies repeated execution of a block.
- `range`: A `for` loop can be controlled by a `range` clause.

Although it is not a keyword, `nil` is a globally defined constant in Go, which is comparable to `Null` in other garbage collected language.

8.2.3. Builtin Functions

- `make()`: The `make` built-in function allocates and initializes a value of a reference type. The first argument is a type. Its return type is the same as the type of its argument.

8.2.4. Grammar

We have discussed `const` and `var` variables before.

In programming in general, a *variable* is something that holds a value. A `const` can be viewed as a variable as well. In Go, the value of a constant is fixed at build time, and it cannot change.

In some programming languages, there are different kinds of variables, like "immutable variables" and "mutable variables", etc. But, in Go, all variables are mutable. The values of all variables in a Go program can potentially change during the execution of a program, e.g., through an assignment or in other ways. You cannot make Go variables immutable (like a constant).

Using `const` constants is generally preferred over `var` variables whenever possible. But, in Go, you can only use `const` for values that are known at compile time. `const` is limited to number types and `string`s.

Constants and variables can be declared globally, that is, within a "package scope", or in a function or in a block.

The constants of the example code are declared in a package scope, and they are available to any functions or expressions within the main package.

Note that these values are "hard-coded" in this example. If these numbers have to be set via a program's action, e.g., by reading from a user input, then these could not have been declared as `const`.

A `slice` is one of the most important constructs in Go. A `slice` is a collection type, so to speak. A variable of a type `slice` can hold multiple values of the *same* type. A slice variable is declared with the following syntax:

```
var myIntSlice []int
```

The elements, or items, in a slice are ordered, and they can be accessed via an index notation (`[]`). They occupy consecutive spaces in memory.

A slice is defined over an (implicit or explicit) array. An array is a fixed-size type. Once declared, the length of an array type variable cannot change.

A variable of an array type can be declared as follows:

```
var myIntArrayWithSize10 [10]int
```

This declares a variable `myIntArrayWithSize10` with a type `[10]int`. It is a 10-element array with each element of `int` type. When an array variable is declared, all of its elements are initialized by default, 0 in this example.

Note that the size of an array is a part of a type. That is, for example, `[10]int` and `[11]int` are two different types. On the other hand, three is only one `int` slice type, namely, `[]int`. There is only one `float64` slice type, `[]float64`, etc.

A newly declared slice variable with no explicit initialization has a `nil` value. Or, a

slice can be initialized with an empty collection. For instance,

```
var myEmptySlice = []int{}
```

Since the variable's type can be inferred from the right-hand side expression, there is no need to explicitly specify the type (`[]int`) in this case. This statement declares a variable `myEmptySlice` and initializes it with an empty slice (e.g., its length == 0).

The size of a slice can change. A slice can grow as more elements are added to the slice. A slice has two size-related attributes, `len` (length) and `cap` (capacity).

The length is the current size of the slice. The `len` is always less than, or equal to, the `cap` of a slice because a slice may contain an extra "room to grow". The capacity of a slice is essentially the length of the underlying array (implicit or explicit).

It should be noted that a given slice cannot grow beyond its capacity, which is a constant.

The builtin functions `len()` and `cap()` are used to get the length and the capacity of a slice, respectively.

The `slice` type is one of the few "reference types" in Go, which means that variables of type `slice` follow the reference semantics. A slice variable is essentially a pointer to the underlying `array`, which is incidentally a value type.

As the size of a slice grows, that is, beyond the current capacity, for example, the runtime may need to allocate new memory and create a different (and larger) array. A new slice variable may be needed to point to the newly allocated array in such a case.

The values from the old array are copied to the new one, and the old array may be garbage collected unless the array is still being used, e.g., by other slices, etc.

You can initialize a slice with a desired length, and a capacity, using builtin function `make()`. For example,

```
var myByteSlice = make([]byte, 10, 20)
```

The make function on the right hand side accepts 3 arguments, the type of the slice to be created (`[]byte`), the initial length (`10`), and the capacity (`20`). All values of its elements are initialized with the element type's default value, `0b0` in this case.

The capacity argument can be omitted. If the capacity is not provided, the length is also optional. `make([]byte)` will create a byte slice of `len == 0` and `cap == 0`.

All programming languages have constructs for facilitating repeated execution of a statement or a set of statements. It is known as a "loop".

Go supports multiple kinds of loops, with a keyword `for`.

One of them is a "classic" `for` loop. Another is a `for range` loop.

```
const N = 10
for i := 0; i < N; i++ {
    fmt.Println("i =", i)
}
```

This "classic" `for` loop is based on C's for loop. There are three statements after the keyword `for` and before the "for block" (`{ … }`). The first statement is typically used for initializing a "loop variable".

The second statement must be a Boolean expression, if present. As long as this value evaluates to `true` the statements in the for-block will be repeatedly executed. If this expression is missing, then it is considered `true`.

The third statement is executed between iterations. `` ` is an increment operator. Unlike in many C-style languages, Go does not have a prefix version. It only has a postfix version. (The ` `` operator is placed after the variable.) One other thing to note is that `i++` is a statement, not an expression.

`i++` is equivalent to

```
i += 1
```

This is also equivalent to

```
i = i + 1
```

That is, `i++` adds `1` to the current value of `i`, and assigns the result back to `i`. Hence, the name "increment" operator.

Notice that the loop variable `i` is initialized with the "short variable assignment" syntax.

One other thing to note is that, in Go, there is no "comma (`,`) operator". (You can ignore this if you don't know what the comma operator is.) You cannot have more than one statements in each of these three slots. But, you can have an empty statement.

For example, the following is a perfectly valid statement.

```
for ; ; { }
```

In Go, you can even omit all semicolons, like this:

```
for { }
```

There are a few different variations of `for` loops in Go, but syntactically, you cannot have a single semicolon. Either both or none should be present.

Another form of `for` loop is so-called "for range loop". The syntax looks like this:

```
for i, v := range X {
    // Statements here.
}
```

`X` must be a type that supports `range`, e.g., a slice. (And, a `map`, as we will see in later lessons.)

Variable `i` is a (0-based) index. Variable `v` is a value of the elements from `X`.

For example,

```
slice := make([]int, 10)
for i, v := range slice {
    fmt.Println("i =", i, "v =", v, "slice[i] =", slice[i])
}
```

In this example, an `int` slice is used for the range. `i` runs from 0 to 9 in this example, and values of `v` will be all zero because the slice has been only initialized with default values. Likewise `slice[i]` will all be zero through the range of `i`.

One thing to note is that `v` is a "copy" of each element in `slice`. Although the values of `v` and the corresponding `slice[i]` have the same values, they can behave differently.

If an element of slice is a value type as in this case (`int`), you cannot change the values of `slice` by manipulating `v` because it is just copies. That's value semantics.

If the elements are a reference type (e.g., pointers or other slices), on the other hand, then the "copies" may actually be pointing to the same underlying values, according to the reference semantics, and changing the content of `v` may indeed affect the content of the `slice`.

In the range loop, you may end up using only one variables. In Go, unused variables cannot be declared.

In case you are using only one, you can use the blank identifier (`_`) for the other.

For example,

```
for _, v := range X {
    // Use v here
}
```

If you do not use the second variable, you can just omit it. Like this:

```
for i := range X {
    // Use i here
}
```

If you use neither, then you can do away with whole variable declarations:

```
for range X { }
```

Unlike other C-style languages like Java, C#, or JavaScript there

> are no parentheses after the `for` keyword. The braces `{ }` are always required. This is true even when there is only one statement in the block.

8.2.5. APIs

The `fmt` package exports many functions. We have seen `fmt.Println()` and `fmt.Printf()`. This lesson's sample code includes another function `fmt.Print()`. `fmt.Print()` is similar to `fmt.Println()`, but it does not add a newline at the end.

- func Print [https://golang.org/pkg/fmt/#Print]: `Print` formats using the default formats for its operands and writes to standard output. Spaces are added between operands when neither is a string. It returns the number of bytes written and any write error encountered.

8.2.6. Deep Dive

The goal of this program is to print a multiplication table as shown earlier.

You start from the requirements, and go backward. In this case, the desired output is the requirement.

First, we initialize the range (from `low` to `high`). You can use the classic `for` loop, or you can do it using `range` as in the example code of this lesson.

You can assign values to a slice using a loop:

```
axis := make([]int, high-low+1)
for i := 0; i < high-low+1; i++ {
    axis[i] = low + i
}
```

We happen to know the length of the slice which we are going to use, `high-low+1`. Hence we use that information when calling `make()`. (Make sure that you understand why the length is `high-low+1`.)

Go functions are "closures". There is little difference for top-level functions like those from this lesson. (Top level constants/variables are accessible from a function because their scope extends into the function.) But when you have a function defined inside another function, you will have to be mindful of the use of the outer scope variables (e.g., defined in the enclosing function) inside the inner function.

None of the examples in this book uses the "closure" characteristics of Go functions, but the readers are encouraged to explore further from other resources.

Then, we "initialize" the slice with the values from the integer range. All variables in Go are initialized by default when declared, and the loop here is really a series of assignments. But, conceptually, this loop amounts to "initialization".

Once we create this `axis` variable, we can use it for looping in subsequent statements. Using the "classic" `for` loops are generally more error-prone (e.g., because one needs to specify more parameters, etc.).

Using this `axis` slice, we print "headers".

```
fmt.Print("        ")
for _, v := range axis {
    fmt.Printf("%4d", v)
}
fmt.Println("")
```

In this example, we use 4 spaces for the width of each "table cell", as indicated by the formatting verb `"%4d"`.

Then, the "divider":

```
fmt.Print("       -")
for range axis {
    fmt.Printf("%4s", "--")
}
fmt.Println("")
```

Again, 4 spaces per "cell". (We could have used a `const` for this fixed length.)

How does one know if this prints out what one wants? It's generally done via "trials and errors". Print out something first and, based the result, change the formatting slightly, etc.

It sounds tedious. But, a lot of programming tasks involve tedious work in case you are new to programming and have romantic ideas. 😊

Because this printing does not involve any numbers from the `axis` slice, we can just ignore the loop variables. We just use the `for range` syntax in this example.

Finally, the body of the table is printed with "row headers" for each row:

```
for _, l := range axis {
    fmt.Printf("%4d", l)        ①
    fmt.Printf("%4c", '|')      ②
    for _, r := range axis {    ③
        m := l * r
        fmt.Printf("%4d", m)
    }
    fmt.Println("")
```

```
}
```

① "Row header".

② Divider. The verb `"%c"` is for "characters" or bytes in Go.

③ Inner loop where the actual multiplication numbers are computed and printed.

As in most other programming languages, the `for` loops in Go can be nested.

The multiplication table is "2-dimensional". One loop goes over the horizontal axis, or across columns (the inner loop in this example), and the other loop goes over the vertical axis, or across rows (the outer loop in this example).

The value `l` corresponds to a number printed on the left hand side. The value `r` corresponds to a number printed on top. The value of each "cell", at a given row and column, is a product of two numbers, one from the row and the other from the column.

```
m := l * r
```

This number is printed with 4 space width. That's the multiplication table. The nested loops produce the "two-dimensional" printout.

8.3. Summary

We learned some basic features of `slice`. A `slice` is a reference type. A slice variable is a `pointer` pointing to the underlying array.

We also reviewed a couple of different forms of `for` loops. In particular, we used the nested `for … range` loops to create a "table" printout.

8.4. Questions

1. If you change the multiplication table range to a range from 6 to 12, what happens? How would you change the formatting to make the table look "regular"?
2. How can you "initialize" values in an `int` array of 10 elements with non-zero values?

8.5. Exercises

1. A slice can include elements of a slice type, which makes it sort of a "two-dimensional" slice. For example,

   ```
   var my2DSlice = [][]int{}
   ```

   This statement declares and initializes `my2DSlice`, whose type is a slice of a slice of `int`.

   Create a 2-D slice of `string`s and store the multiplication table printout first. Then print out the slice using nested loops.

Author's Note

Many Faces of Go

When you start learning Go, especially if you are coming from other (backend) programming language background, things can be rather confusing.

For example, the language specification seems to require semicolons at the

end of each statement. But, in some contexts, they seem optional. And, in some other contexts, it seems that you cannot use semicolons at all.

Which is it?

This is not limited to the use of semicolons. In all C-style languages, for example, white spaces (including tabs and newlines) generally have very little significance, other than as a token separator. You can put a statement in two lines, or you can put two statements in one line.

This is also true with Go. At least, in theory.

But, in practice, it does not work that way. You'll have to follow a particular style when programming in Go. You cannot, for example, put two statements in one line. You will have to put opening braces in particular places. And so forth.

Go uses a formatter, `go fmt`. It is not mandatory. The Go compiler mostly does not care (as long as it ultimately receives grammatically correct code). But, virtually everybody uses it. If you use an IDE or other editor tools that understand Go (or, gofmt), as most programmers do, then you are bound by these rules.

This is comparable to using linting in languages like Javascript. This is reminiscent of programming in Python where white spaces (e.g., indentations) are crucial part of the grammar/syntax.

We will not discuss `go fmt` in this book. As stated, if you use an IDE or a programmer's editor, like VSCode, then you will most likely not have to pay attention to these. As far as the programmers are concerned, the `gofmt` rules are part of the grammar.

You cannot, for example, use semicolons at the end of a statement. You cannot use spaces for indentation. You have to use tabs. And so forth.

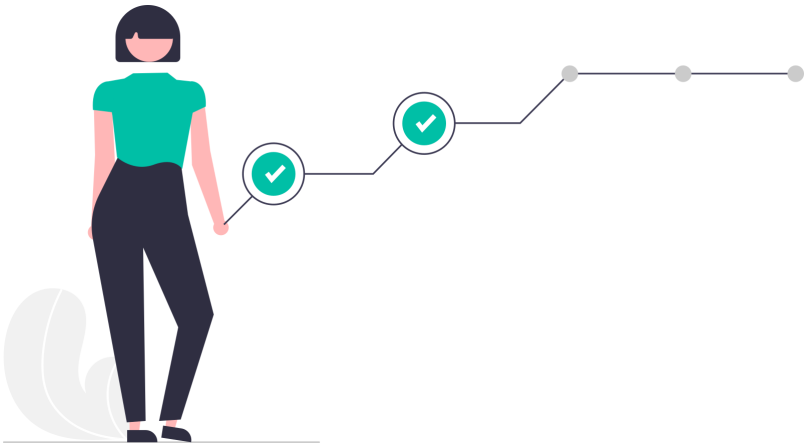
For some people, this may seem rather strange, but after a bit of initial resistance, you will end up accepting it. 😀 It will eventually feel natural to you.

That *is* Go.

Lesson 9. Find the Largest Number

9.1. Agenda

We will learn basics of error handling in Go, among other things.



9.2. Code Reading I

This program comprises two source files, *main.go,* and *findmax.go.*

find-largest-1/main.go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     sequence := []int{17, 7, 29, 3, 11}
7     fmt.Println("Input sequence =", sequence)
8
```

```
 9     max := findMax1(sequence)
10     fmt.Println("Largest =", max)
11 }
```

find-largest-1/findmax.go

```
 1 package main
 2
 3 func findMax1(s []int) int {
 4     max := s[0]
 5     for _, v := range s {
 6         if v > max {
 7             max = v
 8         }
 9     }
10     return max
11 }
```

As stated before, the file names are of little consequences in Go.

There is a source code file which contains the `main` function, named *main.go* in this example, and there is another file which contains a function `findMax1()`, whose name is *findmax.go*.

Both files belong to the same package, `main`.

9.2.1. Explanation

This is the first example in this book where a program includes more than one file (albeit both being small).

In order to build a program with `main` package containing more than one source files, you specify the package location in the build command. For example,

```
go build .
```

The `.` argument refers to the "current directory" where the main package resides (e.g., on a Unix shell).

It is not specified in the Go language specification, but the Go compiler tool chain requires that all source files of a package live in a single folder.

Another requirement is that you cannot have more than one packages in the same folder. (There are exceptions, as we will see shortly.)

Therefore, there is roughly a one-to-one correspondence between the physical construct, a folder/directory, and the Go's language construct, `package`.

In this example, you could not have put *findmax.go* in any other directory. It has to be in the directory where the other files reside that belong to the same `main` package.

`go build .`, by default, uses the directory name, *find-largest-1* in this case, as a default executable name when the build succeeds.

Try *go help build* to get more information on the "go build" command. You can also use just *go help* to list all available go commands.

As a shortcut during the development, one can use "go run" as well, which builds and runs the program in one go.

```
go run .
```

Again, you provide the location of the `main` package as an argument, "." in this case.

Running this program produces the following output:

```
Input sequence = [17 7 29 3 11]
Largest = 29
```

9.2.2. Grammar

A slice is essentially a sequence of values. Or, more precisely, a pointer to a sequence of values.

A variable of a slice type can be initialized in a number of different ways. Creating a slice with builtin `make()` function initializes all elements with a default value of the element type.

Another way is to initialize each element with explicit values:

```
hello := []byte{'h', 'e', 'l', 'l', 'o'}
```

This statement declares a new variable `hello` of type `[]byte` (a byte slice) of length 5, and initializes the values with `'h', 'e', 'l', 'l', 'o'`. That is, after the initialization, `hello[0] == 'h'`, `hello[1] == 'e'`, `hello[2] == 'l'`, `hello[3] == 'l'`, and `hello[4] == 'o'`.

A slice can also be created based on an existing array.

First of all, an array can be initialized in a similar way:

```
arr := [5]byte{'h', 'e', 'l', 'l', 'o'}
```

Notice the difference in syntax. This statement creates a new array `arr` of type

`[5]byte` (`5` as in 5 elements). The length of an array cannot change once it is created.

Since the number of elements for the array, and hence its type, is clear from the right hand side expression, we can omit the number of elements in the declaration. That is,

```
arr := [...]byte{'h', 'e', 'l', 'l', 'o'}
```

A pair of empty square brackets (`[]`) represents a slice type. Square brackets with three dots (`[...]`) indicates that it is for an array initialization.

A slice of a "related" type (e.g., `[]byte` from `[5]byte`) can be created from an array. Here's an example:

```
array := [...]int{101, 102, 103, 104, 105, 106, 107, 108}
slice := array[:]
```

An array with name `array` has been declared and initialized as before. A new variable `slice` is created based on `array` using the syntax `<array_name>[:]` (a pair of square brackets with a colon inside them).

The type of this `slice` is `[]int` (since it is based on the array of type `[8]int`), and its length is 8 (since it is based on the array of type `[8]int`). In this example, the capacity of the slice will also be 8 since it cannot grow beyond the size of the underlying array.

One can create a slice of a different length as well, by specifying the starting (inclusive) and/or ending (exclusive) indices. For example,

```
array := [...]int{101, 102, 103, 104, 105, 106, 107, 108}
```

```
slice := array[0:3]
```

The length of the `slice` in this example would be 3, with `slice[0] == 101`, `slice[1] == 102`, and `slice[2] == 103`. Its capacity is 8.

`0` is used for a missing starting index, and `len(array)` is used for a missing ending index. For instance, `array[:]` would be equivalent to `array[0:len(array)]`, or `array[0:8]` in this particular example.

The same syntax can also be used to create a slice variable from another slice. e.g., `<slice_name>[s:e]` where `s` and `e` are optional beginning and end indices, respectively. For example,

```
array := [...]int{101, 102, 103, 104, 105, 106, 107, 108}
slice1 := array[1:3]
slice2 := slice1[0:5]
```

Here `slice2` is created from `slice1`. Its underlying array of `slice2` is the same array `array`.

One interesting thing to note in this example is that `len(slice1) == 2` whereas `len(slice2) == 5`. We have created a bigger slice from a smaller slice. However, you cannot create a slice beyond the underlying array's capacity.

One other thing to note is that because `slice1` is a slice starting from index 1 of `array`, its starting element `slice1[0]` refers to the value of `array[1]`. Therefore `slice1[0:5]` is equivalent to `array[1:6]`. `slice2` in this example points to a "slice" of these 5 elements in `array`, `102, 103, 104, 105, 106`.

Interestingly, in this particular example, there is no way, syntactically, to access `array[0]` from the slice variable `slice1`.

Although we can access beyond its rightmost element of a slice (up to the last element of `array`), Go does not provide a way to access the elements on the left hand side of its leftmost element of a slice. This is the case as of this writing (version 1.17).

Here's an interesting puzzle. What would be the capacity of `slice1`? Would it be `8`? Or, would it be `7`? Or, something else? 😉

As of Go 1.17, now we can get the underlying array from a slice variable. Here's the relevant part from the Go language reference: Conversions from slice to array pointer [https://golang.org/ref/spec#Conversions_from_slice_to_array_pointer].

9.2.3. Deep Dive

The `main()` function creates a test slice of 5 elements, `sequence`, and calls `findMax1()` function with this slice as an argument.

The `findMax1()` function is defined in file *find-largest-1/findmax.go* in this example. Note first that this source file does not include any `import` statements (because none is needed).

`findMax1()` has a signature `func([]int) int`. It takes an argument of type `[]int` (a slice of `int`) and it returns an `int` value.

One thing to note is that we use a slice type (e.g., `[]int`) not an array type value (e.g., `[5]int`) as an argument of the function `findMax1()`.

This is very common in Go. Arrays are rarely used (other than as an underlying storage for a slice).

In this particular example, passing in an array, not a slice, could have limited the use of the function to a specific size array. The function's implementation can be

clearly more general than that.

The `main()` function of this example happens to use an array `[5]int`, but that is just incidental. That is just for illustration. There is really no reason to write separate functions for different array types: one for `[1]int`, another for `[2]int`, and another for `[3]int`, etc.

Using a slice `[]int` can potentially cover all these use cases.

The same, or similar, arguments can be made for functions using `[]int` vs `[]int32` vs `[]float64`, ... Unfortunately, Go does not support "generics", as of this writing. It is, however, expected that generics will be a part of Go in the near future (version 1.18).

Furthermore, as stated, an array is a value type. Every time we call functions like this with an array argument, it will need to be "copied", which can be rather expensive, especially for big arrays.

The function `findMax1()` goes through each element of the given slice, starting from the first element `s[0]`, and find the largest value.

Notice how it is done in this example. It checks the value of each element in the slice, and every time we see a value greater than the "current max" (`max`) it replaces the current max with the new largest value.

At the end of the loop, you will end up with the largest value from the given slice.

This is an example of an "algorithm".

This `findMax1()` function has an issue. If the passed-in argument is `nil` or if it has zero elements, then the program will crash. Go programs "panic" in situations like this.

Here's an example error message in case the input `sequence` is empty.

```
panic: runtime error: index out of range [0] with length 0
...
exit status 2
```

One way to deal with issues like this is making it the caller's responsibility.

For example, in this simple example, the caller can check the size of the argument, and only if it has at least one element, the caller calls the `findMax1()` function.

Here's an example:

```
// Receive sequence, say, from the user input.
var max int
if len(sequence) > 0 { // check if it
        max = findMax1(sequence)
}
// otherwise do something else.
```

In general, however, functions, especially the ones designed to be used by other programs, need to be able to deal with certain "exceptional" cases in some way, including notifying the caller of the exception.

We will take a look at this issue next.

9.3. Code Reading II

This program also comprises two source files, *main.go* and *findmax.go,* under a folder named `find-largest-2` (on the author's computer).

find-largest-2/main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      sequence := []int{17, 7, 29, 3, 11}
7      fmt.Println("Input sequence =", sequence)
8
9      index, max := findMax2(sequence)
10     if index == -1 {
11         fmt.Println("Empty input. No max found.")
12         return
13     }
14     fmt.Println("Largest =", max)
15 }
```

The *findmax.go,* file includes one function `findMax2()`.

find-largest-2/findmax.go

```
1  package main
2
3  func findMax2(s []int) (index, max int) {
4      if len(s) == 0 {
5          return -1, 0
6      }
7      index = -1
8      max = s[0]
9      for i, v := range s {
10         if v > max {
11             index, max = i, v
12         }
```

```
13      }
14      return
15  }
```

Note that the `findMax2()` function uses "named return values", `(index int, max int)`.

9.3.1. Explanation

We can run the program in the same way, using go run.

```
go run .
```

We get the same result:

```
Input sequence = [17 7 29 3 11]
Largest = 29
```

9.3.2. Deep Dive

In the previous example, there was no easy way to convey the information that something unusual happened.

The function `findMax1()` is expecting a non-nil, non-empty slice, and if it gets something unexpected, what is it supposed to do?

One way to handle situations like this is to return an unlikely, or invalid, value as a normal return value. Suppose that all input values should be positive integers. Then, returning a non-positive value like `-1` can potentially indicate some kind of errors.

In this particular case, however, that is not an option since we are dealing with all `int`s, positive or negative, in general.

One possibility is to modify the program to also return the index of the max value. This is the implementation of `findMax2()`. It returns a pair of numbers `(index int, max int)`.

Since the index cannot be negative, returning a negative integer, like `-1`, could indicate an error.

The caller can now check the return value and see if the function has worked as expected, in which case it can use the return value `max`, or something unexpected happened, in which case it can deal with the situation in some way.

That is what the `main()` function does in this example.

This is a usable option, but Go does it better.

9.4. Code Reading III

Go provides a type `error` for representing an error value. It is an "interface type", but that is not significant at this point. `error` is not a builtin type like `int` or `string`. But that distinction is not that significant either.

The `error` type is always available just like other primitive types.

This third example uses the `error` type to convey the unexpected, or unusual, situations.

find-largest-3/main.go

```
1 package main
2
3 import "fmt"
```

```
4
5  func main() {
6      // sequence := []int{}
7      sequence := []int{17, 7, 29, 3, 11}
8      max, err := findMax3(sequence)
9      if err != nil {
10         fmt.Printf("Error: %v\n", err)
11         return
12     }
13     fmt.Println(max)
14 }
```

The `findMax3()` function returns two values of types `(int, error)`.

find-largest-3/findmax.go

```
1  package main
2
3  import (
4      "errors"
5  )
6
7  func findMax3(s []int) (int, error) {
8      if len(s) == 0 {
9          return 0, errors.New("Empty input")
10     }
11     max := s[0]
12     for _, v := range s {
13         if v > max {
14             max = v
15         }
16     }
17     return max, nil
```

```
18 }
```

9.4.1. Explanation

If we run the program, we get the same result as before:

```
Input sequence = [17 7 29 3 11]
Largest = 29
```

If we run the program with an empty or nil slice, that is, by modifying the value of `sequence` in `main()` to something like `[]int{}`, then we get the following result:

```
Error: Empty input
```

9.4.2. APIs

- Package errors [https://golang.org/pkg/errors/]: Package `errors` implements functions to manipulate errors. The New function creates errors whose only content is a text message.
    - func New [https://golang.org/pkg/errors/#New]: `New` returns an error that formats as the given text. Each call to New returns a distinct error value even if the text is identical.

9.4.3. Deep Dive

Many modern programming languages use "exceptions" for error handling.

Go doesn't.

Go uses a convention in which one of the return values from a function is used to

convey an error, or unexpected, situation. The error return value should be the last one in the set of return values, and its type has to be the `error` type.

That is the convention.

The caller of a function then checks this special return value to see if something unexpected happened within the function (other than what is expected from calling the function).

For example, in the `main()` function, we do the following:

```
max, err := findMax3(sequence)
if err != nil {
    // Handle the "error" here
}
```

Returning from `main()` terminates the program.

The use of this `if err != nil {}` statement is idiomatic after calling a function that can potentially return a non-nil error.

Although we use the terms, "errors" or "exceptions", in programming, they do not necessarily mean that something bad has happened. Or, some kind of mistakes. As we will see throughout this book, the caller-callee relationship is complicated. ☺

"Error handling" is essentially a way of communication between the caller and the callee, in particular, from the called function to the calling function (or, to everyone upstream in the call chain, including the system/runtime).

In the `findMax3()` function, we check the "validity" of the passed-in argument `s`.

If the length of `s` is zero, we simply return with an `error` value. In this case, the normal return value, `max` in this case, is irrelevant since the caller function is expected to check the error value and if there has been an error, in general, it is not to use the normal return value. (There are always exceptions.)

In this example, we use a special function `errors.New()` from the standard library package `errors`, for convenience, to create a value of an `error` type. But, that is not strictly necessary. As we will cover later in the book, any value of a type which "behaves" like `error` will do. In this particular case, the error value has to be of a type that implements the `Error()` "method", whose signature is `func() string`.

In case of a normal execution of the function, without "errors", the `findMax3()` function simply returns `nil` value as an error, in addition to the normal return value, `max`.

`nil` is a predefined constant in Go, which indicates that the value has "no value". `nil` cannot be used as a value for a value type. All variables of a value type has real values. A `nil` value for a pointer type indicates that the pointer "points to nothing". A `nil` value for a variable of an interface type, like `error`, indicates that the variable references no real value of any compatible concrete type.

When a called function encounters a situation which it does not know how to handle, etc., there are a number of options.

It can just terminate the program, for example. It is a valid option. But it may not be the best option since the caller function has a better context and it may know how to handle the situation better.

Normally, a better way to handle an unexpected situation which is beyond the purview of the normal functionality of a function, is just send a signal to the caller function that indicates such a situation.

In Go, a non-nil error return value signals an "error".

We will cover Go's `panic` (and, `recover`) in later lessons, which is another way to handle error situations in Go.

In the `main()` function of this example, we use a special verb `%v` to print the error:

```
fmt.Printf("Error: %v\n", err)
```

The verb `%v` (`v` for verbose, presumably) is mainly used for debugging. The format `%+v`, with `+` in front of `v`, prints out more information. `%#v` (with four `+`'s, essentially) can print out even more information, if available.

In this example, this is equivalent to

```
fmt.Printf("Error: %s\n", err)
```

The returned error has no more information than what we have provided, a string "Empty input". Or, we can just use `Println()`, which simply prints the "string value" of each of its arguments.

```
fmt.Println("Error:", err)
```

Incidentally, the error interface type includes a method `Error()`, which returns a value of `string` type. The returned error value can be of any type, in general, as long as the type implements this `Error()` method of type `func() string`. If we need to get its string-equivalent value, then we can call its `Error()` method.

```
fmt.Println("Error:", err.Error())
```

We will review the `interface` types in later lessons.

9.5. Summary

We learned a basic error handling mechanism in Go, which uses the "error return value" convention.

A function which can potentially run into an unexpected, or exceptional, situation can return a possible error information as the last return value of type `error`.

The caller of this function is then expected to check the error return value to see if it can use the normal return value(s). If the error return value is non-nil, then it should treat the normal return values with caution. Normally, it should discard the normal return values if it receives a non-nil error.

Author's Note

"Thought Programming"

The author loves books. He owns thousands of Kindle books (although he has read only a tiny fraction of them 😀).

The Art of Go - Basics: Introduction to Programming in Golang - Beginner to Intermediate is a book for reading. Keep it on your night stand. Keep it on your coffee table. Take it to lunch.

You do not need a computer to read this book. Read, and if you need to practice, then do it in your head. Like "thought programming". As in a "thought experiment".

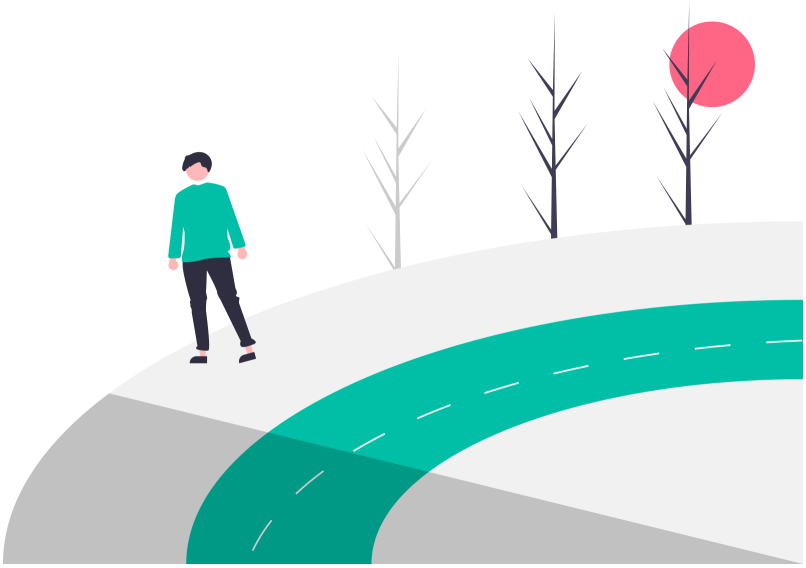
Obviously, this is an oxymoron. But, it is possible. And, this is a much better alternative to making excuses and postponing. You may say, "Oh, I don't have access to computer right now. I'll do it later". And, you may never do it.

> Just do "thought programming" when you need to do programming.

Lesson 10. Rotate Numbers

10.1. Agenda

We will explore the slice types a little bit more in this lesson.



10.2. Code Reading

Suppose that we have an array or slice of `int`s, say, `2, 4, 2, 6, 8`. We would like to "rotate" elements by 1, to the left. For example, from `2, 4, 2, 6, 8` to `4, 2, 6, 8, 2`.

Here's an example code to solve this problem:

rotate-numbers/main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      sequence := [7]int{1, 2, 3, 4, 5, 6, 7}
7      fmt.Println("Original sequence:", sequence)
8
9      rotated := rotateBy1(sequence[:])
10     fmt.Println("Rotated sequence:", rotated)
11 }
```

The *rotate1.go* file includes two functions, `rotateBy1()` and `rotateByK()`:

rotate-numbers/rotate1.go

```
1  package main
2
3  func rotateBy1(s []int) []int {
4      return rotateByK(s, 1)
5  }
6
7  func rotateByK(s []int, k int) []int {
8      l := len(s)
9      if l <= 1 && k <= 0 {
10         return s
11     }
12     k = k % l
13     if k == 0 {
14         return s
15     }
16     rotated := append(s[k:], s[0:k]...)
```

```
17     return rotated
18 }
```

10.2.1. Explanation

In this example, we define a slightly more general function `rotateByK()` and use it for the problem at hand, namely, rotating by `1`.

If you run the program as before:

```
go run .
```

You get the following output:

```
Original sequence: [1 2 3 4 5 6 7]
Rotated sequence: [2 3 4 5 6 7 1]
```

10.2.2. Builtin Functions

We have used the builtin `len()` and `cap()` functions before. `append()` is a new function, which we are going to look at in some detail.

- `len()`: It returns the length of its argument. In case of an array or slice, it returns the number of elements in the collection.
- `cap()`: It returns the capacity of its argument. In case of an array, `cap()` is equivalent to `len()`. In case of a slice, it returns the maximum possible length of the slice when re-sliced;
- `append()`: The `append()` function appends elements to the end of a given slice. If it has a sufficient capacity, the destination is re-sliced to accommodate the

new elements. Otherwise, a new underlying array will be allocated, and it returns a slice pointing to this new array.

10.2.3. Grammar

The `slice` type is one of the most interesting types in Go. But dealing with slices can get a little bit tricky.

A slice is a reference to an underlying array. A slice "points to" the underlying array. The underlying array can be implicitly of explicitly defined. By changing values in a slice, one ends up changing the values in the array as well.

You can "reslice" a slice, or "slice" an array, using the `[b:e]` syntax, where `b` and `e` are optional beginning and end indices, respectively. But this re-slicing operation is limited by the size of the underlying array.

In order to add an element to a slice, which may require allocating a new array, we use the builtin `append()` function.

```
func append(slice []Type, elems ...Type) []Type
```

The append function takes a slice as its first argument, as a destination, and one or more elements to be added to the slice in the following positions.

`append()` returns another slice as a return value. For instance,

```
seq := [...]int{1, 2, 3, 4, 5}        ①
before1 := seq[:2]
after1 := append(before1, 11, 12)
```

Or

```
before2 := []int{1, 2}                ②
after2 := append(before2, 11, 12)
```

Or

```
before3 := make([]int, 2)             ③
before3[0] = 1
before3[1] = 2
after3 := append(before3, 11, 12)
```

In all three examples, the slice values have changed. The `before1` slice is different from `after1`, `before2` is different from `after2`, and `before3` is different from `after3`. Not only that, all three "after" variables in these examples now point to completely different arrays than those of the "before" variables.

In the first example, a slice `before1` is taken from an array `seq`. In the second example, a slice `before2` is explicitly initialized (with an implicit underlying array). In the third example, a slice `before3` of length 2 is initialized with default values using the `make()` function. `before3` will also be implicitly associated with an array (of length 2). Values are then assigned to each element in `before3`.

In these examples, all three `after` slices are associated with arrays, whose values happen to be all “1, 2, 11, 12”. And, they are all different from initial arrays, explicit or implicit, “1, 2, 3, 4, 5”, “1, 2”, and “1, 2”, respectively.

This is generally the case. `append()` may (likely) need to re-allocate a slice’s underlying array, which can be expensive.

There is an exception though. If you use a "capacity" value when creating a new slice with `make()`. The slice can grow up to the capacity without requiring a new underlying array.

For example,

```
slice1 := make([]int, 0, 10)
slice2 := append(slice1, 1, 2, 3, 4, 5)
slice3 := append(slice2, 6, 7)
slice4 := append(slice3, 8, 9)
```

All four slice variables have the same underlying array (implicitly). They are different variables, and they have different lengths. But they all point to (a part of) the same array.

This is because `make()` in this example have created an array with capacity 10, which is big enough to accommodate all 9 elements, `1` through `9`.

Now, if we add a few more elements,

```
slice5 := append(slice4, 10, 11, 12)
```

Then the new slice `slice5` no longer shares the same underlying array. The original (implicit) array has only 10 elements (initialized with zero values). `slice5` requires 12 elements. The `append()` operation in this last line, therefore, has to create a new array.

Why is this important? It is because a slice is a reference type. If we change a value of an element in slice3 for example,

```
slice3[0] = 100
```

Slices, `slice2` and `slice3`, will have the same value in index `0`. That is, `slice2[0] == 100` and `slice4[0] == 100`. (Since the length of `slice1` is zero, this element is

not accessible to `slice1`.)

On the other hand, the zero-th element of `slice5` remains unchanged. That is, `slice5[0] == 1`.

The Go programming language does not provide a way to tell explicitly which slices share the same underlying array and which ones do not. Hence, as a programmer, you have to know what you are doing.

Allocating new memory (and copying old values) can be expensive. Hence, it is a good practice to use the make function with a specific capacity if the likely size of the (final) slice is more or less known.

One other thing to note is that Go is a garbage collected language. Every time we create a new variable (possibly with new allocated memory) and leave the old ones around, it becomes a work for the Go runtime, potentially decreasing the overall performance.

One common idiom when using `append()` is that we use the same variables for the existing and the new slices.

```
sliceX :=[]int{100}
sliceX = append(sliceX, 200)
sliceX = append(sliceX, 300)
```

`append()` returns a different slice (which may or may not point to the same underlying array in general). By re-using the variable, `sliceX` in this case, the old slice becomes inaccessible in the program. And, if the old slice happens to have a different array than the new slice (now assigned to the variable `sliceX`), then the old array becomes inaccessible, and it can be garbage-collected.

It should be noted that this idiomatic pattern also alleviates the issue mentioned earlier: the language itself does not provide an explicit way to tell which slices

share the same underlying array.

Also, calling `append()` as few times as possible is preferred over calling it many times. For instance, the above example could have been done with one `append()` call:

```
sliceX = append(sliceX, 200, 300)
```

(In fact, we could have initialized `sliceX` without calling `append()` in this particular example, but it should be noted that these examples are primarily for illustration purposes.)

The `append()` function has an interesting signature, `func([]int, ...int) []int` in case of `int` slices.

The `...` notation, *before* the argument type `int`, in the function signature indicates that the function can take an arbitrary number of `int`s as arguments after the first `[]int` argument. This kind of functions are often known as "variadic functions".

The variable number arguments should be in the last position in the function argument list.

We have seen some examples from the standard library. In particular, the `Print` family of functions in the `fmt` package. We can pass in zero or more arguments to `fmt.Print()`, `fmt.Println()`, and `fmt.Printf()`, etc. The `Printf()` function is even more special in that the verbs in the formatter have to match the arguments.

One interesting thing about variadic functions in Go is that one can use a slice in place of a list of values (in the variable number argument position). For example, the above example could be written as follows:

```
args := []int{200, 300}
```

```
sliceX = append(sliceX, args...)
```

Note the ... notation *after* the variable name (a slice `int`s in this case).

If you are coming from languages like Javascript/Typescript, you may have seen similar notations. But, their uses are not exactly the same as ... of Go.

10.2.4. Deep Dive

In this lesson, the task is to "rotate" elements of a given slice by 1 to the left.

We tackle a slightly more general problem, rotating the elements by `k` and use the general solution for the specific problem, namely, rotating by `1`, which is a special case of the more general problem.

In programming, this is a common practice to tackle a set of related problems, which are expected to have more or less the same solution.

For example, rotating numbers by 1 might have a similar solution to that of rotating numbers by 2, and rotating numbers by 3, etc.

We saw a similar situation in the previous lesson, Find the Largest Number, for instance. We do not implement different functions for `[1]int`, and `[2]int`, etc. We just implement a generic function that can handle all these situations as special cases. In that case, the solution was using a slice, `[]int`, rather than using fixed size arrays, when we define, and implement, a function.

In the example of this lesson, we implement the general solution as `rotateByK(s []int, k int) []int`, and use it in our specific problem, `rotateBy1(s []int) []int`.

```
func rotateBy1(s []int) []int {
    return rotateByK(s, 1)
```

```
}
```

When `k == 1`, `rotateByK(s, 1)` is equivalent to `rotateBy1(s)`.

In fact, this is the first example of our non-main function calling another non-main function in the main package. As stated, a Go program essentially executes through these call chains, one function calling another, and this in turn calling another, etc.

There can be many different ways to shift elements in a slice (or, an array) by an integer `k`. We illustrate one solution using Go's `append()` function in this lesson.

The implementation of `rotateByK(s, k)` starts by checking some edge cases. When the argument slice `s` is `nil` or empty, we cannot rotate the elements. Hence we just return `s`.

When `len(s) == 1`, we can make a similar argument. That is, "rotating" a list of one element does not make much sense. Hence, we just return the input slice, without change. For simplicity, we will also ignore the case when `k <= 0`. We can either design a function to accept only non-negative `k` (e.g., by declaring `k` as `uint`, for instance) or we can interpret a negative `k` as rotating to the right by `-k` (`-k > 0`).

```
l := len(s)
if l <= 1 && k <= 0 {
    return s
}
```

When `k` is bigger than `l`, we are only interested in the "net rotation". Hence we can set `k` to its modulo with `l`. And, if the resulting `k` is `0`, again no rotation. Just return the same input slice.

```
k = k % l
```

```
if k == 0 {
    return s
}
```

Note that, as mentioned before, it is a "copy" of the slice that is returned. But they point to the same array in this special case because slices follow reference semantics.

At this point, we can make a few assumptions. One of them is the constraints `k > 0` and `k < l` where `l = len(s)` is bigger than 1.

Now, to rotate the slice of `l` elements to the left by `k`, we simply move the first part (from `0` to `k`) to the end of the second part (from `k` to `l`). That is exactly what the `rotateByK(s, k)` function does.

```
rotated := append(s[k:], s[0:k]...)
```

As required by the `append()` function signature (a variadic function), the second slice argument (the beginning part of the original slice) has been "spread" as `int`s using the `...` notation.

The core of the function happens in this line. But, we know that calling `append()` is safe because we already checked the range of possible `k` values, etc. More importantly, `0 <= k <= l`, which is satisfied by the constraints that we have imposed through some initial processing.

A new slice which points to a new array that has the "rotated" content is then returned to the caller, `rotateBy1()` in this case. Then, the `rotateBy1()` function returns its returned value to its own caller, `main()` in this example.

The `main()` function then prints the result, and it returns to its own caller, which is the Go runtime or the operating system. Thus the program terminates.

Go is a somewhat strange language. The apparent simplicity can be deceiving.

For instance, some programmers with background in C/C++, for instance, might find the implementation of `rotateByK()` rather strange.

The `append()` function possibly allocates memory. And, in this case, it does. It happens in a function scope of the `rotateByK()` function. And yet, we return its pointer (a slice, `rotated`) to the (presumably) locally allocated memory.

Clearly, this is an absolute no-no in C/C++. An yet, you can do this kind of things in Go. In fact, Go allocates memory globally (on the heap), not locally (on the stack), in this kind of situations.

If your background is in the garbage collected languages like Java, C#, and Python, then that is how it works most of the time in those languages.

To repeat, every type in Go but a few are all value types. They follow the value semantics. (And, to use the reference semantics, we use pointers.) On the other hand, `slice` is one of the exceptions. A `slice` variable follows reference semantics, and its memory is automatically managed by the Go runtime.

As we will see later in the book, the Go runtime does memory management in other cases as well. We will see some examples in later lessons.

This "dual" nature of Go can be confusing at first to new comers to Go. In certain contexts, we have to distinguish value and reference semantics. In certain contexts, like memory management, Go takes care of it behind the scene.

10.3. Summary

We further explored various aspects of `slice` types in ths lesson.

In particular, we took a look at the `append()` function in some detail.

10.4. Exercises

1. Implement "rotate by 1" without using `append()`. In fact, without having to use big new space. (In the example used in this lesson, we essentially needed an extra space equal to the size of the original slice/array, which may or may not be feasible in certain situations.) This can be done, for instance, by swapping two consecutive elements, one at a time, across all pairs of the neighboring elements.
2. In this kind of implementation, a more general problem "rotate by k" can be a bit more complicated. Can you solve this problem in a similar manner?

Author's Note

What You Don't Know Won't Hurt You

Throughout this book, we make frequent comparisons of Go with other programming languages.

This is to the benefit of the readers who have some familiarity with those languages. Comparisons, and analogies, and contrasts, can be rather useful when you learn new subjects.

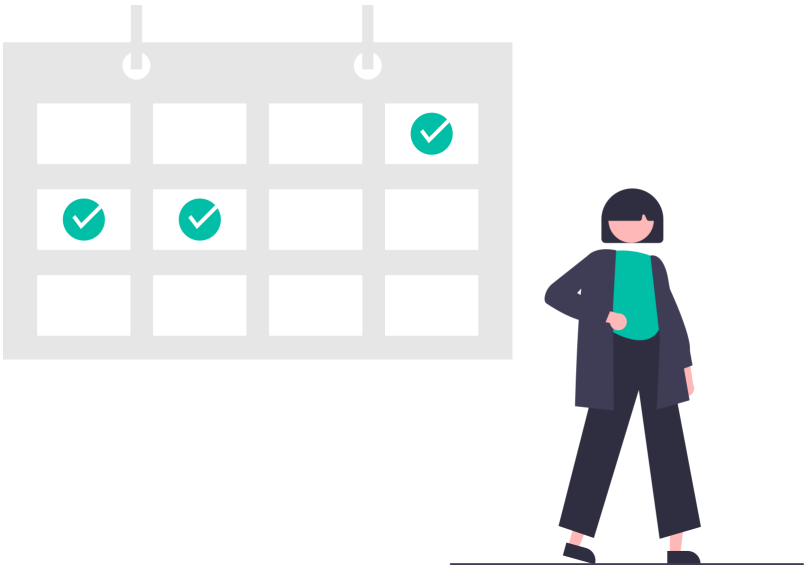
If those comparisons/contrasts do not make sense to you, or if Go is your first programming language, then you can ignore those comments.

No harm done. 👍

Lesson 11. Leap Years

11.1. Agenda

A little bit more about functions.



11.2. Code Reading

Let's create a program that checks if a given year is a leap year.

leap-year/main.go

```
1 package main
2
3 import "fmt"
4
5 func main() {
```

```
    isLeapYear := isLeapYear1

    answer := isLeapYear(1900)
    fmt.Println("Is 1900 leap year?", answer)

    answer = isLeapYear(1984)
    fmt.Println("Is 1984 leap year?", answer)

    answer = isLeapYear(2000)
    fmt.Println("Is 2000 leap year?", answer)

    answer = isLeapYear(2021)
    fmt.Println("Is 2021 leap year?", answer)
}
```

We will look at three different implementations.

The first implementation:

leap-year/leapyear1.go

```
package main

func isLeapYear1(year int) bool {
    var isLeap bool
    if year%4 == 0 {
        if year%100 == 0 {
            if year%400 == 0 {
                isLeap = true
            } else {
                isLeap = false
            }
        } else {
            isLeap = true
```

```
        }
    } else {
        isLeap = false
    }
    return isLeap
}
```

The second implementation:

leap-year/leapyear2.go

```
package main

func isLeapYear2(year int) bool {
    if year%400 == 0 {
        return true
    } else if year%100 == 0 {
        return false
    } else if year%4 == 0 {
        return true
    } else {
        return false
    }
}
```

The third implementation:

leap-year/leapyear3.go

```
package main

func isLeapYear3(year int) bool {
    if year%4 == 0 && year%100 != 0 || year%400 == 0 {
        return true
```

```
6     } else {
7         return false
8     }
9 }
```

11.2.1. Explanation

The program contains three different implementations of `isLeapYear()`.

In the included example of the `main()` function, the first version `isLeapYear1()` is hard-coded.

If you run the program with *go run*:

```
go run .
```

You get the following output:

```
Is 1900 leap year? false
Is 1984 leap year? true
Is 2000 leap year? true
Is 2021 leap year? false
```

If you want to use the second version instead, for instance, then you can assign the name of function, `isLeapYear2`, to the variable `isLeapYear` in the `main()` function:

```
isLeapYear := isLeapYear2
```

11.2.2. Grammar

A function has a type. Just like any other variables or constants, functions have types.

You can even declare a function type and use it to declare a new variable of that function type.

We have seen only simple types like `int` or `string` so far, and we have not really discussed how to define a new type. We will come back to this topic later in the book. But, for now, it is important to realize that a function has a type.

It may seem a bit confusing, but a function declaration is comparable to any other variable or literal definitions.

Just the syntax is different.

```
func isLeapYear(year int) bool {
    // ...
}
```

In this example, the name `isLeapYear` has a type `func(int) bool` (that is, its function signature). The function definition is given inside the matching curly brackets.

Compare this, for instance, with the following:

```
const leapYear bool = false
```

This statement declares a new name `leapYear` as a bool type and initializes it with a boolean value `false`.

The names `isLeapYear` and `leapYear` in these declaration are more or less the same in that both declarations are introducing these new names and their definitions. They just have different types. `isLeapYear` is of type `func(int) bool` whereas `leapYear` is of type `bool`.

You can use the function names just like any other identifiers in Go programs. Note, however, that function names are more like literals than variables.

You can create a variable of a function type.

```
var newVar func(int) bool
```

In this example, `newVar` is a variable of type `func(int) bool`. It is not initialized with any explicit function definition. The default value for a variable of (any) function type is `nil`.

Since the function `isLeapYear` happens to have the same type, in this case, we can assign `isLeapYear` to `newVar`, or use it to initialize `newVar` in the first place.

```
var newVar func(int) bool = isLeapYear
```

Or, even

```
var newVar = isLeapYear
```

Since the type can be inferred from the right hand side expression.

Now, you can use the `newVar` variable as if it is a function name:

```
leapYear := newVar(2020)
```

This will all seem natural to programmers with background in languages like Javascript/Typescript because they have a similar syntax.

Other languages have different constructs to support similar functionalities. For instance, C has function pointers. C++ uses "functors" in addition to function pointers. Java uses interfaces to support function types. C# has a construct called "delegate", among other things.

In Go, a function is just a type.

11.2.3. Deep Dive

It is hard to explain how to write a program. If it was easy, then we could have just taught computers to write a program. (Some day. Some day soon, maybe. But, not yet. ☺)

The best advice to beginning programmers is the same three words that you hear in any art:

"Practice, practice, practice."

Modern programs are rather complex. We use "frameworks", or other libraries, and we often write only a part of a program. The rest is sometimes implicit, hidden in frameworks or runtimes, etc.

Regardless, at the core of a program is an "algorithm". Modern programming does not really fit well into the classic definition of an algorithm, which can be represented by a flow chart, for instance. An object-oriented programming style, for instance, cannot really be described with algorithms, or at least with algorithms alone.

But, in the broadest possible sense of the word, programming is all about "algorithms".

Let's take a look at the "leap year problem" of this lesson: Given a year, determine if the year is a leap year.

A leap year has 366 days instead of 365. Interestingly, which year is a leap year is *defined algorithmically*. For instance, refer to en.wikipedia.org/wiki/Leap_year for the definition of *leap year*.

```
If the year is divisible by 4, continue.
If the year is not divisible by 4, it is not a leap year. End.
    If the year is divisible by 100, continue.
    If the year is not divisible by 100, it is a leap year. End.
        If the year is divisible by 400, it is a leap year. End.
        If the year is not divisible by 400, it is not a leap year.
End.
```

This is an algorithm. We translate this algorithm into a program in Go. That is the `isLeapYear1()` function.

If we write the above algorithm slightly differently, then it is easier to compare:

```
If the year is divisible by 4, continue.
    If the year is divisible by 100, continue.
        If the year is divisible by 400, it is a leap year. End.
        Else (if the year is not divisible by 400), it is not a leap
year. End.
    Else (if the year is not divisible by 100), it is a leap year.
End.
Else (if the year is not divisible by 4), it is not a leap year. End.
```

That is precisely the `isLeapYear1()` function.

This function can be rewritten in the following way, by changing the order of the `if` statements:

```
func isLeapYear1Alt(year int) bool {
    var isLeap bool
    if year%400 == 0 {
        isLeap = true
    } else {
        if year%100 == 0 {
            isLeap = false
        } else {
            if year%4 == 0 {
                isLeap = true
            } else {
                isLeap = false
            }
        }
    }
    return isLeap
}
```

Or, this way, by removing the local variable `isLeap`:

```
func isLeapYear1Alt(year int) bool {
    if year%400 == 0 {
        return true
    } else {
        if year%100 == 0 {
            return false
        } else {
            if year%4 == 0 {
```

```
                return true
            } else {
                return false
            }
        }
    }
}
```

The nested `if-else` statements can be written without nesting.

```
func isLeapYear1Alt(year int) bool {
    if year%400 == 0 {
        return true
    } else if year%100 == 0 {
        return false
    } else if year%4 == 0 {
        return true
    } else {
        return false
    }
}
```

These two functions are exactly the same. But, this version is "flatter" and it is easier to read, and this form is generally preferred over the nested version. This is the version presented earlier, the `isLeapYear2()` function.

Now, all three Boolean expressions can be combined into a single Boolean expression.

```
func isLeapYear1Alt(year int) bool {
    if (year%4 == 0 && year%100 != 0) || year%400 == 0 {
        return true
```

```
    } else {
        return false
    }
}
```

You can easily convince yourself that these two functions behave exactly the same way for all eight different conditions. (`true/false` for 3 boolean expressions yields 8. `2 * 2 * 2 = 8`.) Hence the two functions are equivalent for all possible input, `year`.

The Boolean `&&` operator has a higher "precedence" than the `||` operator in Go. Hence the the parentheses around the `&&` expression can be omitted.

That's the function `isLeapYear3()` presented earlier, and it is the final version.

Operator precedence determines which parts of an expression get evaluated first. For example, `2 + 3 * 4` is evaluated to be `2 + (3 * 4)`, which is `14`. This is because the multiplication operator (`*`) has a higher "precedence" than the addition operator (`+`). The explicit use of the parentheses can change the evaluation order. For instance, `(2 + 3) * 4` is evaluated to `20`.

Unary operators (e.g., `-` in front of a number) have the highest precedence. You can refer to the Go language reference for the complete list of binary operator precedence rules, if necessary. But, you do not have to memorize these precedence rules. When in doubt, use the parentheses to make your intentions clear (e.g., to the human readers), rather than relying on (possibly obscure) precedence rules.

Note that multiple binary operators may have the same precedence. In that case, operators of the same precedence associate from left to right. For example, `6.0 / 2.0 * 3.0` is the

> same as `(6.0 / 2.0) * 3.0`, which evaluates to `9.0`.

11.3. Summary

A function in Go has a type, and functions can be assigned to variables, or they can otherwise be manipulated.

11.4. Exercises

1. Create a new `main()` function to accept a value of `year` as a user input. Write your own `isLeapYear()` function without referring to the sample code. Test your program with a few `year` values, and verify that your program works as expected.
2. Create a new `main()` function to take a value of `year` as a command line argument. Do the same tests as the previous exercise.

Author's Note

"It Does Not Work"

One of the most frequent complaints that we hear from beginning programmers is "it does not work". "My program does not work", "this function does not work", etc.

This is not limited to beginners. Even experienced programmers fall into this trap. For example, many complain, "this API does not work", "this backend component does not work", etc.

What they really mean by that is that something does not work as he/she has expected.

> The interesting thing is, in many cases, their expectations are wrong. Not the programs.
>
> It is often helpful to think carefully what exactly you are doing instead of jumping into programming right away.

Lesson 12. BMI Calculator

12.1. Agenda

We will discuss type conversion in this lesson.

12.2. Code Reading

We accept user inputs and calculate the user's body mass index (BMI), en.wikipedia.org/wiki/Body_mass_index.

bmi-calculator/main.go

```
1 package main
2
3 import (
```

```
    "fmt"
    "os"
)

func main() {
    w, err := readInput("Weight (in pounds)")
    if err != nil {
        fmt.Fprintf(os.Stderr, "Error = %v\n", err)
        os.Exit(1)
    }

    h, err := readInput("Height (in inches)")
    if err != nil {
        fmt.Fprintf(os.Stderr, "Error = %v\n", err)
        os.Exit(1)
    }

    bmi := bmi(w, h)
    fmt.Printf("Your BMI is %.2f kg/m2\n", bmi)
}
```

It is not really necessary to use multiple small files in a package, but for illustration purposes the `main` package in this lesson has been divided into 3 files. One for the "main program", and one for the core calculation (*bmi.go*), and the other for the input handling (*input.go*).

We can even put these files in a different package(s), as we will see in the next lesson.

bmi-calculator/bmi.go

```
package main

func bmi(w, h float32) float32 {
```

```
    wInKilos := float64(w) * 0.453592
    hInMeters := float64(h) * 0.0254
    bmi := wInKilos / (hInMeters * hInMeters)
    return float32(bmi)
}
```

bmi-calculator/input.go

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strconv"
    "strings"
)

var reader = bufio.NewReader(os.Stdin)

func readInput(prompt string) (float32, error) {
    fmt.Printf("%s: ", prompt)
    str, err := reader.ReadString('\n')
    if err != nil {
        return 0, err
    }
    str = strings.TrimSuffix(str, "\n")

    value, err := strconv.ParseFloat(str, 32)
    if err != nil {
        return 0, err
    }
    input := float32(value)

```

```
27     return input, nil
28 }
```

12.2.1. Explanation

We can run the program as before:

```
go run .
```

Heres a sample output:

```
Weight (in pounds): 300          ①
Height (in inches): 75
Your BMI is 37.50 kg/m2
```

① Numbers *300* and *75* are user inputs.

This is a big person. (Clearly, we are just using arbitrary numbers.) Normally, BMI of 25 or less, and above 18.5, is *considered* a healthy weight (although it is not entirely scientific).

12.2.2. Grammar

There are so many different integer types, from `uint8` and `int8` to `uint64` and `int64`. And, even machine architecture dependent `uint` and `int` types. Which one to use?

In general, it is a difficult question to answer. We will show a few examples later in the book. For now, you can just use `int` in most cases. (Or, `uint`, if you are specifically dealing with unsigned integers.)

For floating point numbers, things are much easier. There are only two floating point number types to begin with, `float32` and `float64`.

If you are inclined, you can just use `float64` for all floating point numbers.

The downside is, though, the values of `float64` take up the double amount of space than those of `float32`. If there is memory or storage constraint, then you may have to choose `float32` for some, or all, floating numbers. Besides, in most cases, you won't need the 64 bit precision. `float32` will suffice unless you have special requirements.

There ia a gotcha, however. All computations should be done using `float64`. This is not a must, but highly recommended.

During computations, floating point numbers lose precision, as a general rule. As you do more calculations, the resulting numbers become less and less precise.

Therefore, it is important to use higher precision numbers during calculations. You can convert them back to `float32` type, if you need to, before storing them, etc.

This is why various functions in the `math` package use `float64`.

This is a general rule. But, there is a small twist in Go. Go does not allow implicit conversion from `float32` ("single precision") to `float64` ("double precision"). This implicit "widening" conversion or casting, e.g., for integer and floating point numbers, is generally allowed in most C-style languages. But not so in Go.

You'll have to explicitly convert between each and every value which has a different type. This can be rather cumbersome.

So, going back to our general rule of thumb, just use the `float64` type for all floating numbers, that is, unless you have a particular reason otherwise. As a matter of fact, memory is cheap. Storage is cheap. Computation is cheap. Your time spent in writing a program might be the most valuable resources.

Having said that, however, we will try to use `float32` as much as possible in this book. These examples are for illustration purposes, and depending on the context, they may or may not be as realistic or even practical.

When a program terminates, it returns an "exit code" to the operating system. This is a convention, or a requirement, on Unix-like systems.

An integer return value of `0` indicates that the program has run successfully. Any non-zero value indicates some kind of errors.

In many C-style programming languages, the `main()` function returns the exit code as its function return value. When no value is returned, it is assumed that the exit code is `0`, that is, "success".

In Go, the `main()` function does not return any value. Instead, it uses a system function, `os.Exit()`.

Normally, we do not need to call this function with `0` since that is the normal termination of a program.

In cases of exceptions, or errors, we call `os.Exit(<exit_code>)` with a non-zero value. Unless you are using a particular error code, the convention is just to use `1` or something comparable, as is done in the example of this lesson.

Any non-zero value will do for this purpose, that is, to indicate an unspecified error to the operating system, or the runtime. As is indicated in the API documentation, however, a value in the range of `[0, 125]` is recommended.

12.2.3. APIs

- Package strconv [https://golang.org/pkg/strconv/]: Package `strconv` implements conversions to and from string representations of basic data types.
    - func ParseFloat [https://golang.org/pkg/strconv/#ParseFloat]: `ParseFloat` converts the string s to a floating-point number with the precision specified by

bitSize: 32 for float32, or 64 for float64. When bitSize=32, the result still has type float64, but it will be convertible to float32 without changing its value.

- Package os [https://golang.org/pkg/os/]: Package `os` provides a platform-independent interface to operating system functionality. The design is Unix-like, although the error handling is Go-like; failing calls return values of type error rather than error numbers.
  - func Exit [https://golang.org/pkg/os/#Exit]: Exit causes the current program to exit with the given status code. Conventionally, code zero indicates success, non-zero an error. The program terminates immediately; deferred functions are not run. For portability, the status code should be in the range [0, 125].

12.2.4. Deep Dive

The United States is one of the very few countries in the world, where we still use the "imperial" system of units rather than the metric system.

The *BMI* index is computed using the numbers in the metric system, in particular, kilograms and meters. So, we will need to do some conversion.

The user inputs in the command line interface are all strings. They do not have a type in fact, but we treat them as text, or strings, because CLI is, by definition, character- or text-based.

The input, the body weight and height, are numbers. So, we will need to convert the input text to numbers.

We use, in this example, `parseFloat()` from the `strconv` package. This package includes a number of helper functions to convert between strings and other primitive types.

The `parseFloat()` function returns an error, as a second return value, if the given argument string is not convertible to a floating point number. The second argument `parseFloat()` decides the precision of the parsed number. The `parseFloat()`

function always returns `float64` regardless of the value of this argument.

Here's a simple function that demonstrates some of the `strconv` package functions:

```
func parseDemo() {
    f, e1 := strconv.ParseFloat("12.99", 64)
    fmt.Println(f, e1)

    i, e2 := strconv.ParseInt("1234", 32, 64)
    fmt.Println(i, e2)

    d, e3 := strconv.ParseInt("0xabc", 0, 64)
    fmt.Println(d, e3)

    u, e4 := strconv.ParseUint("0555", 0, 64)
    fmt.Println(u, e4)

    n, e5 := strconv.Atoi("234")
    fmt.Println(n, e5)

    n, e5 = strconv.Atoi("1oo")
    fmt.Println(n, e5)
}
```

Refer to the API documentation, golang.org/pkg/, for more information.

Reading an input for the weight is more or less the same as reading for the height. Therefore, it makes sense to encapsulate this functionality of reading inputs in a separate function. In the example, that is the `readInput()` function. Then we can re-use this function in multiple places.

This is a reasonable thing to do, say, to increase readability, even if the function might not be intended to be used outside of the program, as in this case. The `main()` function becomes easier to read.

Although it is not strictly necessary, we put the part of the program that calculates the actual BMI number in a separate function `bmi()`, which we could have called `computeBMI()` or by some other names. We even put this function in a separate file, *bmi.go*. This is not generally necessary in small programs like this.

Now, the `main()` function becomes much easier to read. If we ignore the error handling,

```
func main() {
    w, _ := readInput("Weight (in pounds)")       ①
    h, _ := readInput("Height (in inches)")       ②
    bmi := bmi(w, h)                              ③
    fmt.Printf("Your BMI is %.2f kg/m2\n", bmi)   ④
}
```

① Read the user's weight.

② Read the user's height.

③ Compute the BMI value.

④ Print out the result.

That's it. That is the whole program. We could have gone even further. Something like this:

```
func main() {
    w, h := readWeightAndHeight()
    bmi := computeBMI(w, h)
    writeOutput(bmi)
}
```

Or, even

```
func main() {
    readInputAndComputeBMIAndWriteOutput()
}
```

Clearly, there should be a sweet spot somewhere in between, depending on the requirements and based on other considerations and context, etc.

In Go programs, a function is a basic unit of "code reusability".

The implementation of the `bmi()` function is straightforward. Its function signature is `func(float32, float32) float32`.

The body mass index is defined to be a person's weight divided by a square of the person's height, in kilograms and meters, respectively. Since we accept the user inputs in pounds and inches, we convert them first, and then compute the BMI.

As stated, as a general rule, it is best to use `float64` for computation even if the argument and return values are in `float32`. The function-like notation, `float64()`, represents type conversion to `float64`. Likewise, `float32()` represents type conversion to `float32`.

We use `bufio.Reader` to read the user input, which we used in earlier lessons.

Both `bufio.Reader.ReadString()` and `strconv.ParseFloat()` can return an error. It is a good practice to check the error values unless you are sure that it is safe to ignore them, or that there is little consequence in doing so.

A function can handle the errors, if it knows how to, or it can pass them to its caller.

In this example, the `main` function has the big picture, and it is probably best to defer the proper error handling to the `main` function.

```
if err != nil {
```

```
    return 0, err
}
```

In this particular example, the `main()` function simply terminates the program when any of the input values, weight or height, is invalid. The `readInput()` function could have done the same, but normally that is not the function's job whose primary responsibility is limited to reading an input text and convert it to `float32`.

Incidentally, this function `readInput()` could have been written in a somewhat simpler form using `fmt.Scanf()`, which we have used before.

bmi-calculator/input2.go

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func readInput2(prompt string) (input float32, err error) {
8      fmt.Printf("%s: ", prompt)
9      _, err = fmt.Scanf("%f", &input)
10     return
11 }
```

The `readInput2()` function uses named return values to further simplify the implementation.

12.3. Summary

We reviewed basics of type conversion in this lesson, for example, using

`strconv.ParseFloat()`.

We also reviewed error handling some more.

To terminate a program before the entire program ends in a normal fashion, we use the `os.Exit()` function.

Author's Note

Software Stack

Building software is not unlike building a skyscraper, or a pyramid.

There are things that go near the ground, and there are things that go near the top. Viewing software as a vertical stack of blocks is a very useful metaphor. Sometimes we look at things from top to bottom, and sometime we look at things from bottom to top.

In Go, the building blocks are `packages`. When you use a package from the standard library, for example, you are putting your block on top of the standard package block.

The closer to the ground, the packages do smaller but more generic tasks. The packages in the higher up do broader but more specific tasks.

At the top of the pyramid is the `main` package of your program, which nobody else can use.

In Go, packages cannot have circular dependencies. That is, if a package `A` uses/imports package `B`, and the package `B` uses/imports package `C`, then the package `C` cannot use/import the package `A`. Doing so would create a dependency cycle.

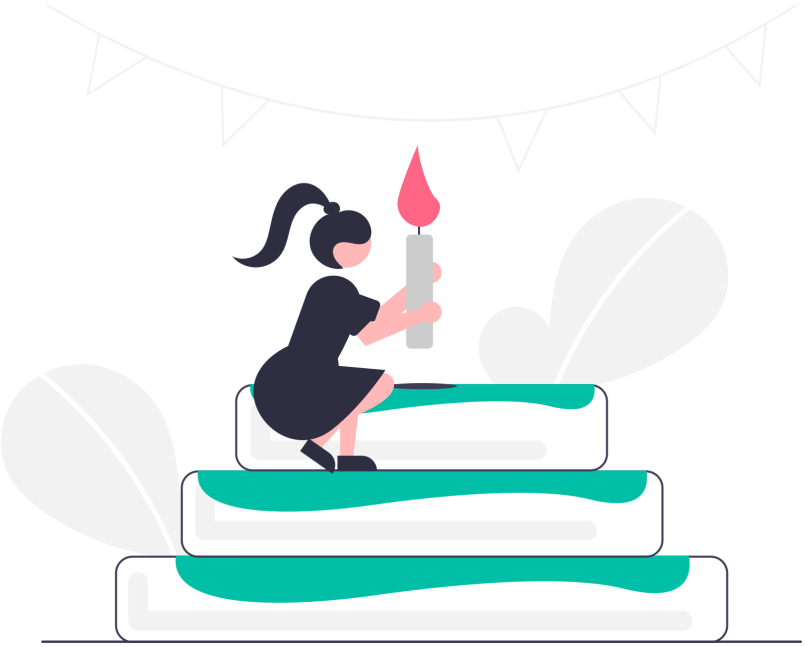
This is consistent with our "pyramid" view. A package at a higher level may depend on the packages below, but not the other way around.

Lesson 13. Birth Date

13.1. Agenda

We will learn how to use more than one packages in a program. That is, we will start using our first non-main package in this lesson.

We will also introduce "Go modules" for the first time in this book.



13.2. Code Reading

What day of the week was it when you were born?

The program accepts year, month, and day as an input, and prints out the day of the

week for the given date.

It uses the Go module.

birth-date/go.mod

```
module examples/birth-date

go 1.17
```

The *go.mod* file is in the same directory as the file that includes the `main()` function.

birth-date/main.go

```
package main

import (
    "fmt"
    "log"

    "examples/birth-date/week"
)

func main() {
    fmt.Print("Enter year (e.g., 2000), month (1~12), and date
(1~31): ")

    var y, m, d int
    if _, err := fmt.Scan(&y, &m, &d); err != nil {
        log.Fatalln("Scan for y, m, and d failed:", err)
    }

    weekday := week.Weekday(y, m, d)
```

```
19     fmt.Println("weekday =", weekday)
20 }
```

The *weekday.go* file is put in a subdirectory, *week*.

birth-date/week/weekday.go

```
 1 package week
 2
 3 import (
 4     "time"
 5 )
 6
 7 func Weekday(year, month, day int) time.Weekday {
 8     date := time.Date(year, time.Month(month), day, 0, 0, 0, 0,
   time.Local)
 9     return date.Weekday()
10 }
```

13.2.1. Explanation

You can run the program as before using *go run*:

```
go run .
```

You get the following output:

```
Enter year (e.g., 2000), month (1~12), and date (1~31): 2021 5 5
①
weekday = Wednesday
```

① Numbers *2021*, *5*, and *5* are user inputs. The date happens to be the day when this book is first published.

13.2.2. Grammar

Go was originally created to be used *uniformly* over the Internet. All libraries, or packages, across the world (with proper permission) can be used just like any other packages from the standard libraries or those from your own computers.

It was a lofty goal, but it was just an ideal. In practice, the libraries change. Their code change. They are often released as different versions. For example, instead of "version 1.0.0", a newer version might be released with "1.1.0" or even "2.0.0", etc.

The dependency management is one of the most important aspects of software engineering.

Go, as was initially released, did not have a tool for dependency management. From the early days, the need was obvious, and many developers started using "home grown" dependency management systems.

Eventually, the Go team released an official tool for dependency management. That is the "Go modules".

With a bit of over-generalization, if you write more than one programs on your computer, then you will need to use Go modules. That is, every Go programmer, for every program, needs to use Go modules.

We will not discuss, in this book, the old system of using the global `GOPATH` variable.

Go does not have a concept of a "project" or something similar. The `package` is the fundamental unit as far as the Go programming language is concerned.

A Go module provides a higher-level construct, which behaves like a "project".

A Go module can include one or more packages. It includes special files like "go.mod" in its root directory, and it provides a uniform dependency management over all packages under that go module directory.

The Go module is comparable to "venv", or other virtual environment tools, in Python. (But, not to pip packages, which more or less correspond to `packages`'s in Go.)

It is comparable to solutions and projects in the DotNet framework (e.g., for C#). It is comparable to a package (e.g., "project.json") in Node.js (Javascript/Typescript). Rust has a cargo project. Java uses tools like Maven and Gradle to manage their project dependencies.

Go uses a "module" to manage library/package dependencies, among other things.

You can create a Go module with the "go mod" command.

```
go mod init <module_name>
```

The name of a module can be a string with some restrictions. A path- or subpath-like string works fine in most cases. A URL-like string works fine in most cases. The module name, however, cannot start or end with a slash `/`.

The names of the packages within a module are interpreted relative to the module name.

Here's an example go.mod file:

```
module first-steps/example-33                ①

go 1.17                                      ②
```

```
require example.com/other/package v1.0.1      ③
```

① It declares a module with name "first-steps/example-33".

② It declares the Go language version used in this module.

③ The packages in this module depend on the version "1.0.1" of the library "example.com/other/package".

You need not include dependencies on the packages in the standard library. Also, you do not specify inter-dependencies among the packages within the same module.

Most of the example code used in this book do not have external dependencies, and hence Go modules are not needed, strictly speaking. However, as indicated, a Go module does a little bit more than external dependency management. It is still a good practice to use modules to manage your "projects".

If you have an external dependency in your project, specify the dependencies in the "go.mod" file, as shown in the require line in the above example, and you can use the Go command *go get* to download those dependent packages to your system. You can then `import` those packages in your program packages and use their exported names.

A Go module, or a Go program, can include multiple packages. A runnable Go program should have one and only one special package, `main package`. A Go module can have at most one `main package` within its subfolder hierarchy, starting from the module root directory.

Although the language specification does not explicitly specify, a package in Go corresponds to a folder in a file system. All source files in a folder should belong to the same package (with one possible exception, as we will see later). All source files of a package should be in one directory.

This is how it works with the current Go compiler tools.

If you have the `main` package in a module, then the main package directory should be the root directory of the module, where the "go.mod" file is.

When you refer to a package in a subfolder of a module root folder, you use the module name as a prefix to the package name.

If you have a package under a directory "sub", within a module with name "first-steps/example-33", for instance, then you can refer to that package as "first-steps/example-33/sub" from other packages in the module.

For example,

```
import (
    "first-steps/example-33/sub"
)
```

It is generally a convention to use the subfolder's name, the last segment in the file path, as a package name, if possible.

In a Go package, the names (e.g., identifiers of variables, constants, or functions) that are capitalized are exported. Go does not use special keywords like "export" or "public" as in other programming languages.

Exported names of a package can be used by other packages by importing the package.

Names that start with lowercase letters are not exported, and they can be accessed only within the package in which they are declared.

13.2.3. APIs

- Package log [https://golang.org/pkg/log/]: Package `log` implements a simple logging package. It defines a type, Logger, with methods for formatting output. It also

has a predefined 'standard' Logger accessible through helper functions Print[f|ln], Fatal[f|ln], and Panic[f|ln], which are easier to use than creating a Logger manually. That logger writes to standard error and prints the date and time of each logged message.

    - func Fatalln [https://golang.org/pkg/log/#Fatalln]: `Fatalln()` is equivalent to Println() followed by a call to os.Exit(1).

- Package time [https://golang.org/pkg/time/]: Package `time` provides functionality for measuring and displaying time. The calendrical calculations always assume a Gregorian calendar, with no leap seconds.

    - func Date [https://golang.org/pkg/time/#Date]: `Date` returns the Time corresponding to `yyyy-mm-dd hh:mm:ss + nsec nanoseconds` in the appropriate zone for that time in the given location. The month, day, hour, min, sec, and nsec values may be outside their usual ranges and will be normalized during the conversion. For example, October 32 converts to November 1.

    - type Time [https://golang.org/pkg/time/#Time]: A `Time` represents an instant in time with nanosecond precision. Programs using times should typically store and pass them as values, not pointers. That is, time variables and struct fields should be of type time.Time, not *time.Time.

    - type Month [https://golang.org/pkg/time/#Month]: A `Month` specifies a month of the year (January = 1, ...).

    - type Weekday [https://golang.org/pkg/time/#Weekday]: A `Weekday` specifies a day of the week (Sunday = 0, ...).

        - func Weekday [https://golang.org/pkg/time/#Time.Weekday]: `Weekday()` returns the day of the week for the given `time`.

    - var Local [https://golang.org/pkg/time/#Location]: `Local` represents the system's local time zone. On Unix systems, Local consults the TZ environment variable to find the time zone to use. No TZ means use the system default /etc/localtime. TZ="" means use UTC. TZ="foo" means use file foo in the

system timezone directory.

13.2.4. Deep Dive

In Go, a `package` is a basic unit of code sharing.

The program of this lesson include an extra `week` package in addition to the main package. The `week` package includes one source file, *weekday.go*, in this example.

```
package week
// ...
```

The source file includes one function, `func Weekday(year, month, day int) time.Weekday`. This function `Weekday()` is exported from the `week` package since its name starts with a capital letter, `W`, in this case.

The `main()` function uses this function via *import package* declaration.

```
import "examples/birth-date/week"
```

The import path is a concatenation of the module name, "examples/birth-date", and the last segment of the directory path, "week", where the source file(s) of the `week` package reside. They are combined as if the "week" folder is a subdirectory of a (hypothetical) folder named "examples/birth-date", the module name.

Through the import statement, all exported names of the `week` package are now available in this file, "main.go".

The names in the `week` package can be accessed using the package name as a prefix. For example,

```
day := week.Weekday(y, m, d)
```

It should be noted that the `week` in this prefix comes from the `package week` line of the source file *weekday.go,* which belong to the `week` package, not from the `import "examples/birth-date/week"` statement. By convention, we generally use the name of the package as a folder name of the package, but they could be different.

The import spec syntax is dictated by the Go tool chain, whereas the `package` names and the use of their exported names are governed by the Go language specification.

If you would like to use a different name than the default package name, then you can add a desired name to the import declaration. For example,

```
import w "examples/birth-date/week"
```

Now, the `week` package can be referred to as `w` in this source file. For example,

```
package main

import w "examples/birth-date/week"

func main() {
    // ...
    day := w.Weekday(y, m, d)
    // ...
}
```

The `Weekday()` function of the `week` package essentially "looks up" the given date (on a calendar), and it returns the day of the week for the specified date.

The `Date()` function from the time package return a value of type `time.Time`. The `Time` type has a method `Weekday()`, which returns the day of the week of the given date.

We have not really discussed "methods" yet, but a method is a function, with a slightly different syntax. In this example, we call the `Weekday()` function, or method, on the variable `date` (of type `Time`):

```
weekday := date.Weekday()
```

Its returned value, via `week.Weekday(y, m, d)`, is then printed in the `main()` function, and the program terminates.

The if statement in Go can have an initialization clause. In this example,

```
var y, m, d int
if _, err := fmt.Scan(&y, &m, &d); err != nil {
    log.Fatalln("Scan for y, m, and d failed:", err)
}
```

The statement `_, err := fmt.Scan(&y, &m, &d)` is executed first before the Boolean expression, `err != nil` in this case, is evaluated.

The `fmt.Scan()` function can return an error. If there is no error, then we proceed with the read numbers, `y`, `m`, and `d` after the `if` statement block.

If there is a non-nil error, on the other hand, then we log the error and terminate the program. The `log.Fataln()` function outputs the error (just like `fmt.Println()`) and then it calls `os.Exit()` with non-zero error code `1`.

A statements like `if err := doSomething(); err != nil { /* Do error handling */}` is one of the commonly used "idioms" in Go.

13.3. Summary

We introduced Go modules in this lesson. A module helps manage dependent external packages, among other things.

A Go module can contain more than one packages. Packages that are in the same module and are used by the main package should be `import`ed.

Author's Note

Don't Be a Parrot

It is not uncommon to see a beginning programmer copy code from a book to a computer. Or, copy code on the Internet to his/her computer.

Often they type the code, not even just copy and paste, and they claim that they *learn better* by actually typing.

There is no evidence for that. If anything, that will be a very inefficient way to learn programming. While imitation is an important part of learning a new language, or a new skill, mindless imitation would not enhance your speaking or writing skills very much.

The author made a conscious decision not to release the sample code of this book. For one thing, it has little value, really. But, there are other reasons as well. In his opinion, downloadable code samples do more harm than good.

Often learning students download a code sample and run it on their computers. And, they think that that's the end of of it. *It works. Now, let's move on.* In doing so, however, they have learned very little. On the contrary, they only ended up with the false sense that they were able to "write" the same code because they compiled the code and ran it. Or, because they even

"typed" the code.

Obviously, they did not *write* the code.

Although the author has asserted that this book is "for reading", if you are inclined to try out some sample code in this book, then here's a suggestion.

1. Learn the main points of the lesson.
2. Try to understand the sample code, and what it does.
3. Then *close the book*.
4. Recall the problem which the sample code is trying to solve.
5. Create your solution to the problem.

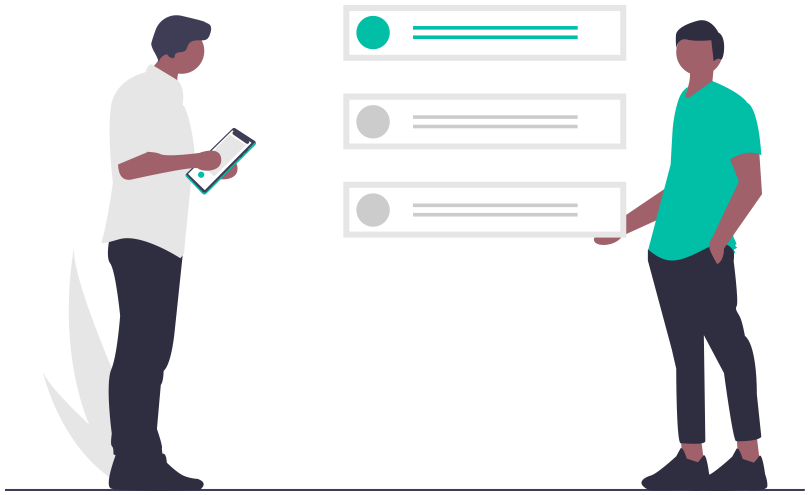
You may, or more likely may not, end up with the same code. But, that's perfectly all right.

If you get stuck, then refer back to the example code. Try to understand what it does, and how it does it. And then, close the book and try again.

Lesson 14. Greatest Common Divisor

14.1. Agenda

We will take a look at recursion in this lesson.



14.2. Code Reading

We will implement a function to find the greatest common divisor of two given numbers.

14.2.1. "go.mod"

We start by creating a Go module:

```
go mod init examples/greatest-common-divisor
```

The module names are largely arbitrary unless you are planning to let others use one or more packages in your module.

greatest-common-divisor/go.mod

```
1 module examples/greatest-common-divisor
2
3 go 1.17
```

14.2.2. Package *main*

We will see two versions of the greatest common divisor functions, one using recursion and the other using iteration.

We use a constant `useRecursive` to switch between these two implementations.

greatest-common-divisor/main.go

```
 1 package main
 2
 3 import (
 4     "examples/greatest-common-divisor/gcd"
 5     "fmt"
 6 )
 7
 8 const useRecursive = true
 9
10 func main() {
11     var a, b int64 = 30, 12
12     fmt.Printf("a = %d, b = %d\n", a, b)
```

```
13
14      var fn func(int64, int64) int64
15      if useRecursive {
16          fn = gcd.GCD1
17      } else {
18          fn = gcd.GCD2
19      }
20
21      g := fn(a, b)
22      fmt.Printf("gcd = %d\n", g)
23  }
```

14.2.3. Package *gcd*

The first implementation using recursion:

greatest-common-divisor/gcd/gcd1.go

```
1 package gcd
2
3 func GCD1(a, b int64) int64 {
4     if b == 0 {
5         return a
6     } else {
7         return GCD1(b, a%b)
8     }
9 }
```

The second implementation using iteration:

greatest-common-divisor/gcd/gcd2.go

```
1 package gcd
```

```
2
3 func GCD2(a, b int64) int64 {
4     for b != 0 {
5         a, b = b, a%b
6     }
7     return a
8 }
```

14.2.4. Explanation

The program includes two different implementations for computing the greatest common divisor of two integers.

Depending on the value of `useRecursive`, one or the other implementation is used.

If you run the program with *go run .*, with either `useRecursive == true` or `useRecursive == false`, you get the following output:

```
a = 30, b = 12
gcd = 6
```

14.2.5. Deep Dive

The program uses a Go module, as in the previous lesson. In fact, all the examples in this book use Go modules although its benefit is not entirely obvious in certain situations.

You can refer back to the the previous lesson, or you can refer to the official doc, Go Modules Reference [https://golang.org/ref/mod], for more information.

It includes two packages, `main` and `gcd`.

Each source file in a directory, which happens to be named "gcd", starts with the package declaration:

```
package gcd
```

The name of the `GCD1()` function starts with a capital letter `G`, and hence it is exported. Its signature is `func(int64, int64) int64`.

One interesting thing about the `GCD1()` function is that it calls itself in its function body.

```
func GCD1(a, b int64) int64 {
    if b == 0 {
        return a
    } else {
        return GCD1(b, a%b)
    }
}
```

It is called "recursion" in programming.

The greatest common divisor (GCD) of two positive integers is the largest positive integer that divides each of the integers. For example, the GCD of 8 and 12 is 4, and the GCD of 9 and 12 is 3.

The `GCD1()` function implements what is known as the Euclidean algorithm. Here's a link to the Wikipedia article: en.wikipedia.org/wiki/Greatest_common_divisor.

As Euclid first discovered, the GCD of two positive numbers, `a` and `b`, is the same as that of `b` and the remainder of `a` divided by `b`. When the remainder becomes zero, the other value in the pair, `a` in this case, is the greatest common divisor.

That is precisely what the `GCD1()` function implements.

You do not have to "understand" why, or how exactly, an algorithm works in order to be able to use it. You will just have to know what the exact steps are to implement the algorithm.

Recursive algorithms, or implementations, say, for a given problem, tend to appear more natural in many cases. They are not, however, the most efficient implementations, for the given problem, in general.

The second version of the GCD function, `GCD2()`, implements the logic in an iterative way.

```
func GCD2(a, b int64) int64 {
    for b != 0 {
        a, b = b, a%b
    }
    return a
}
```

This is the same Euclidean algorithm. It just uses the `for` loop iteration to find the greatest common divisor.

Note that, in each iteration of the loop, the pair `a, b` is replaced by `b, a%b`. Eventually, the remainder operation `b = a%b` will yield `0`, and the other value of the pair, `a`, is the GCD.

In the `main()` function, we declare a variable `fn` of type `func(int64, int64) int64`. Both `GCD1()` and `GCD2()` functions have the same type, and hence they can be assigned to this variable, `fn`.

Calling `fn()` will be the same as calling `GCD1()` or `GCD2()` depending on the value of `useRecursive`.

The program then prints out the result, and it terminates.

14.3. Summary

We introduced recursion in this lesson. We also reviewed function types.

Author's Note

What It Takes to Be a Good Programmer

Programming, or more broadly software development, involves a lot of different skills and talents.

First, you will need to be an expert in the language you use in programming. This is actually the easiest skill you can learn. Programming languages have well-defined grammar, unlike spoken languages, with a finite set of rules. Resources like this book can help you learn the languages.

Second, you will need to be familiar with commonly used libraries and APIs. It takes experience. The longer you program, the more familiar you will become with various libraries. It is not "difficult", but it just takes time.

Third, you will need some problem solving skills. This is not something you can learn by reading books or anything like that. But, you will get better over time as you train yourself by working on more problems. It helps to learn some common algorithms as well. "Familiarity" with diverse set of problems will be definitely useful when you face a new problem.

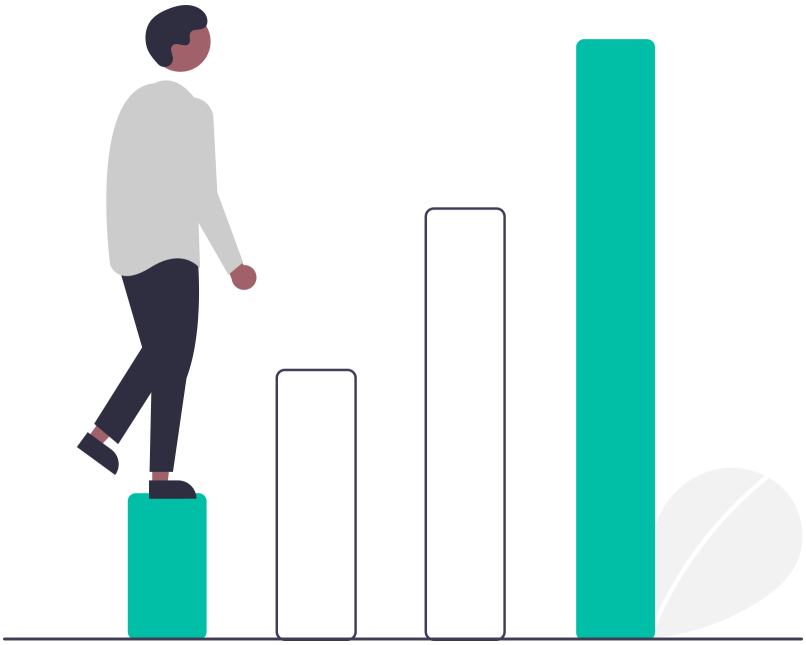
And, eventually, you will need to develop system design skills. This is different from programming skills. Creating a large software is like "building a pyramid", using the analogy we used before, or building a "starship" using lego blocks. You can only learn this skill by actually doing it, by building a

large scale software.

Lesson 15. Reverse a Number

15.1. Agenda

We will review Go's testing framework in this lesson.



15.2. Code Reading

This program "reverses" an integer number. That is, given a number `1234`, it produces another number `4321`.

15.2.1. Package *main*

The `main()` function handles input and output, and it delegates the core logic to

another package.

reverse-number/main.go

```
1  package main
2
3  import (
4      rn "examples/reverse-number/reverse"
5      "fmt"
6      "log"
7  )
8
9  func main() {
10     fmt.Print("Enter a number: ")
11
12     var num int64
13     if _, err := fmt.Scan(&num); err != nil {
14         log.Fatalln("Scan for number failed:", err)
15     }
16
17     reversed := rn.ReverseNumber(num)
18     fmt.Printf("Reversed number: %d\n", reversed)
19 }
```

15.2.2. Package *reverse*

The core function, `ReverseNumber()`, of this program is defined in a different package, `reverse`.

reverse-number/reverse/reverse.go

```
1  package reverse
2
3  func ReverseNumber(num int64) int64 {
```

```
    var reversed int64 = 0
    for num != 0 {
        reversed = reversed*10 + num%10
        num /= 10
    }
    return reversed
}
```

Note that the main.go file is located in a folder *reverse-number* (e.g., in a certain directory path) and the reverse.go file is located in its subfolder *reverse-number/reverse*.

15.2.3. Package *reverse_test*

We have a couple of simple unit tests for the `ReverseNumber()` function, depending on how you count the "tests". The test file is put in the same directory as *reverse.go*.

reverse-number/reverse/reverse_test.go

```
package reverse_test

import (
    "examples/reverse-number/reverse"
    "testing"
)

func TestReverseNumber(t *testing.T) {
    var number int64 = 1234
    var expected int64 = 4321
    got := reverse.ReverseNumber(number)
    if got != expected {
        t.Errorf("ReverseNumber(%d) = %d; want %d", number, got,
expected)
```

```
14      }
15
16      number = 24356879
17      expected = 97865342
18      got = reverse.ReverseNumber(number)
19      if got != expected {
20          t.Errorf("ReverseNumber(%d) = %d; want %d", number, got,
    expected)
21      }
22  }
```

15.2.4. Explanation

You can run the program as before from the main package directory:

```
go run .
```

You get the following output:

```
Enter an integer number: 1235678       ①
Reversed number: 8765321
```

① `1235678` is a user input.

You can run the test program(s) in the *reverse* folder as follows:

```
go test ./reverse
```

It will output something like this,

```
ok      examples/reverse-number/reverse 0.001s
```

Or, if the test fails,

```
--- FAIL: TestReverseNumber (0.00s)
    reverse_test.go:13: ReverseNumber(1234) = 1111; want 4321
FAIL
FAIL    examples/reverse-number/reverse 0.001s
FAIL
```

15.2.5. Deep Dive

Testing is built into the standard Go tool chain. This is one of the nicest features of Go.

It is beyond the scope of this book to discuss the program testing methodologies in general. We will do some unit testing in this lesson, and possibly in some future lessons.

We will try to provide an absolute minimum so that you can start testing your code right away. There are more resources on the Web if you want more thorough introduction to testing in Go.

The Go testing framework uses certain conventions.

For example, a file that includes test code should have a name that ends with *_test.go*. A function that starts with `Test` *and* a capital letter after that is a test function, which will run with the `go test` command.

A Go test function has a signature `func(*testing.T)`. The argument of type `*testing.T` will be used to manage the test states, and what not, by the testing

framework.

The `testing` package needs to be imported in all test files.

```
import "testing"
```

As for where to put the test files, there are a few different conventions across different programming communities. Some prefer to put `src` files and `test` files in separate folders, for example.

In Go, it is best to put the test file(s) in the same directory as the file being tested.

You can put the test files in the same package as the source files. Or, you can put the test files in a different package (but in the same directory).

As mentioned, this is an exception to the "one package - one folder" rule imposed by the Go compiler tools. You can put test files in a special test package named `<sourcepackage>_test`. That is, if the source files belong to a package named `reverse`, then you can put the test files in a package named `reverse_test`.

If you are primarily interested in testing exported names, as if you are an client of the source package, then it is better to use the `xxx_test` package.

If you want to test all internal implementations, then the test files should be put in the same package as the source files.

You cannot mix, however. You can have only one test package in a folder for all test files, whether it is the source file package or it is the `_test` package.

Now the type `testing.T` provides a number of functions and other types to make testing easier.

One thing to note is that Go testing framework does not provide the "assert" type

APIs.

It is almost universal across different test frameworks, across different programming languages, to have some kind of "assert" functions, which determine "pass" or "fail" of a test scenario. But Go does not follow such conventions.

In fact, Go's standard testing framework is much simpler.

You simply use conditional statements to test, for instance, if a certain evaluation works as expected.

When you determine that your code does not work as expected, you explicitly call `testing.T.Fail()`, or its variations, to let the testing framework know.

In this example,

```
func TestReverseNumber(t *testing.T) {
    var number int64 = 1234
    var expected int64 = 4321
    got := reverse.ReverseNumber(number)
    if got != expected {
        t.Errorf("ReverseNumber(%d) = %d; want %d", number, got,
expected)
    }
}
```

The `t.Errorf()` function first calls `t.Logf()` to log the error message, and then it calls `t.Fail()` to report it as a test failure.

In order to view test log messages even when a test does not fail, use `go test -v`.

- Package testing [https://golang.org/pkg/testing/]: Package `testing` provides support

for automated testing of Go packages. It is intended to be used in concert with the "go test" command, which automates execution of any function of the form `func TestXxx(*testing.T)` where Xxx does not start with a lowercase letter. The function name serves to identify the test routine.

- type T [https://golang.org/pkg/testing/#T]: `T` is a type passed to Test functions to manage test state and support formatted test logs. A test ends when its Test function returns or calls any of the methods FailNow, Fatal, Fatalf, SkipNow, Skip, or Skipf.
  - func (*T) Fail [https://golang.org/pkg/testing/#T.Fail]: `Fail` marks the function as having failed but continues execution.

The implementation of the example test function above is otherwise straightforward. Except for the use of type `testing.T`, it is just a normal Go function.

As a general comment, you do *not* have to use testing frameworks to test your code. In certain cases, you can just manually test your code, and that can be sufficient. As we have been doing in this book, you can just use the `main()` function for quick and dirty testing as well.

Use of testing frameworks is recommended in general, however. For many projects, automated testing is a must, for example, to catch regressions during an active development, etc.

If you have multiple packages, e.g., in the same Go module, then you can run *go test* for all packages as follows:

```
go test ./...
```

The full coverage of testing is beyond the scope of this book, but the `testing` package includes a lot of features which will help make your testing easier, among other things. You can refer to the official documentations for more information. For

example, the package doc, golang.org/pkg/testing/, includes all the APIs that you will need to create/run test cases.

The `main()` function of this lesson is simple. It uses `fmt.Scan()` to get an integer input, and it calls `ReverseNumber()`.

One thing to note is that the source file import statement uses a syntax that we have not seen before.

```
import rn "examples/reverse-number/reverse"
```

This import declaration renames the default package name of `reverse` to `rn`. Then, in this source file, we can refer to the package as `rn` rather than `reverse`.

That is what we do when we call `ReverseNumber()`:

```
reversed := rn.ReverseNumber(num)
```

15.3. Summary

We reviewed testing in Go.

The Go testing framework uses certain naming conventions. For example, a file that includes test code should have a name that ends with *_test.go*. A function that starts with `Test` (and a capital letter after that) is a test function.

You use *go test* command to run the test cases.

15.4. Exercises

1. The example code of this lesson has minimal error handling. What would you do if the reversed number overflows (when the input number is a valid `int64`)? Modify the code to handle such an error.
2. Implement `ReverseNumber()` using recursion.
3. Run the unit test for your recursive `ReverseNumber()` function.

Author's Note

Request for Review

Congratulations! You just finished the first part of this book. This could have been the most difficult part, depending on where you are coming from. The real fun starts from the second part, Moving Forward, where we cover Go's unique features like structs and methods as well as interfaces.

If you find this book helpful in any way, then please leave an honest review for other people, who may find this book useful in learning programming, and programming in Go.

Now, let's move forward! ☺

Review - Packages, Functions, Variables

Key Concepts

Packages

`Packages` are what Go programs are made up of. Programs start running in package `main`. You can import other packages using an `import` declaration. Your non-main packages may be imported by other programs as well.

A package comprises one ore more source files. Each source file must begin with the package declaration and the import statement(s), if needed.

Exported Names

A top-level name in a package that starts with a capital letter is exported. When importing a package, you can refer only to its exported names. Non-exported names are not accessible from outside the package.

Functions

A function can take zero or more parameters. Function arguments are declared with zero or more pairs of a variable name and its type, separated with comma `,`. A function can return zero or more results. Functions declare their return values using their types. Return variables can be optionally declared with names as well.

Basic Types

Go's basic types includes the following:

```
bool string int  int8  int16  int32  int64 uint uint8 uint16 uint32
```

```
uint64 byte rune float32 float64
```

All types but a few have fixed widths. `int` and `uint` types can be 32 or 64 bit wide depending on the system architecture.

Variables

The `var` statement declares a list of variables. In a var declaration, variable names are followed by their respective types. A var declaration can include initializers. If an initializer is present, the type can be omitted. The type will be "inferred" based on the type of the initializer.

A var statement can be at package or function level. Inside a function, the `:=` short assignment statement can be used in place of a var declaration with implicit type. At a package level, every statement must begin with a keyword (var, func, ...), and the short variable declaration cannot be used.

Zero Values

Variables declared without an explicit initial value are given their *zero* value, for example, `0` for numeric types, `false` for the boolean type, and "" (the empty string) for strings.

Constants

Constants are declared like variables, but with the `const` keyword. Constants can be string, boolean, or numeric values, including bytes and runes. Constants cannot be declared using the short variable assignment (`:=`) syntax.

Type Conversions

There is no implicit conversion between items of different types in Go. For example, an assignment of a value of one type to a variable with another type requires an explicit conversion. The expression `T(v)` converts the value `v` to the type `T`.

Type Inferences

When declaring a variable without specifying an explicit type, the variable's type is inferred from the value on the right hand side. When it is typed, the new variable is of that same type.

Flow Control

For Statement

The classic `for` loop has three components separated by semicolons, (1) the (optional) init statement: executed before the first iteration, if present, (2) the condition expression: evaluated before every iteration, and (3) the (optional) post statement: executed at the end of every iteration, if present. Both semicolons are required if either of the init or post statements are present. If neither exists, both semicolons can be omitted. In this case, the `for` loop is like the `while` loop in other programming languages.

If the condition expression is omitted, then it is equivalent to having a Boolean value `true`.

For Range

The for range loop iterates over an array, slice, or map. When ranging over a slice, two values are returned for each iteration. The first is the index, and the second is a copy of the element at that index.

If Statement

The `if` statement can start with a short statement to execute before the condition. Variables declared by the statement are only in scope within the `if` and any of the `else` blocks.

Advanced Types

Pointers

A pointer is a reference type. The type `*T` is a pointer to a value of type `T`. The `&` operator generates a pointer to its operand. The `*` operator denotes the pointer's underlying value. Pointer's zero value is `nil`.

Arrays

An array is a type for a sequence of values of a fixed length. The type `[n]T` is an array of `n` values of type `T`. An array's length is part of its type.

Slices

A slice is similar to an array, but it is dynamically-sized. It is a view into the elements of an underlying array. The type `[]T` is a slice with elements of type `T`. A slice is formed by specifying two indices, a low and high bound, separated by a colon. Slices are like references to arrays. The zero value of a slice is `nil`.

A slice has both a length and a capacity. The length of a slice is the number of elements it contains. The capacity of a slice is the number of elements in the underlying array, counting from the first element in the slice. The length and capacity of a slice s can be obtained using the expressions len(s) and cap(s).

Slices can be created with the built-in `make()` function. A new slice can be also initialized with a slice literal.

To append a new element(s) to a slice, the built-in `append()` function is used, which returns a new slice variable. If the backing array of s is too small to fit all the given values a bigger array will be allocated. The returned slice will point to the newly allocated array.

Error Handling

Errors

Go programs use function's return values to express error states. The `error` type is a built-in interface that is used for error values. When a function returns an error value, the calling code should handle errors by testing whether the error is `nil` or not. A `nil` error denotes success (i.e., "no error"), and a non-`nil` error denotes failure.

Part II: Moving Forward

There is no royal road to learning.

Lesson 16. Hello Morse Code

16.1. Introduction

Morse code is a method for encoding a set of alphabets and numbers used in telecommunications. It uses a combination of two types of signals, short (or, "dot") or long (or, "dash"), along with various length "gaps" (e.g., the silent part), to represent characters, and words.

For more information, refer to other resources on the Web. For example, the wikipedia page: en.wikipedia.org/wiki/Morse_code

In this lesson, we will write an "encoder" (alphabets to Morse code) and a "decoder" (Morse code to alphabets) in the Go programming language.

This is not a realistic program, and it is not intended to be a practical example. However, the concepts presented here could be useful in general programming, especially in the context of communications and encryption.

The program is primarily written as a "library", and it is included in a package named `morse`. The `main()` function in this example, as in many projects in this book, is mainly used as a "quick and dirty" test driver. As stated, the `main` package is not shareable.

16.2. Code Review

16.2.1. Package *main*

Here's a sample solution.

The `morse` package exports two functions, `Encode()` and `Decode()`. The main function calls these two functions with some sample data, and just prints out the results for visual inspection.

morse-code/main.go (lines 8-16)

```
8  func main() {
9      text2 := "Hello, World!"
10     code2 := morse.Encode(text2)
11     fmt.Printf("text: %s => code: %s\n", text2, code2)
12
13     code1 := ".... . .-.. .-.. --- --..--    .-- --- .-. .-.. -.. 
```

```
  -.-.--"
14      text1 := morse.Decode(code1)
15      fmt.Printf("code: %s => text: %s\n", code1, text1)
16  }
```

16.2.2. Package *morse*

For this example, we simply use strings for both English text and Morse code. We define a mapping from alphabets (and numbers and punctuations) to Morse code, and also its reverse mapping.

morse-code/morse/code.go (lines 8-18)

```
 8  var morseCode = make(map[byte]string, 26*2+10+16)
 9
10  func init() {
11      for k, v := range code {
12          morseCode[k] = v
13          if unicode.IsLetter(rune(k)) {
14              u := []byte(strings.ToUpper(string(k)))[0]
15              morseCode[u] = v
16          }
17      }
18  }
```

morse-code/morse/code.go (lines 20-26)

```
20  var reverseCode = make(map[string]byte, 26+10+16)
21
22  func init() {
23      for k, v := range code {
24          reverseCode[v] = k
25      }
```

```
26 }
```

Note the special package function, `init()`, used to initialize two package scope variables, `morseCode` and `reverseCode`.

Here we use an internal variable `code` of a type `map` to define the alphabet to (the string representation of) Morse code.

morse-code/morse/chars.go (lines 3-9)

```
3 var code = map[byte]string{
4     'a': ".-",
5     'b': "-...",
6     'c': "-.-.",
7     'd': "-..",
8     'e': ".",
9     'f': "..-.",
```

morse-code/morse/chars.go (lines 54-60)

```
54     '+':  ".-.-.",
55     '-':  "-....-",
56     '_':  "..--.-",
57     '"':  ".-..-.",
58     '$':  "---.--.",
59     '@':  ".--.-.",
60 }
```

In principle, characters in Go are `runes`, and using the rune type for the English alphabets, and numbers and punctuation symbols, would have been more appropriate. For the ASCII characters, however, runes and bytes are roughly interchangeable when dealing with strings in Go.

As explained earlier, strings in Go have a dual nature. A `string` can be viewed (physically) as a sequence of bytes, or (conceptually) as a sequence of runes or "characters". (The lengths of `byte`, `rune`, and Unicode character are 1, 2, and *2 or 4* bytes, respectively. Strings in Go use the UTF-8 encoding, which uses 1, 2, or 4 bytes for each character. An ASCII character (conceptually, a 2-byte `rune`) occupies 1 byte *in a Go string*. Dealing with the text (and, the date and time) is the "messiest" part in programming. 😀)

"Encoding" a text amounts to mapping the alphabets of the text to the corresponding strings representing the Morse code. In practice, Morse code involves a few different length gaps, etc. We will represent those gaps with spaces.

The implementation of `Encode()` is straightforward.

morse-code/morse/encode.go (lines 7-21)

```
 7 func Encode(text string) string {
 8     var sb strings.Builder
 9     for _, b := range []byte(text) {
10         if c, ok := morseCode[b]; ok {
11             sb.WriteString(c + " ")
12         } else {
13             if b == ' ' {
14                 sb.WriteString("   ")
15             } else {
16                 sb.WriteString("???")
17             }
18         }
19     }
20     return sb.String()
21 }
```

If a character is not representable by a Morse code, then we simply output "???" in this example.

Implementing the `Decode()` function requires a little more thinking. This is not necessarily an artifact of our toy example. Decoding Morse code is inherently more complicated than encoding.

This is because while the unit of a signal is dots and dashes (and gaps), it is a series of these signals (one or more) that represent a single character in English.

The following `Decode()` function implements one of the simplest solutions. One can probably implement this more efficiently using more advanced algorithms.

morse-code/morse/decode.go (lines 9-48)

```
 9 func Decode(code string) string {
10     var sb strings.Builder
11     var char []byte
12     var spaceCount = 0
13     for _, b := range []byte(code) {
14         if b != ' ' {
15             if spaceCount > 0 {
16                 if len(char) > 0 {
17                     letter, err := findChar(char)
18                     if err != nil {
19                         fmt.Println(err)
20                         sb.WriteString("?")
21                     } else {
22                         sb.WriteString(string(letter))
23                     }
24                 }
25                 char = []byte{}
26 
27                 if spaceCount > 1 {
28                     sb.WriteString(" ")
29                 }
30                 spaceCount = 0
31             }
```

```
            char = append(char, b)
        } else {
            spaceCount++
        }
    }
    if len(char) > 0 {
        letter, err := findChar(char)
        if err != nil {
            fmt.Println(err)
            sb.WriteString("?")
        } else {
            sb.WriteString(string(letter))
        }
    }

    return sb.String()
}
```

morse-code/morse/decode.go (lines 50-57)

```
func findChar(bytes []byte) (byte, error) {
    str := string(bytes)
    if c, ok := reverseCode[str]; ok {
        return c, nil
    } else {
        return 0b0, errors.New(fmt.Sprintf("Unrecognized code:
%s", str))
    }
}
```

This particular implementation relies on the assumption (specific to this example) that one space is used between characters and more than one spaces are used between words.

Note that the function `findChar()` is not exported from this package as it uses the lowercase first letter for its function name.

All exported names (variables, functions, etc.) should be documented, as a general practice. As stated, however, the content of the book serves as a documentation for these examples, and the sample programs in this book are mostly undocumented. We will discuss Go's "doc comments" in later lessons.

If you run the program as before:

```
go run .
```

You get the following output:

```
text: Hello, World! => code: .... . .-.. .-.. --- --..--    .-- ---
.-. .-.. -.. -.-.--
code: .... . .-.. .-.. --- --..--    .-- --- .-. .-.. -.. -.-.-- =>
text: hello, world!
```

16.3. Pair Programming

Let's start from the beginning.

When you are given a problem, as a general rule, it is best to solve the problem from top to bottom. That is, from the high-level organization, description, understanding, etc. to the lower-level details. This process often corresponds to what is known as a "design" in software engineering.

Then, you start to implement the lower level components first and build upward.

This is a general guideline in solving computational problems. This kind of strategy

may not work well in all problem domains.

In this particular example, we may need to read an input in some form (presumably, converted from the Morse code signals, electrical or otherwise), and convert them into English. On the reverse side, we can convert an English text to a sequence of symbols in certain formats (which can be read and converted to Morse code signals in some way, for instance).

Without considering specific (executable) programs, therefore, we can imagine that we will need the following "API" to support a broad range of programs.

```
func Encode(text string) (code string) { /* ... */ }
func Decode(code string) (text string) { /* ... */ }
```

In this particular example, as with most examples in this book, we do not have well-defined, and specific, requirements. In reality, we will most likely start from a set of particular system requirements (e.g., based on the business requirements), which will likely constrain our "API design".

In this lesson, we will use the use cases of the `main()` function as a requirement, as shown earlier. Then, this API design will most likely support those use cases.

Now, how do we implement the `Encode()` function? As stated, "encoding" is simply a "dictionary lookup", in this example. The Go programming language provides a builtin data type `map`, which we can use for this purpose. (`map` is a keyword in Go.)

- `map`: A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. The value of an uninitialized map is nil. The comparison operators `==` and `!=` must be fully defined for operands of the key type.

A `map` is another reference type in Go. (One other reference type we have seen so far is `slice`.)

It is like a hashtable, a dictionary, or a map in other languages. It stores key-value pairs, and it provides a way to retrieve the value corresponding to a given key.

A variable of the `map` type can be declared as follows:

```
var myMap1 map[string]int
```

Note the syntax. In this example, the type of the keys is `string`, and the type of the values is `int`.

A map can be initialized with an empty value:

```
myMap2 := map[byte]float64{}
```

Note the similarity with the empty slice initialization syntax (e.g., the use of `{}`). Alternatively, a map can be created with the builtin `make()` function with an (optional) initial capacity:

```
myMap3 := make(map[string][]byte, 10)
```

In this example, the value type is `[]byte`, a slice of bytes, and its initial capacity is 10. The types of map values can be just about anything, but the key types are limited to those that are "comparable".

The language spec defines this more precisely, but in short, comparable types are boolean, numeric, string, pointer, channel, and interface types, and structs or arrays that contain only those types. Slices, maps, and functions cannot be used as keys of a map since these types cannot be compared using `==`.

Note that, unlike in the case of slices, maps cannot be automatically initialized with

default values. The size of the map created this way is always zero, initially.

If you would like to provide some initial values, then you can use a "map literal" using the following syntax:

```
ages := map[string]int{
    "John": 100,
    "Joe":  80,
    "Mary": 70,
    "Jill": 50,
}
```

Now, you can use the square bracket notation `ages["John"]` to access an element, like in many C-style programming languages. In Go, however, accessing a non-existent key does not throw an error. It simply return the default value of the map's value type. For instance, in the above example, `ages["Lisa"]` will return `0`.

To differentiate the cases where the key does not exist in the map and where the value happens to be a default value, Go's map access returns two values, in fact.

It returns the map's value corresponding to the given key, if it exists, and a `bool` value as a second return value, which indicates whether the key exists in the given map or not.

This is an idiom in Go:

```
  if v, ok := ages[name]; ok {
    // v is the value of ages[name] for a given key, name
  } else {
    // the key, name, is not found in ages.
  }
```

Note the `if` statement with a variable initialization (`:=`), which we have discussed before. If the `ok` value of `ages[name]` is `true`, then the `if` branch is executed. The value `v` will be the value of `ages` at index `"name"`. Otherwise, the `else` branch is executed. In that case, the value of `v` will be merely the default/zero value of the type of the map's value, that is, `0` in this example. (Hence, we should normally ignore that value.) Using the variable name `ok` is a convention.

You can add a new element, or overwrite an existing item, this way:

```
ages["Michael"] = 40
ages["Joe"] = 10
```

The builtin function `len()` can be used to get the size of a map.

```
size := len(ages)
```

You can delete an existing element using another builtin function, `delete()`.

- `delete()`: The `delete` built-in function deletes the element with the specified key from a map. If the the map is nil or there is no such element specified by the key, `delete` is a no-op.

For example,

```
delete(ages, "Mary")
```

Deleting a non-existent element in a `map` is a null operation. It does not throw an error.

We can iterate over a map, in a similar way that we do over a slice or an array.

```
for name, age := range ages {
    fmt.Printf("%s is %d years old\n", name, age)
}
```

The first values in the `for-range` loop are the keys and the second values are the values of the map, e.g., `ages` in this example.

In this example, we use a map of `bytes` to `strings` to represent the mapping from the letters (e.g., Alphabets and numbers) to the Morse code:

```
var code = map[byte]string{}
```

The map, `code`, is initialized with all the characters relevant to English.

The Morse code does not distinguish upper and lowercase letters. For convenience, we create a new `map` that includes both lowercase and uppercase Alphabets in the init function (say, instead of converting them on the fly).

A Go package can contain *one or more* `init()` functions to set up whatever initial state is needed.

```
func init() {}
```

The `init()` functions do not return any value like the `main()` function. They run *after* all package scope variables and constants are initialized.

The `init()` functions in a source file are called after all `init()` functions in the `import`ed packages are called.

This new map initialized via the first `init()` function in *morse-code/morse/code.go*

is named `morseCode` in this example. Note that the map is created with an initial capacity of `26*2+10+16` to accommodate `26*2` Alphabets (upper and lowercase letters), `10` numbers, and `16` punctuation symbols.

The `Encode()` function uses a type `strings.Builder`.

- type Builder [https://golang.org/pkg/strings/#Builder]: A `Builder` is used to efficiently build a string using `Write` methods. It minimizes memory copying. The zero value is ready to use. Do not copy a non-zero Builder.
  - func (*Builder) WriteString [https://golang.org/pkg/strings/#Builder.WriteString]: `WriteString` appends the contents of a given string to the builder's buffer. It returns the length of the string argument and a `nil` error.
  - func (*Builder) String [https://golang.org/pkg/strings/#Builder.String]: `String` returns the accumulated string.

It goes through each character or byte in the given `text`, and if it has a corresponding Morse code (`ok == true`), then the code is added to the `strings.Builder`. If not, it adds three spaces for a space (`' '`) or an invalid value, `"???"`, otherwise.

The accumulated string is then returned via the `String()` method defined in the `strings.Builder` type.

Writing, or understanding, the `Decode()` function requires a little bit more thinking. We will leave this as an exercise to the reader. But, it essentially follows the same logic as `Encode()`. It starts with a map `reverseCode` and it iterates over the byte slice of the input `code`.

The only (main) difference is that we will need to read possibly more than one bytes (until the next space) to match the code to a corresponding letter. As before, processing a string as a slice of bytes (e.g., `[]byte(code)`) is kind of an idiom in Go. As indicated, however, a byte from this byte slice may not correspond to a "character" unless we are dealing with ASCII characters only.

16.4. Summary

We reviewed Go's builtin `map` type in this lesson. A map is a reference type.

A `map` stores key-value pairs. Go uses mostly similar syntax to those used in other programming languages for similar data types.

16.5. Exercises

1. Implement the same/similar function as the `Eecode()` method in this lesson (without directly copying it).
2. Study the implementation of the `Decode()` function, and implement the same functionality from memory (again, without directly copying the function, character by character, or even line by line).
3. Write a program that reads an input in Morse code and prints out a warning every time it sees a code corresponding to "SOS" in the input. (This is an open-ended/loosely-defined problem, as is the case with all/most exercise problems in this book. There are no specific requirements other than what is stated in the problem. For example, when you see "SOSOS", should you return one `SOS` or two `SOS`'s? Dealing with this kind of questions is part of software development.)

Lesson 17. "LED" Clock

17.1. Introduction

Before the time of Graphical User Interface, early users of computers often used character-based rendering, known as "ASCII art", to "draw pictures" on the screen, or on paper using "dot matrix printers", for instance: en.wikipedia.org/wiki/ASCII_art.

In this lesson, we are going to write a program that prints out the current time in "big letters".

Here's, for example, a big letter `0` in a two dimensional slice, that is, a slice of slices:

```
zero := [][]byte{
    {' ', '0', ' '},
    {'0', ' ', '0'},
    {'0', ' ', '0'},
    {'0', ' ', '0'},
    {' ', '0', ' '},
}
```

We just use the smallest possible dimension, 5 by 3, in this example, which can clearly represent all 10 digits.

Go does not have "multi-dimensional" arrays or slices, but one can define a type of a slice/array of slices/arrays, whose elements can be, in turn, of a type slice/array, etc.

```
var jagged [][]int
```

This declaration is equivalent to

```
type intSlice = []int
var jagged []intSlice
```

Note that a slice of slices can be "jagged". That is, each element (a slice) in the slice `[]intSlice` can have different lengths. (We will discuss the keyword `type` later in this lesson.)

The purpose of the program in this lesson is to write the current time in these "big letters". For instance, here's a sample display from the example program.

```
====================
22   0      33  555
  2 0 0  *    3 5
 2  0 0     33  55
2   0 0  *    3   5
222  0      33  55
====================
```

In case it's not very clear to you, it says *20:35* in the ""military time", which is 8:35PM in the "civilian time" 😀. It is rather "low tech", but you can see it better if you squint your eyes. 😊

Once we have a set of `[][]byte` for all pertinent letters, printing them out in a vertical way is trivial. For example,

```
 0
0 0
0 0
0 0
```

```
 0

22
  2
 2
2
222

 44
4 4
4 4
444
  4
```

This can be done by printing one big letter (`[][]byte`) after another, 0, 2, and 4, in this example.

Printing them horizontally, however, as is needed for our time display, requires a bit of thinking.

Other than that, the design of this program is straightforward. [1] Get the current time first, [2] "translate" it into *big letters*, and then [3] print out all the letters, horizontally put together.

17.2. Code Review

17.2.1. Package *main*

The main function of this program is simple:

led-clock/main.go (lines 7-9)

```
7 func main() {
8     big.DisplayTime()
9 }
```

17.2.2. Package *big*

The core functionality is included in the `DisplayTime()` function in the `big` package.

led-clock/big/time.go (lines 8-13)

```
8 func DisplayTime() {
9     now := time.Now()
10     displayStr := fmt.Sprintf("%02d:%02d", now.Local().Hour(),
  now.Local().Minute())
11     tm := CreateBigDigits([]byte(displayStr)...)
12     tm.Print()
13 }
```

(As stated, file names/paths have generally no relevance in go programs.)

The `DisplayTime()` function gets the current time via `time.Now()` and it converts the time into a string in a displayable format, namely, "HH:MM".

- Package time [https://golang.org/pkg/time/]: Package `time` provides functionality for measuring and displaying time. The calendrical calculations always assume a Gregorian calendar, with no leap seconds.
  - func Now [https://golang.org/pkg/time/#Now]: `Now` returns the current local time.
  - type Time [https://golang.org/pkg/time/#Time]: A `Time` represents an instant in time with nanosecond precision. Programs using times should typically store and pass them as values, not pointers. That is, time variables and struct fields should be of type time.Time, not *time.Time.
    - func (Time) Local [https://golang.org/pkg/time/#Time.Local]: Local returns the `Time` with the location set to local time.
    - func (Time) Hour [https://golang.org/pkg/time/#Time.Hour]: `Hour` returns the hour within the day specified by the `Time`, in the range [0, 23].
    - func (Time) Minute [https://golang.org/pkg/time/#Time.Minute]: `Minute` returns the minute offset within the hour specified by the `Time`, in the range [0, 59].

Then `DisplayTime()` passes the formatted string to the `CreateBigDigits()` function, which takes a variable number of bytes (or, "characters") as arguments.

led-clock/big/text.go (lines 8-39)

```
 8 type BigText [][]byte
 9
10 const height int = 5
11
12 var le = [height][]byte{}
13
14 func (c BigText) append(c1 BigText) BigText {
15     lx := make(BigText, height)
16     for i := range le {
17         lx[i] = append(c[i], c1[i]...)
18     }
19     return lx
20 }
21
22 func (c BigText) Print() {
23     len := len(c[0])
24     bar := strings.Repeat("=", len)
25
26     fmt.Printf("%s\n", bar)
27     for _, v := range c {
28         fmt.Printf("%s\n", v)
29     }
30     fmt.Printf("%s\n", bar)
31 }
32
33 func CreateBigDigits(digits ...byte) BigText {
34     lx := make(BigText, height)
35     for _, d := range digits {
36         lx = lx.append(let[d]).append(let[' '])
37     }
```

```
    return lx
}
```

The "big letters" are defined here:

led-clock/big/digits.go (lines 3-24)

```
var let = map[byte]BigText{
    '0': {
        {' ', '0', ' '},
        {'0', ' ', '0'},
        {'0', ' ', '0'},
        {'0', ' ', '0'},
        {' ', '0', ' '},
    },
    '1': {
        {' ', '1', ' '},
        {' ', '1', ' '},
        {' ', '1', ' '},
        {' ', '1', ' '},
        {'1', '1', '1'},
    },
    '2': {
        {'2', '2', ' '},
        {' ', ' ', '2'},
        {' ', '2', ' '},
        {'2', ' ', ' '},
        {'2', '2', '2'},
    },
```

led-clock/big/digits.go (lines 74-88)

```
    ' ': {
```

```
75          {' '},
76          {' '},
77          {' '},
78          {' '},
79          {' '},
80      },
81      ':': {
82          {' ', ' ', ' '},
83          {' ', '*', ' '},
84          {' ', ' ', ' '},
85          {' ', '*', ' '},
86          {' ', ' ', ' '},
87      },
88  }
```

The `big` package includes a couple of methods defined on the type `BigText`, which is a new type defined based off `[][]byte`.

The magic happens in the `BigText.append()` function. When big letters are concatenated, we combine the byte slices of each row. There are 5 rows in this example, and we use an array of 5 empty byte slices, `le`, for the `for` loop ranges.

Then, implementing `BigText.Print()` is straightforward. Just print all rows in the given BigText, from top to bottom. Printing the `bar` string, before and after the time, is merely for display purposes.

17.3. Pair Programming

We have discussed "types" in Go in the earlier lessons. We have dealt with primitive types like `int` and `float64`. We have used builtin reference types like `slice` and `map`.

A `type` plays a central role in programming in Go. This is similar to the roles that a

"class" plays in the object-oriented programming languages.

A type can be associated with a set of "methods", or functions. You can add methods to the types that you define (in the same package).

We will start tackling this problem by defining a new type `BigText`, which is equivalent to `[][]byte`. This is merely for convenience (and for illustration), and it is not strictly required. But, it leads to an idiomatic Go program. (The "convenience" here does not merely refer to naming. As we will see shortly, without this type definition our program could have ended up "less elegant".)

```
type BigText [][]byte
```

In this statement, using the keyword `type`, `BigText` is defined to be (more or less) equivalent to `[][]byte` (a slice of slices of bytes).

- `type`: A type definition creates a new, distinct type with the same underlying type and operations as the given type, and binds an identifier to it. The new type is called a defined type. It is different from any other type, including the type it is created from.

This statement does not create a type alias per se. It defines a *new type* `BigText`, which happens to behave just like `[][]byte`.

The keyword `type` is similar to `typedef` in C/C++. The `typedef` creates a type alias, just a different name for an existing type. On the other hand, the `type` definition in Go defines a new and distinct type.

The new type `BitText`, in this example, is identical to the type `[][]byte`, in every aspect (at least, as of this definition). Nonetheless, variables/constants of these types cannot be used interchangeably. Explicit type conversion is required.

A C-style typedef alias can be created using a different syntax. For example,

```
type smallText = [][]byte
```

The name `smallText` is just an alias to `[][]byte` in this case, and they can be used interchangeably (e.g., without requiring type conversions).

Although it is typical to create type definitions or aliases in a package scope, it is also possible to create new types/aliases in a function scope, or even in a block scope. A defined or alias type is valid only within the scope where it is defined/declared.

Types defined in a package scope can be exported. The `BigText` type is exported, in this example, as denoted by the capital `B`.

As indicated in the previous section, the design of this program is rather straightforward. We define "big letters", and create a way to write a "big string", or a sentence, using these big letters. Then we print each row of the byte slice as a string.

All this logic can be encapsulated into the new type `BigText`.

A method of a type is defined as a function with a "receiver" with that type, or its pointer type.

```
func (c BigText) Print() { /* ... */ }
```

In this example, `c` of type `BigText` is the receiver. The type of this method is `func(BigText)`. Note that the receiver type is the type of the first argument, before the function name.

You can call the method using the dot notation.

```
text := BigText{}           ①
text.Print()                ②
```

① Note the initialization syntax. This is equivalent to `[][]byte{}`.

② The function `Print()` is called on the variable `text`.

Note that we could not have declared a method like `Print()` on the type `[][]byte`. Methods of a type can be only defined in the package where the type is defined.

The `append()` method on type `BigText` takes an argument of type `BigText` and returns a value of type `BigText`. Note that `BigText` is a reference type since `[][]byte` is a reference type.

```
func (c BigText) append(c1 BigText) BigText {
    lx := make(BigText, height)
    for i := range le {
        lx[i] = append(c[i], c1[i]...)
    }
    return lx
}
```

Note that the type of the `append()` method is `func(BigText, BigText) BigText`. (The first parameter is the receiver.)

This function signature, and the fact that `BigText` is a reference type, allows us to do "chaining". For example, in the function definition of `CreateBigDigits()`,

```
func CreateBigDigits(digits ...byte) BigText {
    lx := make(BigText, height)
    for _, d := range digits {
        lx = lx.append(let[d]).append(let[' '])
```

```
    }
    return lx
}
```

We call the `append()` method twice in each iteration over the byte slice. (If you paid attention in the first part 😀, then you should know why this kind of chaining (e.g., `x.append().append()`) is possible for the reference variables, but not for the value type variables. Otherwise, this is a good time to stop and review the relevant lessons again. 😊)

Now, at this point, all we have to do is to get the current time and display it as `BigText`. That is done in the `DisplayTime()` function.

The `main()` function of this program simply calls `big.DisplayTime()`.

17.4. Summary

We introduced a type definition in this lesson. One can define a set of methods on a new type. The dot notation is used on a variable of that type to call its methods.

We will introduce a couple of different kinds of types in the coming lessons, including `struct`s and `interface`s.

Lesson 18. Euclidean Distance

18.1. Introduction

A distance between two points in a Euclidean space can be calculated using the Pythagorean theorem.

In this lesson, we will define a coordinate in a 2-D Euclidean space as a `Point` using Go's `struct`. Then, we will create a function which returns the distance between two given points.

The goal of this lesson is to introduce a `struct` and various features of Go that are relevant to `struct`s.

As an exercise, we will write a program that takes a `Point` as an input and writes the distance between the given `Point` and the previous `Point`. This continues in a loop. For the first `Point`, we will compute the distance of the point from the "origin", a `Point` with the coordinate, `(0, 0)`, which we call a "radius" of the `Point`.

The program terminates when an input `Point` is the origin.

18.2. Code Review

An "infinite loop" is a scary thing. Especially, to beginning programmers.

If you run your program and if it keeps running and running, and it does not terminate, then it is probably because there is a problem with your code. Generally, however, many programs are written to run "forever", that is, unless otherwise instructed.

At the heart of many programs with user interactions, for instance, are infinite loops. Many server programs run indefinitely, say, until they crash or they are explicitly stopped.

18.2.1. Package *main*

We use an infinite loop in this example to implement a user input handling. This is similar to an "event loop" typically found in a GUI program (or, in a GUI framework, hidden from the application developers).

Although we do not use "events" per se, the idea is the same. There is an infinite `for` loop at the heart of the `main()` function. (One may call it the "main loop".)

euclid-distance/main.go (lines 10~31)

```
10 func main() {
11     prev := euclid.Origin
12     for {
13         p, err := readPoint(repeat)
14         if err != nil {
15             log.Fatalln(err)
16         }
17 
18         if p == euclid.Origin {
19             fmt.Println("Now your are back to the origin.
```

```
    Exiting...")
20              os.Exit(0)
21          }
22
23          if prev == euclid.Origin {
24              fmt.Printf("The \"radius\" of the point is %.4f\n",
    p.Radius())
25          } else {
26              fmt.Printf("The distance of the new point from the
    previous point is %.4f\n", euclid.Distance(prev, p))
27          }
28
29          prev = p
30      }
31  }
```

We read an input (a `Point`) and prints out its distance from the previous point, or from the "origin" `(0, 0)` if it happens to be first input point. This `for` loop runs forever until the program is stopped (e.g., using Ctrl+C) until the input is the origin.

In the `main()` function, `os.Exit(0)` with the exit code `0`, i.e., a normal exit, is equivalent to a simple `return`. It is sometimes more, or less, readable to use one or the other.

The `main()` function uses the `readPoint()` function to read two numbers as a `Point`.

euclid-distance/main.go (lines 33-51)

```
33  const repeat = 3
34
35  func readPoint(repeat int) (euclid.Point, error) {
36      var x, y float32
37      for attempts := 0; ; {
```

```
38          fmt.Print("Input a point (x, y): ")
39          if _, err := fmt.Scanf("%f,%f", &x, &y); err != nil {
40              attempts++
41              if attempts <= repeat {
42                  fmt.Println("The input point should be a form \"x,
    y\", including the comma.")
43                  continue
44              } else {
45                  return euclid.Origin, err
46              }
47          }
48          break
49      }
50      return euclid.Point{X: x, Y: y}, nil
51  }
```

We use `fmt.Scanf()` to read a pair of floating point numbers and convert it to a `Point`. As specified by the format `"%f,%f"`, the two numbers have to be separated by a comma `,`. (Spaces are ignored.)

When there is an error in the input, we give the user a few more chances. If the user fails for `repeat` times, then we return the error to the caller, which "gracefully" terminates the program in this example.

We could have used an infinite loop in this case as well, but that would have been too intrusive even for this simple program. The only way to terminate the program (other than inputting an input in the correct format, which the user appears to be having a trouble in this case) would have been using "Ctrl+C" (or, whatever the termination signal is on the user's computer).

18.2.2. Package *euclid*

We define `Point` as a `struct` of two `float32` numbers. This type is exported, and

other packages can use the type `Point` as long as they have proper access to do so.

euclid-distance/euclid/point.go (lines 5-7)

```
5 type Point struct {
6     X, Y float32
7 }
```

18.3. Pair Programming

We have stated that types play a crucial role in Go programs.

One of the ways to create a `type` is using the `struct` keyword.

- `struct`: A `struct` is a sequence of named elements, called fields, each of which has a name and a type.

Field names may be specified explicitly or implicitly. A field declared with a type but no explicit field name is called an embedded field.

An embedded field must be specified as a type name `T` or as a pointer to a non-interface type name `*T`. The unqualified type name, without the package name prefix, acts as the field name.

A field or method `f` of an embedded field in a struct `x` is called promoted if `x.f` is a legal selector that denotes that field or method `f`. Promoted fields act like ordinary fields of a struct. But they cannot be used as field names in composite literals of the struct.

Given a struct type `S` and a defined type `T`, promoted methods are included in the method set of the struct as follows:

- If `S` contains an embedded field `T`, the method sets of `S` and `*S` both include

promoted methods with receiver `T`. The method set of `*S` also includes promoted methods with receiver `*T`.

- If `S` contains an embedded field `*T`, the method sets of `S` and `*S` both include promoted methods with receiver `T` or `*T`.

A field declaration may be followed by an optional string literal tag, which becomes an attribute for all the fields in the corresponding field declaration. An empty tag string is equivalent to an absent tag.

A struct type is a value type. A variable of a struct type can be declared just like those of any other types.

```
var p Point
```

In this declaration, the value of `p` is initialized with default values. The default value of a struct type comprises the default value of each field.

For example, in the case of `Point`, the default value will be `X: 0.0, Y: 0.0`.

A variable of a struct can be initialized this way, using a struct literal:

```
p := Point{X: 1.5, Y: 2.5}
```

Or, using a different formatting,

```
p := Point{
        X: 2.1,
        Y: 5.5,
    }
```

In this example, notice the trailing comma `,` at the end of the last field. This is required by *go fmt*.

The expression on the right hand side (a "struct literal") is an example of a " composite literal".

Values of not all fields will need to be specified in this literal syntax. The missing fields will have the "zero values" of the corresponding types. You can even change the order of the fields. For example, `Point{Y: 3.0}` is equivalent to `Point{X: 0.0, Y: 3.0}`. And, `Point{Y: 3.0, X: 1.0}` is equivalent to `Point{X: 1.0, Y: 3.0}`.

A struct can also be initialized using positional arguments. For example, `Point{1.0, 2.0}` is equivalent to `Point{X: 1.0, Y: 2.0}`. In this syntax, however, all field values are required. And, the order matters. The positional argument initializers are not as commonly used as the "field:value" pair initializers.

It will be a good exercise for the readers to think about why one would prefer one syntax over the other. In what situations. (Hint: There is no one "correct" answer.)

You can access the fields with *the dot notation,* for read and write:

```
p := Point{X: 1.0, Y: -2.0}
x := p.X
p.Y = 3.0
```

Go's `struct` is based on C's struct. It has some similarities to `class` in object oriented programming languages. But, they are very different constructs, in many respects.

Go's struct types have no constructors or destructors.

In the case of "complex" struct types, it is often a convention to create a builder function for the type, using a name `New()` or names that start with `New...()`.

In the example code, we have `NewPoint()` function for the `Point` type.

euclid-distance/euclid/new.go (lines 3-9)

```
3 func NewPoint(x, y float32) *Point {
4     p := Point{
5         X: x,
6         Y: y,
7     }
8     return &p
9 }
```

We can return a type or a pointer type to the given type. Many programmers prefer to return a pointer because the word "new" is often associated with functions that create and return a pointer or reference type, including Go's builtin new() function.

- `func new(Type) *Type`: The `new()` function allocates memory. The first argument is a type, not a value, and the value returned is a pointer to a newly allocated zero value of that type.

In this example, the `NewPoint()` function returns a pointer type of `Point`.

As mentioned before, returning a pointer to a local, or "auto", variable is generally not allowed in most C-style block-scoped languages that support pointer types.

In this `NewPoint()` function, for instance, when the function returns, the local variable `p` of value type `Point` (which is allocated on the stack) will be deleted. And, accessing its pointer is a disaster waiting to happen. Generally speaking.

Go, however, automatically takes care of the situations like this. The memory of the value `p` is allocated in the heap, and its value is copied. Its pointer `&p` is now safe to use even outside the scope of this function.

Use of struct literals as initializers is perfectly valid, and is often preferred. In fact, using the `New`-style constructors is generally discouraged for "small" types like `Point`, as we will discuss further later in the book.

Here's a test code for `NewPoint()`:

euclid-distance/euclid/new_test.go (lines 8~15)

```
8 func TestNew(t *testing.T) {
9     p := euclid.NewPoint(1.0, 2.0)
10     t.Logf("New point created: %s", *p)
11
12     if p.X != 1.0 || p.Y != 2.0 {
13         t.Fail()
14     }
15 }
```

The type of `p` in this code is `*Point`. Note that we use the same dot notation to access its fields regardless of whether a type is a struct or a pointer type to a struct. This is also true when we access its methods. For example, the syntax `p.X` is equivalent to `(*p).X` when `p` is a variable of a pointer type.

One thing to note in this case is that we use the formatting verb `%s` when we print the point's value using the "Printf"-like functions, `testing.T.Logf()` in this case.

Variables of any type `T` which has a function `String()` of a type `func(T) string` can be used in formatted print functions with the `%s` verb (s for string).

In our `Point` example, this is possible because we have defined the `String()` function for the type:

euclid-distance/euclid/string.go (lines 5-7)

```
5 func (p Point) String() string {
6     return fmt.Sprintf("(%.4f, %.4f)", p.X, p.Y)
7 }
```

`String()` is a method defined in the Stringer interface. We will cover `interface`s in later lessons.

The `Distance()` function between two points is implemented using `math.Hypot()` function:

euclid-distance/euclid/distance.go (lines 5-9)

```
5 func Distance(p1, p2 Point) float32 {
6     dx := float64(p1.X) - float64(p2.X)
7     dy := float64(p1.Y) - float64(p2.Y)
8     return float32(math.Hypot(float64(dx), float64(dy)))
9 }
```

We could have implemented the distance function as a method to `Point`. In this particular case, a function seems more natural since their symmetry is more obvious. That is, in principle,

```
Distance(p1, p2) == Distance(p2, p1)
```

This symmetry would have been less obvious if we used methods. E.g.,

```
p1.Distance(p2) == p2.Distance(p1)  // Not so obvious
```

As stated, floating point operations are approximate and there could be rounding errors. The strict equality test between (computed) float numbers are not generally used.

The `Radius()` method of `Point` is then defined using `Distance()`:

euclid-distance/euclid/point.go (lines 9-11)

```
9 func (p Point) Radius() float32 {
10     return Distance(Origin, p)
11 }
```

As we discussed in the previous lesson, "LED" Clock), the method declaration syntax includes a receiver, `(p Point)` in this case.

You can have either a value type or its pointer type as a receiver type. For example,

```
func (p *Point) MoveToOrigin() {
    p.X, p.Y = 0, 0
}
```

This method, `MoveToOrigin()`, using the receive `(p *Point)` resets the values of both `X` and `Y` to `0`.

```
p := Point{X: 3.0, Y: 4.0}
p.MoveToOrigin()
fmt.Println("p =", p)
```

The `Println()` statement will print out `(0.0000, 0.0000)`. This would not have worked if we used a value receiver `(p Point)`.

Note that `func (p Point) Radius() float32 {}` is more or less equivalent to `func Radius(p Point) float32 {}`. And, `func (p *Point) MoveToOrigin() {}` is more or less equivalent to `func MoveToOrigin(p *Point) {}`.

As explained before, for example, in A Tale of Two Numbers, you will have to pass arguments of pointer types to a function if you need to change the "content" of the variables. Variables of value types are *copied*. Changes to the copies have no effect to the original values.

We will come back to this question, in later lessons, as to when to use value receivers and when to use pointer receivers.

As will be further discussed later, `func (p *Point) MoveToOrigin()` may not be an ideal method to use for "small" types like `Point`, which can be consistently used as a value type. In this case, functions like `func MoveToOrigin(p Point) Point` or `func Move(target Point, delta Distance) Point` (where `type Distance Point`) might be a better choice.

Now, we have all the building blocks. In the spirit of the bottom-up approach, let's start building "the program" using these components, that is, the top-level `main()` function.

We have an infinite `for` loop that handles the user input.

```
func main() {
    prev := euclid.Origin
    for {
        p, err := readPoint(repeat)
        // ...
```

```
            prev = p
        }
    }
```

First, in each iteration, we get the user input as a `Point` variable.

The implementation of the `readPoint()` is straightforward. One thing to note here is the location of the `repeat` declaration in the file, "main.go".

```
const repeat = 3
```

Some styles prefer putting all `const`s and `var`s in the beginning of a source file. Some styles prefer putting them in the places closest to where they are used.

In this example, we could have done it either way. The const `repeat` happens to be used by the `readPoint()` function only, and placing it before `readPoint()` rather than, say, before `main()` seems appropriate.

In fact, since nobody else uses it (as is currently written), we can even put it inside the `readPoint()` function. It is generally, however, easier to read, and modify, the code when you place variables and constants where they are more easily visible. And, programs change. We update code over time, etc.

In this particular example, the number `3` is more or less arbitrary. And, you may decide to use a different number in the future. (That is why we use the const `repeat` in the first place instead of hard coding the number into the `if` Boolean expression.)

Next, when we receive a valid input point, we first compare it with the `Origin` because that is the program termination input in this example.

If the user inputs `Origin` (that is, `0, 0`), then we terminate the program using

`os.Exit(0)`.

If not, we print the "radius" of the point if it is the first iteration. Or, we print the distance between this point and the previous point in subsequent iterations. `Radius()` and `Distance()` are the functions/methods we created earlier, that is, they are the building blocks.

```
func main() {
    prev := euclid.Origin
    for {
        p, err := readPoint(repeat)
        // ...
        if prev == euclid.Origin {
            r := p.Radius()
            // ...
        } else {
            d := euclid.Distance(prev, p)
            // ...
        }

        prev = p
    }
}
```

Note the comparison operator (`==`) between two points. Equality comparison between variables of a user-defined struct type is automatically made available by Go based on the field-wise comparison.

In this example, `p == Origin` if and only `p.X = 0.0` and `p.Y == 0.0`.

(Note that `euclid.Distance(prev, p)` is the same as `p.Radius()` when `prev == euclid.Origin` in this example. Hence we did not need the `if` statement. We could have just called `euclid.Distance(prev, p)`.)

At the end of each iteration, we replace the `prev` var with the current point `p`.

```
prev = p
```

As stated, `Point` is a value type, and this assignment copies the values of `p.X` and `p.Y` to `prev.X` and `prev.Y`, respectively.

Before we move on to the next lesson, here's an interesting question.

Is Go an object oriented programming language?

The answer is, Yes and No. ☺

Obviously, the answer depends on what we mean by "object oriented programming" (OOP). It is debateable, but one of the most important aspects of OOP is "data encapsulation". And, an ability to expose the data only through a well defined "interface".

All object oriented programming languages have a construct called "class" or something comparable. A class can have "private" fields. A class can have "public" methods. They all satisfy this fundamental requirement of data encapsulation. And, that's why they are called "object oriented programming" languages.

Other characteristics like inheritance, polymorphism, etc., are all secondary.

Now, let's take a look at Go's `struct`.

Within a package, there is no such support. There is no data encapsulation. There is no private or public methods. Everything is visible to everyone. Go's `struct` is not a class. You cannot do OOP within a package.

On the other hand, there are some level of access controls across different packages. You can access only the exported names of another package. Nobody can

access non-exported names of your package. You can hide what you need to hide within a package and you can expose only the types (and methods, etc.) which you want to expose to the outside world.

You can definitely practice OOP in Go if you understand that it is `package` not `struct` that is comparable to `class` in other OOP languages.

Go does not really use the term "object". An object is an abstract concept in "object oriented programming". But it is also a term used in many OOP languages to refer to a value of a reference type. In Go, if you use a value of the pointer type corresponding to a struct type, then that is the closest thing to an "object" in the OOP languages. But, regardless, Go does not support inheritance-based polymorphism. So, the concept of "object" has less significance.

As stated, Go supports an "object" oriented programming style using `types`, and methods. Hence, a better terminology would be "type-oriented programming". An `interface` in Go can represent either a value type or a pointer type, as we will discuss in later lessons. Go's *type-oriented programming* style is not limited to "objects" or pointers.

18.4. Summary

We learned `struct` types in this lesson.

A `struct` type is a sequence of fields. `struct` allows us to create a new type, known as a composite type, from other existing types. A set of "methods" can be defined on a `struct` type.

18.5. Exercises

1. Write a function that accepts a slice of `Point`s and finds a pair that has the shortest distance among all the pairs of points.

Lesson 19. Area Calculation

19.1. Introduction

This lesson also includes a somewhat artificial example to demonstrate certain features of the Go programming language. In particular, `structs` and `interfaces`.

We will define a few types that represent geometric "shapes" like a rectangle, an isosceles (a triangle with two equal sides), and a circle.

We will define a couple of methods to compute its area and perimeter for each shape.

Now, the problem that we are going to tackle in this lesson is to create a function that accepts an arbitrary shape (among the three we have defined) and computes its area. Likewise, create a function that accepts an arbitrary shape and computes its perimeter.

This is a kind of "polymorphism", and this is where Go's `interface` is used.

19.2. Code Review

As before, we take a top-down approach to discuss the problem.

19.2.1. Package *main*

We define a few "demo" functions in the `main` package to test-drive the functions we are going to create. We could just use unit test functions for this purpose, but it should be easier to read these normal functions, as presented in the book.

The `main` function of this program simply calls these demo functions, one at a time.

The first demo function will create a slice of "shapes" and compute their total areas and total perimeters. It simply uses the `area()` and `perimeter()` methods of each shape.

area-calculation/shapes.go (lines 11-26)

```
11 func shapesDemo() {
12     shapes := []shape.Shape{}
13     shapes = append(shapes, isosceles.New(1.0, 2.0))
14     shapes = append(shapes, rectangle.New(2, 3))
15     shapes = append(shapes, circle.New(2))
16
17     totalArea := 0.0
18     totalPerimeter := 0.0
19     for _, s := range shapes {
20         totalArea += float64(shape.Area(s))
21         totalPerimeter += float64(shape.Perimeter(s))
22     }
23
24     fmt.Printf("totalArea = %.5f\n", totalArea)
25     fmt.Printf("totalPerimeter = %.5f\n", totalPerimeter)
```

```
26 }
```

`Shape` is an interface type as we will see shortly.

The second demo will illustrate a use of a function which accepts an arbitrary number of variables of any type that has `Area()` method defined and returns their total area.

area-calculation/areas.go (lines 11-19)

```
11 func areasDemo1() {
12     areaer := []shape.Areaer{}
13     areaer = append(areaer, isosceles.New(1.0, 2.0))
14     areaer = append(areaer, rectangle.New(2, 3))
15     areaer = append(areaer, circle.New(2))
16
17     totalArea1 := shape.Areas(areaer...)
18     fmt.Printf("totalArea1 = %.5f\n", totalArea1)
19 }
```

`Areaer` is another interface type which we will introduce shortly.

The third demo is the same except that we weill use the `Perimeter()` functions instead of `Area()`.

area-calculation/perimeters.go (lines 11-19)

```
11 func perimetersDemo1() {
12     a := []shape.Perimeterer{}
13     a = append(a, isosceles.New(1.0, 2.0))
14     a = append(a, rectangle.New(2, 3))
15     a = append(a, circle.New(2))
16
```

```
    totalPerimeter1 := shape.Perimeters(a...)
    fmt.Printf("totalPerimeter1 = %.5f\n", totalPerimeter1)
}
```

In this example, `Perimeterer` is another interface type.

An `interface` is essentially a set of methods, with a name. Go's `interface` is similar to the interfaces in other object-oriented programming languages. But, there are some crucial differences as well.

For one thing, we do *not* have to define an interface when we create a type. We only need an interface when we use the types. We will give more detailed explanation later in this lesson.

19.2.2. Package *rect*

We define the `Rectangle` type as follows:

area-calculation/rect/rectangle.go

```
package rect

type Rectangle struct {
    width  float32
    height float32
}

func New(w, h float32) Rectangle {
    return Rectangle{width: w, height: h}
}

func (r Rectangle) Area() float32 {
    a := float64(r.width) * float64(r.height)
    return float32(a)
```

```
15 }
16
17 func (r Rectangle) Perimeter() float32 {
18     p := 2 * (float64(r.width) + float64(r.height))
19     return float32(p)
20 }
```

The `Area()` and `Perimeter` methods are defined on this type, `rectangle`, as well as the `New()` function.

19.2.3. Package *iso*

We define the `Isosceles` type as follows:

area-calculation/iso/isosceles.go

```
1  package iso
2
3  import "math"
4
5  type Isosceles struct {
6      base   float32
7      height float32
8  }
9
10 func New(w, h float32) Isosceles {
11     return Isosceles{base: w, height: h}
12 }
13
14 func (t Isosceles) Area() float32 {
15     a := 0.5 * float64(t.base) * float64(t.height)
16     return float32(a)
17 }
18
```

```
19 func (t Isosceles) Perimeter() float32 {
20     h := math.Hypot(0.5*float64(t.base), float64(t.height))
21     p := float64(t.base) + 2*h
22     return float32(p)
23 }
```

The `Area()` and `Perimeter` methods are defined on this type `isosceles` as well.

19.2.4. Package *circ*

The type `Circle` looks like this:

area-calculation/circ/circle.go

```
 1 package circ
 2
 3 import "math"
 4
 5 type Circle struct {
 6     radius float32
 7 }
 8
 9 func New(r float32) Circle {
10     return Circle{radius: r}
11 }
12
13 func (c Circle) Area() float32 {
14     a := 0.5 * math.Pi * float64(c.radius) * float64(c.radius)
15     return float32(a)
16 }
17
18 func (c Circle) Perimeter() float32 {
19     p := math.Pi * float64(c.radius)
20     return float32(p)
```

```
21 }
```

The `Circle` type also has a similar set of methods, `Area()` and `Perimeter`.

19.2.5. Package *shape*

In order to create functions with polymorphic behavior in Go, we will need to introduce interfaces.

The interface type definition uses the following syntax, which is similar to those used in defining other types. It uses the `interface` keyword:

area-calculation/shape/area.go (lines 3-5)

```
3 type Areaer interface {
4     Area() float32
5 }
```

The interface type `Areaer` includes one method `Area()`. The name `Areaer` looks rather strange, but it is generally a convention that, in case of an interface with one method, we use the name of the method + "er" as the interface name. We mentioned the Stringer interface before, which includes one method `String()`.

- `interface`: An interface type specifies a method set called its interface. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to implement the interface. An interface type may specify methods explicitly through method specifications, or it may embed methods of other interfaces through interface type names.

Now we can define an `Area()` function that takes an argument of type `Areaer`.

area-calculation/shape/area.go (lines 7-9)

```
7 func Area(shape Areaer) float32 {
8     return shape.Area()
9 }
```

The function signature `func(shape Areaer) float32` indicates that this function accepts, as an argument, a variable of any type which includes the `Area()` method, as declared by the `Areaer` interface, and it returns a `float32` value. That is the "contract".

As can be seen from the `Area()`'s implementation, in this example, the function does use the fact that it can call the argument's `Area()` method. Variables of any type that does not include the `Area()` method with the same signature could not have been passed in to this function.

We can define a similar interface type for the types that have the `Perimeter()` method.

area-calculation/shape/perimeter.go (lines 3-5)

```
3 type Perimeterer interface {
4     Perimeter() float32
5 }
```

The name of the interface `Perimeterer` is again based on the name of the method `Perimeter()`.

Now, we can define a polymorphic function, `func Perimeter(shape Perimeterer) float32`, that computes the perimeter of an argument as long as its type is `Perimeterer`, that is, as long as it has a method `Perimeter()`.

area-calculation/shape/perimeter.go (lines 7-9)

```
7 func Perimeter(shape Perimeterer) float32 {
8     return shape.Perimeter()
9 }
```

As in the case of the `Area(Areaer)` function, this function's implementation also relies on the constraint that the argument `shape` has the method `Perimeter()` with the same type.

Note that, in case of "concrete" types like structs, the function signature of a method includes a receiver. On the other hand, a method in the interface type does not include receivers.

That is just a syntactic difference. As long as everything else is the same, the functions types of the methods of a concrete type and an interface type are considered the same.

Interfaces can be combined. For example,

area-calculation/shape/shape.go (lines 3-6)

```
3 type Shape interface {
4     Areaer
5     Perimeterer
6 }
```

The interface type `Shape` has a method set that includes both `Area()` (from `Areaer`) and `Perimeter` (from `Perimeterer`).

For illustration, we also define two functions that take an arbitrary number of `Areaer` or `Perimeterer` arguments and return their total areas and perimeters, respectively.

area-calculation/shape/area.go (lines 11-17)

```
11 package shape
12
13 type Areaer interface {
14     Area() float32
15 }
16
17 func Area(shape Areaer) float32 {
18     return shape.Area()
19 }
20
21 func Areas(shapes ...Areaer) float32 {
22     totalArea := 0.0
23     for _, s := range shapes {
24         totalArea += float64(s.Area())
25     }
26     return float32(totalArea)
27 }
```

area-calculation/shape/perimeter.go (lines 11-17)

```
11 func Perimeters(shapes ...Perimeterer) float32 {
12     totalPerimeter := 0.0
13     for _, s := range shapes {
14         totalPerimeter += float64(s.Perimeter())
15     }
16     return float32(totalPerimeter)
17 }
```

19.3. Pair Programming

Let's start from the `shapesDemo()` function.

The type of the variable `shapes` is a slice of `Shape`s.

```
shapes := []shape.Shape{}
```

We add a few "shapes" to `shapes`, a `Rectangle`, an `Isosceles`, and a `Circle`. Then we iterate over `shapes`.

```
for _, s := range shapes {
    totalArea += float64(shape.Area(s))
    totalPerimeter += float64(shape.Perimeter(s))
}
```

The type of `s` is the interface type `Shape`. The function `shape.Area()` is polymorphic. Although `s` is declared to be of interface type `Shape`, the implementation of `shape.Area()` does the proper area calculation based on the specific type of `s`.

The same with `shape.Perimeter()`.

Now, as for the `areasDemo1()` function of *area-calculation/areas.go*, `shape.Areas()` is a variadic function, which takes a variable number of arguments of type `Areaer`.

When we pass variables of types, `Rectangle`, `Isosceles`, and `Circle`, to `shape.Areas()` (as defined in *area-calculation/shape/area.go*), the implementation uses the "correct" implementation of the `Area()` method based on the type of each argument. The behavior is again polymorphic.

Go uses `interface` types for polymorphism, *at the point of use,* so to speak. In other object programming languages, interfaces are used to constrain a type, e.g., a `class`, *at the point of type definition.* Java, for example, uses the keyword `implements` (or, `extends`) to declare that an object of a class can behave like a certain interface. C# uses a slightly different syntax (e.g., using ":"), but the idea is the same.

Go, on the other hand, does not use `interface`s to add a behavior to a type.

For example, the type `Rectangle`, in this example, does not know anything, and does not care, about the `Areaer` interface, or the `Perimeterer` interface.

The polymorphic behavior happens at the point of use. In the `shapesDemo()` function, for example, functions `shape.Area()` and `shape.Perimeter()` called with arguments of an interface type correctly identifies their concrete type.

As long as a variable of a type `X` has a method `Do()` which has the same name and function type as `Do()` defined in an interface `Doer`, the variable can be used as if it is a type `Doer`, in any place where a `Doer` is required.

The same holds true for interfaces with more than one methods. As long as a type implements *all* methods in the "method set" of an interface, a variable (or, a constant or a literal) of that type can be used in any place where a type of that interface is required.

The empty interface, `interface{}`, is special in that any variable (constant, literal) can behave like the empty interface type, since there is no requirement imposed by its method set (which is empty).

In many object oriented programming languages that support inheritance, there is something like an "Object" type that (almost) everything else inherits from. In Go, the empty interface, `interface{}`, is sort of like that top-level Object. Any variable can be cast to the `interface{}` type.

One thing to note is that an interface provides a polymorphic behavior, but not a slice of interfaces. A `Shape` is an `Areaer`, and it is a `Perimeterer`. But, a slice of `Shape`s is not a slice of `Areaer`s. It is not a slice of `Perimeterer`s. It is not a slice of `Rectangle`s.

If we need to use `[]Shape` for `[]Areaer`, for example, you will need to explicitly convert them. There is no polymorphic behavior.

area-calculation/areas.go (lines 21-33)

```
21 func areasDemo2() {
22     s := []shape.Shape{}
23     s = append(s, isosceles.New(1.0, 2.0))
24     s = append(s, rectangle.New(2, 3))
25     s = append(s, circle.New(2))
26 
27     b := make([]shape.Areaer, len(s))
28     for i, d := range s {
29         b[i] = d
30     }
31     totalArea2 := shape.Areas(b...)
32     fmt.Printf("totalArea2 = %.5f\n", totalArea2)
33 }
```

The same with `[]Shape` vs `[]Perimeterer`.

area-calculation/perimeters.go (lines 21-33)

```
21 func perimetersDemo2() {
22     s := []shape.Shape{}
23     s = append(s, isosceles.New(1.0, 2.0))
24     s = append(s, rectangle.New(2, 3))
25     s = append(s, circle.New(2))
26 
```

```
27      b := make([]shape.Perimeterer, len(s))
28      for i, d := range s {
29          b[i] = d
30      }
31      totalPerimeter2 := shape.Perimeters(b...)
32      fmt.Printf("totalPerimeter2 = %.5f\n", totalPerimeter2)
33  }
```

19.4. Summary

We introduced `interface`s in this lesson.

An `interface` defines a "behavior" in terms of a set of methods which a type has to satisfy, and it can be used to support polymorphism in Go.

19.5. Questions

1. What is `interface` in Go?
2. What is `struct`?

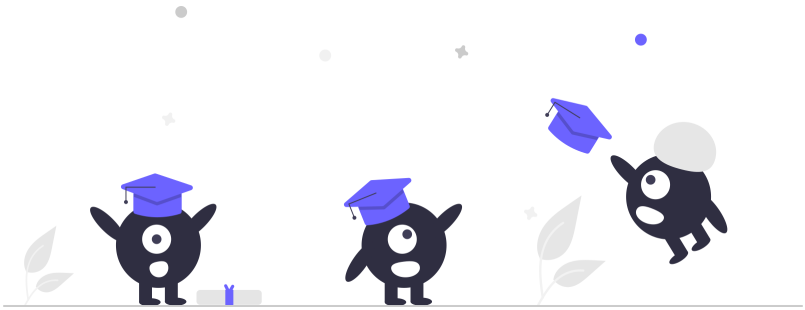
Lesson 20. Rock Paper Scissors

20.1. Introduction

We have covered a lot of important topics in the last few lessons. `Types`, `structs`, and `interfaces`. And, more.

In this lesson, we are going to take it a little bit easy, and work on a rock paper scissors game. Rock paper scissors is one of the most well-known games, which does not really require introduction.

The program of this lesson lets you play rock paper scissors with the computer.



20.2. Code Review

20.2.1. Package *main*

The `main` function of this program merely creates a variable of type `Game`, and it calls the `Game`'s main method, `Start()`.

rock-paper-scissors/main.go (lines 7-10)

```
7 func main() {
```

```
8     game := rps.NewGame()
9     game.Start()
10 }
```

20.2.2. Package *rps*

The `rps` package includes a package init function, which is called when the package is imported.

rock-paper-scissors/rps/random.go (lines 8-10)

```
8 func init() {
9     rand.Seed(time.Now().UnixNano())
10 }
```

The `rand.Seed()` function seeds the random number generator from the `math/rand` package.

A struct `Game` is the main type in this example:

rock-paper-scissors/rps/game.go (lines 7-9)

```
7 type Game struct {
8     wins, losses, ties int
9 }
```

The main logic of the program is implemented in the `Start()` function:

rock-paper-scissors/rps/game.go (lines 20-49)

```
20 func (g *Game) Start() {
21
```

```
    fmt.Println("------------------------------------------------------
    ")
22      fmt.Println("Welcome to Rock Paper Scissors!")
23      fmt.Println("Type X or Q to end the game.")
24
    fmt.Println("------------------------------------------------------
    ")
25
26      for {
27          playerHand, err := readHand()
28          if err != nil {
29              fmt.Println("Error:", err)
30              continue
31          }
32          fmt.Printf("Your Hand = %s\n", playerHand)
33
34          myHand := randomHand()
35          fmt.Printf("My Hand = %s\n", myHand)
36
37          wol := compareHands(playerHand, myHand)
38          if wol == Win {
39              g.wins++
40          } else if wol == Lose {
41              g.losses++
42          } else {
43              g.ties++
44          }
45
46          fmt.Printf("Your wins: %d, losses: %d out of %d plays\n",
    g.wins, g.losses, g.wins+g.losses+g.ties)
47
    fmt.Println("------------------------------------------------------
    ")
48      }
```

```
49 }
```

This function, when invoked, prints out a welcome banner first, and then it starts a game loop (the infinite `for` loop).

Each iteration of the loop corresponds to one "play" of hands. It first reads the player's hand, it generates its hand (the computer's hand), and it compares the two hands to decide who wins. It then updates the `struct`'s fields, wins, losses, and ties.

The game continues until the player inputs `Q` (Quit) or `X` (eXit).

If you run the program,

```
go run .
```

It prints out the banner and the first prompt.

```
--------------------------------------------------
Welcome to Rock Paper Scissors!
Type X or Q to end the game.
--------------------------------------------------
Rock (R), Paper (P), or Scissors (S)?
```

Here's an example of a round of plays.

```
--------------------------------------------------
Welcome to Rock Paper Scissors!
Type X or Q to end the game.
--------------------------------------------------
Rock (R), Paper (P), or Scissors (S)?
r
```

```
Your Hand = Rock
My Hand = Rock
Your wins: 0, losses: 0 out of 1 plays
-----------------------------------------------------
Rock (R), Paper (P), or Scissors (S)?
p
Your Hand = Paper
My Hand = Paper
Your wins: 0, losses: 0 out of 2 plays
-----------------------------------------------------
Rock (R), Paper (P), or Scissors (S)?
s
Your Hand = Scissors
My Hand = Paper
Your wins: 1, losses: 0 out of 3 plays
-----------------------------------------------------
Rock (R), Paper (P), or Scissors (S)?
s
Your Hand = Scissors
My Hand = Scissors
Your wins: 1, losses: 0 out of 4 plays
-----------------------------------------------------
Rock (R), Paper (P), or Scissors (S)?
r
Your Hand = Rock
My Hand = Scissors
Your wins: 2, losses: 0 out of 5 plays
-----------------------------------------------------
Rock (R), Paper (P), or Scissors (S)?
q
Thanks for playing the game!
```

20.2.3. Pair Programming

An iteration of the game loop in the `Game.Start()` function essentially consists of three actions:

1. Read the player's hand,
2. Generate the computer's hand, and
3. Compare the two hands.

Depending on the outcome, the values of the three fields of the type `Game`, `wins`, `losses`, and `ties`, are updated.

The main functionality of each of these steps is implemented in `readHand()`, `randomHand()`, and `compareHands()` functions, respectively.

The `readHand()` function is defined in the file, *rps/input.go*.

rock-paper-scissors/rps/input.go (lines 17-32)

```
17 func readHand() (Hand, error) {
18     reader := bufio.NewReader(os.Stdin)
19
20     fmt.Println("Rock (R), Paper (P), or Scissors (S)?")
21     str, err := reader.ReadString('\n')
22     if err != nil {
23         return NullHand, err
24     }
25     str = strings.TrimSuffix(str, "\n")
26     if s := strings.ToUpper(str); strings.HasPrefix(s, "Q") ||
   strings.HasPrefix(s, "X") {
27         fmt.Println("Thanks for playing the game!")
28         os.Exit(0)
29     }
30     hand, err := parseHand(str)
```

```
31     return hand, err
32 }
```

It essentially reads the text input and translates it into a `Hand`.

rock-paper-scissors/rps/hand.go (lines 7-11)

```
7 const (
8     Rock Hand = iota + 1
9     Paper
10     Scissors
11 )
```

Go does not support "enum" types. Instead, Go uses `const`s to represent a set of related constants, as in this example. It is customary to use a single "factored" `const` statement to defines a set of related constants.

`iota` is a special value that starts from 0 and increments by 1 within a const statement. For `Hand`, which is a type defined to be `uint8`, there are three constants defined: `Rock == 1`, `Paper == 2`, and `Scissors == 3`.

Note that we use values, 1, 2, and 3 in this example, not 0, 1, and 2. `0` can be a good "default value", e.g., to indicate an invalid `Hand`.

The implementation of `randomHand()` is simple:

rock-paper-scissors/rps/random.go (lines 12-14)

```
12 func randomHand() Hand {
13     return Hand(rand.Intn(3) + 1)
14 }
```

It just picks a random `Hand` based on a random number in the range `{1, 2, 3}`.

The expression `Hand()` represents a type conversion or casting.

The `compareHands()` function implements the rock paper scissors logic.

rock-paper-scissors/rps/hand.go (lines 13-33)

```
13 type WinOrLose uint8
14
15 const (
16     Tie WinOrLose = iota
17     Win
18     Lose
19 )
20
21 func compareHands(h1, h2 Hand) WinOrLose {
22     if h1 == h2 {
23         return Tie
24     } else {
25         if (h1 == Rock && h2 == Scissors) ||
26             (h1 == Paper && h2 == Rock) ||
27             (h1 == Scissors && h2 == Paper) {
28             return Win
29         } else {
30             return Lose
31         }
32     }
33 }
```

That is, Rock beats Scissors, Scissors bests Paper, and Paper beats Rock.

The type `Hand` implements the `Stringer` method, `String()`:

rock-paper-scissors/rps/hand.go (lines 35-46)

```
35 func (h Hand) String() string {
36     switch h {
37     case Rock:
38         return "Rock"
39     case Paper:
40         return "Paper"
41     case Scissors:
42         return "Scissors"
43     default:
44         return "?"
45     }
46 }
```

Note that we use the `switch` statement to map a `Hand` value to a `string`. We could have alternatively used a `map` to do the same, in this particular example. Or, in general, we could have implemented the same logic using the `if-else` statement.

The switch statement in Go is based on C's switch. But, there are differences.

- `switch`: `Switch` statements provide multi-way execution. An expression or type specifier is compared to the `cases` inside the `switch` to determine which branch to execute.

A `switch` statement comprises a condition expression and a number of `case`s. It runs the first case whose value is equal to the condition expression. Switch cases evaluate cases from top to bottom, stopping when a case succeeds. Go's switch does not need `break` after each case unlike in other C-style languages.

Instead, Go has `fallthrough` in cases where the execution needs to cross over the case boundaries.

- `fallthrough`: A `fallthrough` statement transfers control to the first statement

> of the next case clause in an expression `switch` statement. It may be used only as the final non-empty statement in such a clause.

Now that we have implemented all important building blocks, let's go back to the `Game.Start()` method.

```
func (g *Game) Start() {
    for {
        playerHand, _ := readHand()
        fmt.Printf("Your Hand = %s\n", playerHand)

        myHand := randomHand()
        fmt.Printf("My Hand = %s\n", myHand)

        wol := compareHands(playerHand, myHand)
    // Update the win-loss stats.
    }
}
```

Incidentally, we could have implemented the part where the stats (wins/losses) are computed (lines 38-44)

```
if wol == Win {
    g.wins++
} else if wol == Lose {
    g.losses++
} else {
    g.ties++
}
```

using `switch` as follows:

```
switch wol {
case Win:
    g.wins++
case Lose:
    g.losses++
default:
    g.ties++
}
```

Generally, switch statements are easier to read than long if-then-else chains. Even in this simple case, the `switch` version appears preferable.

One other thing to note is that, as stated before, increment/decrement is a statement, not an expression. Go has only the postfix increment and decrement operators (`x` or `x--`), and not the prefix increment and decrement operators. You cannot place the ` or `--` operators before the variable.

In each iteration of the for loop in the `Game.Start()` method, it does the three things we mentioned earlier, which correspond to `readHand()`, `randomHand()`, and `compareHands()`, respectively.

Then, the `main()` calls this `rps.NewGame().Start()` function. That is the entire program.

One thing to note regarding the `Game.Start()` method is that it uses a pointer receiver `(g *Game)`, not a value receiver `(g Game)`. What is the difference?

As we stated in the earlier lessons, for example, in Euclidean Distance, if you may have to possibly change the value of a variable, then you will have to use a pointer receiver. The argument value of a value receiver is copied, and hence any changes to the copied value (including its field values, etc.) will not affect the original receiver value.

A pointer receiver is also generally preferred when the copying of the receiver argument is expensive, for instance, because the receiver is a large struct.

We will revisit this question of value receiver vs pointer receiver in later lessons, but as a general rule of thumb, a pointer receiver is preferred, for a number of reasons including the above two.

20.3. Summary

We created a simple game using a random number generator from the `math/rand` package. We also quickly reviewed Go's `switch` statement.

Lesson 21. File Cat

21.1. Introduction

The *cat* command on the Unix-like platforms prints out the content of one or more files together. Hence the name *conCATenate.*

If the argument is not provided, then it reads from the standard input.

```
NAME
       cat - concatenate files and print on the standard output

SYNOPSIS
       cat [OPTION]... [FILE]...

DESCRIPTION
       Concatenate FILE(s) to standard output.

       With no FILE, or when FILE is -, read standard input.
```

Let's implement a simple version of *cat* in Go.

21.2. Code Review

21.2.1. Package *main*

Here's the "program":

file-cat/main.go (lines 11-28)

```
11 func main() {
12     if len(os.Args) == 1 {
13         if _, err := io.Copy(os.Stdout, os.Stdin); err != nil {
14             if err != io.EOF {
15                 log.Fatalln(err)
16             }
17         }
18         return
19     }
20
21     for _, fname := range os.Args[1:] {
22         err := file.Copy(os.Stdout, fname)
```

```
23          if err != nil {
24              fmt.Fprintln(os.Stderr, err)
25              continue
26          }
27      }
28  }
```

The `main()` function first checks if a command line argument is provided. If there is none, then it simply copies the input from `stdin` to `stdout`, using the `Copy()` function in the `io` package in the standard library.

If there is one or more arguments, then it copies the content of each file to `stdout`. The logic is implemented in the `file.Copy()` function.

21.2.2. Package *file*

The `file.Copy()` function is exported, and it has a type `func(*os.File, string) error`.

file-cat/file/copy.go (lines 8-28)

```
 8  // Copy finds a named file and copies its content
 9  // to the destination file.
10  // It returns error if opening the file or copying fails.
11  func Copy(dest *os.File, name string) error {
12      var file *os.File
13      if name == "-" {
14          file = os.Stdin
15      } else {
16          var err error
17          file, err = os.Open(name)
18          if err != nil {
19              return err
20          }
```

```
21          defer file.Close()
22      }
23
24      if _, err := io.Copy(dest, file); err != nil {
25          return err
26      }
27      return nil
28  }
```

This function takes two arguments, a destination *file,* `dest` of type `*os.File`, and a source *file name,* `name` of type `string`.

Note that the function is written slightly more generally than required by the `main()` function. The program only uses `os.Stdout` as destination, and yet `Copy()` can be used with an arbitrary destination (of type `*os.File`).

`Copy()` is slightly more useful (e.g., since it can be used in a wider range of situations), at the expense of being slightly more complicated (e.g., since the caller has to provide two arguments instead of one). This is a design choice.

Note also that the source and destination arguments are not "symmetric". They have different types, and they should be interpreted differently. In cases like this, a function name like `Copy` might be too generic. A more descriptive name might be useful.

The function includes what is called "doc comments". Any comment immediately preceding exported names (or, other unexported top-level constructs, etc.), without any blank lines, is considered part of the API documentation.

You, or anyone who uses this package, can generate documentation using the *go doc* command.

The `Open()` function first tries to open the file of the given name (or, file path), and if it fails, it returns an error. If it successfully opens the file, then it copies the

content of the file to the destination file, using `io.Copy()`.

One small twist is that, just like in "real cat", if the given file name is "-", then we read from `os.Stdin` instead of trying to open a file with that name.

21.3. Pair Programming

Much of the file-related APIs is included in the `os` and `io` packages in the standard library. Getting familiar with file handling APIs, and IO in general, is essential to be a proficient programmer in Go.

The example program deals with a few basic functions for file opening and file copying. The function's logic is straightforward.

In the implementation of `file.Copy()`, there are a couple of things to note.

First, `errors` in Go do not literally mean that they are "errors" as commonly used in English. This is true across all programming languages, regardless of what kind of error or exception handling mechanisms are used.

As stated before, an `error` is a signaling mechanism from a callee function to its caller.

If the callee function does not know how to handle certain situations (e.g., due to lack of broader context), or if it decides that its caller may be the better one to handle those situations, etc., then it can return an `error` so that the caller function can pay special attention, if necessary.

When dealing with files, encountering `EOF` ("end of file") while reading a file is not an "error" per se. All files have endings. Some of Go's IO functions (but not `io.Copy()`) happen to use the error return value to convey the information that they have read all content of the file, using the variable `io.EOF`.

It is worth repeating that the error handling in Go is simply a convention. Functions

return an error as a normal return value (as the last one, if there are multiple return values), and they use the interface type `error` for the error return value. It is just a convention, not hard-wired into the Go programming language.

It is based on C's convention, and it is a big improvement. But, there are cases where simply returning error values to the caller (and the caller returning the errors to its own caller, etc.) through the normal call chain has limitations.

Need for frequent error checking can also lead to code bloating.

Many other modern languages use "exceptions", which has a slightly different semantics. Exception mechanisms (e.g., using *try* and *catch*) have some limitations as well. The current "trend" in the programming language design seems to be using "option" style values, as commonly found in functional programming languages.

In Go, if something really disastrous happens that the program cannot recover from, then it is often best to just terminate the program (e.g., using `os.Exit(1)`).

Between these two extremes, there is another mechanism for handling exceptional or unexpected situations: `panic()` and `recover()`. This is similar, although rather limited, to the exception handling mechanism in other programming languages.

We will cover some essential features of `panic` and `recover` in later lessons.

One other thing to note in this `file.Copy()` function implementation is the defer statement.

We use the `defer` keyword in this example that we have not seen in the previous lessons.

- `defer`: A `defer` statement invokes a function whose execution is deferred to the moment the surrounding function returns.

A defer statement starts with the `defer` keyword, which is followed by a certain kind of expression, namely a function or method call. The expression cannot be put

in parentheses.

Deferred functions are invoked immediately before the surrounding function returns, in the reverse order they were deferred.

It is very common to use defer statements to clean up resources.

In this example, opening a file uses resources (e.g., the file handle, etc.). It is best to close the file before returning. A function can return in many different ways (either explicitly or implicitly).

The `defer` statement guarantees that the deferred function will be called regardless of how the function returns, even through panics. The exception is `os.Exit()`, which immediately terminates the program.

When we open the named file using `os.Open()`, we use the following two statements:

```
var err error
file, err = os.Open(name)
```

Rather than, say, using the more common short variable declaration:

```
file, err := os.Open(name)
```

This is because the compiler sometimes treats the left-hand side of := as all new variables, and sometimes it does not. (The short variable declaration syntax is valid as long as there is at least one new variable.)

If a variable, `file` in this case, happen to be in the *same* scope, it is considered as the same variable (because a variable cannot be declared twice in the same scope). And, `err` is a new variable in this case, the `:=` declaration is valid. Because of the "at

least one new variable" rule on the left-hand side of the short variable declaration, the compile does not complain.

On the other hand, if a variable, `file` in this case, happens to exist outside the block, that is, in the surrounding block, then the variable on the left hand side of `:=` is considered new because the "variable shadowing" is allowed. In this case, in the second one-liner example, the compiler treats both `file` and `err` as new variables. The `file` variable declared before the `if` statement is different from the `file` variable which assign using the `:=` statement.

A variable can be re-declared in a block even if a variable with the same name exists in an outer scope, e.g., in a block surrounding this block. Within the scope of this variable in the inner block, the name refers to the variable declared (or, redeclared) in this block, not the variable declared outside (which would have been "in scope" if not for the redelaration). This is called *variable shadowing*.

Clearly, that is not what we intend here. Hence, in cases like this, you cannot use the short variable declaration syntax.

This is one of the frequent gotchas for new Go programmers. This is one of the rare cases in Go where you build/compile a program successfully and yet you still have a trivial, but potentially critical, bug that could have been easily caught by a compiler.

One other thing worth noting is that the package `ioutil` in the standard library has been deprecated as of this writhing (Go version 1.16). Many of the functions which was in the `ioutil` have been moved other packages, including `io`.

Although the Go programming language rarely changes, libraries, even the (stable) standard libraries, change.

The `io.Copy()` function is a utility function. It performs certain tasks behind the scene, like reading the source file and writing to the destination file.

Let's try to rewrite our `file.Copy()` function without using `io.Copy()`.

file-cat/file/copy2.go (lines 8-29)

```
 8 // Copy2 finds a named file and copies its content to the
   destination file.
 9 // It returns the number of bytes copied and a possible error.
10 // It return non-nil error if opening the file or reading and
   writing the content fails.
11 func Copy2(dest *os.File, name string) (int, error) {
12     file, err := os.Open(name)
13     if err != nil {
14         return -1, err
15     }
16     defer file.Close()
17
18     data, err := io.ReadAll(file)
19     if err != nil {
20         return -1, err
21     }
22
23     written, err := dest.Write(data)
24     if err != nil {
25         return -1, err
26     }
27
28     return written, nil
29 }
```

The `file.Copy2()` function, not the best descriptive name, uses the `io.ReadAll()` and `File.Write()` functions instead of `io.Copy()`.

Since `io.ReadAll()` is supposed to read *the whole content* of the file into memory, it does not return `io.EOF` as an `error` value. And, because of this, `io.ReadAll()` may not, in general, be the most suitable function to use. See the exercise at the end of this lesson.

`File.Write()` returns an error if the size of the input content `len(data)` is different from the size of the written content, `written`. Hence an error checking like this is not required when using the `File.Write()` function:

```
if len(data) != written {
    return -1, errors.New(
        fmt.Sprintf("The numbers of bytes read (%d) and written (%d)
are different/n",
            len(data), written))
}
```

Note the way in which we split a single statement into multiple lines.

The implementation of `file.Copy2()` is straightforward. It opens a file with a given name, reads it, and then writes the content to the given output file. Most of the code is for error handling.

The `file.Copy2()` function is also preceded by a doc comment.

You can generate, or view, the documentation for the `file` package using the *go doc* command.

```
go doc --all file
```

Note that the *file* argument is the name of the package (or, the last segment of the path if it follows the package naming convention), not the path of the package folder, as is the case with most other go commands like *go build, go run,* and *go test.*

Here's a sample output:

```
package file // import "examples/file-cat/file"

FUNCTIONS

func Copy(dest *os.File, name string) error
    Copy finds a named file and copies its content to the destination
file. It
    returns error if opening the file or copying fails.

func Copy2(dest *os.File, name string) (int, error)
    Copy2 finds a named file and copies its content to the
destination file. It
    returns the number of bytes copied and a possible error. It
return non-nil
    error if opening the file or reading and writing the content
fails.
```

You can include un-exported names in the documentation using `-u` flag.

You can also use *go doc* for viewing documentations for other libraries. For the standard library, for instance, you just specify a package name, or qualified names other symbols.

```
go doc io
```

It prints the information on all exported names from the `io` package, constants, variables, types, and functions.

```
go doc io.Copy
```

It prints the information on the `io.Copy()` function.

The Go tools support what is called "testable examples". You can run *go test* on the example code, which can be included in the documentation. This is a fantastic feature that will help you maintain the docs as well as the code over time.

Although this book cannot cover all the details of Go, it'll be worthwhile for you to look this up on the Web if you tend to write a lot of (public) library packages. For example, refer to an article, blog.golang.org/examples, on the official Go blog.

21.4. Summary

We learned some basic file and IO-related functions, from the `os` and `io` packages.

A `defer` statement is used to clean up resources before returning from the function. The deferred functions execute just before the function returns.

We also covered some basics of the *go doc* command. Readers are encouraged to fully utilize the go doc features, both from the library user's and the provider's perspectives.

21.5. Exercises

1. The `io.ReadAll()` function may not be the best option when you read a file, especially if the file is large. Create a copy function that copies the content of one file to the other, one line at a time, without having to read all content into memory.

Lesson 22. World Time API

22.1. Introduction

Go is one of the most popular languages for Web backend development.

In fact, Web backend is one of most important uses of the Go programming language. Go is rarely used in GUI programming, for instance. Go's support is minimal for data science or machine learning. At least as of now. Not many programmers use Go for developing mobile games.

If you use Go, then Web backend, or other server side programming, is the sweet spot. We will briefly touch on a few examples that are in the area of Web development in this book.

In this lesson, we look at a simple example that demonstrates the use of some simple functions of the `net/http` package in the standard library.

22.2. Code Review

22.2.1. Package *main*

The `main()` function calls the "World Time API" service to retrieve the current time every 10 seconds.

world-time-api/main.go (lines 9-24)

```
 9 const maxErrorCount = 5
10 const interval = 10 * time.Second
11 const url =
   "http://worldtimeapi.org/api/timezone/America/New_York.txt"
12
```

```
13 func main() {
14     for errorCount := 0; errorCount < maxErrorCount; {
15         datetime, err := world.Datetime(url)
16         if err != nil {
17             errorCount++
18             fmt.Println("Error:", err)
19             continue
20         }
21         fmt.Printf("datetime: %s\n", datetime)
22         time.Sleep(interval)
23     }
24 }
```

The particular endpoint we use for this example is `"http://worldtimeapi.org/api/timezone/America/New_York.txt"`.

It serves the time in the New York time zone, and the output is plain text (e.g., as opposed to JSON format), as indicated by the ".txt" suffix in the URL.

22.2.2. Package *world*

The `world.Datetime()` function fetches the content on the Web, reads the content of the response body, and parses the content to find the desired data, *datetime* in our example.

world-time-api/world/datetime.go (lines 11-33)

```
11 func Datetime(url string) (string, error) {
12     response, err := http.Get(url)
13     if err != nil {
14         return "", err
15     }
16 
17     responseData, err := io.ReadAll(response.Body)
```

```
18      if err != nil {
19          return "", err
20      }
21
22      scanner := bufio.NewScanner(strings.NewReader(
    string(responseData)))
23      for scanner.Scan() {
24          text := scanner.Text()
25          s := strings.SplitN(text, ":", 2)
26
27          if s[0] == "datetime" {
28              return s[1], nil
29          }
30      }
31
32      return "", errors.New("Datetime not found!")
33  }
```

The `Datetime()` function uses `http.Get()`, one of the simplest functions in the `net/http` package.

It returns the datetime string to the caller, the `main()` function, which simply prints out the result and waits for 10 seconds before it calls `Datetime()` again.

22.3. Pair Programming

Given a URL, the first thing to do is to fetch the content from the URL. One of the easiest way to do is using `http.Get()` function.

Since we covered how to read Go docs in the previous lesson, let's try that:

```
go doc http
```

This lists all exported names from the `http` package. Now, let's try the `Get()` function:

```
go doc http.Get
```

Here's the output:

```
package http // import "net/http"

func Get(url string) (resp *Response, err error)
    Get issues a GET to the specified URL. If the response is one of
the
    following redirect codes, Get follows the redirect, up to a
maximum of 10
    redirects:

        301 (Moved Permanently)
        302 (Found)
        303 (See Other)
        307 (Temporary Redirect)
        308 (Permanent Redirect)

    An error is returned if there were too many redirects or if there
was an
    HTTP protocol error. A non-2xx response doesn't cause an error.
Any returned
    error will be of type *url.Error. The url.Error value's Timeout
method will
    report true if request timed out or was canceled.

    When err is nil, resp always contains a non-nil resp.Body. Caller
should
    close resp.Body when done reading from it.
```

```
    Get is a wrapper around DefaultClient.Get.

    To make a request with custom headers, use NewRequest and
DefaultClient.Do.
```

That is useful. In particular, we can confirm that this is indeed a function which we can use: `Get()` issues a HTTP GET command to the target URL.

`http.Get()` returns the response of type `*http.Response` (along with a potential error value). You can read the body content using `io.ReadAll(response.Body)` which we used in the previous lesson, File Cat.

In case you are not familiar, an HTTP response comprises the header part and the rest, "body", separated by a blank line.

The `io.ReadAll()` function returns the content in a byte slice type.

It is generally a good idea to spend some time to understand the API and figure out how the API service works. Reading the documentation can be useful. Trying out some endpoints using command line tools such as `curl` can be useful.

For the World Time API service, everything you need to know to use the API is available on their home page: worldtimeapi.org.

Let's try to get the London time using one of their examples.

```
curl -X GET -i
"http://worldtimeapi.org/api/timezone/Europe/London.txt"
```

You can use tools other than *curl* as well. The point of this exercise is to understand the response format from the service. Since we already implemented `http.Get()` at this point, we can actually

> use the (work-in-progress) program we are writing for this purpose. We can just print out the received content (both the headers and the body) to the console output.

A typical response from the server will look like this:

```
HTTP/1.1 200 OK
Connection: keep-alive
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers:
Cache-Control: max-age=0, private, must-revalidate
Content-Length: 359
Content-Type: text/plain; charset=utf-8
Cross-Origin-Window-Policy: deny
Date: Fri, 23 Apr 2021 20:41:02 GMT
Server: Cowboy
X-Content-Type-Options: nosniff
X-Download-Options: noopen
X-Frame-Options: SAMEORIGIN
X-Permitted-Cross-Domain-Policies: none
X-Request-Id: 06b5bce3-27fb-4abb-9407-cea8865d6345
X-Runtime: 28ms
X-Xss-Protection: 1; mode=block
Via: 1.1 vegur

abbreviation: BST
client_ip: 187.144.22.133
datetime: 2021-04-23T21:41:03.119143+01:00
day_of_week: 5
day_of_year: 113
dst: true
dst_from: 2021-03-28T01:00:00+00:00
dst_offset: 3600
```

```
dst_until: 2021-10-31T01:00:00+00:00
raw_offset: 0
timezone: Europe/London
unixtime: 1619210463
utc_datetime: 2021-04-23T20:41:03.119143+00:00
utc_offset: +01:00
```

Anything above the blank line is headers. Anything that follows the blank line is the body.

Now, where is the London time?

As you can easily figure out, the desired data is in a line that starts with "datetime: ". What we need to get is the value after this prefix, "2021-04-23T21:41:03.119143+01:00" in this particular response.

So, one way to get the datetime is to go through all lines from the response body, line by line, and find a line that starts with `"datetime: "`. It is easy to write such a function using library functions like `strings.HasPrefix()`.

In this example, we are using a slightly more specialized API, namely, `bufio.Scanner`, for demonstration.

Let's browse the doc:

```
go doc bufio.Scanner
```

Here's an output:

```
package bufio // import "bufio"

type Scanner struct {
```

```
        // Has unexported fields.
}
    Scanner provides a convenient interface for reading data such as
a file of
    newline-delimited lines of text. Successive calls to the Scan
method will
    step through the 'tokens' of a file, skipping the bytes between
the tokens.
    The specification of a token is defined by a split function of
type
    SplitFunc; the default split function breaks the input into lines
with line
    termination stripped. Split functions are defined in this package
for
    scanning a file into lines, bytes, UTF-8-encoded runes, and
space-delimited
    words. The client may instead provide a custom split function.

    Scanning stops unrecoverably at EOF, the first I/O error, or a
token too
    large to fit in the buffer. When a scan stops, the reader may
have advanced
    arbitrarily far past the last token. Programs that need more
control over
    error handling or large tokens, or must run sequential scans on a
reader,
    should use bufio.Reader instead.

func NewScanner(r io.Reader) *Scanner
func (s *Scanner) Buffer(buf []byte, max int)
func (s *Scanner) Bytes() []byte
func (s *Scanner) Err() error
func (s *Scanner) Scan() bool
func (s *Scanner) Split(split SplitFunc)
```

```
func (s *Scanner) Text() string
```

The Go documentation is also available online. For the standard library, the best place is golang.org/pkg.

As you can see from the doc, a new `Scanner` can be created using `NewScanner()` function. (The naming convention, as well as its function signature, clearly indicates that this is a function we can use to *create* a new `Scanner`.)

```
$ go doc bufio.NewScanner

package bufio // import "bufio"

func NewScanner(r io.Reader) *Scanner
    NewScanner returns a new Scanner to read from r. The split
function defaults
    to ScanLines.
```

The `NewScanner()` function takes an argument of type `io.Reader`.

We have `responseDate` which is a type of `[]byte`. A byte slice can be cast to a `string`. A new `io.Reader` can be created from a `string` using `strings.NewReader()` function.

```
$ go doc strings.NewReader

package strings // import "strings"

func NewReader(s string) *Reader
    NewReader returns a new Reader reading from s. It is similar to
    bytes.NewBufferString but more efficient and read-only.
```

`strings.Reader` is an interface type of `io.Reader`, among other things.

Hence, we get `scanner` of type `*bufio.Scanner` this way:

```
    scanner := bufio.NewScanner(strings.NewReader(
string(responseData)))
```

`bufio.NewScanner()` creates a `Scanner` which by default "tokenizes" the content using newlines.

Now, we can use the `Scan()` method of `bufio.Scanner` to read `responseData`, line by line.

```
$ go doc bufio.Scanner.Scan

package bufio // import "bufio"

func (s *Scanner) Scan() bool
    Scan advances the Scanner to the next token, which will then be
available
    through the Bytes or Text method. It returns false when the scan
stops,
    either by reaching the end of the input or an error. After Scan
returns
    false, the Err method will return any error that occurred during
scanning,
    except that if it was io.EOF, Err will return nil. Scan panics if
the split
    function returns too many empty tokens without advancing the
input. This is
    a common error mode for scanners.
```

We use the `scanner.Text()` method to get the next "token":

```
$ go doc bufio.Scanner.Text

package bufio // import "bufio"

func (s *Scanner) Text() string
    Text returns the most recent token generated by a call to Scan as
a newly
    allocated string holding its bytes.
```

The next thing we will need to do is to split each line into a "key-value" pair, This is based on the knowledge we gained by browsing the sample response body.

```
$ go doc strings.SplitN

package strings // import "strings"

func SplitN(s, sep string, n int) []string
    SplitN slices s into substrings separated by sep and returns a
slice of the
    substrings between those separators.

    The count determines the number of substrings to return:

        n > 0: at most n substrings; the last substring will be the
unsplit remainder.
        n == 0: the result is nil (zero substrings)
        n < 0: all substrings

    Edge cases for s and sep (for example, empty strings) are handled
as
    described in the documentation for Split.
```

Based on this doc, we know we can use something like this to get the key and the rest (i.e., the value). Note the value `2` for `n` (the third argument):

```
s := strings.SplitN(text, ":", 2)
```

The `for scanner.Scan()` loop continues until we find the desired key "datetime" or it runs out of tokens.

In the latter case, we return a non-nil error to indicate that we have not found what we are looking for. You can use `errors.New()` to create a value of an interface type `error`:

```
$ go doc errors.New

package errors // import "errors"

func New(text string) error
    New returns an error that formats as the given text. Each call to
New
    returns a distinct error value even if the text is identical.
```

That is our `Datetime` function implementation.

In the `main` function, we keep calling `world.Datetime()` on a regular interval. We "sleep" for 10 seconds.

```
$ go doc time.Sleep

package time // import "time"

func Sleep(d Duration)
    Sleep pauses the current goroutine for at least the duration d. A
```

```
negative
    or zero duration causes Sleep to return immediately.
```

`time.Second` is a constant (of type `time.Duration`) defined in the `time` package:

```
$ go doc time.Second

package time // import "time"

const (
        Nanosecond  Duration = 1
        Microsecond          = 1000 * Nanosecond
        Millisecond          = 1000 * Microsecond
        Second               = 1000 * Millisecond
        Minute               = 60 * Second
        Hour                 = 60 * Minute
)
    Common durations. There is no definition for units of Day or
larger to avoid
    confusion across daylight savings time zone transitions.

    To count the number of units in a Duration, divide:

        second := time.Second
        fmt.Print(int64(second/time.Millisecond)) // prints 1000

    To convert an integer number of units to a Duration, multiply:

        seconds := 10
        fmt.Print(time.Duration(seconds)*time.Second) // prints 10s
```

If we run into a certain number of errors, then we terminate the program.

Let's run the program. Here's a sample output:

```
$ go run .

Error: Datetime not found!
datetime:  2021-04-23T17:32:42.207099-04:00
datetime:  2021-04-23T17:32:52.418983-04:00
Error: Datetime not found!
datetime:  2021-04-23T17:33:33.179857-04:00
datetime:  2021-04-23T17:33:43.432816-04:00
datetime:  2021-04-23T17:33:53.671061-04:00
datetime:  2021-04-23T17:34:03.901718-04:00
signal: interrupt
```

One of the things that we see by browsing the documentation for the `http.Get()` function is that "Get is a wrapper around DefaultClient.Get".

`http.Get` uses the variable `DefaultClient`. In the next lesson, we will look into the http client in some more detail.

22.4. Summary

We used the `http.Get()` function to fetch Web content. We parsed the response body to get the current time in a particular timezone.

22.5. Exercises

1. Modify the `url` in the `main` package to get the time in your own timezone.
2. Use the number consecutive errors not the total error count as a criterion to terminate the program.
3. Parse the datetime string and print it out in a particular format, e.g., "April 21st, 2:15PM".

4. Use the "big digit LED clock", "LED" Clock, to display the datetime from the server.

Lesson 23. Where the ISS at

23.1. Introduction

The website, Where the ISS at? [https://wheretheiss.at], provides the real-time location data of the International Space Station (ISS) in REST API.

Here's the REST API documentation: wheretheiss.at/w/developer. There are five API endpoints. They do not require authentication, as of this writing,

One thing to note is that they have rate limits, around 1 request per second.

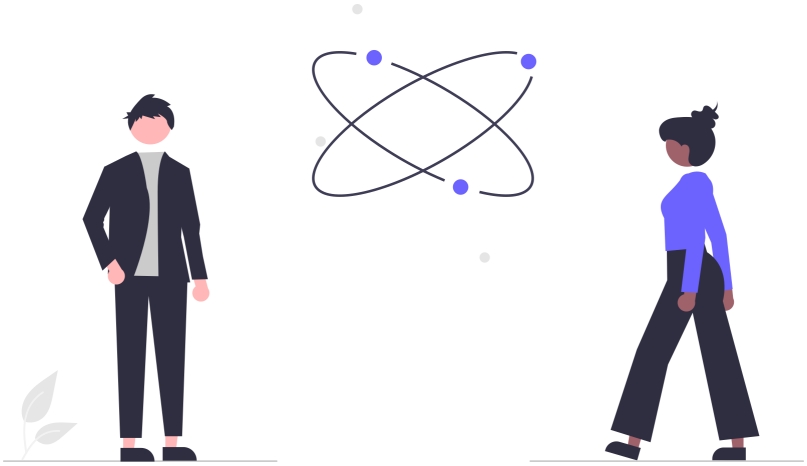
The *satellites* endpoint returns a list of satellites and their IDs.

```
$ curl https://api.wheretheiss.at/v1/satellites

[{"name":"iss","id":25544}]
```

There is currently only one satellite supported, and that is the ISS. Its ID is `25544`.

You can get the current position of the ISS using the *satellites/[ID]* endpoint. Let's write a program which displays the current location and altitude of the ISS on a regular interval.



23.2. Code Review

First, let's try out this endpoint:

```
curl -X GET -i https://api.whereTheISS.at/v1/satellites/25544
```

Here's a sample response:

```
HTTP/1.1 200 OK
Date: Sat, 24 Apr 2021 03:28:21 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.26
X-Rate-Limit-Limit: 350
X-Rate-Limit-Remaining: 349
X-Rate-Limit-Interval: 5 minutes
Access-Control-Allow-Origin: *
X-Apache-Time: D=23228
Cache-Control: max-age=0, no-cache
Content-Length: 312
```

```
Content-Type: application/json

{"name":"iss","id":25544,"latitude":-21.409188508186,"longitude":-
32.420309123332,"altitude":424.89524314303,"velocity":27565.45635825,
"visibility":"eclipsed","footprint":4532.2848807191,"timestamp":16192
34901,"daynum":2459328.6446875,"solar_lat":12.915312498887,"solar_lon
":127.45847703932,"units":"kilometers"}
```

Again, note the empty line separating the headers and the body. By default, it sends the data in the JSON format in the response body. We can "pretty print" the body of the response:

```
{
  "name": "iss",
  "id": 25544,
  "latitude": -21.409188508186,
  "longitude": -32.420309123332,
  "altitude": 424.89524314303,
  "velocity": 27565.45635825,
  "visibility": "eclipsed",
  "footprint": 4532.2848807191,
  "timestamp": 1619234901,
  "daynum": 2459328.6446875,
  "solar_lat": 12.915312498887,
  "solar_lon": 127.45847703932,
  "units": "kilometers"
}
```

We are only interested in the four fields, `latitude`, `longitude`, `altitude`, and `timestamp` (in Unix epoch seconds), in this example.

This program's logic/structure will be pretty much the same as that of the previous lesson, World Time API.

- Call the API endpoint.
- Fetch the data, and parse the response.
- Print out the result.

The primary difference will be, in this example, we will deal with JSON responses.

23.2.1. Package *main*

Here's a portion of the source file, *main.go*, which includes the `main()` function:

whereistheiss-api/main.go (lines 9-15)

```
9  const issID = 25544
10 const endpoint = "https://api.whereistheiss.at/v1/satellites"
11 const interval = 10 * time.Second
12
13 func main() {
14     trackISS()
15 }
```

A few constants are defined here, and then the `main()` function merely calls another function, `trackISS()`, in the same package. We are going to work on a few different versions of the program, and we will use a different "track ISS" function for each version.

The `trackISS()` function is defined as follows:

whereistheiss-api/main.go (lines 17-28)

```
17 func trackISS() {
18     for {
19         sat, err := iss.Track(endpoint, issID)
20         if err != nil {
```

```
21              fmt.Println("Error while tracking ISS:", err)
22              continue
23          }
24          fmt.Println(*sat)
25
26          time.Sleep(interval)
27      }
28  }
```

We have an infinite loop, as before, and in each iteration, we call the `iss.Track()` function defined in the `iss` package. Then, we print out the result, and repeat the same after waiting about 10 seconds.

The `iss.Track()` function does all the heavy lifting.

23.2.2. Package *client*

First, let's create an `http.Client`. In the previous lesson, we used `http.Get()` to fetch the data, which indirectly calls `http.DefaultClient.Get()`.

This time, we will explicitly create a variable of type `http.Client` (or, its pointer type), and use its methods.

whereistheiss-api/client/client.go (lines 8-15)

```
8  const timeout = 5 * time.Second
9
10 func New() *http.Client {
11     client := http.Client{
12         Timeout: timeout,
13     }
14     return &client
15 }
```

The type, `http.Client`, includes a few exported (or, public) fields, and their values can be customized. In this example, we set the request timeout duration to 5 seconds.

The conventional `New()` function returns a pointer to an `http.Client` value.

23.2.3. Package *iss*

Based on the sample JSON response data, and according to our requirements, we create a struct type `SatelliteData`.

This will be used to store the ISS position data.

wheretheiss-api/iss/iss.go (lines 9-14)

```
9 type SatelliteData struct {
10     Timestamp int64   `json:"timestamp"`
11     Latitude  float64 `json:"latitude"`
12     Longitude float64 `json:"longitude"`
13     Altitude  float64 `json:"altitude"`
14 }
```

Each field of a struct definition can include an optional "tag", after the field type, a string which has no real significance to the compiler. Other applications, or packages, can use the tags as needed.

Go supports two styles of string literals, * the double-quote style, as we primarily use in the examples in this book, and * the back-quote style, known as "raw string literals".

The raw string literals using back quotes can directly include various characters which otherwise would have needed escaping. Raw string literals can even include newline characters without

> escaping, thereby making them effectively "multi-line strings".
>
> Note that it would have been necessary to escape the double quotes in the field tags (e.g., `"json:\"timestamp\""`) if we had used the normal double quote strings.

In this example, the tags are to be used for marshaling and unmarshaling JSON data.

wheretheiss-api/iss/iss.go (lines 16-23)

```
16 func Unmarshal(data []byte) (*SatelliteData, error) {
17     sat := SatelliteData{}
18     err := json.Unmarshal(data, &sat)
19     if err != nil {
20         return nil, err
21     }
22     return &sat, nil
23 }
```

The `iss.Unmarshal()` function converts the response byte slice (JSON string) into a value of type `SatelliteData`.

The magic happens in `json.Unmarshal()`. The `json` package in the Go standard library includes functions for marshaling/encoding and unmarshaling/decoding JSON strings.

```
$ go doc json.Unmarshal

package json // import "encoding/json"

func Unmarshal(data []byte, v interface{}) error
    Unmarshal parses the JSON-encoded data and stores the result in
```

```
the value
    pointed to by v. If v is nil or not a pointer, Unmarshal returns
an
    InvalidUnmarshalError.
    ...
```

It should be noted that you can pick and choose a set of fields that you want to serialize or deserialize. You do not have to deserialize every field that is included in the response, for instance.

Another thing to note is that, in order to be able to use the marshall/unmarshall functions in the `json` package, the fields of a struct should be exported. That is, their names should be capitalized. For example, in the `SatelliteData` type, all four fields `Timestamp`, `Latitude`, `Longitude`, and `Altitude` are exported although no other packages use them.

In order to use the response JSON data, which follows the typical JSON field name convention (e.g., *camelCase*), we use the tags with the corresponding target names in the JSON response. These tags are solely for the standard `json` package.

For convenience, we also implement the `Stringer` interface method, `String()`, on the type `SatelliteData`. Functions like `fmt.Printf()` uses the Stringer interface.

whereetheiss-api/iss/iss.go (lines 25-28)

```
25 func (s SatelliteData) String() string {
26     t := time.Unix(s.Timestamp, 0)
27     return fmt.Sprintf("%s: (Lat:%.4f, Lon:%.4f, Alt:%.4f)",
   t.Format(time.RFC1123), s.Latitude, s.Longitude, s.Altitude)
28 }
```

The implementation of this method uses a couple of exported functions from the `time` package.

```
$ go doc time.Unix

package time // import "time"

func Unix(sec int64, nsec int64) Time
    Unix returns the local Time corresponding to the given Unix time,
sec
    seconds and nsec nanoseconds since January 1, 1970 UTC. It is
valid to pass
    nsec outside the range [0, 999999999]. Not all sec values have a
    corresponding time value. One such value is 1<<63-1 (the largest
int64
    value).
```

`time.Unix()` is used to create a value of type `time.Time` from the timestamp, and `Time.Format()` is used to format the time for display.

```
$ go doc time.Format

package time // import "time"

func (t Time) Format(layout string) string
    Format returns a textual representation of the time value
formatted
    according to layout, which defines the format by showing how the
reference
    time, defined to be "Mon Jan 2 15:04:05 -0700 MST 2006",
    would be displayed if it were the value.
        ...
```

Here's the `iss.Track()` function:

wheretheiss-api/iss/track.go (lines 10-41)

```
10 func Track(endpoint string, issID int) (*SatelliteData, error) {
11     url := fmt.Sprintf("%s/%d", endpoint, issID)
12 
13     req, err := http.NewRequest(
14         http.MethodGet,
15         url,
16         nil,
17     )
18     if err != nil {
19         return nil, err
20     }
21 
22     req.Header.Add("Accept", "application/json")
23 
24     httpClient := client.New()
25     res, err := httpClient.Do(req)
26     if err != nil {
27         return nil, err
28     }
29 
30     data, err := io.ReadAll(res.Body)
31     if err != nil {
32         return nil, err
33     }
34 
35     sat, err := Unmarshal(data)
36     if err != nil {
37         return nil, err
38     }
39 
40     return sat, err
41 }
```

The `Track()` function calls the *ISS Satellite* endpoint, it reads the response body as a JSON string, and deserializes it into a variable of type `SatelliteData`.

23.3. Pair Programming

As indicated, the `Track()` function uses a value of `http.Client` directly rather than using higher level functions such as `http.Get()`, which uses `http.DefaultClient` in its implementation. This gives us some more control.

```
var httpClient = &http.Client{
  Timeout: 5 * time.Second,
}
```

Furthermore, we use `Client.Do()` method instead of simpler `Client.Get()`. If we need to customize a request, e.g., set a header, or add a cookie, etc, then we will need to create a value of `http.Request` and use the `Client.Do(request)` method.

In this example, we set a header `"Accept: application/json"` to the request. This is not entirely required in this case since the *Where the ISS at* API happens to return JSON response by default. Code samples included in this book are primarily for illustration.

`Track()` then uses `io.ReadAll()` to read the body of the response (a JSON string). As stated before, `io.ReadAll()` might not be the best choice in some situations, but in this case, it is perfectly all right. We cannot parse JSON until we see the whole string.

The `iss.Unmarshal()` function is a simple wrapper around `json.Unmarshal()`, which is generally not necessary. In this example, it saves us one line. ☺

In some cases, you may want to have more control over deserialization over the

default behavior provided by the implementation of `json.Unmarshal()`.

The `iss.Track()` function includes some boilerplate code for error handling:

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

This 3-liner `if` statement is included four times in this small function.

The purpose of the if statement in this example is just to "relay" any non-nil errors returned by the functions that `Track()` calls to its own caller, and nothing else.

Unfortunately, there is no easy way to reduce this clutter, that is, as long as we use this error-as-a-return-value convention in Go. This is a simple example, but one can easily imagine a long call chain where a function calls a function, which calls another function, and so forth.

As alluded before, Go provides another way of handling exceptional, or unexpected, situations.

Panics automatically "bubble up", or propagate upstream in the call chain. When a panic happens in a function, the execution stops at this point, and it immediately returns, after calling only the deferred functions. The same thing happens to its caller, and its caller, ..., until it reaches the `main()` function, at which point the program terminates, with a non-zero exit code.

A good example of the use of `panic` is when dividing a number by a variable whose value is 0. (As a side note, if you attempt to divide a number by a constant `0` or `0.0`, or numeric literals equivalent to 0, the Go compiler catches it at a build time. It's a compile error, not a run-time error.)

- `func panic(v interface{})`: The panic built-in function stops normal

> execution of the current goroutine. When a function F calls panic, normal execution of F stops immediately. Any functions whose execution was deferred by F are run in the usual way, and then F returns to its caller.

We can call `panic()` with an arbitrary value, of any type.

During the unwinding of the call stack, or during the "panicking", as is called in Go, any function in the call chain may decide to handle the panic from downstream, using Go's builtin `recover()` function.

- `func recover() interface{}`: The recover built-in function allows a program to manage behavior of a panicking goroutine. Executing a call to recover inside a deferred function stops the panicking sequence by restoring normal execution.

The `recover()` function is normally, and almost always, used in a deferred function. `recover()` returns the same value as the parameter used when `panic()` was called. If `recover()` is called when the goroutine is not panicking, then it merely returns `nil`.

As an example, let's try to rewrite the `Track()` function so that we do not have to include the 3-liner error handling `if` statement everywhere. Simply putting the `if` statement in a separate function, and calling that function, will not work since that function call has to go through the normal call chain as well.

Here, in the second version of the `Track` function, we use `panic()`:

whereistheiss-api/iss/track.go (lines 43-65)

```
43 func Track2(endpoint string, issID int) (sat *SatelliteData, err
   error) {
44     defer func() {
45         if r := recover(); r != nil {
46             err = r.(error)
```

```
47          }
48      }()
49
50      url := fmt.Sprintf("%s/%d", endpoint, issID)
51      req, err := http.NewRequest(http.MethodGet, url, nil)
52      panicOnError(err)
53
54      req.Header.Add("Accept", "application/json")
55      res, err := client.New().Do(req)
56      panicOnError(err)
57
58      data, err := io.ReadAll(res.Body)
59      panicOnError(err)
60
61      sat, err = Unmarshal(data)
62      panicOnError(err)
63
64      return
65  }
```

The `panicOnError()` function is defined as follows:

whereetheiss-api/iss/track.go (lines 85-89)

```
85  func panicOnError(err error) {
86      if err != nil {
87          panic(err)
88      }
89  }
```

Every time we run into an error, we call `panic(err)`. In a deferred function, we then call `recover()` to check if there has been an error. And, if so, `recover() != nil`, then we "convert the return value `r` back to an error type" and return that

`error` to the caller.

The expression `r.(error)` casts `r`, which is of type `interface{}`, to `error` type. This is known as a "type assertion".

At the risk of sounding like a broken record, all example programs in this book are primarily for illustration purposes, and this example, in particular, only demonstrates the use of `panic()` and `recover()`. Use of panics is often reserved for situations where the normal convention using the error return values is not suitable.

Incidentally, the defer statement uses an anonymous function literal, and it calls the function immediately at the point of its definition.

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

This is equivalent to

```
f := func() { /* ... */ }
f()
```

This variation of `Track()`, or `Track2()`, does not require any real changes to its caller(s). Refer to `trackISS2()` in the code listing at the end of the book, [appendix-code-listing-part2].

We can even go a little bit further with this example.

This example of using panic/recover in the same function, `Track2()`, does not add much value, beyond the body of this function. Its callers will still have to deal with errors.

We can just use `panics` throughout our program. Every time we encounter an

error (from the functions we use), we convert it to `panic` and deal with it later. In situations where a long call chain is involved, it can potentially reduce the code clutter significantly.

The example still uses one caller and one callee, but it can be used in multiple function call sequences.

Here's a new version of `Track()`:

wheretheiss-api/iss/track.go (lines 67-83)

```
67 func Track3(endpoint string, issID int) *SatelliteData {
68     url := fmt.Sprintf("%s/%d", endpoint, issID)
69     req, err := http.NewRequest(http.MethodGet, url, nil)
70     panicOnError(err)
71
72     req.Header.Add("Accept", "application/json")
73     res, err := client.New().Do(req)
74     panicOnError(err)
75
76     data, err := io.ReadAll(res.Body)
77     panicOnError(err)
78
79     sat, err := Unmarshal(data)
80     panicOnError(err)
81
82     return sat
83 }
```

We use the same `panicOnError()` function as before. It simply calls `panic()` when it receives a non-nil error.

The difference between `Track3()` and `Track2()` is that, in this new version, we do not call `recover()`. It lets its caller, or the caller's caller, etc., handle the panicking

situation as they see fit.

Now, one of the functions in the call chain, `trackISS3()` in this case, which is really the `main()` function of the program (this additional layer of function call is somewhat artificial in this example), may decide to do something with `panics`.

For example,

whereistheiss-api/main.go (lines 43-52)

```
43 func trackISS3() {
44     defer restartOnPanic()
45
46     for {
47         sat := iss.Track3(endpoint, issID)
48         fmt.Println(*sat)
49
50         time.Sleep(interval)
51     }
52 }
```

whereistheiss-api/main.go (lines 54-59)

```
54 func restartOnPanic() {
55     if r := recover(); r != nil {
56         fmt.Println("Error while tracking ISS:", r.(error))
57         trackISS3()
58     }
59 }
```

The `restartOnPanic()` simply restarts the `for` loop, essentially by calling `restartOnPanic()` again, in the error situation.

The three versions of `trackISS()` behave the same way.

Here's a sample output:

```
go run .
Sat, 24 Apr 2021 09:35:12 CST: (Lat:-51.7867, Lon:69.1035,
Alt:435.8655)
Sat, 24 Apr 2021 09:35:22 CST: (Lat:-51.7975, Lon:70.0997,
Alt:435.8687)
Sat, 24 Apr 2021 09:35:33 CST: (Lat:-51.7987, Lon:71.1959,
Alt:435.8684)
Error while tracking ISS: Get
"https://api.wheretheiss.at/v1/satellites/25544": context deadline
exceeded (Client.Timeout exceeded while awaiting headers)
Sat, 24 Apr 2021 09:35:54 CST: (Lat:-51.7704, Lon:73.2874,
Alt:435.8565)
Sat, 24 Apr 2021 09:36:04 CST: (Lat:-51.7428, Lon:74.2820,
Alt:435.8457)
signal: interrupt
```

Note that we can change the request timeout duration by changing the `http.Client`'s `Timeout` value. For example,

```
client := http.Client{
  Timeout: 500 * time.Millisecond,
}
```

23.4. Summary

We learned how to make simple HTTP GET requests in a Go program using a value of type `http.Request`.

We also reviewed the error handling mechanism, using `panic()` and `recover()`.

23.5. Exercises

1. Although we do not cover GUI programming, including the Web frontend, if you are familiar with Web mapping APIs, such as Google Maps, then you can plot the (projections of) the ISS's positions on the map of the planet, real time.

Lesson 24. Simple Web Server

24.1. Introduction

Let's write a simple HTTP server.

There are many "Web framework" libraries in Go. Many Go programmers use these Web frameworks for various reasons. For example, some frameworks may have better performance. Some frameworks may expose APIs which are "easier" to use (although "easy" is really a subjective characterization in this context). Etc.

But, Go's standard library has a perfectly good support for Web backend programming.

We will try out a few simple APIs from the `net/http` package in this lesson.

24.2. Code Review

As always, let's start from the `main()` function.

When you are trying to read and understand a program source code, which is new to you, the `main` package is the best place to start, especially the `main()` function.

Generally, but not always, the top-down approach works best when designing a new program, or when trying to get a high-level understanding of an existing program.

24.2.1. Package *main*

The main function in this example is very simple:

web-server-simple/main.go (lines 9-17)

```
9 func main() {
10     fmt.Println("Server starting...")
11
12     http.HandleFunc("/", handler.Handler1)
13
14     if err := http.ListenAndServe(":8080", nil); err != nil {
15         panic(err)
16     }
17 }
```

It sets up a handler function,

```
http.HandleFunc("/", handler.Handler1)
```

And, it calls `http.ListenAndServe()`

```
if err := http.ListenAndServe(":8080", nil); err != nil {
    panic(err)
}
```

This if statement is pretty idiomatic when you start a long-running server program.

The `http.HandleFunc()` maps a URL path, or route, the "root" `/` in this case, to a handler function of type `func(http.ResponseWriter, *http.Request)`. All requests will be handled by the `Handler1()` function in the `handler` package, in this example.

In general, the Web server program may include multiple handlers for different paths.

```
$ go doc http.HandleFunc

package http // import "net/http"

func HandleFunc(pattern string, handler func(ResponseWriter,
*Request))
    HandleFunc registers the handler function for the given pattern
in the
    DefaultServeMux. The documentation for ServeMux explains how
patterns are
    matched.
```

The doc says that `http.HandleFunc()` uses `http.DefaultServeMux`, which is of a type `http.ServeMux`. You can *go doc* each of these names, and then the names that are included in those docs, and so forth, if you need to.

But, in this case, all we need is essentially the type of the `handler` argument. We will write, or review, a handler function shortly.

Likewise, we can look up the `http.ListenAndServe()` function:

```
$ go doc http.ListenAndServe

package http // import "net/http"

func ListenAndServe(addr string, handler Handler) error
    ListenAndServe listens on the TCP network address addr and then
calls Serve
    with handler to handle requests on incoming connections. Accepted
    connections are configured to enable TCP keep-alives.

    The handler is typically nil, in which case the DefaultServeMux
is used.

    ListenAndServe always returns a non-nil error.
```

Again, we are using the default `http.DefaultServeMux` variable defined in the `net/http` package when calling the `http.ListenAndServe()` function with `nil Handler` second argument, as in this example.

We can easily create a custom handler type (e.g., using `struct`) by implementing a method, `ServeHTTP(http.ResponseWriter, *http.Request)`, as defined in the `http.Handler` interface type:

```
$ go doc http.Handler

package http // import "net/http"

type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
    A Handler responds to an HTTP request.
```

```
    ...
```

The `http.ListenAndServe()` function does not return, except for the case of an error. When this function returns, the `error` value will always be non-`nil`.

Therefore, the following implementation is also idiomatic:

```
func main() {
    http.HandleFunc("/", handler.Handler1)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

When `http.ListenAndServe()` returns, if ever, it logs the error (which is always non-nil) and terminates the program with a non-zero exit code.

24.2.2. Package *handler*

Here's the simplest handler function of type `func(ResponseWriter, *Request)`:

web-server-simple/handler/handlers.go (lines 8-10)

```
8 func Handler1(w http.ResponseWriter, r *http.Request) {
9     io.WriteString(w, "Hello\n")
10 }
```

This function merely returns a string "Hello" as a plain text.

Here's a slightly more complex version:

web-server-simple/handler/handlers.go (lines 12-15)

```
12 func Handler2(w http.ResponseWriter, r *http.Request) {
```

```
13     w.Header().Add("Content-Type", "application/json")
14     io.WriteString(w, `{"greeting":"hello"}`)
15 }
```

The `Handler2` function sets a header, `"Content-Type: application/json"`, and it returns a simple JSON string `{"greeting":"hello"}`.

When you start this simple Web server, it print out the message, "Server starting...", and it just waits.

```
$ go run .

Server starting...
```

Let's try to access this Web server using, for example, *curl*.

```
$ curl http://localhost:8080/

Hello
```

It just returns the string "Hello" regardless of the URL path.

```
$ curl http://localhost:8080/abc

Hello
```

You can also try to access the server using a web browser.

24.3. Pair Programming

A function has a type. A variable or a literal of a function type can be used just like any other values.

Functions like `http.HandleFunc()`, for example, accept arguments of a function type. In particular, the second argument of `http.HandleFunc()` is a type `func(http.ResponseWriter, *http.Request)`.

As an exercise, let's create a kind of "universal" function that does any kind of binary operation, which takes two `int`s and returns an `int`.

```
func doBinaryOperation(l, r int, f func(int, int) int) int {
    return f(l, r)
}
```

This function accepts a value of function type, `func(int, int) int`, as its third argument, and calls the argument function using its first two arguments, which are `int`s.

If we pass a function argument that does addition, then `doBinaryOperation()` does addition. If we pass a function argument that does multiplication, then `doBinaryOperation()` does multiplication. It does anything that the argument function does.

Here's a simple test code:

```
func TestDoBinaryOperation(t *testing.T) {
    l, r := 1, 2

    sum := func(l int, r int) int {
        return l + r
```

```
    }
    p := doBinaryOperation(l, r, sum)
    if p != 3 {
        t.Fail()
    }

    m := doBinaryOperation(l, r, func(l int, r int) int {
        return l - r
    })
    if m != -1 {
        t.Fail()
    }
}
```

Both of these two tests "pass". The first test case calls `doBinaryOperation()` with a variable `sum` which has a type `func(l int, r int) int`. The second test case uses a function literal, `func(l int, r int) int { return l - r })`.

We can go a little bit further.

In this Web server example, the `http.HandleFunc()` function maps a path to a single handler function. You can use different handlers for different paths, but as is written, you can call only one handler per path.

Many Web server frameworks support constructs called "middleware". A middleware is a function that runs in the "middle" from the time when a request is received to the time when a response is sent. You can run multiple middlewares in series.

Let's build, as an exercise, a middleware framework into our simple Web server.

We can define a "middleware" as follows:

```
type middleware = func(http.Handler) http.Handler
```

It is a function that takes a `http.Handler` function and returns another `http.Handler` function. `http.Handler` is an interface type from the `http` package, and it is being used as a reference type. We can "chain-call" our `middleware`s one after another.

Here's an example of our `middleware` function:

```
func LogMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            log.Printf("%v\n", *r)
            next.ServeHTTP(w, r)
        })
}
```

This function *return a function* that simply logs the request `r` and calls the "next" handler function. Note that the type of `LogMiddleware()` is `middleware`, that is, `func(http.Handler) http.Handler`.

Now, let's create a function that takes a set of `middleware`s and return a `http.Handler` function, which we can use as the second argument for `http.Handle()`.

```
func RunMiddlewares(h http.Handler, middlewares ...middleware)
http.Handler {
    for _, mw := range middlewares {
        h = mw(h)
    }
    return h
```

```
}
```

We can use this function as follows:

```
func main() {
    http.Handle("/", handler.RunMiddlewares(
        http.HandlerFunc(handler.Handler1),
        handler.Middleware1,
        handler.Middleware2,
    ))

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Note that we use `http.Handle()` instead of `http.HandleFunc()` in this example.

24.4. Summary

We implemented a simple Web server using APIs of the `http` package. We also implemented a simple "middleware" framework using a function that takes a function and returns a function.

24.5. Exercises

1. Implement a handler that uses a query parameter, say, `name` and returns a string "Hello <name>". We have not discussed how to read URL query parameters, but *go doc* (or, the online Go documentations) is your best friend.
2. Create a simple Web server using a custom `http.ServeMux`, not the `http.DefaultServeMux`.
3. The `RunMiddlewares()` function calls middleware function in the reverse

order. And, it calls the "main handler" last (e.g., `handler.Handler1` in the example). How would you change its implementation so that middlewares are called in the order they are added but the main handler is still called last?

Lesson 25. TCP Client and Server

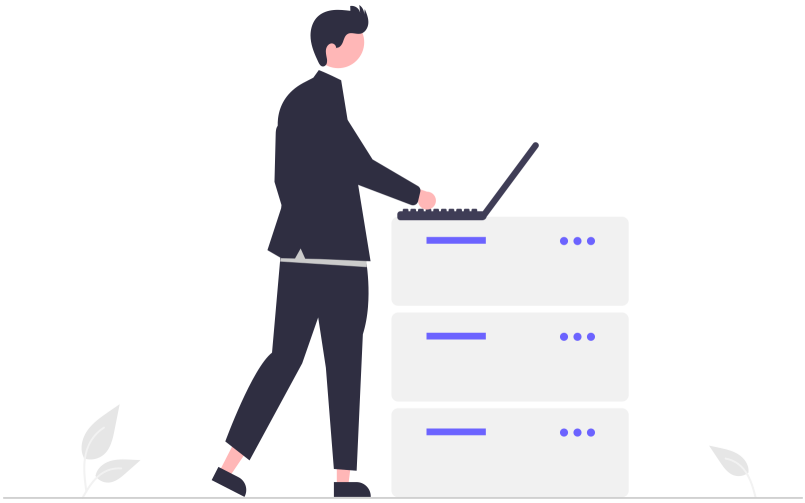
25.1. Introduction

The Internet is built on top of the *TCP/IP* protocol. The *HTTP* application layer protocol is built on top of *TCP/IP*, among other things.

Telnet is another popular application layer protocol that is built on *TCP/IP*. It used to be one of the main ways to connect to a remote machine, e.g., to get a shell access, but that use is mostly superseded by `SSH` at this point, which is more secure.

The Telnet protocol, however, still remains to be one of the most basic methods to facilitate "text-based" communications among remote hosts on the Internet. Here's a Wikipedia article, if you need more information: en.wikipedia.org/wiki/Telnet.

We are going to write a simple Telnet client, as well as a server, in this lesson.



25.2. Code Review - Client

Let's start with the client.

25.2.1. Package *main*

Here's the client's `main()` function:

telnet-client-simple/main.go (lines 10-31)

```
10 const (
11     host = "localhost"
12     port = 2323
13 )
14 
15 func main() {
16     client := telnet.NewClient(host, port)
17     err := client.Connect(false)
18     if err != nil {
19         log.Fatalln(err)
20     }
21     defer client.Close()
22 
23     go client.Listen()
24 
25     err = client.ProcessInput()
26     if err == io.EOF {
27         os.Exit(0)
28     } else if err != nil {
29         log.Fatalln(err)
30     }
31 }
```

The `main()` function first creates a `client` of type `telnet.Client` (or, its pointer type) from the `telnet` package. It uses its methods to connect and process input/output.

25.2.2. Package *telnet*

First, we create a new type `Client` to represent a Telnet client program and its behavior.

telnet-client-simple/telnet/client.go (lines 15-19)

```
15 type Client struct {
16     address       string
17     connection    net.Conn
18     ignoreNewline bool
19 }
```

The `Client` struct includes three fields. For example, `connection` is a network connection of type `net.Conn`. Note that none of them is exported.

`Client` implements the following set of methods:

```
func (c *Client) Connect(newline bool) (err error) {}
func (c *Client) Close() {}
func (c *Client) Listen() {}
func (c *Client) ProcessInput() error {}
```

The `Connect()` method is implemented as follows:

telnet-client-simple/telnet/client.go (lines 30-34)

```
30 func (c *Client) Connect(ignoreNewline bool) (err error) {
```

```
31      c.connection, err = net.Dial(TCP, c.address)
32      c.ignoreNewline = ignoreNewline
33      return
34  }
```

This method is just a simple wrapper around `net.Dial()`, which does all the heavy lifting.

- Package net [https://golang.org/pkg/net/]: Package `net` provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets. Although the package provides access to low-level networking primitives, most clients will need only the basic interface provided by the Dial, Listen, and Accept functions and the associated Conn and Listener interfaces.
    - func Dial [https://golang.org/pkg/net/#Dial]: Dial connects to the address on the named network. For TCP and UDP networks, the address has the form "host:port". The host must be a literal IP address, or a host name that can be resolved to IP addresses. The port must be a literal port number or a service name.

Note that the `Client.Connect()` method accepts a Boolean argument, `ignoreNewline`. We use this variable in the implementation of `ProcessInput()`.

The `Close()` method is a also simple wrapper around `net.Conn.Close()`:

telnet-client-simple/telnet/client.go (lines 36-41)

```
36  func (c *Client) Close() {
37      if c.connection != nil {
38          c.connection.Close()
39          c.connection = nil
40      }
41  }
```

The `Listen()` method starts a "listener loop".

telnet-client-simple/telnet/client.go (lines 43-62)

```
43 func (c *Client) Listen() {
44     for {
45         if c.connection != nil {
46             err := doCopy(c.connection)
47             if err != nil {
48                 fmt.Println(err)
49                 return
50             }
51         } else {
52             return
53         }
54     }
55 }
56 
57 func doCopy(src io.Reader) error {
58     if _, err := io.Copy(os.Stdout, src); err != nil {
59         return err
60     }
61     return nil
62 }
```

This listener looping continues until the connection is broken, or until it encounters an error.

The `ProcessInput()` method is defined as follows:

telnet-client-simple/telnet/client.go (lines 64-84)

```
64 func (c *Client) ProcessInput() error {
65     reader := bufio.NewReader(os.Stdin)
```

```
66      for {
67          cmd, err := reader.ReadString('\n')
68          if err != nil {
69              return err
70          }
71          if c.ignoreNewline {
72              cmd = strings.TrimRight(cmd, "\n")
73          }
74          c.send(cmd)
75      }
76  }
77
78  func (c *Client) send(cmd string) error {
79      if c.connection == nil {
80          return errors.New("Connection not established")
81      }
82      fmt.Fprint(c.connection, cmd)
83      return nil
84  }
```

`ProcessInput()` simply copies the user input to the destination `connection`, using a helper method, `send()`.

25.3. Pair Programming - Client

The implementation of this client program is straightforward.

The `main()` function creates a `client` for a particular remote address. The `host` and `port` are hard-coded in this case as `const`s. Normally, it will be better to read these values from the command line as we have done in some previous lessons.

The `main()` function first starts a goroutine to listen to the messages coming from a server, `client.Listen()`. Then (in parallel), it processes the user input from the

terminal, `client.ProcessInput()`.

The `go` keyword starts a function or method in a "goroutine", a thread of execution different from the one where the `main()` function is running. Note that we run the `ProcessInput()` function within the main goroutine. In this example, we ignore synchronization between these two goroutines.

The `client.Listen()` method starts an infinite loop. In each iteration, it reads an input from the connection and it copies the content to the `os.Stdout`, if any. The `doCopy(c.connection)` function simply calls `io.Copy(os.Stdout, c.connection)`.

The `ProcessInput()` method starts another infinite loop. It processes the user input on a line-by-line basis. It calls `send()` to send the user input to the connection, e.g., to the remote server.

Clearly, we cannot run both in the same thread, or goroutine. At least one, or both, functions will have to be run as separate goroutines.

Note the use of the `ignoreNewline` flag in the `ProcessInput()` method.

When a user types a text and presses an Enter key on the terminal, a newline is automatically added. In general, it may not be the user's intention to add a newline. It is just a signal indicating that the user is done with this particular line of input.

When you issue a command "ls" on a Unix shell, for instance, you press the `l` key, the `s` key, and the Enter key. The shell receives three characters corresponding to these three keystrokes. But, the command is actually "ls", without the trailing newline.

The same holds true with telnet. In fact, a telnet server like *telnetd* expects a newline (`\n` or `\r\n`) after each line of input.

By default, the `ignoreNewline` is set to `false`. But, there can be situations where

that is not very convenient.

For instance, in some communications, a newline may have a special meaning. If we use the newline as an end of line symbol, then it will be hard to tell which is a real newline and which is just an end of line signal.

As another example, if you develop a game server and a client that communicate over Telnet (or, over TCP), using some kind of a predefined text-based protocol, then it will be more desirable not to include the newline characters after each line.

HTTP is a text based protocol. In fact, we can use our "Telnet client" to communicate with an HTTP server. It is, strictly speaking, not a Telnet client per se in this use case. It is more of a TCP client. But, the distinction is not that important in practice.

The only thing we need to be careful about is the newlines. In HTTP, a "line" does not end with a newline character.

We built a simple Web server in the previous lesson, Simple Web Server. Let's run it on the localhost with port 8080.

Then, we can use our client to "talk" to this Web server. Run the client with the server address, `"localhost:8080"`. Note that we will need to change the `ignoreNewline` value to `true` in the `Connect()` method since we are not really using the "Telnet" protocol.

Once the client program runs, type the following in the input:

```
GET / HTTP/1.1          ①
Host: localhost         ②
```

① Type `GET / HTTP/1.1 ` and Enter.

② Type `Host: localhost` and Enter.

③ Just Enter.

You will get the same output as in the previous lesson:

```
Hello
```

This is an example request in the HTTP/1.1 protocol. Note the empty "line" at the end.

Obviously, our "Telnet client" does not know anything about HTTP, which is an application layer protocol. But, it knows TCP, thanks to the library we use, e.g., the `net` package.

We can use this client app to talk to any TCP server that uses text-based protocols as long as we understand those protocols.

25.4. Code Review - Server

25.4.1. Package *main*

The `main()` function creates a `server` of type `echo.Server` (or, its pointer type) from the `echo` package. The `host` and `port` are also hard-coded in this example.

tcp-server-echo/main.go (lines 8-21)

```
 8 const (
 9     host = "localhost"
10     port = 2323
11 )
12
13 func main() {
14     server, err := echo.NewServer(host, port)
```

```
15      if err != nil {
16          log.Fatalln("Failed to create a server:", err)
17      }
18      defer server.Close()
19
20      log.Fatalln(server.Listen())
21  }
```

The use of the port number `2323` is arbitrary. Telnet servers use port `23` by default.

25.4.2. Package *echo*

As with `Client`, let's create a new type `Server` to represent a TCP server's behavior:

tcp-server-echo/echo/server.go (lines 12-15)

```
12  type Server struct {
13      Address string
14      net.Listener
15  }
```

The `Server` struct includes two fields. Note that it has an embedded field `net.Listener`, which allows us to use a simpler syntax using "promotion".

The type `Server` implements the following method:

```
func (s *Server) Listen() error {}
```

Note that we have deliberately chosen an API that is rather similar to `http.ListenAndServe()` from the `http` package. This function does not return unless there is an error.

tcp-server-echo/echo/server.go (lines 30-47)

```
30 func (s *Server) Listen() error {
31     fmt.Printf("Listening on %s....\n", s.Address)
32 
33     for {
34         conn, err := s.Accept()
35         if err != nil {
36             return err
37         }
38 
39         err = handleConnection(conn)
40         if err == io.EOF {
41             continue
42         }
43         if err != nil {
44             return err
45         }
46     }
47 }
```

Here's `handleConnection()`:

tcp-server-echo/echo/server.go (lines 49-58)

```
49 func handleConnection(conn net.Conn) error {
50     defer conn.Close()
51 
52     for {
53         err := echo(conn)
54         if err != nil {
55             return err
56         }
57     }
```

```
58 }
```

The `handleConnection()` function also includes an "infinite loop" to process user inputs. The main service this this "echo server" provides is to `echo` the user input back to the client:

tcp-server-echo/echo/server.go (lines 60-78)

```
60 func echo(conn net.Conn) error {
61     buf := make([]byte, 1024)
62     read, err := conn.Read(buf)
63     if err != nil {
64         return err
65     }
66 
67     bytes := buf[:read]
68     input := strings.TrimRight(string(bytes), "\n")
69     fmt.Println("Received:", input)
70 
71     output := fmt.Sprintf("ECHO: %s\n", input)
72     _, err = conn.Write([]byte(output))
73     if err != nil {
74         return err
75     }
76 
77     return nil
78 }
```

25.5. Pair Programming - Server

Note the idiomatic way to start a server in the `main()` function:

```
log.Fatalln(server.Listen())
```

We will leave the implementation to the reader. As always, try to understand the sample code first, close the book, and create the same or similar program on your own. Your program does not have to be exactly the same as the sample.

You can just do "thought programming". You'll likely learn more by actually programming on computer, but that is not always required. Depending on where you are right now while reading this book, do whatever works best for you.

Once you are done with the server implementation, you can run both the server and the client, and test them to see if they really work.

Here's a sample session:

Server

```
tcp-server-echo$ go run .
Listening on localhost:2323....
Received: Hello server!
Received: How are you?
Received: Bye bye
```

Client

```
telnet-client-simple$ go run .
Hello server!                    ①
ECHO: Hello server!
How are you?                     ②
ECHO: How are you?
Bye bye                          ③
```

```
ECHO: Bye bye
```

① The client-side user types "Hello server!" and Enter.

② The user types "How are you?" + Enter.

③ The user types "Bye bye" + Enter.

Before we end this lesson, let's go back to the question, the value receiver vs the pointer receiver.

In the case of `telnet.Client` and `tcp.Server`, all the methods are implemented on the pointer types.

```
func (c *Client) Connect(newline bool) (err error) {}
func (c *Client) Close() {}
func (c *Client) Listen() {}
func (c *Client) ProcessInput() error {}
```

```
func (s *Server) Listen() error {}
```

As explained, there are a couple of reasons to use a pointer receiver. First, in order for the method to be able to modify the receiver value, it needs to use the pointer receiver. Second, to avoid copying the value on each method call, the pointer receiver is preferred.

We then suggested, as a general rule, prefer using pointer receivers.

So, when do we use value receivers?

In order to answer this question, we will have to go back to our discussion on value types vs reference types.

In languages like Java, this is essentially a non-issue. All custom types are reference types. There is no choice. In C#, the choice is limited: `struct` for value types, and `class` for reference types.

In languages like C++ and Go, however, it is much more complicated.

First of all, all custom types in Go are value types. But, for every value type, we can define a reference type corresponding to the value type, that is, its pointer type.

In general, all types in Go have this dual nature. (The builtin reference types like `slice` and `map` are exceptions.)

And, you can use a type either way, as a value type (or, "value-semantics" type) or as a reference type (or, "reference-semantics" type), e.g., using pointers. In some sense, this is a good thing. It gives a programmer more flexibility, more freedom, and more power.

Well...

> With great power comes great responsibility. ☺
>
> — Ben Parker, Spider-Man

It is generally a best practice to create a new type for one or the other use, but not for both. Not to mix their uses.

When you create a type, you will have to think of this question: Is this more like a value type (or, a value-semantics type), or more like a reference type (or, a reference-semantics type)?

"Small" types like `Point` and `Hand` that we used in the earlier lessons are value-semantics types, by their nature. Anything that are, in some ways, comparable to primitive types like `int` is a value type. Anything that it makes sense to "copy" is a value type.

Everything else should be a reference-semantics type. That is, you will (almost) exclusively use its pointer type `*T` rather than type `T` itself. The `Client` and `Server` types in this lesson belong to this category.

One other thing to note is that a value type should not generally include a reference type. For example, if a `struct` type includes a pointer type field, or a slice, then it will be probably best to use that type as a reference type. There are exceptions, however.

This is a general guideline. The compiler does not care, but you will end up writing better code if you stick to this rule or guideline, especially if you are new to the Go programming language.

An experienced C programmer intuitively knows this even if he/she may not have explicitly thought about it. In C, there is also an issue of memory management, which is tied to this issue of value vs reference, and stack vs heap memory, etc. It's much simpler in Go.

In Go, unlike in C++, however, there is one additional twist. You cannot implement an interface method for both a type and its pointer type. Go does not allow function overloading. You'll have to choose one or the other.

If you stick to the above guideline, then everything will work out fine. For a value-semantics type, you use value receivers *for all its methods.* (Or, for almost all.) For a reference-semantics type, you use pointer receivers *for all its methods.* (Or, for almost all.)

This general rule will encompass the two special cases mentioned before where only pointer receivers can be used (or, pointer receivers are preferred).

This discussion is really beyond the scope of an introductory book like this, but it is essential to understand this concept to become a proficient Go programmer. In the long run.

To reiterate, when you create a new type, you decide: Is the value semantics, or the reference semantics, more suitable for the values of this type?

As stated, this is a general guideline, especially for beginners. As you gain more experience, you will gain "better intuition" when it comes to making design decisions like this.

For "references types" (which will be primarily used as pointer types), you can generally mix pointer and value receivers. For example, you may decide to use pointer receivers for all but the `String()` method, etc. For "value value types" (which will be primarily used as value types), it is still preferred to exclusively use value receivers. There are, obviously, exceptions. For instance, the builtin type `string` is a value type. Syntactically. It is a `struct`. But, it includes a field of a reference type, a pointer to a byte array. So, in general, treating types like like this as reference types is better. And yet, `string` behaves like a value-semantics type in most cases. This is because of the fact that a `string` value is immutable in Go.

As indicated before, the Go language does not provide a way to make values of a user defined type immutable. We will not discuss the immutability aspect of programming languages in this book because it is largely irrelevant in Go.

A type can be made effectively immutable, outside a package, by not exposing the fields and not providing any methods that can mutate the values of the fields. This is similar to using the package level access control for OOP in Go. However, this "trick" has limitations. For example, all fields of a struct that are to be marshalled or unmarshalled should be exported, that is, if you use the standard `json` package.

25.6. Summary

We touched upon basic client-server programming in this lesson. We used APIs in the `net` package in the Go standard library.

We also spent some time discussing value receivers vs pointer receivers. The general rule is that, for a type, you use one or the other type receivers (almost) exclusively for all its methods and do not mix them.

25.7. Exercises

1. Try running `Client.ProcessInput()` in a separate goroutine. What changes are required? The simplest way to do this is probably using `sync.WaitGroup`. Refer to the API doc online, or use `go doc sync.WaitGroup`.
2. Create an echo server, as in this example, and create another server that does "relaying". A client talks to the relay server, and it merely re-sends the message to the echo server. Once the relay server receives the response from the echo server, it relays the response back to the client. How would you write this "relay server"?

Author's Note

"Polyglotting"

How was it? Was it fun? That was the end of Part II. Hope you learned something new from the lessons in this part. If any of the topics we covered here was not clear to you, then you can always go back and repeat the lessons (remember, "if you read a book 100 times, ..." ☺), or you can refer to different resources on the Web.

Many people learn foreign languages for fun. Even if you have no plans to

travel to Japan, for instance, at least not in the near future, you can still learn the Japanese language just for the fun of it. A lot of people do. A lot of people speak many languages.

Likewise, many programmers learn and use many different programming languages. Often, for practical reasons. But, sometimes, just for fun.

All programming languages are different. They offer different perspectives. They have different strengths and weaknesses. They have different areas of primary uses. For example, Python is now becoming *the* language for AI/machine learning. If you are interested in low-level systems programming, then you will have to use C/C++ or Rust.

Learn a new programming language just for the fun of it. If you use Java, for instance, at your work, then learn C#. Use C# to build something fun.

The author has used over 20 different programming languages over the years, sometimes by necessity, sometimes just for fun.

Here's a list of other (upcoming) books by the author, if you are interested:

- The Art of C# - Basics: Introduction to Programming in Modern C#
- The Art of Python - Basics: Introduction to Programming in Modern Python
- The Art of Typescript - Basics: Introduction to Programming in Typescript and Javascript
- The Art of Rust - Basics: Introduction to Programming in Rust
- The Art of C++ - Basics: Introduction to Programming in Modern C++

Yes, they all have more or less the same titles, except for the language part. ☺

They are part of the *"Learn Programming (Languages) for Fun"* series.

Review - Structs, Methods, Interfaces

Key Concepts

Function Types and Values

In Go, a function is a value, with a function type. Functions can be passed around just like other values. Function values may be used as function arguments and return values.

Flow Control

Switch Statements

A switch statement comprises a condition expression and a number of `case`s. It runs the first case whose value is equal to the condition expression. Switch cases evaluate cases from top to bottom, stopping when a case succeeds. Go's `switch` does not need `break` after each case.

Switch without a condition is the same as `switch true`. This construct can be a clean way to write long if-then-else chains.

Defer Statements

A `defer` statement defers the execution of a function until the surrounding function returns. The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns. Deferred function calls are pushed onto a stack. When a function returns, its deferred calls are executed in last-in-first-out order.

Advanced Types

Maps

A map stores a key to value mapping. The make function returns a map of the given type, initialized and ready for use. A map can also be initialized with a map literal. If the top-level type is just a type name, you can omit it from the elements of the literal.

Structs

A struct is a collection of fields of one or more types. Struct fields are accessed using the dot notation (`.`). They can be accessed through a struct pointer with the same dot notation without the explicit dereference.

A struct literal denotes a newly allocated struct value by listing the values of its fields. You can list just a subset of fields by using the `Name:` syntax. The order of named fields is irrelevant.

Methods

You can define methods on types, including struct types. A method is a function with a special receiver argument. The receiver appears in its own argument list between the func keyword and the method name. Methods are just functions with a different syntax.

Receivers

You can declare methods with value or pointer receivers. You can only declare a method with a receiver whose type is defined in the same package as the method.

Methods with a pointer receiver can modify the value to which the receiver points. Since methods often need to modify their receiver, pointer receivers are more common than value receivers. With a value receiver, the methods operate on a copy of the original value.

Interfaces

A set of method types defines an interface type. A type *implicitly* implements an interface by implementing its methods. An interface value holds a value of a specific underlying concrete type, which implements those methods. Calling a method on an interface value executes the method of the same name on its underlying type.

Implicit interfaces decouple the definition of an interface from its implementation, which could then appear in any package without prearrangement.

The Empty Interface

The interface type that specifies zero methods is known as the empty interface, `interface{}`.` An empty interface may hold values of any type. The empty interface type can be used by code that handles values of unknown type at build time.

Type Assertions

A type assertion provides access to an interface value's underlying concrete value. For example, the statement `t := i.(T)` asserts that the interface value i holds the concrete type T and assigns the underlying T value to the variable t. If i does not hold a T, the statement will trigger a panic.

To test whether an interface value holds a specific type, a type assertion can return two values: the underlying value and a boolean value that reports whether the assertion succeeded. E.g., in `t, ok := i.(T)`, if `i` holds a `T`, then `t` will be the underlying value and `o`k will be true. If not, `ok` will be false and `t` will be the zero value of type T, and no panic occurs.

Part III: Having Fun

There is no royal road to programming.

Lesson 26. Folder Tree

26.1. Problem

There is a Unix command *tree*, which lists the content of a directory, files and subdirectories, in a tree-like format.

Implement a simple *tree* command in Go.

Here's a relevant part from the *tree* man page.

```
DESCRIPTION
       Tree is a recursive directory listing program that produces a
depth indented listing of files. With no arguments, tree lists the
files in the current directory. When directory arguments are given,
tree lists all the files and/or directories found in the given
directories each in turn. Upon completion of listing all
files/directories found, tree returns the total number of files
and/or directories listed.
```

Here's a sample output from the *tree* command:

```
.
├── go.mod
├── main.go
├── temp
│   └── temp.txt
└── tree
    ├── print.go
    ├── tree1.go
    ├── tree1_test.go
```

```
    └── tree2.go

2 directories, 7 files
```

26.2. Discussion

The Go standard library includes various functions to facilitate file system related operations. We will use those APIs to traverse a directory hierarchy.

One of the the most convenient functions to use in this context is `filepath.WalkDir()`. Or, we can manually traverse the directory tree using the `fs` package APIs.

- Package filepath [https://golang.org/pkg/path/filepath/]: Package `filepath` implements utility routines for manipulating filename paths in a way compatible with the target operating system-defined file paths.
    - func WalkDir [https://golang.org/pkg/path/filepath/#WalkDir]: `WalkDir` walks the file tree rooted at root, calling fn for each file or directory in the tree, including root. All errors that arise visiting files and directories are filtered by fn: see the fs.WalkDirFunc documentation for details. The files are

walked in lexical order, which makes the output deterministic but requires WalkDir to read an entire directory into memory before proceeding to walk that directory. WalkDir does not follow symbolic links.

- Package fs [https://golang.org/pkg/io/fs/]: Package `fs` defines basic interfaces to a file system. A file system can be provided by the host operating system but also by other packages.
  - type WalkDirFunc [https://golang.org/pkg/io/fs/#WalkDirFunc]: `WalkDirFunc` is the type of the function called by WalkDir to visit each file or directory. The path argument contains the argument to WalkDir as a prefix. That is, if WalkDir is called with root argument "dir" and finds a file named "a" in that directory, the walk function will be called with argument "dir/a".
  - type DirEntry [https://golang.org/pkg/io/fs/#DirEntry]: A DirEntry is an entry read from a directory (using the ReadDir function or a ReadDirFile's ReadDir method).

26.3. Sample Code Snippets

The main function parses the command line argument, and if there is at least one argument, then it uses the first argument as a starting directory.

Otherwise it uses the current folder (`.`) by default.

folder-tree/main.go (lines 9-19)

```
 9 func main() {
10     folder := "."
11     if len(os.Args[1:]) > 0 {
12         folder = os.Args[1]
13     }
14
15     err := tree.Tree1(folder)
16     if err != nil {
```

```
17         log.Fatalln(err)
18     }
19 }
```

The easiest way to traverse a directory hierarchy is using `filepath.WalkDir()`. Here's a sample function, `Tree1()`.

folder-tree/tree/tree1.go (lines=9-25)

```
 9 func Tree1(folder string) error {
10     refDepth := computeDepth(folder, 0)
11     err := filepath.WalkDir(folder,
12         func(path string, d fs.DirEntry, err error) error {
13             if err != nil {
14                 return err
15             }
16             depth := computeDepth(path, refDepth)
17             prefix := buildPrefix(d.IsDir(), depth)
18             printTree(prefix, d.Name())
19             return nil
20         })
21     if err != nil {
22         return err
23     }
24     return nil
25 }
```

Here's an alternative implementation to traverse the directory structure in recursive way.

folder-tree/tree/tree2.go (lines=21-42)

```
21 func list(parentPath string, folder string, depth int) error {
```

```
    path, err := filepath.Abs(parentPath +
string(filepath.Separator) + folder)
    if err != nil {
        return err
    }

    files, err := os.ReadDir(path)
    if err != nil {
        return err
    }

    for _, file := range files {
        prefix := buildPrefix(file.IsDir(), depth)
        printTree(prefix, file.Name())

        if file.IsDir() {
            list(path, file.Name(), depth+1)
        }
    }

    return nil
}
```

The full code is included in the appendix at the end of the book.

Here's a sample output:

```
.
.. go.mod
.. main.go
== temp
.... temp.txt
== tree
.... print.go
```

```
.... tree1.go
.... tree1_test.go
.... tree2.go
```

26.4. Exercises

1. The sample code of this lesson does not include the counts of dirs and files, as in "real tree". Modify either or both `Tree()` functions to display that information.
2. Implement your own version of the `tree` command.

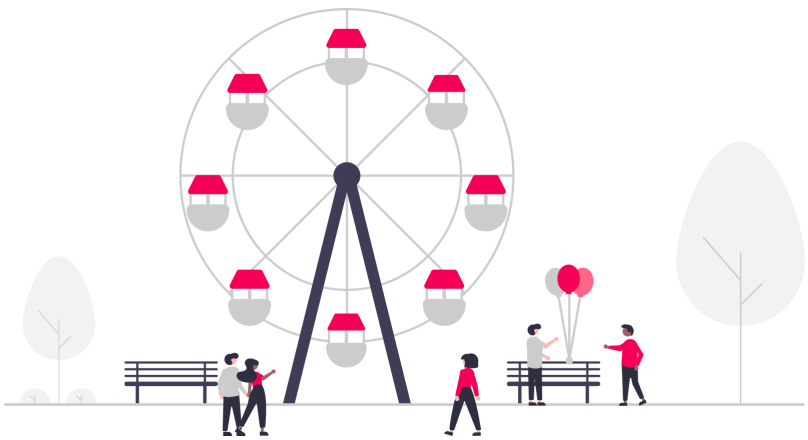
Lesson 27. Stack Interface

27.1. Problem

A *stack* is a data structure commonly used in programming. Here's a Wikipedia article if you need an intro or refresher: en.wikipedia.org/wiki/Stack_(abstract_data_type).

1. Implement a stack data structure using a slice.
2. Implement another stack using a linked list.
3. Create `Push()` and `Pop()` functions which take either type of stack as arguments.

Readers are encouraged to think about this problem before continuing.



27.2. Discussion

A stack, by definition, supports "push" and "pop" operations, in a LIFO fashion (last in, first out).

It is easy to implement a stack using a slice. We can embed a slice inside a stack structure, and support `push()` and `pop()` via slice operations.

We can `append()` to a slice to emulate the stack `push`. We can re-slice the slice to remove the last element and return it to emulate the stack `pop`.

It is a little bit harder to implement a stack using a linked list. Go standard library includes a doubly linked list container type, but not a singly linked list. We can either use the standard library doubly linked list or implement our own singly linked list.

Once we have a linked list data structure, we can map stack's *push* to a list operation of adding an item to the front. And, we can map stack's *pop* to a list operation of removing an item from the front.

In order to provide polymorphic behavior for `Push()` and `Pop()` functions, we will need to define an interface that encapsulate the "`stack`ness" of slice-based stacks and linked-list-based stacks.

27.3. Sample Code Snippets

The `main()` function demonstrates the uses of the polymorphic functions named `PushToStack()` and `PopFromStack()`. These two functions call the corresponding methods in the stack types.

stack-interface/main.go (lines 10-30)

```
10 func main() {
11     sliceStack := slice.New()
12
13     stack.PushToStack(sliceStack, []int{1, 2, 3})
14     stack.PushToStack(sliceStack, []int{4, 5})
15     stack.PrintStack(sliceStack)
16
```

```
17      sliceItem := stack.PopFromStack(sliceStack)
18      fmt.Printf("Item = %v\n", sliceItem)
19      stack.PrintStack(sliceStack)
20
21      linkedStack := linked.New()
22
23      stack.PushToStack(linkedStack, []int{1, 2, 3})
24      stack.PushToStack(linkedStack, []int{4, 5})
25      stack.PrintStack(linkedStack)
26
27      linkedItem := stack.PopFromStack(linkedStack)
28      fmt.Printf("Item = %v\n", linkedItem)
29      stack.PrintStack(linkedStack)
30  }
```

We define a stack using a slice as follows:

stack-interface/slice/slicestack.go (lines 5-9)

```
5  type data = interface{}
6
7  type SliceStack struct {
8      slice []data
9  }
```

Note that we use the empty interface type, `interface{}`, the mother of all types, for our data type. This is a common practice in Go since the language does not support generics (as of yet). You can just pick a simple type like `int` or `string`. There is no difference for this exercise.

Generics will be included in Go 1.18.

The `SliceStack` type implements the `Push()` and `Pop()` methods as follows:

stack-interface/slice/slicestack.go (lines 18-30)

```
18 func (s *SliceStack) Push(item interface{}) {
19     s.slice = append(s.slice, item.(data))
20 }
21
22 func (s *SliceStack) Pop() interface{} {
23     l := len(s.slice)
24     if l == 0 {
25         return nil
26     }
27     item := s.slice[l-1]
28     s.slice = s.slice[:l-1]
29     return item
30 }
```

`SliceStack` does not need to be concerned about formal `interface`s at this point. But, it is important to keep in mind that we are implementing a "behavior". A stack's behavior is defined by "push" and "pop".

Next, let's take a look at another example, a stack which internally uses a singly linked list. For this, we will need to create a type that represents a singly linked list.

stack-interface/linked/list.go (lines 3-5)

```
3 type list struct {
4     head *node
5 }
```

Here a `node` is defined as follows:

stack-interface/linked/node.go (lines 3-8)

```
3 type data = interface{}
4
5 type node struct {
6     item data
7     next *node
8 }
```

The `list` implements the following two methods:

stack-interface/linked/list.go (lines 13-24)

```
13 func (l *list) pushFront(n *node) {
14     n.next = l.head
15     l.head = n
16 }
17
18 func (l *list) popFront() (n *node) {
19     if l.head == nil {
20         return nil
21     }
22     n, l.head = l.head, l.head.next
23     return
24 }
```

A generic linked list will support more general API, but these two methods suffice for our purposes.

Then, we define a stack using this list, which we call `LinkedStack`, as follows:

stack-interface/linked/linkedstack.go (lines 8-10)

```
8 type LinkedStack struct {
9     *list
10 }
```

Note that we use "embedding" for the `list` field. This is semantically equivalent to the following although the embedding provides some syntactic convenience.

```
type LinkedStack struct {
    list *list
}
```

The `LinkedStack` type implements the `Push()` and `Pop()` methods as follows:

stack-interface/linked/linkedstack.go (lines 19-31)

```
19 func (s *LinkedStack) Push(item interface{}) {
20     n := node{
21         item: item.(data),
22     }
23     s.pushFront(&n)
24 }
25
26 func (s *LinkedStack) Pop() interface{} {
27     n := s.popFront()
28     if n == nil {
29         return nil
30     }
31     return n.item
32 }
```

Note that we use `s.pushFront(&n)`, for instance, instead of `s.list.pushFront(&n)`. Syntactically, it is almost as if `LinkedStack` "inherits" from `list`, in object oriented programming languages. This is called "promotion" in Go.

Now, let's create a function that takes a stack and pushes an element into the given stack.

For this, we will need to define an `interface` that captures this behavior. That is, a stack is something which we can push an element into.

stack-interface/stack/stack.go (lines 3-5)

```
3 type Pusher interface {
4     Push(item interface{})
5 }
```

Following the convention, we named this interface `Pusher` (because it includes one method `Push()`). We just use the broadest possible type `interface{}` for the item type.

Note that this is not type-aliased as in the case of `data` in concrete stack implementations. We have chosen the broadest possible API for these functions. Also, using a custom type name (type alias or type definition) will make the interface less generic.

We implement our polymorphic function as follows:

stack-interface/stack/operations.go (lines 5-7)

```
5 func PushToStack(s Pusher, item interface{}) {
6     s.Push(item)
7 }
```

The `PushToStack()` function just pushes the given item into the given `Pusher`. Now, since both `SliceStack` and `LinkedStack` implement the `Push()` method with the same signature as that in the `Pusher` interface, we can use a value of either of these stack types as an argument to the `PushToStack()` function.

We can do the same with a function that takes a stack and pops the head element from the given stack.

For this, we will need to define an `interface` that captures this behavior. That is, a stack is something which we can pop an element from. We name this interface `Popper`.

stack-interface/stack/stack.go (lines 7-9)

```
7 type Popper interface {
8     Pop() interface{}
9 }
```

Now, the `PopFromStack()` function:

stack-interface/stack/operations.go (lines 9-11)

```
9 func PopFromStack(s Popper) interface{} {
10     return s.Pop()
11 }
```

This function pops an item from the given `Popper` and returns the item.

Since both `SliceStack` and `LinkedStack` implement the `Pop()` method with the same signature as that in the `Popper` interface, we can use a value of either of these stack types as an argument to the `PopFromStack()` function.

You can find the full code listing in the appendix at the end of the book.

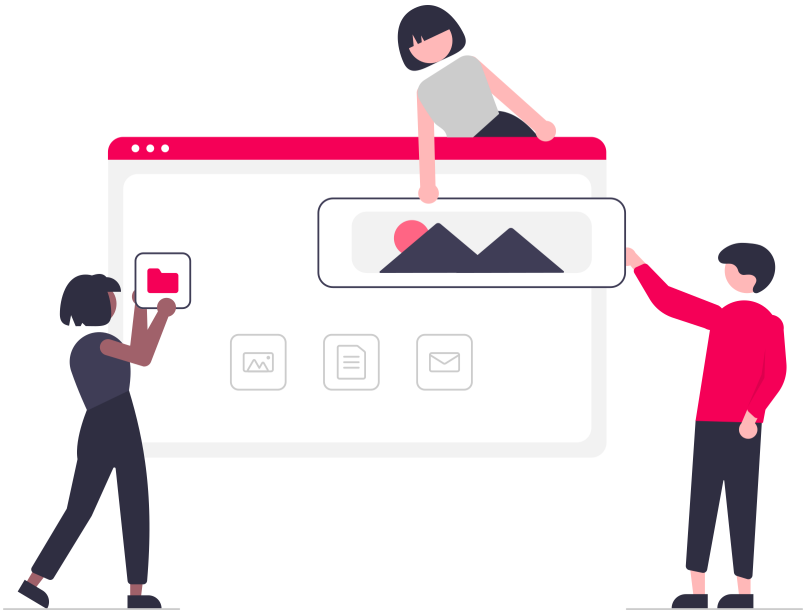
27.4. Exercises

1. Implement a slice-based stack type using a concrete data type, not `interface{}`. Is there any difference when you use a value type vs a pointer type?
2. Implement a linked list type using a concrete data type, not `interface{}`. Is there any difference when you use a value type vs a pointer type?
3. Implement a stack using a new linked list with the concrete data type. Is there any difference when you use a value data type vs a pointer type?
4. Implement a `Peek()` function for each stack type. A "peek" function returns an element at the top of a stack, if any, without "popping" the element. What are the considerations when you use a value data type vs a pointer type?
5. Create a stack type whose implementation can be switched at runtime, e.g., using a flag.

Lesson 28. Web Page Scraping

28.1. Problem

Given a web page URL, retrieve the page's information, in particular, its title and its description and keywords meta tags.



28.2. Discussion

HTML pages, written in the HTML markup language, are primarily to be consumed by end users, after a Web browser renders them in a user readable format.

Sometimes, however, a machine may need to consume the Web content in HTML, just like the Web browser program.

This is often known as "web scraping". We can extract some useful data from the websites in various ways.

Web scraping of certain websites can potentially violate the copyright laws in your jurisdiction, for example, depending on how you use the scraped data.

In our sample code, we get the HTML page from a given website, and extract its page title, and the description, keywords, and author fields from meta tags, if present.

Here's a sample output from the Yahoo home page:

```
$ go run . https://www.yahoo.com

2021/04/26 16:16:51 [ 0] URL: https://www.yahoo.com
2021/04/26 16:17:01 Extracted: {"title":"Yahoo","description":"News,
email and search are just the beginning. Discover more every day.
Find your yodel.","keywords":["yahoo","yahoo home page","yahoo
homepage","yahoo search","yahoo mail","yahoo messenger","yahoo
games","news","finance","sport","entertainment"],"author":""}
```

We internally create a JSON object based on the extracted data (e.g., for further processing down the line). If we "pretty print" this data,

```
{
  "title": "Yahoo",
  "description": "News, email and search are just the beginning.
Discover more every day. Find your yodel.",
  "keywords": [
    "yahoo",
    "yahoo home page",
```

```
    "yahoo homepage",
    "yahoo search",
    "yahoo mail",
    "yahoo messenger",
    "yahoo games",
    "news",
    "finance",
    "sport",
    "entertainment"
  ],
  "author": ""
}
```

In this example, the Yahoo home page does not include a meta tag for "author".

28.3. Sample Code Snippets

The `main()` function reads one or more URL arguments from the command line, and processes each URL sequentially.

website-title-single/main.go (lines 10-26)

```
10 func main() {
11     if len(os.Args) == 1 {
12         log.Fatalln("Provide URLs in the command line argument")
13     }
14
15     for i, url := range os.Args[1:] {
16         log.Printf("[%2d] URL: %s\n", i, url)
17
18         htmlmeta, err := processWebsite(url)
19         if err != nil {
20             log.Println("Error:", err)
```

```
            continue
        }

        log.Printf("Extracted: %s\n", htmlmeta)
    }
}
```

The `processWebsite()` function fetches the HTML content from the given URL and it calls `Extract()` in the `meta` package.

website-title-single/main.go (lines 28-41)

```
func processWebsite(url string) (*meta.HTMLMeta, error) {
    res, err := http.Get(url)
    if err != nil {
        return nil, err
    }

    htmlmeta, err := meta.Extract(res.Body)
    if err != nil {
        return nil, err
    }
    defer res.Body.Close()

    return htmlmeta, nil
}
```

The `meta.Extract()` does the "scraping" on the given HTML page content (`res.Body`). It returns a pointer value of `HTMLMeta` along with a potential `error`.

The `HTMLMeta` struct is defined in the `meta` package:

website-title-single/meta/htmlmeta.go (lines 5-10)

```
type HTMLMeta struct {
    Title       string   `json:"title"`
    Description string   `json:"description"`
    Keywords    []string `json:"keywords"`
    Author      string   `json:"author"`
}
```

Although the page title is not "meta" per se, we just lump them together in one struct for convenience.

The `Extract()` function parses the HTML content using the "golang.org/x/net/html" package.

website-title-single/meta/extract.go (lines 10-53)

```
func Extract(resp io.Reader) (*HTMLMeta, error) {
    tkzer := html.NewTokenizer(resp)

    hm := NewMeta()
    inTitleTag := false
    for token := tkzer.Next(); token != html.ErrorToken; token = tkzer.Next() {
        switch token {
        case html.StartTagToken, html.SelfClosingTagToken:
            t := tkzer.Token()
            if t.Data == "body" {
                return hm, nil
            } else if t.Data == "title" {
                inTitleTag = true
            } else if t.Data == "meta" {
                desc, ok := extractMetaProperty(t, "description")
                if ok {
```

```
                    hm.Description = desc
                }

                keywords, ok := extractMetaProperty(t, "keywords")
                if ok {
                    hm.Keywords = regexp.MustCompile(
  `(\s*,\s*)+`).Split(keywords, -1)
                }

                author, ok := extractMetaProperty(t, "author")
                if ok {
                    hm.Author = author
                }
            }
        case html.TextToken:
            if inTitleTag {
                t := tkzer.Token()
                hm.Title = t.Data
                inTitleTag = false
            }
        }
    }

    err := tkzer.Err()
    if err != nil && err != io.EOF {
        return nil, tkzer.Err()
    }
    return hm, nil
}
```

This is the first and the only time we use packages that are not from the standard library or from our own modules.

```
import (
```

```
    "golang.org/x/net/html"
)
```

The Go compiler tools use this string to find the package to import. Currently, the import spec has to be a URL-like string pointing to a source code repository (aka "go-gettable") so that the tools can fetch the necessary package code. The Go tools currently support a few source control systems including *git*.

The location "golang.org/x" contains a special set of packages. Some of them might be moved to the standard library in the future.

Here's the "go.mod" file:

website-title-single/go.mod

```
module examples/website-title-single

go 1.17

require golang.org/x/net v0.0.0-20210428140749-89ef3d95e781
```

Note the `require` line which includes a specific version information for the `golang.org/x/net` package.

You can download the external packages to your computer using the "go get" command:

```
go get
```

Or, if you want to update the dependencies to the latest versions, you can use the `-u` flag:

```
go get -u
```

When you have external package dependencies in your module, the go module tools also create a file called "go.sum", which includes information on indirect/transitive dependencies. Normally, you do not have to look at this file. This is primarily used, and managed, by the tools.

website-title-single/go.sum

```
golang.org/x/net v0.0.0-20210428140749-89ef3d95e781
h1:DzZ89McO9/gWPsQXS/FVKAlG02ZjaQ6AlZRBimEYOd0=
golang.org/x/net v0.0.0-20210428140749-89ef3d95e781/go.mod
h1:OJAsFXCWl8Ukc7SiCT/9KSuxbyM7479/AVlXFRxuMCk=
golang.org/x/sys v0.0.0-20201119102817-f84b799fce68/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
golang.org/x/sys v0.0.0-20210423082822-04245dca01da/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
golang.org/x/term v0.0.0-20201126162022-7de9c90e9dd1/go.mod
h1:bj7SfCRtBDWHUb9snDiAeCFNEtKQo2Wmx5Cou7ajbmo=
golang.org/x/text v0.3.6/go.mod
h1:5Zoc/QRtKVWzQhOtBMvqHzDpF6irO9z98xDceosuGiQ=
golang.org/x/tools v0.0.0-20180917221912-90fa682c2a6e/go.mod
h1:n7NCudcB/nEzxVGmLbDWY5pfWTLqBcC2KZ6jyYvM4mQ=
```

In order to clean up your dependencies, you do

```
go mod tidy
```

The implementation of `Extract()` should be easy to understand. It uses the `Tokenizer` type from the `net/http` package.

It iterates over the "tokens" using the “for ;;” loop. Note that this use of the `for` loop is similar to the `do while` loop in other C-style languages.

It uses the `switch` statement to find what we are looking for, e.g., `<title>...</title>` and other meta tags.

When you scrape Web content, it is best to start by browsing the HTML source of the target page. How to scrape certain data depends on how the data is embedded in the HTML markup.

In this particular example, we are only interested in the content in the `<head>...<head>` part, and we ignore the `<body>...</body>` part. In general, however, we will more likely want to extract data from the HTML body.

One thing to note in this function is that we use the `regex` package, which we haven’t covered in this book.

```
hm.Keywords = regexp.MustCompile(`(\s*,\s*)+`).Split(keywords, -1)
```

We will leave it as an exercise to the reader to understand what this statement does. (Remember, "documentation, documentation, documentation".)

We could have used a simpler function `strings.Split()`, but then we would have to `trim` all keywords. (Take a look at an HTML page source to see why that is.)

```
hm.Keywords = strings.Split(keywords, ",")
```

If we reach the end of the token list, as signaled by `err == io.EOF` by the `html/Tokenizer`, or if we run into an error, then we terminate the parsing.

The full code is included in the appendix at the end of the book.

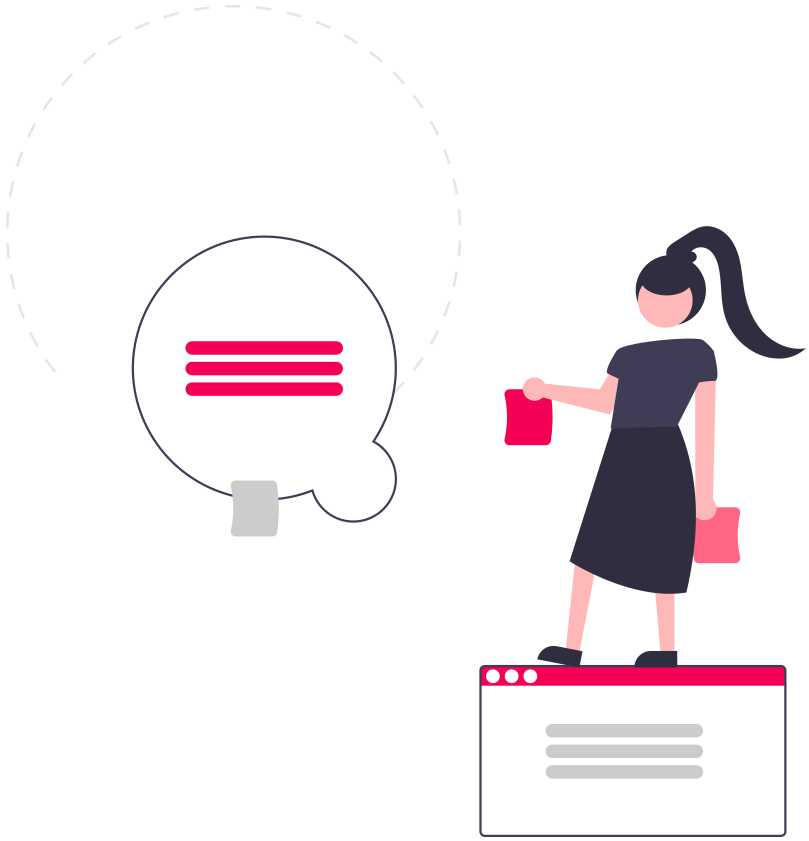
28.4. Exercises

1. In the example code, we call `extractMetaProperty()` multiple times, once for each meta tag we are looking for. How can we improve this implementation?
2. Get a list of all congressmen in your state, or any state if you live outside the United States, from the U.S. Congress website, www.congress.gov.
3. Create a Web crawler. Fetch all URLs (from `<a>` tags) from a given site, follow them recursively, up to a certain depth, and extract each site's title and description, among other things.

Lesson 29. QR Code Generator

29.1. Problem

Given a text such as a URL string, generate a QR code for the given text, as a PNG image.



29.2. Discussion

A QR code is a type of matrix barcode (or two-dimensional barcode). A barcode is a machine-readable optical label that contains information about the item to which it is attached such as the price of the item. QR codes often contain data for a locator, identifier, or tracker that points to a website or application.

You can find more information on the Web, for example, QR Code [https://en.wikipedia.org/wiki/QR_code].

The Go standard library includes a very interesting set of packages, namely, `image` and its sub-packages. They seem almost out of place because Go's support for graphics and/or user interfaces is in general pretty minimal.

However, if you think about it, the image manipulation is one of the most important tasks of the Web backend services. As stated, Go's forte is the Web backend development.

We are going to use some APIs from golang.org/pkg/image/ and golang.org/pkg/image/png/ to create PNG images.

In addition, we will use the `flag` package, golang.org/pkg/flag/, again from the Go standard library, to process the command line flags.

Many programs that use the command line arguments follow certain conventions that are more or less universal across all command line tools. The flag package comes in handy when you need to provide some systematic command line options for your Go programs.

Interestingly, there are many command line argument processing libraries written in, and for, Go. They are essentially "better" versions of the flag package. And yet, many of them are wildly popular.

> This will likely indicate that Go is one of the most favorite languages among the developers when it comes to creating command line tools, or server programs.

29.3. Sample Code Snippets

The main function handles the command line flags, and it calls the `qrcode.GeneratePNG()` function to generate, and save, a QR code image file which encodes the input string.

qrcode-generator/main.go (lines 10-25)

```
10 var pFile = flag.String("file", "", "QR Code image file name")
11 var pSize = flag.Int("size", 256, "QR Code size")
12
13 func main() {
14     flag.Parse()
15
16     if flag.NArg() == 0 {
17         log.Fatalln("Need to specify the content for the QR
   Code.")
18     }
19     content := strings.Join(flag.Args(), " ")
20
21     err := qrcode.GeneratePNG(content, *pFile, *pSize)
22     if err != nil {
23         log.Fatal(err)
24     }
25 }
```

As stated, we use various functions from the `flag` package to process the command line options. We will leave the usage of these functions from the flag package to the readers.

As emphasized before, the API doc is always your best friend. E.g., refer to Package flag [https://golang.org/pkg/flag/] to figure out how to use these functions. Alternatively, you can always use the `go doc` command.

For QR code generation, we use a third party package, *github.com/skip2/go-qrcode*.

As we discussed before, in order to be able to use a third party library in our program, we will need to indicate that in the `go.mod` file. For example,

```
require github.com/skip2/go-qrcode v0.0.0-20200617195104-da1b6568686e
```

And, we will need to do `go get`. (Alternatively, if we do `go get github.com/skip2/go-qrcode`, it will automatically update the go.mod file if the command runs successfully.)

Then, we will need to import it in the source code file that uses the package. For example,

```
import (
    qr "github.com/skip2/go-qrcode"
)
```

The `GeneratePNG()` function does a few things.

qrcode-generator/qrcode/png.go (lines 12-39)

```
12 func GeneratePNG(content string, imgFile string, size int) error {
13     code, err := qr.Encode(content, qr.Medium, size)
14     if err != nil {
15         return err
16     } else {
17         img, _, err := image.Decode(bytes.NewReader(code))
```

```
18          if err != nil {
19              return err
20          }
21
22          if imgFile == "" {
23              if err := png.Encode(os.Stdout, img); err != nil {
24                  return err
25              }
26          } else {
27              f, err := os.Create(imgFile)
28              if err != nil {
29                  return err
30              }
31              defer f.Close()
32
33              if err := png.Encode(f, img); err != nil {
34                  return err
35              }
36          }
37      }
38      return nil
39  }
```

First, it calls the `go-qrcode` library to encode the given string into a QR code, in `[]byte` format. Next, it converts the byte slice into an instance of `image.Image` type. Finally, it creates an (empty) file, or just uses stdout, to save the image in PNG format.

There is some duplication of code, e.g., we have `png.Encode(os.Stdout, img)` in two places, and it can be simplified, if desired. We will leave it to the readers.

The full code is included in the appendix at the end of the book.

In fact, the `github.com/skip2/go-qrcode` package includes a

> high-level function `WriteFile()`, which does more or less the same thing as `GeneratePNG()`.

So, how do you know how to use this CLI program? Well, it comes for free when we use a library like the `flag` package.

```
$ go run . -h

Usage of /tmp/go-build364500019/b001/exe/qrcode-generator:
  -file string
        QR Code image file name
  -size int
        QR Code size (default 256)
```

This option `-h`, or `--help`, is automatically generated by the `flag` package based on our program (e.g., the main() function).

Unfortunately, this help message only cares about the "flags", and it does not tell you that the program actually needs an argument (the text to be encoded as a QR code).

Integrating program arguments and flags will require a little bit more work than this code sample. We will leave it to the readers, as an exercise.

In any case, if you run the program without any arguments,

```
$ go run .

2021/07/04 14:40:45 Need to specify the content for the QR Code.
exit status 1
```

It tells you that an argument is needed. This is from the error check that we added

in the `main()` function:

```
if flag.NArg() == 0 {
    log.Fatalln("Need to specify the content for the QR Code.")
}
```

You can actually "install" this program, in fact all of the Go programs you create on your computer, on your system using the `go install` command. One nice thing about `go install` is that, once you install a Go program, it is available from your PATH so that you do not have to use `go run` or even specify the path to the executable.

(This will only work if you have installed your Go tools properly. Typically, the binaries are installed under `$GOPATH/bin`. If the `GOPATH` env variable is not explicitly set (which is rarely required in recent versions of Go), then the go tools use a folder named `go/bin` under the user's home directory.)

Try `go help install` for more information.

29.4. Exercises

1. Write a QR code generator program which outputs the code as a JPG image. The PNG and JPG packages have essentially the same APIs. For more information, refer to the API doc, golang.org/pkg/image/jpeg/. Or, `go doc image/jpeg`.
2. Add an additional flag to indicate the output format. That is, depending on its value, we will generate either a png image of jpeg image. If the flag is not specified, then generate a png image by default.

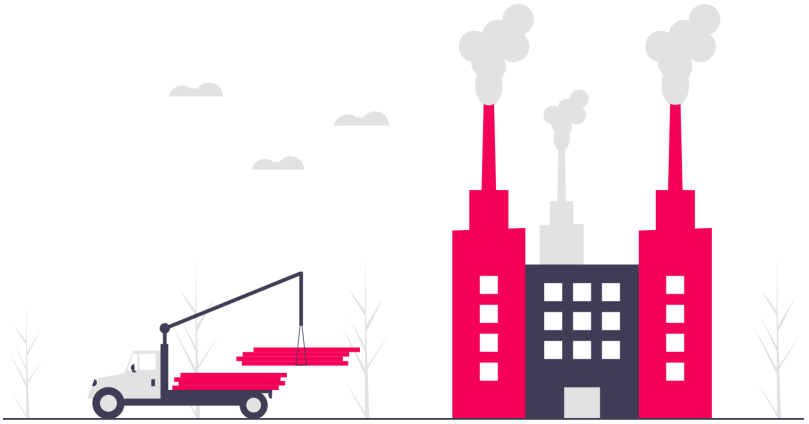
Lesson 30. Producer Consumer

30.1. Problem

Implement the classic producer consumer problem using `goroutines`: Producer-consumer problem [https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem].

Let the producer generate 10 integer numbers, and let the consumer print out those 10 numbers.

Once the task is done, terminate the program.



30.2. Discussion

An idiomatic way to tackle this kind of problem is using channels.

Goroutines communicate via channels, which is considered a better practice than sharing data directly in memory. Using shared memory requires exclusiveness. Using channels requires coordination.

We used goroutines in this book before.

A goroutine is a (lightweight) thread of execution managed by the Go runtime. Goroutines run in the same memory space, and the memory shared across different goroutines must be synchronized, in some way.

The Go standard library includes package `sync`, which provides basic synchronization primitives such as mutual exclusion locks (e.g., `Mutex`). Using these synchronization primitives are the traditional, and most common, ways of managing multiple threads. You can use them in Go as well if you need to, or if you want to.

In Go, however, sharing data via `channels`, instead of using shared memory, is considered a "better" way.

Channels are a typed conduit through which goroutines can send and receive values of a specific element type. The keyword `chan` is used to declare a channel type.

The optional `<-` operator specifies the channel direction, send or receive. If no direction is given, the channel is bidirectional. A channel may be constrained only to send or only to receive by assignment or explicit conversion.

The following channel can be used to send and receive values of type `string`:

```
var ch chan string
```

The following channel can only used to send `int64` values:

```
var ch chan<- int64
```

The following channel can only used to receive `float64` values:

```
var ch <-chan float64
```

A channel is a reference type. A `nil` channel is never ready for communication. Channels must be initialized before use, just like `slice`s or `map`s.

Channels act as FIFO queues (first in, first out). The values sent on a channel, from a goroutine, are received, say, by another goroutine, in the order they are sent. Channels do not need synchronization, unlike slices or maps.

A new, initialized channel value can be made using the built-in function `make()`, which takes a channel type and an optional capacity as arguments:

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

A channel with zero capacity is "unbuffered". Communication succeeds only when both the sender and receiver are ready.

Otherwise, the channel is buffered and communication succeeds without blocking if the buffer is not full (for *sends*) or not empty (for *receives*).

A channel may be closed with the built-in function `close()`. The multi-valued assignment form of the receive operator reports whether a received value was sent before the channel was closed.

A sender goroutine can close a channel to indicate that no more values will be sent. A receiver goroutine can test whether a channel has been closed by assigning a second parameter to the receive expression:

```
v, ok := <-ch
```

The value of `ok` will be false if there are no more values to receive and the channel is closed.

The loop “for i := range ch”" receives values from the channel repeatedly until it is closed.

30.3. Sample Code Snippets

Here’s the `main()` function, which creates two (unbuffered) channels, one for data communication, `msgs`, and another for sending the "done" signal, `done`.

producer-consumer/main.go (lines 9-23)

```
 9 const buff int = 0
10
11 func main() {
12     var msgs = make(chan int, buff)
13     var done = make(chan bool)
14
15     var p producer.Producer = producer.MakeProducer(msgs, done)
16     var c consumer.Consumer = consumer.MakeConsumer(msgs, done)
17
18     go p.Produce()
19     go c.Consume()
20
21     b := <-done
22     fmt.Println("DONE", b)
23 }
```

It creates a variable of the interface type `Producer` (as defined in the `producer` package) and a variable of the interface type `Consumer` (as defined in the `consumer` package).

And, it starts goroutines `Produce()` and `Consume()` on the producer and the consumer, respectively.

It should be noted that since `chan` is a reference type, the same `msgs` and `done` channels are shared by both producer `p` and consumer `c`.

Finally, it waits for the "done" message on the `done` channel.

```
b := <-done
```

When it receives the "done" value (`true` or `false`), it terminates the program.

In situations like this where we do not care about the actual values, a value of an empty struct type `struct{}` is often used. In this particular example, we will use the `bool` value to indicate a certain termination state.

The `Producer` interface type is introduced to demonstrate polymorphic behaviors. This is not really necessary for the operations of the goroutines and channels.

producer-consumer/producer/producer.go (lines 10-12)

```
10 type Producer interface {
11     Produce()
12 }
```

The `QuickProducer` struct is a concrete type, which implements the `Produce()` method.

producer-consumer/producer/producer.go (lines 14-17)

```
14 type QuickProducer struct {
```

```
15      msgs chan int
16      done chan bool
17  }
```

The `Produce()` method of the `QuickProducer` type generates ten integers, from `1` to `10`, and it returns.

producer-consumer/producer/producer.go (lines 27-35)

```
27  func (p *QuickProducer) Produce() {
28      for i := 1; i <= 10; i++ {
29          fmt.Printf("P: Sending Msg %d\n", i)
30          p.msgs <- i
31          fmt.Printf("P: Sent %d\n", i)
32          time.Sleep(delay)
33      }
34      close(p.msgs)
35  }
```

Closing the channel is not needed in this case.

Likewise, the `Consumer` interface type is introduced to demonstrate polymorphic behaviors.

producer-consumer/consumer/consumer.go (lines 10-12)

```
10  type Consumer interface {
11      Consume()
12  }
```

The `QuickConsumer` struct is a concrete type, which implements the `Consume()` method.

producer-consumer/consumer/consumer.go (lines 14-17)

```
14 type QuickConsumer struct {
15     msgs chan int
16     done chan bool
17 }
```

The `Consume()` method of the `QuickConsumer` type waits for `int` values on the `msgs` channel. When it has received 10 `int` values, it sends the "done" message to the `done` channel.

producer-consumer/consumer/consumer.go (lines 27-45)

```
27 func (c *QuickConsumer) Consume() {
28     count := 0
29     for {
30         fmt.Println("C: Waiting...")
31         msg, ok := <-c.msgs
32         if !ok {
33             c.done <- false
34             break
35         }
36         count++
37         fmt.Printf("C: Msg received %d\n", msg)
38         time.Sleep(delay)
39 
40         if count >= 10 {
41             c.done <- true
42             break
43         }
44     }
45 }
```

We can let the producer produce more than 10 values. But, the overall behavior would not change, in this particular example. Once the consumer receives 10 `int`s, it will send "done".

On the other hand, if the producer does not produce 10 values, then it can potentially lead to a deadlock since the consumer is waiting for 10 values.

The producer explicitly closing the channel in that case would help.

```
close(p.msgs)
```

The consumer can detect the channel closure, and act appropriately. In this example, it sends the "done" message (with value `false`), effectively terminating the program.

If you are new to goroutines and channels, then it would be instructive to see how its behavior changes when you change various parameters in the program (e.g., the message channel capacity, the relative size of the `delay` variables on the consumer and producer sides, etc.).

Here's a sample output (with `buff == 0`):

```
$ go run .

C: Waiting...
P: Sending Msg 1
P: Sent 1
C: Msg received 1
P: Sending Msg 2
C: Waiting...
C: Msg received 2
P: Sent 2
...
```

```
P: Sending Msg 10
C: Waiting...
P: Sent 10
C: Msg received 10
DONE true
```

You can find the full code listing in the appendix.

30.4. Exercises

1. Update the consumer `for` loop using the "channel range" loop. We have not discussed this in the book. You'll need to do some research to find out what that is.
2. Modify the web scraper to get the site metadata in parallel using goroutines. Use a map `map[string]*HTMLMeta` to store the scraped data. (Use URLs as keys.) There are a number of different ways you can do this. Let's suppose that the goal is to get the data from many websites (say, 1000 websites) as fast as possible.

Author's Note

Request for Feedback

The author is constantly looking to improve the book.

If you have any suggestions, then please let us know. We, and the future readers, will really appreciate it.

It can be anything from simple typos, unclear sentences, and formatting errors to bugs in the sample code and maybe downright incorrect explanations. Here's the author's email:

- harry@codingbookspress.com.

The author will try to correct the errors as soon as possible, if needed.

Thank you! 😀

Review - Goroutines, Channels

Key Concepts

Goroutines

A goroutine is a lightweight thread managed by the Go runtime. You can start a new goroutine by calling a function or a method with the `go` keyword. Goroutines run in the same address space, so access to shared memory must be synchronized.

Channels

Channels are a typed conduit through which you can send and receive values with the channel operator, `<-`. (The data flows in the direction of the arrow.) A channel can be created using the `make()` function. You cannot send or receive data via an unbuffered channel until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

Channels with non-zero length are buffered. You can send data to a buffered channel unless the buffer is full. You can receive data from a buffered channel unless the buffer is empty.

Part IV: Final Projects

All's well that ends well.

Lesson 31. Go Fish

31.1. Project

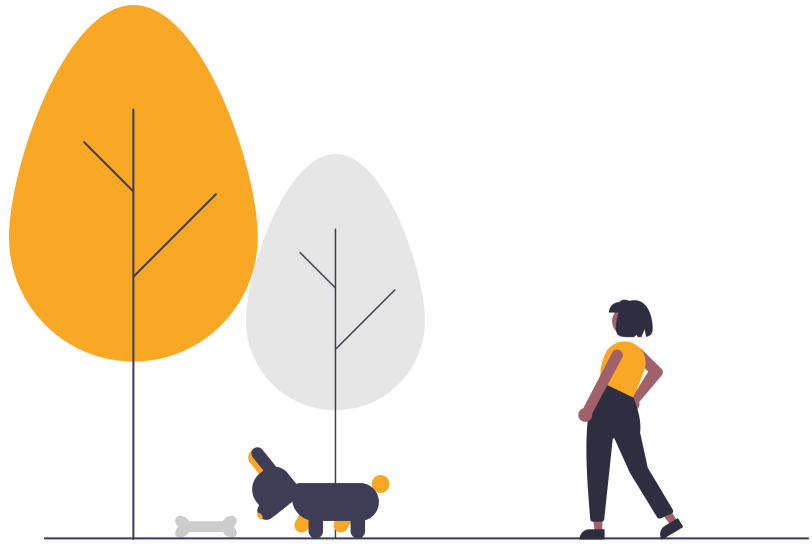
As a final project, we will work on a card game, in particular, "Go Fish".

Go Fish is one of the most popular card games played around the world: en.wikipedia.org/wiki/Go_Fish. We will design and implement this game, to be played on the terminal (e.g., via "CLI"), in this lesson.

Here's an except from the Wikipedia article, in case you are not familiar with the game:

```
Five cards are dealt from a standard 52-card deck (54 Counting
Jokers) to each player, or seven cards if there are three or fewer
players. The remaining cards are shared between the players, usually
spread out in a disorderly pile referred to as the "ocean" or "pool".

The player whose turn it is to play asks another player for their
cards of a particular face value. For example, Player A may ask,
"Player B, do you have any threes?" Player A must have at least one
card of the rank they requested. Player B must hand over all cards of
that rank if possible. If they have none, Player B tells Player A to
"go fish" (or just simply "fish"), and Player A draws a card from the
pool and places it in their own hand. Then it is the next player's
turn – unless the card Player A drew is the card they asked for, in
which case they show it to the other players, and they get another
turn. When any player at any time has four cards of one face value,
it forms a book, and the cards must be placed face up in front of
that player. Play proceeds to the left. When all sets of cards have
been laid down in books, the game ends. The player with the most
books wins.
```



31.2. Design

We will create a "single player" version of the game. More precisely, it will be a "one on one" play between a player and the computer.

The game, Go Fish, is available on Unix/Linux platforms.

```
$ man go-fish

NAME
     go-fish — play “Go Fish”

SYNOPSIS
     go-fish [-p]

DESCRIPTION
     go-fish is the game “Go Fish”, a traditional children's card
game.
```

```
     The computer deals the player and itself seven cards, and places
the rest of the deck face-down (figuratively).  The object of the
     game is to collect “books”, or all of the members of a single
rank.  For example, collecting four 2's would give the player a “book
     of 2's”.

     The options are as follows:

     -p      Professional mode.

     The computer makes a random decision as to who gets to start the
game, and then the computer and player take turns asking each other
     for cards of a specified rank.  If the asked player has any
cards of the requested rank, they give them up to the asking player.
A
     player must have at least one of the cards of the rank they
request in their hand.  When a player asks for a rank of which the
other
     player has no cards, the asker is told to “Go Fish!”.  Then, the
asker draws a card from the non-dealt cards.  If they draw the card
     they asked for, they continue their turn, asking for more ranks
from the other player.  Otherwise, the other player gets a turn.

     When a player completes a book, either by getting cards from the
other player or drawing from the deck, they set those cards aside
     and the rank is no longer in play.

     The game ends when either player no longer has any cards in
their hand.  The player with the most books wins.

     go-fish provides instructions as to what input it accepts.
```

Incidentally, Go Fish was the first game that the author played on Unix. ☺

Now how would you implement a game like this?

First, we will need some types to represent cards, hands, books, etc. This is not a must, but it helps to create an idiomatic Go program.

The game will have a "loop", as in most games, to process the user input, etc.

A Go Fish game will go through a number of different "phases". A game starts by dealing an initial set of cards to each player, that is, the user ("you") and the computer ("me").

Then, in the game loop, we need to determine which player's turn it is.

Then, if it's the player's turn (or "your turn"), read the player's input (e.g., a request for cards of a certain rank), handle the input according to the rules of the game, and determine the outcome.

The outcome can be one of the following:

- The computer does not have any cards of the rank that is requested. That is, "Go Fish", or
- The computer has one or more cards of the requested rank, and it hands over those cards to the player.

Based on the outcome, it determines who plays the next round.

If it's the computer's turn (or "my turn"), then we need first to pick a rank to ask for from the player. And, the game proceeds in a similar way.

We keep track of the books made throughout the game, and when ultimately all cards are exhausted and all books are made, the game ends.

Whoever has more books at the end wins the game.

31.3. Implementation

We will leave the implementation to the reader, as an exercise. An example program is included in the appendix: [appendix-code-listing-part4].

Here's a sample session:

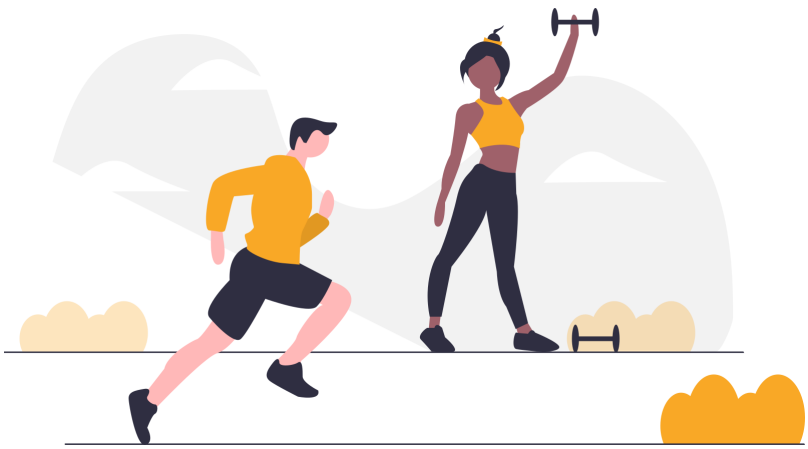
```
Starting a new game.
Dealing cards.
You were dealt 7 cards
Your hand: [2: 1], [3: 2], [4: 2], [8: 2],
I now have 7 cards as well
Determining the turn... I play first.
----------------------------------------------------------------
Me: 0 books, You: 0 books
----------------------------------------------------------------
It's my turn:
Give me A
You say, "Go Fish!"
I drew a card
----------------------------------------------------------------
Me: 0 books, You: 0 books
----------------------------------------------------------------
It's your turn:
Your hand: [2: 1], [3: 2], [4: 2], [8: 2],
Ask me for a card by rank (1 ~ 13)
8
You took 1 8s from me
----------------------------------------------------------------
Me: 0 books, You: 0 books
----------------------------------------------------------------
It's your turn:
Your hand: [2: 1], [3: 2], [4: 2], [8: 3],
Ask me for a card by rank (1 ~ 13)
----------------------------------------------------------------
```

```
4
"Go Fish!"
You drew 8 of Spades
You made a book: 8.
Your hand: [2: 1], [3: 2], [4: 2],
-----------------------------------------------------------------
Me: 0 books, You: 1 books
-----------------------------------------------------------------
It's my turn:
Give me Q
You say, "Go Fish!"
I drew a card
-----------------------------------------------------------------
Me: 0 books, You: 1 books
-----------------------------------------------------------------
...
```

The example code uses `labels`, which we did not cover in this book. Statements like `continue`, `break`, and `goto` (yes, Go has `goto`) can use the labels as their targets.

Lesson 32. Go Fish Galore

A few more Go Fish projects. After all, this is a book on Go.



There is no sample code for these final projects. This is a time for you to test your Go skills! 🚀

32.1. Project A

In the previous lesson, Go Fish, we created a program that lets you play a Go Fish game against a computer.

Modify the program so that you can play with multiple computer players, say, 1 to 5.

For example, you can specify the number of computer players as a command line argument:

```
./gofish 3
```

Note that when there are more than two players in the game, a player has to pick one of the other players before asking for cards.

32.2. Project B

Create a Go Fish game server and client programs. You can use the basic TCP server and client programs from TCP Client and Server as a basis for these programs.

The server can accept game requests from multiple users across the network. Use goroutines to handle each game.

Each player/client plays a one-on-one game against the game server.

32.3. Project C

Now, create a Go Fish game server that lets two or more users play in the same game. The game server can support multiple simultaneous games, with each game allowing multiple players. You can also add zero or more computer players in the mix.

This project requires a bit of design.

How will a user create a new game? How will a user start the game? How will a user find games that are currently accepting new players? Etc. etc.

All "project ideas" in this lesson are "open-ended". Use your imagination.

32.4. Project D

Make the computer player "smarter". What is the best strategy to win in Go Fish? Implement the strategy for the computer player.

Use your smart player in your server implementations in the previous projects, A, B, and C.

These are not easy projects for beginners. But, we covered everything you need to know to work on these problems in this book. Good luck!

Author's Note

Final Remarks

Congratulations! You made it! 🔥

It is not easy to read a technical book like this from beginning to end, regardless of your skill levels. This is a big achievement. Congrats!

As stated, knowledge is familiarity. The more you read, and the more you practice, the better you will become. In any art. Especially, in the art of programming.

Hope you had as much fun reading this book as I did writing it. ♡

Index

G

H

I

Credits

Images

All drawings used in this book are taken from undraw.co, an amazing service with an amazing open source license. Many thanks to the creator of the site: twitter.com/ninaLimpi!

Icons

All emoji icons used in this book are from fontawesome.com. Fontawesome is a very popular tool, probably used by almost everyone who does Web or mobile programming.

Typesetting

Here's another absolutely fantastic software, asciidoctor.org, which is used to create an ebook as well as paperback versions of this book. AsciiDoc [https://asciidoc.org] is like a Markdown on steroid. You can follow them on Twitter: twitter.com/asciidoctor.

Other Resources

The author has relied on many resources on the Web in writing this book, in particular, golang.org. If the book includes any material from these resources, then the copyright of those content belong to the respective owners.

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his academic and professional career. His experience spans broad areas from scientific programming and machine learning to enterprise software and Web and mobile app development.

He *occasionally* hangs out on social media:

- Instagram: @codeandtips [https://www.instagram.com/codeandtips/]
- TikTok: @codeandtips [https://tiktok.com/@codeandtips]
- Twitter: @codeandtips [https://twitter.com/codeandtips]
- YouTube: @codeandtips [https://www.youtube.com/@codeandtips]
- Reddit: r/codeandtips [https://www.reddit.com/r/codeandtips/]

Mini Programming Language References

We are creating a number of books under the series title, *A Hitchhiker's Guide to the Modern Programming Languages*. We cover essential syntax of the 12 select languages in 100 pages or so, Go, C#, Python, Typescript, Rust, C++, Java, Julia, Javascript, Haskell, Scala, and Lua.

These are all very interesting and widely used languages that can teach you different ways of programming, and more importantly, different ways of thinking.

All Books in the Series

> Already published, or to be published, throughout 2023

- Go Mini Reference
- Modern {cs} Mini Reference
- Python Mini Reference
- Typescript Mini Reference
- Rust Mini Reference
- C++20 Mini Reference
- Modern Java Mini Reference
- Julia Mini Reference
- Javascript Mini Reference
- Haskell Mini Reference
- Scala 3 Mini Reference

- Lua Mini Reference