Essentials of Go Programming

First Edition

Baiju Muthukadan

Essentials of Go Programming
First Edition
(English)

Baiju Muthukadan

Source repository: `https://github.com/baijum/essential-go`

To my mother Sulochana M.I.

Contents

Preface **ix**

Acknowledgements **xi**

1 Introduction **1**

1.1 Preparations 2

1.2 Organization of Chapters 5

1.3 Suggestions to Use this Book 7

2 Quick Start **9**

2.1 Hello World! 9

2.2 Building and Running Programs 10

2.3 The Example Explained 11

2.4 Organizing Code 12

2.5 Basics 13

2.6 Exercises 25

3 Control Structures **29**

3.1 If 29

3.2 For 33

3.3 Switch Cases 39

3.4 Defer Statements 43

3.5 Deffered Panic Recover 44

3.6 Goto 45

3.7 Exercises 46

4 Data Structures 49
4.1 Primitive Data Types 49
4.2 Constants 54
4.3 Arrays 58
4.4 Slices 59
4.5 Maps 63
4.6 Custom Data Types 66
4.7 Pointers 68
4.8 Exercises 70

5 Functions 73
5.1 Parameters 75
5.2 Return Values 76
5.3 Variadic Functions 77
5.4 Anonymous Functions 78
5.5 Function as Value 79
5.6 Methods 79
5.7 Exercises 83

6 Objects 85
6.1 Type with Multiple Interfaces 87
6.2 Empty Interface 87
6.3 Pointer Receiver 88
6.4 Type Assertions 89
6.5 Type Switches 90
6.6 Exercises 90

7 Concurrency 93
7.1 Goroutine 94
7.2 Channels 94
7.3 Waitgroups 96
7.4 Select 97
7.5 Buffered Channels 98
7.6 Channel Direction 99

7.7 Lazy Initialization Using sync.Once 100
7.8 Exercises 100

8 Packages 103
8.1 Creating a Package 104
8.2 Package Initialization 105
8.3 Documenting Packages 106
8.4 Publishing Packages 107
8.5 Module 108
8.6 Moving Type Across Packages 109
8.7 Exercises 110

9 Input/Output 113
9.1 Command Line Arguments 113
9.2 Files 114
9.3 Standard Streams 116
9.4 Using flag Package 118
9.5 String Formatting 119
9.6 Exercises 122

10 Testing 125
10.1 Failing a Test 127
10.2 Logging Message 127
10.3 Failing with Log Message 128
10.4 Skipping Test 129
10.5 Parallel Running 130
10.6 Sub Tests 130
10.7 Exercises 131

11 Tooling 133
11.1 Getting Help 133
11.2 Basic Information 134
11.3 Building and Running 136
11.4 Running Test 137

11.5 Formatting Code 139
11.6 Generating Code 140
11.7 Embedding Code 140
11.8 Displaying Documentation 141
11.9 Find Suspicious Code Using Vet 141
11.10 Exercises 142

Appendix A: Answers 145

Further Readings 155

Index 157

Preface

For surely there is an end; and thine expectation shall not be cut off. — Proverbs 23:18

If you are searching for a straightforward and robust programming language suitable for a variety of applications, Go is an excellent option to consider. Go is an open-source language developed by Google with significant contributions from the community. The project originated in 2007 through the efforts of Robert Griesemer, Rob Pike, and Ken Thompson. It was subsequently released as open-source software by Google in November 2009. Go has gained popularity among numerous organizations across diverse problem domains.

This book serves as an introduction to the fundamentals of Go programming. Whether you are a novice programmer or someone seeking a refresher, I hope this book proves to be valuable. If you are entirely new to programming, I recommend exploring the Scratch programming language website at `https://scratch.mit.edu`. It provides helpful resources for those who have not yet delved deeply into programming. Once you have grasped the fundamental programming concepts, you can return to this book.

My journey with programming began around 2003 when I started working with Python. Over the course of a decade, I gained extensive experience in Python programming. In 2013, a former colleague introduced me to Go, and it proved to be a refreshing experience. Although there are notable differences between Python and Go, I was particularly impressed by Go's simplicity in the early stages. Compared to other languages I had explored, the learning curve for Go was remarkably smooth.

Upon developing an interest in Go, one of my initial aspirations was to write a book about it. Writing has long been my passion, with my first foray being a blog on LiveJournal in 2004. In 2007, I authored my first book on Zope component architecture. Writing can be an enjoyable activity, although at times it can become demanding. Since this book is self-published, I had the freedom to take my time and ensure its quality.

Throughout the years, I have conducted numerous Go workshops in various parts of India. During these workshops, I always desired to offer something more to the participants, and I believe this book will fulfill that aspiration.

Software development encompasses more than just programming languages. It involves acquiring additional skills such as proficiently using text editors/IDEs, version control systems, and your preferred operating system. Furthermore, it is beneficial to continually expand your knowledge by exploring different languages and technologies throughout your career. While Go leans toward an object-oriented programming style, you may also find it worthwhile to explore functional programming languages like Scheme and Haskell.

Apart from technical skills, business domain knowledge and soft skills play a pivotal role in your career growth. However, discussing these aspects in detail falls beyond the scope of this context. With that said, I conclude by wishing you a successful career.

Baiju Muthukadan
Kozhikode, Kerala, India
May 2023

Acknowledgements

First and foremost, I express my gratitude to my God for granting me the ability to write this book. I would like to extend my heartfelt thanks to my wife, Nithya, and our two sons, Aadidev and Aarav, for their unwavering love, care, and support throughout this endeavor. I am also indebted to my colleagues and friends for their invaluable assistance.

I am incredibly grateful for the abundance of feedback I have received from my colleagues, friends, and fellow members of the Go community. Special recognition goes to Satish V.J., Nurali Virani, Preeti Chandrashekar, Parvathy V.R., Sunil Malagi, Bilal M.K, Iliyas Shirol, Pratik Shah, and Andrei Tudor Călin for their contributions.

The encouragement I have received from the Go community has been truly remarkable, and I feel privileged to be a part of it. My sincere appreciation goes out to the entire Go community for their support and camaraderie.

Chapter 1

Introduction

I try to say everything at least three times: first, to introduce it; second, to show it in context; and third, to show it in a different context, or to review it. — Robert J. Chassell (An Introduction to Programming in Emacs Lisp)

Computer programming skill helps you to solve many real-world problems. Programming is a process starting from the formulation of a computing problem to producing computer programs (software). Computer programming is part of a more extensive software development process.

Programming involves analysis, design, and implementation of the software. Coding is the activity of implementing software. Sometimes coding involves more than one programming language and use of other technologies. Learning a programming language is a crucial part of the skill required for computer programming or software development in general.

Using a programming language, we are preparing instructions for a computing machine. The computing machine includes a desktop computer, laptop, and mobile phone.

There are many programming languages in use today with different feature sets. You should learn to pick the right programming language for the problem at hand. Some languages are more suitable for specific problems. This book provides an introduction to the Go programming language. Studying the Go programming should help you to make the right decision about programming language choice for your projects. If you have already decided to use Go, this book should give an excellent introduction to Go programming.

Go, also commonly referred to as Golang, is a general-purpose programming language. Go was initially developed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. Go was publicly released as an open source software in November 2009 by Google.

Many other programming languages including C, Python, and occam has inspired the design of the Go programming language. Go programs can run on many operating systems including GNU/Linux, Windows and Mac OS X.

You require some preparations to learn Go programming. The next section explains the preparations required.

This book is for beginners who want to learn to programme. The readers are expected to have a basic knowledge of computers. This book covers all the major topics in the Go programming language. Each chapter in this book covers one or two major topics. However many minor topics are introduced intermittently.

This chapter provides an introduction to the language, preparations required to start practicing the exercises in this book, organizing code and walk through of remaining chapters. Few suggestions for learning Go using this book is given towards the end of this chapter.

1.1 Preparations

Learning a natural language like English, Chinese, and Spanish is not just understanding the alphabet and grammar of that particular language. Similarly, there are many things that you need to learn to become a good programmer. Even though this book is going to focus on Go programming; now we are going to see some other topics that you need to learn. This section also explains the installation of the Go compiler and setting up the necessary development environment.

Text editors like Notepad, Notepad++, Gedit, and Vim can be used to write Go programs. The file that you create using the text file is called source file. The Go compiler creates executable programs from the source file. You can run the executable program and get the output. So, you need a text editor and Go compiler installed in your system.

Depending on your operating system, follow the instruction given below to install the Go compiler. If you have difficulty following this, you may get some help from your friends to do this step. Later we write a simple Go program and run it to validate the steps.

You can use any text editor to write code. If you are not familiar with any text editor, consider using Vim. You can bootstrap Vim configuration for Go programming language from this website: `https://vim-bootstrap.com`.

Using a source code management system like Git would be helpful. Keeping all your code under version control is highly recommended. You can use a public code hosting service like GitHub, Bitbucket, and GitLab to store your examples.

Linux Installation

Go project provides binaries for major operating systems including GNU/Linux. You can find 64 bit binaries for GNU/Linux here: `https://go.dev/dl`

The following commands will download and install Go compiler in a 64 bit GNU/Linux system. Before performing these steps, ensure Go compiler is not installed by running `go` command. If it prints `command not found...`, you can proceed with these steps.

These commands must be run as *root* or through *sudo*. If you do not know how to do it, you can get help from somebody else.

```
cd /tmp
wget https://go.dev/dl/go1.x.linux-amd64.tar.gz
tar -C /usr/local -zxvf go1.x.linux-amd64.tar.gz
```

The first line ensure that current working directory is the `/tmp` directory.

In the second line, replace the version number from the Go downloads website. It's going to download the 64 bit binary for GNU/Linux. The `wget` is a command line download manager. Alternatively you can use `curl` or any other download manager to download the tar ball.

The third line extract the downloaded tar ball into `/usr/local/go` directory.

Now you can exit from the *root* user or stop using *sudo*.

By default Go packages are installed under `$HOME/go` directory. This directory can be overridden using `GOPATH` environment variable. Any binaries installed using `go install` goes into `$GOPATH/bin` directory.

You can also update PATH environment variable to include new binary locations. Open the `$HOME/.bashrc` file in a text editor and enter this lines at the bottom.

```
export PATH=$HOME/go/bin:/usr/local/go/bin:$PATH
```

Windows Installation

There are separate installers (MSI files) available for 32 bit and 64 bit versions of Windows. The 32 bit version MSI file will be named like this: *go1.x.y.windows-386.msi* (Replace `x.y` with the current version). Similarly for 64 bit version, the MSI file will be named like this: *go1.x.y.windows-amd64.msi* (Replace `x.y` with the current version).

You can download the installers (MSI files) from here: `https://go.dev/dl`

After downloading the installer file, you can open the MSI file by double clicking on that file. This should prompts few things about the installation of the Go compiler. The installer place the Go related files in the `C:\Go` directory.

The installer also put the `C:\Go\bin` directory in the system `PATH` environment variable. You may need to restart any open command prompts for the change to take effect.

You also need to create a directory to download third party packages from github.com or similar sites. The directory can be created at `C:\mygo` like this:

```
C:\> mkdir C:\mygo
```

After this you can set `GOPATH` environment variable to point to this location.

```
C:\> go env -w GOPATH=C:\mygo
```

You can also append `C:\mygo\bin` into the `PATH` environment variable.

If you do not know how to set environment variable, just do a Google search for: *set windows environment variable*.

Hello World!

It's kind of a tradition in teaching programming to introduce a *Hello World* program as the first program. This program normally prints a *Hello World* to the console when running.

Here is our hello world program. You can type the source code given below to your favorite text editor and save it as `hello.go`.

Listing 1.1 : Hello World! (hello.go)

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Once you saved the above source code into a file. You can open your command line program (bash or cmd.exe) then change to the directory where you saved the program code and run the above program like this:

```
$ go run hello.go
Hello, World!
```

If you see the output as `Hello, World!`, congratulations! Now you have successfully installed Go compiler. In fact, the `go run` command compiled your code to an executable format and then run that program. The next chapter explains more about this example.

Using Git

You should be comfortable using a source code management system. As mentioned before, Git would be a good choice. You can create an account in GitHub and publish your example code there. If you do not have any prior experience, you can spend 2 to 3 days to learn Git.

Using Command Line

You should be comfortable using command line interfaces like GNU Bash or PowerShell. There are many online tutorials available in the Internet to learn shell commands. If you do not have any prior experience, you can spend few days (3 to 4 days) to learn command line usage.

1.2 Organization of Chapters

The rest of the book is organized into the following chapters. You can read the first six chapters in the given order. The remaining chapters can be read in any order.

Chapter 2: Quick Start
This chapter provides a tutorial introduction to the language. It introduce few basic topics in Go programming language. The topics include Data Types, Variables, Comments, For Loop, Range Clause, If, Function, Operators, Slices, and Maps.

Chapter 3: Control Structures
This chapter cover the various control structures like *goto*, *if condition*, *for loop* and *switch case* in the language. It goes into details of each of these topics.

Chapter 4: Data Structures
This chapter cover data structures. The chapter starts with arrays. Then slices, the more useful data structure built on top of array is explained. Then we looked at how to define custom data types using existing primitive types. The struct is introduced which is more useful to create custom data types. Pointer is also covered. like *struct*, *slice* and *map* in the language.

Chapter 5: Functions
This chapter explained all the major aspects of functions in Go. The chapter covered how to send input parameters and return values. It also explained about variadic function and anonymous function. This chapter briefly also covered methods.

Chapter 6: Objects
This chapter explained the concept of objects and interfaces and it's uses. Interface is an important concept in Go. Understanding interfaces and properly using it makes the design robust. The chapter covered empty interface. Also, briefly explained about pointer receiver and its significance. Type assertions and type switches are also explained.

Chapter 7: Concurrency
This chapter explained concurrency features of Go. Based on your problem, you can choose channels or other synchronization techniques. This chapter covered goroutines and channels usage. It covered Waitgroups, Select statement. It also covered buffered channels, channel direction. The chapter also touched upon *sync.Once* function usage.

Chapter 8: Packages
This chapter explained the Go package in detail. Package is one of building block of a reusable Go program. This chapter explained about creating packages, documenting packages, and finally about publish packages. The chapter also covered

modules and its usage. Finally it explained moving types across packages during refactoring.

Chapter 9: Input/Output
This chapter discussed about various input/output related functionalities in Go. The chapter explained using command line arguments and interactive input. The chaptered using *flag* package. It also explained about various string formatting techniques.

Chapter 10: Testing
This chapter explained writing tests using the *testing* package. It covered how to mark a test as a failure, logging, skipping, and parallel running. Also, it briefly touched upon sub-tests.

Chapter 11: Tooling
This chapter introduced the Go tool. All the Go commands were explained in detail. Practical example usage was also given for each command. The chapter coverd how to build and run programs, running tests, formatting code, and displaying documentation. It also touched upon few other handy tools.

In addition to the solved exercises, each chapter contains additional problems. Answers to these additional problems are given in Appendix A.

And finally there is an index at the end of the book.

1.3 Suggestions to Use this Book

Make sure to setup your system with Go compiler and the environment as explained in this chapter. If you are finding it very difficult, you may get help from your friends to setup the environment. Use source code management system like Git to manage your code. You can write exercises and solve additional problems and keep it under version control.

I would suggest not to copy & paste code from the book. Rather, you can type every example in this book. This will help you to familiarize the syntax much quickly.

The first 6 chapters, that is from Introduction to Interfaces should be read in order. The remaining chapters are based on the first 6 chapters. And chapters 7 onward can be read in any order.

Summary

This chapter provided an introduction to Go programming language. We briefly discussed about topics required to become a good programmer.

Then we covered chapter organization in this book. And finally, I offer a few suggestions for how to use this book. The next chapter provides a quick start to programming with the Go language.

Chapter 2

Quick Start

Software engineering is what happens to programming when you add time, and other programmers. – Russ Cox

This chapter walks through a few basic topics in Go. You should be able to write simple programs using Go after reading and practicing the examples given in this chapter. The next 3 sections revisit the hello world program introduced in the last chapter. Later we will move on to a few basic topics in Go. We will learn about data types, variables, comments, For loops, range clauses, If, functions, operators, slices, and maps.

2.1 Hello World!

Here is the hello world program introduced in the previous chapter. You can type the below code to your favorite text editor and save it as **`hello.go`**. This program will print a message, **`Hello, World!`** into your console/terminal.

Listing 2.1 : Hello World

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

You can open your command line program and run the above program like this:

```
$ go run hello.go
Hello, World!
```

What you wrote in the **hello.go** is a structured document. The characters, words, spaces, line breaks and the punctuation characters used all are important. In fact, we followed the ”syntax” of Go language. According to Wikipedia, the syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language.

The **go run** command is easy to use when developing programs. However, when you want to use this program in production environment, it is better to create executable binaries. The next section briefly explain the process of building executable binaries and running it.

2.2 Building and Running Programs

You can use the **go build** command to compile the source and create executable binary programs. Later this executable can run directly or copied to other similar systems and run.

To compile (build) the hello world program, you can use this command:

```
$ go build hello.go
```

This command produce an an executable file named **hello** in the current directory where you run the **build** command. And you can run this program in a GNU/Linux system as given below. The **./** in the beginning of the command ensure that you are running the **hello** program located in the current directory:

```
$ ./hello
Hello, World!
```

In Windows, the executable file name ends with **.exe**. This is how you can run the executable in Windows:

```
C:\> hello.exe
Hello, World!
```

The **go build** command produce a binary file native to the operating system and the architecture of the CPU (i386, x86_64, arm etc.)

2.3 The Example Explained

The first line is a clause defining the name of the package to which the file belongs. In the above hello world program, the `hello.go` file belongs to the the `main` package because of this clause at the beginning of the file:

```
package main
```

A package is a collection of Go source files. Package sources can be spread across multiple source files in a directory. If you want to produce an executable from your program, the name of package should be named as `main`. Always use lowercase letters for the package names.

The second line has kept as blank for readability. The 3rd line is an `import` declaration. The `import` declaration enable accessing external packages from the current package. In the above example, `fmt` package is imported like this.

```
import "fmt"
```

If a package is imported, it must be used somewhere in the source files. Otherwise, the compiler will produce an error. As you can see above, the import declaration starts with a word `import` followed by the name of the package in double quotes. If multiple packages need be imported, you can group the imports into a parenthesis (factored import) to reduce typing. Here is an example for factored import:

```
import (
    "fmt"
    "math"
)
```

The name of the package for the built-in packages will be the name given within quotes of the import statement. If the import string is a path separated by slash, then name of the package will be the last part of the string. For example, ”net/http” package name is `http`. For other third party vendor packages, the name should be verified within the source code.

Names within the imported package can be referred using a dot operator as you can see above: `fmt.Println`. A name is considered as exported if it begins with a capital letter. For example, the name `Area` is an exported name, but `area` is not exported.

The `https://go.dev/play` site can be used to share Go source code publicly. You can also run the programs in the playground.

Again we have added one blank line after the import statement for readability. The fifth line starts with a function definition. In this case, this is a special function named `main`. A function is a collection of instructions or more specifically statements. A function definition starts with `func` keyword followed by function name then arguments (parameters) for the function within parenthesis and finally statements within curly brackets. The `main` function is a special function which doesn't accept any arguments. The starting curly bracket should be in the same line where function definition started and statements should start in the next line. There should be only one `main` function for an executable program.

Inside the main function, we are calling the `Println` function available inside the `fmt` package.

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

The above function call is a complete statement in Go. The `Println` function print the string into standard output of the terminal/console and also add a new line at the end of the string.

2.4 Organizing Code

As mentioned above, a package is a collection of Go source files. Package sources can be spread across multiple source files in a directory. For a given package, all the variables, functions, types, and constants defined in one source file can be directly referrenced from other sources files.

A Git repository normally contain one module, located at the root, however it is possible to add more than one, if necessary. A Go module is a collection of Go packages that are released together.

To understand the code organization, you also need to understand about Go module. A file named *go.mod* declares the module path: the import path prefix for all packages within the module. The module contains the packages in the directory containing its go.mod file as well as subdirectories of that directory, up to the next subdirectory containing another go.mod file (if any).

Note that you don't need to publish your code to a remote repository before you can build it. A module can be defined locally without belonging to a repository. However, it's a good habit to organize your code as if you will publish it someday.

Each module's path not only serves as an import path prefix for its packages, but also indicates where the go command should look to download it. For example, in order to download the module golang.org/x/tools, the go command would consult the repository indicated by https://golang.org/x/tools (described more here).

An import path is a string used to import a package. A package's import path is its module path joined with its subdirectory within the module. For example, the module github.com/google/go-cmp contains a package in the directory cmp/. That package's import path is github.com/google/go-cmp/cmp. Packages in the standard library do not have a module path prefix.

2.5 Basics

Data Types

Data is unorganized facts that requires processing. In programming, the data is processed and organized to be useful. Data type provides a classification for the data. Date type is often simply called as *type*. Data type is one of the fundamental concept in any programming language. In most of the places in this book, we will say data as "value". More advanced data type is often called data structures.

Consider an example, you want to work with names of toys in your programs. So, the values of the "names of toys" is the data. The data type that you can use to represent this data is called "string". If you are literally writing a string in Go, you can use a double quote around the names like this:

```
"Sheriff Woody"
"Buzz Lightyear"
"Jessie"
```

In the hello world example, we used the string "Hello, World!" literally. Representation of a string value within source code is called string literal.

Consider a related example, you want to mark whether the toys are male or not. This type of data is called Boolean data. So, if the toy is male, the value will be `true` otherwise `false` as given below:

```
{"Sheriff Woody",  true}
{"Buzz Lightyear", true}
{"Jessie",        false}
```

Apart from *string*, and *bool*, Go has some other data types like *int*, *byte*, *float64* etc.

Variables

Let's go back to the hello world example, if you want to print the hello world message three times. You will be required to write that sentence three times as given below.

Listing 2.2 : Multiple Hello World

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
    fmt.Println("Hello, World!")
    fmt.Println("Hello, World!")
}
```

This is where the concept called *variable* becoming useful. Instead of using the literal string three times, you can use a short variable name to refer that string value. The variable is like an alias referring to the data. The name of the variable is considered as an identifier for the variable. Consider the example below where a variable named `hw` is used to refer the "Hello, World!" string literal.

Listing 2.3 : Reusing variable

```go
package main

import "fmt"

func main() {
    hw := "Hello, World!"
    fmt.Println(hw)
    fmt.Println(hw)
    fmt.Println(hw)
}
```

As you can see in the above example, we are using two special characters (`:=`) in between the variable name and the string literal. The colon character immediately followed by equal character is what you can use to define a short variable declaration in Go. However, there is a small catch here, the this short syntax for declaring variable will only work inside a function definition. The Go compiler identify the type of variable as string. This process of identifying data type automatically is called *type inference*.

To assign a new value to the variable, you can use = as given in the below example:

Listing 2.4 : Assign new value to variable

```
package main

import "fmt"

func main() {
    hw := "Hello, World!"
    fmt.Println(hw)
    hw = "Hi, New World!"
    fmt.Println(hw)
}
```

The output will look like this:

```
$ go run t4.go
Hello, World!
Hi, New World!
```

You can also explicitly define the type of variable instead of using the `:=` syntax. To define the type of a variable, you can use the keyword `var` followed by the name of the type. Later, to assign a string value for the `hw` variable, you can use `=` symbol instead of `:=`. So, the example we can rewrite like this.

Listing 2.5 : Alternate syntax for variable declaration

```
package main

import "fmt"

func main() {
    var hw string
    hw = "Hello, World!"
    fmt.Println(hw)
    fmt.Println(hw)
    fmt.Println(hw)
}
```

The variable declared outside the function (package level) can access anywhere within the same package.

Variables declared at the function level must be used. Otherwise, the compiler is going to throw an error during compilation.

The keyword *var* can used to declare more than one variable. You can also assign values along with `var` declaration. Unlike `:=` syntax give above, the variable declaration using *var* keyword can be at package level or inside function.

Here are different ways how you can declare a variable:

```
var variable type
var variable type = value
var variable = value
var variable1, variable2 type = value1, value2
```

If value is not given, a default ”zero” value will be assigned. The zero value is: 0 for numeric types (int, int32 etc.), false for Boolean type, and empty string for strings.

Here are a few examples.

```
var name string
var age int = 24
var length = 36
var width, height int = 3, 6
```

The same examples using short declaration look like this.

```
name := ""
age := 24
length := 36
width, height := 3, 6
```

We used names like `hw`, `name`, `age`, `length` etc. as identifiers for variables. An identifier should start with an alphabet or underscore, and it can contain digits afterwards. But there are certain reserved words called keywords which are not allowed to be used as identifiers. We have already seen some keywords like `package`, `import`, `func` and `var`. In the next few sections, we are going to see some more keywords like `for`, `if` etc. These keywords has special meaning in the language.

Comments

Writing documentation helps the users to understand the code better. Go provides syntax to write documentation in the form of comments. The comments will be written along with source code. Comments are ignored by the compiler. Usually comments are written for two purpose:

- To explain complex logic or remarks about part of code
- Application programming interface (API) documentation

There are two kinds of comments, the one form is a multi-line comment and the other form only allows single line comment.

The multi-line comment starts with `/*` and ends with `*/`. And everything in between is considered as comments.

Here is a multi-line comment to document the package named `plus`. As you can see here, the comment is used to give a brief description about the package and two example usages are also given.

Listing 2.6 : Package level comment

```
/*
Package plus provides utilities for Google+
Sign-In (server-side apps)

Examples:

  accessToken, idToken, err := plus.GetTokens(code, clientID,
                                                       clientSecret)
  if err != nil {
      log.Fatal("Error getting tokens: ", err)
  }

  gplusID, err := plus.DecodeIDToken(idToken)
  if err != nil {
      log.Fatal("Error decoding ID token: ", err)
  }
*/
package plus
```

The other form of comments is inline comments and it starts with two forward slashes (`//`). All the characters till end of line is treated as comments. Even if you have any valid code within comment, it will not be considered by compiler to produce the executable binary. Here is an example line comment:

```
// SayHello returns wishing message based on input
func SayHello(name string) string {
    if name == "" { // check for empty string
        return "Hello, World!"
    } else {
        return "Hello, " + name + "!"
    }
}
```

In the above example the first line is a line comment. The “godoc” and similar tool treated this comment as an API documentation.

There is another comment in the line where name equality with empty string is checked. These kind of comment helps the reader of source code to understand what that attribute is used for.

For Loop

Repeating certain process is a common requirement in programming. The repetition process aiming a result is called iteration. In Go, the iteration is performed by using the `for` loop block.

In the previous section about variable, we printed the `Hello, World!` message three times. As you can see there, we repeatedly printed the same message. So, instead of typing the same print statement again and again, we can use a `for` loop as given below.

Listing 2.7 : For loop (sum1.go)

```go
package main

import "fmt"

func main() {
    hw := "Hello, World!"
    for i := 0; i < 3; i++ {
        fmt.Println(hw)
    }
}
```

The for loop starts with a variable initialization, then semi-colon, then a condition which evaluate `true` or `false`, again one more semi-colon and an expression to increment value. After these three parts, the block starts with a curly bracket. You can write any number of statements within the block. In the above example, we are calling the `Println` function from `fmt` package to print the hello world message.

In the above example, the value `i` was initialized an integer value of zero. In the second part, the condition is checking whether the value of `i` is less than 3. Finally, in the last part, the value of `i` is incremented by one using the `++` operator. We will look into operators in another section later in this chapter.

Here is another example `for` loop to get sum of values starting from 0 up to 10.

Listing 2.8 : For loop (sum2.go)

```go
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum)
}
```

The initialization and increment part are optional as you can see below.

Listing 2.9 : For loop (sum3.go)

```
package main

import "fmt"

func main() {
    sum := 1
    for sum < 1000 {
        sum += sum
    }
    fmt.Println(sum)
}
```

An infinite loop can be created using a `for` without any condition as given below.

Listing 2.10 : Infinite For loop

```
package main

func main() {
    for {
    }
}
```

If

One of the common logic that is required for programming is branching logic. Based on certain criteria you may need to perform some actions. This could be a deviation from normal flow of your instructions. Go provides `if` conditions for branching logic.

Consider a simple scenario, based on money available you want to buy vehicles. You want to buy a bike, but if more money is available you also want to buy a car.

Listing 2.11 : If control structure (buy.go)

```
package main
```

```
 3 import "fmt"

 5 func main() {
 6     money := 10000
 7     fmt.Println("I am going to buy a bike.")
 8     if money > 15000 {
 9         fmt.Println("I am also going to buy a car.")
10     }
11 }
```

You can save the above program in a file named **buy.go** and run it using **go run**. It's going to print like this:

```
$ go run buy.go
I am going to buy a bike.
```

As you can see, the print statement in the line number 9 didn't print. Because that statement is within a condition block. The condition is **money > 15000**, which is not correct. You can change the program and alter the money value in line number 7 to an amount higher than 15000. Now you can run the program again and see the output.

Now let's consider another scenario where you either want to buy a bike or car but not both. The **else** block associated with **if** condition will be useful for this.

Listing 2.12 : If with else block

```
package main

import "fmt"

func main() {
    money := 20000
    if money > 15000 {
        fmt.Println("I am going to buy a car.")
    } else {
        fmt.Println("I am going to buy a bike.")
    }
}
```

You can save the above program in a file named **buy2.go** and run it using **go run**. It's going to print like this:

```
$ go run buy2.go
I am going to buy a car.
```

Similar to **for** loop, the **if** statement can start with a short statement to execute before the condition. See the example given below.

Listing 2.13 : If with initialization statement

```
package main

import "fmt"

func main() {
    if money := 20000; money > 15000 {
        fmt.Println("I am going to buy a car.")
    } else {
        fmt.Println("I am going to buy a bike.")
    }
}
```

A variable that is declared along with `if` statement is only available within the `if` and `else` blocks.

Function

Function is a collection of statements. Functions enables code reusability. Function can accept arguments and return values. To understand the idea, consider this mathematical function:

$$f(r) = 3.14r^2$$

Figure 2.1: Mathematical function for area of a circle

This function square the input value and multiply with 3.14. Depending on the input value the output varies.

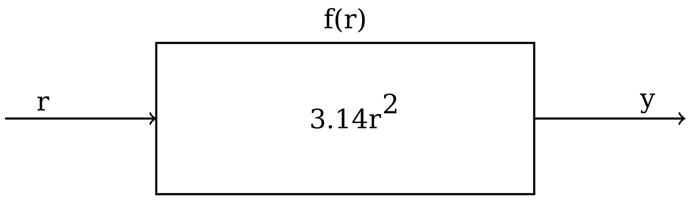

Figure 2.2: Blackbox representation of a function

As you can see in the above diagram, `r` is the input and `y` is the output. A function in Go can take input arguments and perform actions and return values. A simple implementation of this function in Go looks like this.

```
func Area(r float64) float64 {
    return 3.14 * r * r
}
```

The function declaration starts with **func** keyword. In the above example, **Area** is the function name which can be later used to call the function. The arguments that can be received by this function is given within brackets. The line where function definition started should end with an opening curly bracket. The statements can be written in the next line on wards until the closing curly bracket.

Here is a complete example with usage of the Area function.

Listing 2.14 : Function usage

```
 1  package main

 3  import "fmt"

 5  // Area return the area of a circle for the given radius
 6  func Area(r float64) float64 {
 7      return 3.14 * r * r
 8  }

10  func main() {
11      area := Area(5.0)
12      fmt.Println(area)
13  }
```

In the above example, the **Area** function is called in line number 11 with an argument of **5.0**. We are using the short variable declaration. The type of the variable **area** will be **float64** as the **Area** function returns with that type.

Operators

Programming languages use operators to simplify the usage. Operators behave more or less like functions. More specifically, operators combine operands to form expressions. We have already seen few operators like `:=`, `=`, `+=`, `++`, `*`, `>` and `<`.

The `:=`, `=`, `+=` are assignment operators. The `*` is the multiplication operator. The `>` and `<` are comparison operators.

Sometimes logical conditions should be checked to proceed with certain steps. Logical operators does these kind kind of checking. Let's say you want to check whether a particular value is divisible by 3 and 5. You can do it like this.

```
if i%3 == 0 {
    if i%5 == 0 {
        // statements goes here
    }
}
```

The same thing can be achieved using conditional AND logical operator (`&&`) like this.

```
if i%3 == 0 && i%5 == 0 {
    // statements goes here
}
```

Apart from the conditional AND, there are conditional OR (`||`) and NOT (`!`) logical operators. We will see more about operators in the next chapter.

Slices

Slice is a sequence of values of the same type. In computer science terminology, it's a homogeneous aggregate data type. So, a slice can contain elements of only one type of data. However, it can hold a varying number of elements. It can expand and shrink the number of values. `[]T` is a slice with elements of type T.

The number of values in the slice is called the length of that slice. The slice type `[]T` is a slice of type `T`. Here is an example slice of color names:

```
colors := []string{"Red", "Green", "Blue"}
```

In the above example, the length of slice is `3` and the slice values are string data. The `len` function gives the length of slice. See this complete example:

Listing 2.15 : Printing slice values

```
package main

import "fmt"

func main() {
    colors := []string{"Red", "Green", "Blue"}
    fmt.Println("Len:", len(colors))
    for i, v := range colors {
        fmt.Println(i, v)
    }
}
```

If you save the above program in a file named `colors.go` and run it, you will get output like this:

```
$ go run colors.go
Len: 3
0 Red
1 Green
2 Blue
```

The **range** clause loop over through elements in a variety of data structures including slice and map. Range gives index and the value. In the above example, the index is assigned to **i** and value to **v** variables. As you can see above, each iteration change the value of **i** & **v**.

If you are not interested in the index but just the value of string, you can use blank identifier (variable). In Go, underscore is considered as blank identifier which you need not to define and you can assign anything to it. See the example written below to print each string ignoring the index.

Listing 2.16 : Range loop with index ignored

```go
package main

import "fmt"

func main() {
    colors := []string{"Red", "Green", "Blue"}
    fmt.Println("Len:", len(colors))
    for _, v := range colors {
        fmt.Println(v)
    }
}
```

If you just want to get the index without value, you can use just use one variable to the left of range clause as give below.

Listing 2.17 : Range loop without index

```go
package main

import "fmt"

func main() {
    colors := []string{"Red", "Green", "Blue"}
    fmt.Println("Len:", len(colors))
    for i := range colors {
        fmt.Println(i, colors[i])
    }
}
```

In the above example, we are accessing the value using the index syntax: `colors[i]`.

Maps

Map is another commonly used complex data structure in Go. Map is an implementation of hash table which is available in many very high level languages. The data organized like key value pairs. A typical map type looks like this:

```
map[KeyType]ValueType
```

A `KeyType` can be any type that is comparable using the comparison operators. The `ValueType` can be any data type including another map. It is possible add any numbers of key value pairs to the map.

Here is a map definition with some values initialized.

```
var fruits = map[string]int{
      "Apple":  45,
      "Mango":  24,
      "Orange": 34,
  }
```

To access a value corresponding to a key, you can use this syntax:

```
mangoCount := fruits["Mango"]
```

If the key doesn't exist, a zero value will be returned. For example, in the below example, value of `pineappleCount` is going be `0`.

```
pineappleCount := fruits["Pineapple"]
```

More about maps will be explained in the data structure chapter.

2.6 Exercises

Exercise 1: Print multiples of 5 for all even numbers below 10

Solution:

This exercise requires getting all even numbers numbers below 10. As we we have seen above, a `for` loop can be used to get all numbers. Then `if` condition can be used with `%` operator to check whether the number is even or not. The `%` operator given the gives the remainder and we can check it is zero or not for modulus 2. If the number is even use the `*` operator to multiply with 5.

Here is the program.

```
package main

import "fmt"
```

```
func main() {
    for i := 1; i < 10; i++ {
        if i%2 == 0 {
            fmt.Println(i * 5)
        }
    }
}
```

Exercise 2: Create a function to reverse a string

Solution:

```
package main

import "fmt"

func Reverse(s string) string {
    var r string
    for _, c := range s {
        r = string(c) + r
    }
    return r
}

func main() {
    hw := "Hello, World!"
    rhw := Reverse(hw)
    fmt.Println(rhw)
}
```

Exercise 3: Find sum of all numbers below 50 completely divisible by 2 or 3 (i.e., remainder 0).

Hint: The numbers completely divisible by 2 or 3 are 2, 3, 4, 6, 8, 9 ... 45, 46, 48.

Solution:

```
package main

import "fmt"

func main() {
    sum := 0
    for i := 1; i < 50; i++ {
        if i%2 == 0 {
            sum = sum + i
        } else {
            if i%3 == 0 {
                sum = sum + i
            }
        }
    }
    fmt.Println("Sum:", sum)
}
```

The logic can be simplified using a conditional OR operator.

```
package main

import "fmt"

func main() {
    sum := 0
    for i := 1; i < 50; i++ {
        if i%2 == 0 || i%3 == 0 {
            sum = sum + i
        }
    }
    fmt.Println("Sum:", sum)
}
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a function to check whether the first letter in a given string is capital letters in English (A,B,C,D etc).

Hint: The signature of the function definition could be like this: `func StartsCapital(s string) bool`. If the function returns `true`, the string passed starts with a capital letter.

Problem 2: Write a function to generate Fibonacci numbers below a given value.

Hint: Suggested function signature: `func Fib(n int)`. This function can print the values.

Summary

We began with a ”hello world” program and briefly explained it. This chapter then introduced a few basic topics in the Go programming language. We covered data types, variables, comments, for loops, range clauses, if statements, functions, operators, slices, and maps. The following chapters will explain the fundamental concepts in more detail.

Chapter 3

Control Structures

Science is what we understand well enough to explain to a computer. Art is everything else we do. — Donald E. Knuth

Control structure determines how the code block will get executed for the given conditional expressions and parameters. Go provides a number of control structures including *for*, *if*, *switch*, *defer*, and *goto*. The Quick Start chapter has already introduced control structures like *if* and *for*. This chapter will elaborate more about these topics and introduce some other related topics.

3.1 If

Basic If

Programming involves lots of decision making based on the input parameters. The If control structure allows to perform a particular action on certain condition. A conditional expressions is what used for making decisions in the code using the If control structure. So, the If control structure will be always associated with a conditional expression which evaluates to a boolean value. If the boolean value is true, the statements within the block will be executed. Consider this example:

Listing 3.1 : If example program

```
package main
```

```
import "fmt"

func main() {
    if 1 < 2 {
        fmt.Println("1 is less than 2")
     }
}
```

The first line starts with the `if` keyword followed by a conditional expression and the line ends with an opening curly bracket. If the conditional expression is getting evaluated as true, the statements given within the curly brace will get evaluated. In the above example, the conditional expression `1 < 2` will be evaluated as true so the print statement given below that will get executed. In fact, you can add any number of statements within the braces.

If and Else

Sometimes you need to perform different set of action if the condition is not true. Go provides a variation of the `if` syntax with the `else` block for that.

Consider this example with else block:

Listing 3.2 : If with else block

```
package main

import "fmt"

func main() {
    if 3 > 4 {
        fmt.Println("3 is greater than 4")
    } else {
        fmt.Println("3 is not greater than 4")
    }
}
```

In the above code, the code will be evaluated as false. So, the statements within `else` block will get executed.

Example

Now we will go through a complete example, the task is to identify the given number is even or odd. The input can be given as a command line argument.

Listing 3.3 : If with else example

```go
package main

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

func main() {
	i := os.Args[1]
	n, err := strconv.Atoi(i)
	if err != nil {
		fmt.Println("Not a number:", i)
		os.Exit(1)
	}
	if n%2 == 0 {
		fmt.Printf("%d is even\n", n)
	} else {
		fmt.Printf("%d is odd\n", n)
	}
}
```

The `os` package has an attribute named `Args`. The value of `Args` will be a slice of strings which contains all command line arguments passed while running the program. As we have learned from the Quick Start chapter, the values can be accessed using the index syntax. The value at zero index will be the program name itself and the value at 1st index the first argument and the value at 2nd index the second argument and so on. Since we are expecting only one argument, you can access it using the 1st index (`os.Args[1]`).

The `strconv` package provides a function named `Atoi` to convert strings to integers. This function return two values, the first one is the integer value and the second one is the error value. If there is no error during convertion, the error value will be `nil`. If it's a non-nil value, that indicates there is an error occured during conversion.

The `nil` is an identifier in Go which represents the "zero value" for certain built-in types. The `nil` is used as the zero for these types: interfaces, functions, pointers, maps, slices, and channels.

In the above example, the second expected value is an object conforming to built-in `error` interface. We will discuss more about errors and interfaces in later chapters. The zero value for interface, that is `nil` is considered as there is no error.

The `Exit` function within `os` package helps to exit the program prematurely. The argument passed will be exit status code.

Normally exit code `0` is treated as success and non-zero value as error.

The conditional expression use the modulus operator to get remainder and checking it is zero. If the remainder against 2 is zero, the value will be even otherwise the value will odd.

Else If

There is a third alternative syntax available for the If control structure, that is **`else if`** block. The Else If block get executed if the conditional expression gives true value and previous conditions are false. It is possible to add any number of Else If blocks based on the requirements.

Look at this example where we have three choices based on the age group.

Listing 3.4 : if example program output

```go
package main

import "fmt"

func main() {
        age := 10
        if age < 10 {
        fmt.Println("Junior", age)
    } else if age < 20 {
        fmt.Println("Senior", age)
    } else {
        fmt.Println("Other", age)
    }
}
```

In the above example, the value printed will be either `Junior`, `Senior` or `Other`. You can change age value and run the program again and again to see the outputs. The Else If can be repeated here to create more choices.

Inline Statement

In the previous section, the variable *age* was only within the If, Else If and Else blocks. And that variable was not used used afterwards in the function. Go provides a syntax to define a variable along with the If where the scope of that variable will be within the blocks. In fact, the syntax can be used for any valid Go statement. However, it is mostly used for declaring variables.

Here is an example where a variable named **money** is declared along with the If control structure.

Listing 3.5 : If with initialization statement

```go
package main

import "fmt"

func main() {
    if money := 20000; money > 15000 {
        fmt.Println("I am going to buy a car.", money)
    } else {
        fmt.Println("I am going to buy a bike.", money)
    }
    // can't use the variable `money` here
}
```

As mentioned above, the variable declared by the inline statement is available only within the scope of If, Else If and Else blocks. So, the variable **money** cannot be used outside the blocks.

It is possible to make any valid Go statement as part of the If control structure. For example, it is possible to call a function like this:

Listing 3.6 : Variable initialization in If

```go
    if money := someFunction(); money > 15000 {
```

3.2 For

Basic For

As we have seen briefly in the Quick Start, the For control structure helps to create loops to repeat certain actions. The For control structure has few syntax variants.

Consider a program to print few names.

Listing 3.7 : For loop example (forbasic.go)

```go
package main

import "fmt"

func main() {
    names := []string{"Tom", "Polly", "Huck", "Becky"}
    for i := 0; i < len(names); i++  {
        fmt.Println(names[i])
    }
}
```

You can save the above program in a file named **`names.go`** and run it like this:

```
$ go run name.go
Tom
Polly
Huck
Becky
```

In the above example, **`names`** variable hold a slice of strings. The value of **`i`** is initialized to zero and incremented one by one. The **`i++`** statement increment the value of **`i`**. The second part of **`for`** loop check if value of **`i`** is less than length of the slice. The built-in **`len`** gives the length of slice.

Other programming languages offer many ways for iterations. Some of the examples are *while* and *do...while*. But in Go using syntactic variation of *for* loop meets all requirements. Functional languages prefer to use recursion instead of iteration.

Break Loop Prematurely

Sometimes the iteration should be stopped prematurely on certain condition. This can be achieved using the If control structure and break statement. We have already studied If control structure from the previous major section. The **`break`** keyword allows to create a break statement. The break statement end the loop immediately. Though any other code followed by For loop will be executed.

Let's alter the previous program to stop printing after the name **`Polly`** found.

Listing 3.8 : For loop with break

```
package main

import "fmt"

func main() {
    names := []string{"Tom", "Polly", "Huck", "Becky"}
    for i := 0; i < len(names); i++  {
        fmt.Println(names[i])
        if names[i] == "Polly" {
            break
        }
    }
}
```

In the above example, we added an If control structure to check for the value of name during each iteration. If the value matches `Polly`, break statement will be executed. The break statement makes the For loop to end immediately.

As you can see in the above code, the break statement can stand alone without any other input. There is alternate syntax with label similar to how *goto* works, which we are going to see below. This is useful when you have multiple loops and want to break a particular one, may be the outer loop.

To understand this better, let's consider an example. The problem is to to change print the name given the slice until a word with `u` found.

Listing 3.9 : For loop with break and label

```go
package main

import "fmt"

func main() {
    names := []string{"Tom", "Polly", "Huck", "Becky"}
Outer:
    for i := 0; i < len(names); i++ {
        for j := 0; j < len(names[i]); j++ {
            if names[i][j] == 'u' {
                break Outer
            }
        }
        fmt.Println(names[i])
    }
}
```

In the above example, we are declaring a label statement just before the first For loop. There is an inner loop to iterate through the name string and check for the presence of character `u`. If the character `u` is found, then it will break the outer loop. If the label `Outer` is not used in the break statement, then the inner loop will be stopped.

Partially Execute Loop Statements

Sometimes statements within For loop should be executed on certain iterations. Go has a **continue** statement to proceed loop without executing further statements.

Let's modify the previous problem to print all names except **Polly**.

Listing 3.10 : For loop with continue

```
package main

import "fmt"

func main() {
    names := []string{"Tom", "Polly", "Huck", "Becky"}
    for i := 0; i < len(names); i++ {
        if names[i] == "Polly" {
            continue
        }
        fmt.Println(names[i])
    }
}
```

In the above code, the **continue** statement makes it proceed with next iteration in the loop without printing **Polly**.

Similar to **break** statement with label, continue also can be used with a label. This is useful if there are multiple loops and want to continue a particular loop, say the outer one.

Let's consider an example where you need to print names which doesn't have character **u** in it.

Listing 3.11 : For loop with continue and label

```
package main

import "fmt"

func main() {
    names := []string{"Tom", "Polly", "Huck", "Becky"}
Outer:
    for i := 0; i < len(names); i++ {
        for j := 0; j < len(names[i]); j++ {
            if names[i][j] == 'u' {
                continue Outer
            }
        }
        fmt.Println(names[i])
    }
}
```

In the above code, just before the first loop a label is declared. Later inside the inner loop to iterate through the name string and check for the presence of character `u`. If the character `u` is found, then it will continue the outer loop. If the label `Outer` is not used in the continue statement, then the inner loop will be proceed to execute.

For with Outside Initialization

The statement for value initialization and the last pat to increment value can be removed from the For control structure. The value initialization can be moved outside For and value increment can be moved inside loop.

The previous example can be changed like this:

Listing 3.12 : For without initialization and increment

```
package main

import "fmt"

func main() {
    names := []string{"Tom", "Polly", "Huck", "Becky"}
    i := 0
    for i < len(names) {
        i++
        fmt.Println(names[i])
    }
}
```

In the above example, the scope of variable `i` is outside For loop code block. Whereas in the previous section, when the variable declared along with For loop, the scope of that variable was within the loop code block.

Infinite Loop

For loop has yet another syntax variant to support infinite loop. You can create a loop that never ends until explicitly stopped using break or exiting the whole program. To create an infinite loop, you can use the `for` keyword followed by the curly bracket.

If any variable initialization is required, that should be declared outside the loop. Conditions can be added inside the loop.

The previous example can be changed like this:

Listing 3.13 : Infinite For loop

```
package main

import "fmt"

func main() {
    names := []string{"Tom", "Polly", "Huck", "Becky"}
    i := 0
    for {
        if i >= len(names) {
            break
        }
        fmt.Println(names[i])
        i++
    }
}
```

Range Loops

The range clause form of the for loop iterates over a slice or map. When looping over a slice using range, two values are returned for each iteration. The first is the index, and the second is a copy of the element at that index.

The previous example `for` loop can be simplified using the `range` clause like this:

Listing 3.14 : Range loop with slice

```
package main

import "fmt"

func main() {
    characters := []string{"Tom", "Polly", "Huck", "Becky"}
    for _, j := range characters {
        fmt.Println(j)
    }
}
```

The underscore is called blank indentifier, the value assigned to that variable will be ignored. In the above example, the index values will be assigned to the underscore.

The range loop can be used with map. Here is an example:

Listing 3.15 : Range loop with map

```go
package main

import "fmt"

func main() {
    var characters = map[string]int{
                "Tom": 8,
                "Polly": 51,
                "Huck": 9,
                "Becky": 8,
    }
    for name, age := range characters {
        fmt.Println(name, age)
    }
}
```

3.3 Switch Cases

Basic Switch

In addition to the `if` condition, Go provides **`switch case`** control structure for branch instructions. The **`switch case`** is more convenient if many cases need to be handled in the branch instructions.

The below program use a switch case to print number names based on the value.

Listing 3.16 : Switch case example

```go
package main

import "fmt"

func main() {
    v := 1
    switch v {
    case 0:
            fmt.Println("zero")
    case 1:
            fmt.Println("one")
    case 2:
            fmt.Println("two")
    default:
            fmt.Println("unknown")
    }
}
```

In this case, the value of `v` is `1`, so the case that is going to execute is 2nd one. This will be the output.

```
$ go run switchbasic.go
one
```

If you change the value of `v` to `0`, it's going to print `zero` and for `2` it will print `two`. If the value is any number other than `0`, `1` or `2`, it's going to print `unknown`.

Fallthrough

The cases are evaluated top to bottom until a match is found. If a case is matched, the statements within that case will be executed. And no other case will be executed unless a `fallthrough` statement is used. The `fallthrough` must be the last statement within the case.

Here is a modified version with `fallthrough`

Listing 3.17 : Switch case with fallthrough

```go
package main

import "fmt"

func main() {
	v := 1
	switch v {
	case 0:
		fmt.Println("zero")
	case 1:
		fmt.Println("one")
		fallthrough
	case 2:
		fmt.Println("two")
	default:
		fmt.Println("unknown")
	}
}
```

If you run this program, it will print `one` followed by `two`.

```
$ go run switchbasic.go
one
two
```

Break

As you can see from the above examples, the switch statements break implicitly at the end of each cases. The **fallthrough** statement can be used to passdown control to the next case. However, sometimes execution should be stopped early without executing all statements. This can can be achieved using **break** statements.

Here is an example:

Listing 3.18 : Switch case with break

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    v := "Becky"
    t := time.Now()
    switch v {
    case "Huck":
        if t.Hour() < 12 {
            fmt.Println("Good morning,", v)
            break
        }
        fmt.Println("Hello,", v)
    case "Becky":
        if t.Hour() < 12 {
            fmt.Println("Good morning,", v)
            break
        }
        fmt.Println("Hello,", v)
    default:
        fmt.Println("Hello")
    }
}
```

In the above example, morning time greeting is different.

Multiple Cases

Multple cases can be presented in comma-separated lists.

Here is an example.

Listing 3.19 : Switch with multiple cases

```go
package main

import "fmt"

func main() {
	o := "=="
	switch o {
	case "+", "-", "*", "/", "%", "&", "|", "^", "&^", "<<", ">>":
		fmt.Println("Arithmetic operator")
	case "==", "!=", "<", "<=", ">", ">=":
		fmt.Println("Comparison operators")
	case "&&", "||", "!":
		fmt.Println("Logical operators")
	default:
		fmt.Println("Unknown operator")
	}
}
```

In this example, if any of the value is matched in the given list, that case will be executed.

Without Expression

If the switch has no expression it switches on true. This is useful to write an if-else-if-else chain.

Let's take the example program used earlier when Else If was introduced:

Listing 3.20 : Switch without expression

```go
package main

import "fmt"

func main() {
	age := 10
	switch {
	case age < 10:
		fmt.Println("Junior", age)
	case age < 20:
		fmt.Println("Senior", age)
	default:
		fmt.Println("Other", age)
	}
}
```

3.4 Defer Statements

Sometimes it will require to force certain things to do before a function returns. For example, closing an opened file descriptor. Go provides the *defer* statements to do these kind of cleanup actions.

A defer statement add a function call into a stack. The stack of function call executes at the end of the surrounding function in a last-in-first-out (LIFO) order. Defer is commonly used to perform various clean-up actions.

Here is a simple example:

Listing 3.21 : Defer usage

```
package main

import (
    "fmt"
    "time"
)

func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
}
```

The above program is going to print `hello` followed by `world`.

If there are multiple *defer* statements, it will execute in last-in-first-out (LIFO) order.

Here is a simple example to demonstrate it:

Listing 3.22 : Defer usage

```
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        defer fmt.Println(i)
    }
}
```

The above program will print this output:

```
4
3
```

```
2
1
0
```

The arguments passed the the deferred call are evaluated immediately. But the deferred call itself is not executed until the function returns. Here is a simple example to demonstrate it:

Listing 3.23 : Defer argument evaluation

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    defer func(t time.Time) {
        fmt.Println(t, time.Now())
    }(time.Now())
}
```

When you run the above program, you can see a small difference in time. The *defer* can also be used to recover from *panic*, which will be discussed in the next section.

3.5 Deffered Panic Recover

We have discussed the commonly used control structures including if, for, and switch. This section is going to discuss a less commonly used set of control structures: *defer*, *panic*, and *recover*. We have discussed the use of the defer statement in the previous section. In this section, you are going to learn how to use the *defer* along with *panic* and *recover*.

Few important points about defer, panic, and recover:

- A panic causes the program stack to begin unwinding and recover can stop it
- Deferred functions are still executed as the stack unwinds
- If recover is called inside such a deferred function, the stack stops unwinding
- The recover returns the value (as an *interface{}*) that was passed to panic

- A panic cannot be recovered by a different goroutine

Here is an example:

Listing 3.24 : deferred panic recover

```
package main

import "fmt"

func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recoverd", r)
        }
    }()
    panic("panic")
}
```

3.6 Goto

The *goto* statement can be used to jump control to another statement. The location to where the control should be passed is specified using label statements. The *goto* statement and the corresponding label should be within the same function. The *goto* cannot jump to a label inside another code block.

Listing 3.25 : Goto example program (goto.go)

```
package main

import "fmt"

func main() {
    num := 10
    goto Marker
    num = 20
Marker:
    fmt.Println("Value of num:", num)
}
```

You can save the above program in a file named **goto1.go** and run it like this:

Listing 3.26 : Goto example program output

```
$ go run goto1.go
Value of num: 10
```

In the above code `Marker:` is a label statement. A label statement is a valid identifier followed by a colon. A label statement will be target for *goto*, *break* or *continue* statement. We will look at *break* and *continue* statement when we study the For control structure.

The *goto* statement is writen using the `goto` keyword followed by a valid label name. In the above code, immediately after the *goto* statement, there is a statement to assign a different value to `num`. But that statement is never getting executed as the *goto* makes the program to jump to the label.

3.7 Exercises

Exercise 1: Print whether the number given as the command line argument is even or odd.

Solution:

You can store the program with a file named `evenodd.go`. Later you can compile this program and then you will get a binary executable with name as `evenodd`. You can execute this program like this:

```
./evenodd 3
3 is odd
./evenodd 4
4 is even
```

In the above program, the 3 and 4 are the command line arguments. The command line arguments can be accessed from Go using the slice available under `os` package. The arguments will be available with exported name as `Args` and individual items can be accessed using the index. The 0th index contains the program itself, so it can be ignored. To access the 1st command argument use `os.Args[1]`. The values will be of type string which can be converted later.

```
package main

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

func main() {
    i := os.Args[1]
    n, err := strconv.Atoi(i)
    if err != nil {
        fmt.Println("Not a number:", i)
        os.Exit(1)
    }
    if n%2 == 0 {
        fmt.Printf("%d is even\n", n)
    } else {
        fmt.Printf("%d is odd\n", n)
    }
}
```

Exercise 2: Write a program to print numbers below 20 which are multiples of 3 or 5.

Solution:

```
package main

import "fmt"

func main() {
    for i := 1; i < 20; i++ {
        if i%3 == 0 || i%5 == 0 {
            fmt.Println(i)
        }
    }
}
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a program to print greetings based on time. Possible greetings are Good morning, Good afternoon and Good evening.

Problem 2: Write a program to check if the given number is divisible by 2, 3, or 5.

Summary

This chapter explained the control structures available in Go, except those related to concurrency. The *if* control structure was covered first, then the *for* loop was explained. The *switch* cases were discussed later. Then the *defer* statement and finally, the *goto* control structure was explained in detail. This chapter also briefly explained accessing command line arguments from the program.

Control structures are used to control the flow of execution in a program. They allow you to execute code conditionally, repeatedly, or in a specific order.

- The *if* control structure is used to execute code if a condition is met.
- The *for* loop is used to execute code repeatedly until a condition is met.
- The *switch* statement is used to execute code based on the value of a variable.
- The *defer* statement is used to execute code after the current function has finished executing.
- The *goto* statement is used to jump to a specific label in the code.

Chapter 4

Data Structures

> *Bad programmers worry about the code. Good programmers worry about data structures and their relationships.* — Linus Torvalds

In the last last few chapters we have seen some of the primitive data data types. We also introduced few other advanced data types without going to the details. In this chapter, we are going to look into more data structures.

4.1 Primitive Data Types

Advanced data structures are built on top of primitive data types. This section is going to cover the primitive data types in Go.

Zero Value

In the Quick Start chapter, you have learned various ways to declare a variable. When you declare a variable using the `var` statement without assigning a value, a default Zero value will be assigned for certain types. The Zero value is 0 for integers and floats, empty string for strings, and false for boolean. To demonstrate this, here is an example:

Listing 4.1 : Zero values

```
package main
```

```
import "fmt"

func main() {
    var name string
    var age int
    var tall bool
    var weight float64
    fmt.Printf("%#v, %#v, %#v, %#v\n", name, age, tall, weight)
}
```

This is the output:

```
"", 0, false, 0
```

Variable

In the quick start chapter, we have discussed about variables and its usage. The variable declared outside the function (package level) can access anywhere within the same package.

Here is an example:

Listing 4.2 : Package level variable

```
package main

import (
    "fmt"
)

var name string
var country string = "India"

func main() {
    name = "Jack"
    fmt.Println("Name:", name)
    fmt.Println("Country:", country)
}
```

In the above example, the `name` and `country` are two package level variables. As we have seen above the `name` gets zero value, where as value for `country` variable is explicitly initialized.

If the variable has been defined using the `:=` syntax, and the user wants to change the value of that variable, they need to use `=` instead of `:=` syntax.

If you run the below program, it's going to throw an error:

Listing 4.3 : Changing value with wrong syntax

```go
package main

import (
    "fmt"
)

func main() {
    age := 25
    age := 35
    fmt.Println(age)
}
```

```
$ go run update.go
# command-line-arguments
./update.go:9:6: no new variables on left side of :=
```

The above can be fixed like this:

Listing 4.4 : Changing value with wrong syntax

```go
package main

import (
    "fmt"
)

func main() {
    age := 25
    age = 35
    fmt.Println(age)
}
```

Now you should see the output:

```
$ go run update.go
35
```

Using the *reflect* package, you can identify the type of a variable:

Listing 4.5 : Identifying type of a variable

```go
package main

import (
        "fmt"
        "reflect"
)

func main() {
    var pi = 3.41
    fmt.Println("type:", reflect.TypeOf(pi))
}
```

Using one or two letter variable names inside a function is common practice. If the variable name is multi-word, use lower camelCase (initial letter lower and subsequent words capitalized) for unexported variables. If the variable is an exported one, use upper CamelCase (all the words capitalized). If the variable name contains any abbreviations like ID, use capital letters. Here are few examples: pi, w, r, ErrorCode, nodeToDaemonPods, DB, InstanceID.

Unused variables and imports

If you declare a variable inside a function, use that variable somewhere in the same function where it is declared. Otherwise, you are going to get a compile error. Whereas a global variable declared but unused is not going to throw compile time error.

Any package that is getting imported should find a place to use. Unused import also throws compile time error.

Boolean Type

A boolean type represents a pair of truth values. The truth values are denoted by the constants *true* and *false*. These are the three logical operators that can be used with boolean values:

- `&&` – Logical AND
- `||` – Logical OR
- `!` – Logical NOT

Here is an example:

Listing 4.6 : Logical operators

```go
package main

import "fmt"

func main() {
    yes := true
    no := false
    fmt.Println(yes && no)
    fmt.Println(yes || no)
    fmt.Println(!yes)
    fmt.Println(!no)
}
```

The output of the above logical operators are like this:

```
$ go run logical.go
false
true
false
true
```

Numeric Types

The numeric type includes both integer types and floating-point types. The allowed values of numeric types are same across all the CPU architectures.

These are the unsigned integers:

- uint8 – the set of all unsigned 8-bit integers (0 to 255)
- uint16 – the set of all unsigned 16-bit integers (0 to 65535)
- uint32 – the set of all unsigned 32-bit integers (0 to 4294967295)
- uint64 – the set of all unsigned 64-bit integers (0 to 18446744073709551615)

These are the signed integers:

- int8 – the set of all signed 8-bit integers (-128 to 127)
- int16 – the set of all signed 16-bit integers (-32768 to 32767)
- int32 – the set of all signed 32-bit integers (-2147483648 to 2147483647)
- int64 – the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

These are the two floating-point numbers:

- float32 – the set of all IEEE-754 32-bit floating-point numbers

- float64 – the set of all IEEE-754 64-bit floating-point numbers

These are the two complex numbers:

- complex64 – the set of all complex numbers with float32 real and imaginary parts
- complex128 – the set of all complex numbers with float64 real and imaginary parts

These are the two commonly used used aliases:

- byte – alias for uint8
- rune – alias for int32

String Type

A string type is another most import primitive data type. String type represents string values.

4.2 Constants

A constant is an unchanging value. Constants are declared like variables, but with the *const* keyword. Constants can be character, string, boolean, or numeric values. Constants cannot be declared using the `:=` syntax.

In Go, *const* is a keyword introducing a name for a scalar value such as 2 or 3.14159 or ”scrumptious”. Such values, named or otherwise, are called constants in Go. Constants can also be created by expressions built from constants, such as 2+3 or 2+3i or math.Pi/2 or (”go”+”pher”). Constants can be declared are at package level or function level.

This is how to declare constants:

```
package main

import (
    "fmt"
)

const Freezing = true
const Pi = 3.14
const Name = "Tom"

func main() {
    fmt.Println(Pi, Freezing, Name)
}
```

You can also use the factored style declaration:

```
package main

import (
    "fmt"
)

const (
      Freezing = true
      Pi = 3.14
      Name = "Tom"
)

func main() {
    fmt.Println(Pi, Freezing, Name)
}

const (
      Freezing = true
      Pi = 3.14
      Name = "Tom"
)
```

Compiler throws an error if the constant is tried to assign a new value:

```
package main

import (
    "fmt"
)

func main() {
    const Pi = 3.14
    Pi = 6.86
    fmt.Println(Pi)
}
```

The above program throws an error like this:

```
$ go run constants.go
constants:9:5: cannot assign to Pi
```

iota

The *iota* keyword is used to define constants of incrementing numbers. This simplify defining many constants. The values of iota is reset to *0* whenever the reserved word const appears. The value increments by one after each line.

Consider this example:

```
// Token represents a lexical token.
type Token int

const (
    // Illegal represents an illegal/invalid character
    Illegal Token = iota

    // Whitespace represents a white space
    // (" ", \t, \r, \n) character
    Whitespace

    // EOF represents end of file
    EOF

    // MarkerID represents '\id' or '\id1' marker
    MarkerID

    // MarkerIde represents '\ide' marker
    MarkerIde
)
```

In the above example, the `Token` is custom type defined using the primitive *int* type. The constants are defined using the factored syntax (many constants within parenthesis). There are comments for each constant values. Each constant value is be incremented starting from `0`. In the above example, `Illegal` is `0`, Whitespace is `1`, `EOF` is `2` and so on.

The `iota` can be used with expressions. The expression will be repeated. Here is a good example taken from *Effective Go* (`https://go.dev/doc/effective_go#constants`):

```
type ByteSize float64

const (
    // ignore first value (0) by assigning to blank identifier
    _           = iota
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

Using `_` (blank identifier) you can ignore a value, but *iota* increments the value. This can be used to skip certain values. As you can see in the above example, you can use an expression with *iota*.

Iota is reset to `0` whenever the *const* keyword appears in the source code. This means that if you have multiple *const* declarations in a single file, *iota* will start at `0` for each declaration. Iota can only be used in *const* declarations. It cannot be used in other types of declarations, such as *var* declarations. The value of *iota* is only available within the const declaration in which it is used. It cannot be used outside of that declaration.

Blank Identifier

Sometimes you may need to ignore the value returned by a function. Go provides a special identifier called blank identifier to ignore any types of values. In Go, underscore `_` is the blank identifier.

Here is an example usage of blank identifier where the second value returned by the function is discarded.

```
x, _ := someFunc()
```

Blank identifier can be used as import alias to invoke init function without using the package.

```
import (
	"database/sql"

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

In the above example, the `pq` package has some code which need to be invoked to initialize the database driver provided by that package. And the exported functions within the above package is supposed to be not used.

We have already seen another example where blank identifier if used with *iota* to ignore certain constants.

4.3 Arrays

An array is an ordered container type with a fixed number of data. In fact, the arrays are the foundation where slice is built. We will study about slices in the next section. Most of the time, you can use slice instead of an array.

The number of values in the array is called the length of that array. The array type `[n]T` is an array of `n` values of type `T`. Here are two example arrays:

```
colors := [3]string{"Red", "Green", "Blue"}
heights := [4]int{153, 146, 167, 170}
```

In the above example, the length of first array is `3` and the array values are string data. The second array contains `int` values. An array's length is part of its type, so arrays cannot be re-sized. So, if the length is different for two arrays, those are distinct incompatible types. The built-in `len` function gives the length of array.

Array values can be accessed using the index syntax, so the expression `s[n]` accesses the nth element, starting from zero.

An array values can be read like this:

```
colors := [3]string{"Red", "Green", "Blue"}
i := colors[1]
fmt.Println(i)
```

Similarly array values can be set using index syntax. Here is an example:

```
colors := [3]string{"Red", "Green", "Blue"}
colors[1] = "Yellow"
```

Arrays need not be initialized explicitly. The zero value of an array is a usable array with all elements zeroed.

```
var colors [3]string
colors[1] = "Yellow"
```

In this example, the values of colors will be empty strings (zero value). Later we can assign values using the index syntax.

There is a way to declare array literal without specifying the length. When using this syntax variant, the compiler will count and set the array length.

```
colors := [...]string{"Red", "Green", "Blue"}
```

In the chapter on control structures, we have seen how to use For loop for iterating over slices. In the same way, you can iterate over array.

Consider this complete example:

Listing 4.7 : Array example

```
 1 package main

 3 import "fmt"

 5 func main() {
 6     colors := [3]string{"Red", "Green", "Blue"}
 7     fmt.Println("Length:", len(colors))
 8     for i, v := range colors {
 9         fmt.Println(i, v)
10     }
11 }
```

If you save the above program in a file named **`colors.go`** and run it, you will get output like this:

```
$ go run colors.go
Length: 3
0 Red
1 Green
2 Blue
```

In the above program, a string array is declared and initialized with three string values. In the 7th line, the length is printed and it gives 3. The **`range`** clause gives index and value, where the index starts from zero.

4.4 Slices

Slice is one of most important data structure in Go. Slice is more flexible than an array. It is possible to add and remove values from a slice. There will be a length for slice at any time. Though the length vary dynamically as the content value increase or decrease.

The number of values in the slice is called the length of that slice. The slice type `[]T` is a slice of type `T`. Here are two example slices:

```
colors := []string{"Red", "Green", "Blue"}
heights := []int{153, 146, 167, 170}
```

The first one is a slice of strings and the second slice is a slice of integers. The syntax is similar to array except the length of slice is not explicitly specified. You can use built-in `len` function to see the length of slice.

Slice values can be accessed using the index syntax, so the expression `s[n]` accesses the nth element, starting from zero.

A slice values can be read like this:

```
colors := []string{"Red", "Green", "Blue"}
i := colors[1]
fmt.Println(i)
```

Similary slice values can be set using index syntax. Here is an example:

```
colors := []string{"Red", "Green", "Blue"}
colors[1] = "Yellow"
```

Slices should be initialized with a length more than zero to access or set values. In the above examples, we used slice literal syntax for that. If you define a slice using `var` statement without providing default values, the slice will be have a special zero value called `nil`.

Consider this complete example:

Listing 4.8 : Nil slice example

```
package main

import "fmt"

func main() {
    var v []string
    fmt.Printf("%#v, %#v\n", v, v == nil)
    // Output: []string(nil), true
}
```

In the above example, the value of slice `v` is `nil`. Since the slice is nil, values cannot be accessed or set using the index. These operations are going to raise runtime error (index out of range).

Sometimes it may not be possible to initialize a slice with some value using the literal slice syntax given above. Go provides a built-in function named `make` to initialize a slice with a given length and zero values for all items. For example, if you want a slice with 3 items, the syntax is like this:

```
colors := make([]string, 3)
```

In the above example, a slice will be initialized with 3 empty strings as the items. Now it is possible to set and get values using the index as given below:

```
colors[0] = "Red"
colors[1] = "Green"
colors[2] = "Blue"
i := colors[1]
fmt.Println(i)
```

If you try to set value at 3rd index (`colors[3]`), it's going to raise runtime error with a message like this: "index out of range". Go has a built-in function named **append** to add additional values. The append function will increase the length of the slice.

Consider this example:

Listing 4.9 : Append to slice

```go
package main

import (
    "fmt"
)

func main() {
    v := make([]string, 3)
    fmt.Printf("%v\n", len(v))
    v = append(v, "Yellow")
    fmt.Printf("%v\n", len(v))
}
```

In the above example, the slice length is increased by one after append. It is possible to add more values using **append**. See this example:

Listing 4.10 : Append more values to slice

```go
package main

import (
    "fmt"
)

func main() {
    v := make([]string, 3)
    fmt.Printf("%v\n", len(v))
    v = append(v, "Yellow", "Black")
    fmt.Printf("%v\n", len(v))
}
```

The above example append two values. Though you can provide any number of values to append.

You can use the "..." operator to expand a slice. This can be used to append one slice to another slice. See this example:

Listing 4.11 : Append a slice to another

```go
package main

import (
	"fmt"
)

func main() {
	v := make([]string, 3)
	fmt.Printf("%v\n", len(v))
	a := []string{"Yellow", "Black"}
	v = append(v, a...)
	fmt.Printf("%v\n", len(v))
}
```

In the above example, the first slice is appended by all items in another slice.

Slice Append Optimization

If you append too many values to a slice using a for loop, there is one optimization related that you need to be aware.

Consider this example:

Listing 4.12 : Append to a slice inside a loop

```go
package main

import (
	"fmt"
)

func main() {
	v := make([]string, 0)
	for i := 0; i < 9000000; i++ {
		v = append(v, "Yellow")
	}
	fmt.Printf("%v\n", len(v))
}
```

If you run the above program, it's going to take few seconds to execute. To explain this, some understanding of internal structure of slice is required. Slice is implemented as a struct and an array within. The elements in the slice will be stored in the underlying array. As you know, the length of array is part of the array type. So, when appending an item to a slice the a new array will be created.

To optimize, the `append` function actually created an array with double length.

In the above example, the underlying array must be changed many times. This is the reason why it's taking few seconds to execute. The length of underlying array is called the capacity of the slice. Go provides a way to initialize the underlying array with a particular length. The `make` function has a fourth argument to specify the capacity.

In the above example, you can specify the capacity like this:

```
v := make([]string, 0, 9000000)
```

If you make this change and run the program again, you can see that it run much faster than the earlier code. The reason for faster code is that the slice capacity had already set with maximum required length.

4.5 Maps

Map is another important data structure in Go. We have briefly discussed about maps in the Quick Start chapter. As you know, map is an implementation of hash table. The hash table is available in many very high level languages. The data in map is organized like key value pairs.

A variable of map can be declared like this:

```
var fruits map[string]int
```

To make use that variable, it needs to be initialized using *make* function.

```
fruits = make(map[string]int)
```

You can also initialize using the *:=* syntax:

```
fruits := map[string]int{}
```

or with *var* keywod:

```
var fruits = map[string]int{}
```

You can initialize map with values like this:

```
var fruits = map[string]int{
      "Apple":  45,
      "Mango":  24,
      "Orange": 34,
  }
```

After initializing, you can add new key value pairs like this:

```
fruits["Grape"] = 15
```

If you try to add values to maps without initializing, you will get an error like this:

```
panic: assignment to entry in nil map
```

Here is an example that's going to produce panic error:

```
package main

func main() {
    var m map[string]int
    m["k"] = 7
}
```

To access a value corresponding to a key, you can use this syntax:

```
mangoCount := fruits["Mango"]
```

Here is an example:

```
package main

import "fmt"

func main() {
  var fruits = map[string]int{
      "Apple":  45,
      "Mango":  24,
      "Orange": 34,
  }
  fmt.Println(fruits["Apple"])
}
```

If the key doesn't exist, a zero value will be returned. For example, in the below example, value of `pineappleCount` is going be `0`.

```
package main

import "fmt"

func main() {
  var fruits = map[string]int{
      "Apple":  45,
      "Mango":  24,
      "Orange": 34,
  }
  pineappleCount := fruits["Pineapple"]
  fmt.Println(pineappleCount)
}
```

If you need to check if the key exist, the above syntax can be modified to return two values. The first one would be the actual value or zero value and the second one would be a boolean indicating if the key exists.

```
package main

import "fmt"

func main() {
  var fruits = map[string]int{
      "Apple":  45,
      "Mango":  24,
      "Orange": 34,
  }
  _, ok := fruits["Pineapple"]
  fmt.Println(ok)
}
```

In the above program, the first returned value is ignored using a blank identifier. And the value of `ok` variable is `false`. Normally, you can use if condition to check if the key exists like this:

```
package main

import "fmt"

func main() {
    var fruits = map[string]int{
        "Apple":  45,
        "Mango":  24,
        "Orange": 34,
    }
    if _, ok := fruits["Pineapple"]; ok {
        fmt.Println("Key exists.")
    } else {
        fmt.Println("Key doesn't exist.")
    }
}
```

To see the number of key/value pairs, you can use the built-in *len* function.

```
package main

import "fmt"

func main() {
    var fruits = map[string]int{
        "Apple":  45,
        "Mango":  24,
        "Orange": 34,
    }
    fmt.Println(len(fruits))
}
```

The above program should print `3` as the number of items in the map.

To remove an item from the map, use the built-in *delete* function.

```
package main

import "fmt"

func main() {
    var fruits = map[string]int{
        "Apple":  45,
        "Mango":  24,
        "Orange": 34,
    }
    delete(fruits, "Orange")
    fmt.Println(len(fruits))
}
```

The above program should print `2` as the number of items in the map after deleting one item.

4.6 Custom Data Types

Apart from the built-in data types, you can create your own custom data types. The type keyword can be used to create custom types. Here is an example.

```
package main

import "fmt"

type age int

func main() {
     a := age(2)
     fmt.Println(a)
     fmt.Printf("Type: %T\n", a)
}
```

If you run the above program, the output will be like this:

```
$ go run age.go
2
Type: age
```

Structs

Struct is a composite type with multiple fields of different types within the struct. For example, if you want to represent a person with name and age, the `struct` type will be helpful. The `Person` struct definition will look like this:

```
type Person struct {
    Name string
    Age  int
}
```

As you can see above, the `Person` struct is defined using `type` and `struct` keywords. Within the curly brace, attributes with other types are defined. If you avoid attributes, it will become an empty struct.

Here is an example empty struct:

```
type Empty struct {
}
```

Alternatively, the curly brace can be in the same line.

```
type Empty struct {}
```

A struct can be initialized various ways. Using a `var` statement:

```
var p1 Person
p1.Name = "Tom"
p1.Age = 10
```

You can give a literal form with all attribute values:

```
p2 := Person{"Polly", 50}
```

You can also use named attributes. In the case of named attributes, if you miss any values, the default zero value will be initialized.

```
p3 := Person{Name: "Huck"}
p4 := Person{Age: 10}
```

In the next, we are going to learn about funtions and methods. That chapter expands the discussion about custom types bevior changes through funtions associated with custom types called strcuts.

It is possible to embed structs inside other structs. Here is an example:

```
type Person struct {
     Name string
}

type Member struct {
     Person
     ID int
}
```

4.7 Pointers

When you are passing a variable as an argument to a function, Go creates a copy of the value and send it. In some situations, creating a copy will be expensive when the size of object is large. Another scenario where pass by value is not feasible is when you need to modify the original object inside the function. In the case of pass by value, you can modify it as you are getting a new object every time. Go supports another way to pass a reference to the original value using the memory location or address of the object.

To get address of a variable, you can use `&` as a prefix for the variable. Here in an example usage:

```
a := 7
fmt.Printf("%v\n", &a)
```

To get the value back from the address, you can use `*` as a prefix for the variable. Here in an example usage:

```
a := 7
b := &a
fmt.Printf("%v\n", *b)
```

Here is a complete example:

Listing 4.13 : Pass by value vs reference

```
package main

import (
    "fmt"
)

func value(a int) {
    fmt.Printf("%v\n", &a)
}

func pointer(a *int) {
    fmt.Printf("%v\n", a)
}

func main() {
    a := 4
    fmt.Printf("%v\n", &a)
    value(a)
    pointer(&a)
}
```

A typical output will be like this:

```
0xc42000a340
0xc42000a348
0xc42000a340
```

As you can see above, the second output is different from the first and third. This is because a value is passed instead of a pointer. And so when we are printing the address, it's printing the address of the new variable.

In the functions chapter, the section about methods (section 5.6) explains the pointer receiver.

new

The built-in function *new* can be used to allocate memory. It allocates zero values and returns the address of the given data type.

Here is an example:

```
name := new(string)
```

In this example, a *string* pointer value is allocated with zero value, in this case empty string, and assigned to a variable.

This above example is same as this:

```
var name *string
name = new(string)
```

In this one *string* pointer variable is declared, but it's not allocated with zeror value. It will have *nil* value and so it cannot be dereferenced. If you try to reference, without allocating using the *new* function, you will get an error like this:

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

Here is another example using a custom type defined using a primitive type:

```
type Temperature float64
name := new(Temperature)
```

4.8 Exercises

Exercise 1: Create a custom type for circle using float64 and define `Area` and `Perimeter`.

Solution:

```
package main

import "fmt"

type Circle float64

func (r Circle) Area() float64 {
    return float64(3.14 * r * r)
}

func (r Circle) Perimeter() float64 {
    return float64(2 * 3.14 * r)
}

func main() {
    c := Circle(4.0)
    fmt.Println(c.Area())
    fmt.Println(c.Perimeter())
}
```

The custom `Circle` type is created using the built-in `float64` type. It would be better if the circle is defined using a struct. Using struct helps to change the structure later with additional attributes. The struct will look like this:

```
type Circle struct {
    Radius float64
}
```

Exercise 2: Create a slice of structs where the struct represents a person with name and age.

Solution:

```
package main

import "fmt"
```

```
type Person struct {
    Name string
    Age int
}

func main() {
    persons := []Person{
        Person{Name: "Huck", Age: 11},
        Person{Name: "Tom", Age: 10},
        Person{Name: "Polly", Age: 52},
    }
    fmt.Println(persons)
}
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a program to record temperatures for different locations and check if it's freezing for a given place.

Problem 2: Create a map of world nations and details. The key could be the country name and value could be an object with details including capital, currency, and population.

Summary

This chapter introduced various data structures in Go. These data structures help organize data in a way that makes it easier to use and understand. The chapter started with a section about zero values, which are values that are assigned to variables when they are declared but not initialized. Then, constants were explained in detail, including the *iota* keyword, which can be used to define incrementing constants. Next, the chapter briefly explained arrays, which are data structures that can store a fixed number of elements of the same type. Then, slices were explained, which are more flexible data structures that can store a variable number of elements of the same type. Slices are built on top of arrays, and they inherit many of the same properties. Next, the chapter looked at how to define custom data types using existing primitive types. Custom data types can be used to group together related data and make it easier to work with. The struct was introduced as a way to define custom data types. Structs can be used to store a collection of related data items, such as the name, age, and address of a person. Pointers were also covered. Pointers are variables that store the address of another variable. Pointers can be used to access the value of another variable indirectly. The next chapter will explain more about functions and methods.

Chapter 5

Functions

Either mathematics is too big for the human mind, or the human mind is more than a machine. — Kurt Gödel

In this chapter, we are going to learn more about functions. You have already seen functions in previous chapters. Functions provide a way to group statements together and call it multiple times. If you don't use functions, the same statements will be written again and again in different parts of your code. Ultimately functions helps you to reuse code. A function can optionally accept arguments and execute statements and optionally return values.

Mathematical function would be a good analogy to understand the concept of functions in programming. We have seen this mathematical function in the Quick Start chapter.

$$f(r) = 3.14r^2$$

Figure 5.1: Mathematical function for area of a circle

This function square the input value and multiply with 3.14. Depending on the input value the output varies.

As you can see in the diagram, `r` is the input and `y` is the output. A function in Go can take input arguments and perform actions and return values. A minimal implementation of this function in Go looks like this.

```
func Area(r float64) float64 {
    return 3.14 * r * r
}
```

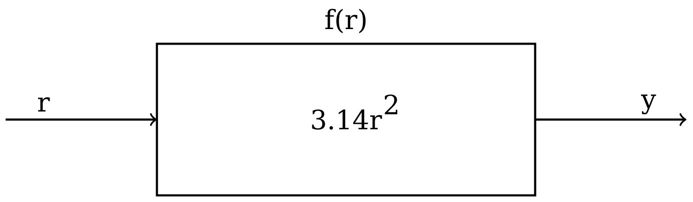

Figure 5.2: Blackbox representation of a function

The function declaration starts with **`func`** keyword. In the above example, **`Area`** is the function name which can be later used to call the function. The arguments that can be received by this function is given within brackets. After the input parameters you can specify the output parameters. If there are more than one output parameter required, use a bracket around that. After the output parameters, add one opening curly bracket. The statements can be written in the next line on wards until the closing curly bracket. It is recommended to start a new line after the opening curly bracket and closing bracket can be in a line by its own.

Here is a complete example with usage of the Area function.

Listing 5.1 : Function to calculate area of circle (area.go)

```
 1 package main

 3 import "fmt"

 5 func Area(r float64) float64 {
 6     return 3.14 * r * r
 7 }

 9 func main() {
10     area := Area(5.0)
11     fmt.Println(area)
12 }
```

In the above example, the **`Area`** function is called with argument value as **`5.0`** (line number 10). And the short variable declaration syntax is used to assign value returned by the function. The type of the variable **`area`** will be **`float64`** as the **`Area`** function returns with that type.

If you run the above program, you will get the output like this:

```
$ go run area.go
78.5
```

5.1 Parameters

A function can accept any number of arguments depending on the parameters defined. The **`Area`** function in the previous section accepts one argument.

If the function definition doesn't have any parameters, you cannot pass any arguments. Parameter is the variable used in the declaration of a function. Where as argument is the actual value of this variable that gets passed to function. Consider this example:

Listing 5.2 : Function without any parameters

```
package main

import (
	"fmt"
	"time"
)

func TimeNow() string {
	t := time.Now()
	h := t.Hour()
	m := t.Minute()
	return fmt.Sprintf("%d:%d", h, m)
}

func main() {
	now := TimeNow()
	fmt.Println(now)
}
```

The **`TimeNow`** function doesn't declare any parameters. So, when the function is called, no arguments are passed. If you try to pass any arguments, you will get an error with this message: **`too many arguments in call to TimeNow`**.

More parameters

A function can accepts more arguments of same or different types.

Listing 5.3 : Function with two parameters

```
package main

import "fmt"

func sum(a int, b int) int {
    return a + b
}

func main() {
    s := sum(5, 2)
    fmt.Println(s)
}
```

The above `sum` function accepts two integer parameters. Since both parameters are integers, the type can be specified once.

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

5.2 Return Values

A function can return any number of values. The calling side should have comma separated variables to receive the return values. If you are only interested in a particular return value, you can use underscore as the variable name for others.

Here is an example function which return two values:

Listing 5.4 : Function with two return values

```
package main

import "fmt"

func div(a, b int) (int, int) {
    return a / b, a % b
}

func main() {
    v, r := div(5, 2)
    fmt.Println(v, r)
}
```

In the above example, the div function return two values. So two variables are used to assign the values. If you use one variable it will produce compile time error. The compile time error will be produced, if more than two variables are used to assigned. However, it is possible to call the function without assigning to any variables.

```
v, _ := div(5, 2)
div(5, 2)
```

By convention, the last return value will be an error value. Here is a modified example.

```
func div(a, b int) (int, int, error) {
    if b == 0 {
        err := errors.New("Zero division error")
        return 0, 0, err
    }
    return a / b, a % b, nil
}
```

In the above example, package `errors` is used to create a new error value. If there is no error, a `nil` value can be returned.

Named output parameters

It is possible to specify name for output parameters. These variables can be used to assign values. With named output parameters, return statement need not to explicitly specify the variables.

Listing 5.5 : Function with named output parameters

```
package main

import "fmt"

func div(a, b int) (int d, int r) {
    d := a / b
    r := a % b
    return
}

func main() {
    v, r := div(5, 2)
    fmt.Println(v, r)
}
```

5.3 Variadic Functions

A function which can receive any number of arguments of a particular type is called variadic function. Variable name along with an ellipsis (`...`) symbol is used to declare variadic parameters. The `fmt.Println` is a commonly used variadic function.

Here is a complete example:

Listing 5.6 : Variadic Function (variadic.go)

```go
package main

import "fmt"

func sum(nums ...int) {
    fmt.Printf("%#v ", nums)
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {
    sum(1, 2)
    sum(1, 2, 3)
    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

If you run the above program, this will be the output:

```
$ go run variadic.go
[]int{1, 2} Sum: 3
[]int{1, 2, 3} Sum: 6
[]int{1, 2, 3, 4} Sum: 10
```

As you can see the arguments are captured into a slice. You can send values in a slice to a variadic function using the ellipsis syntax as a suffix.

5.4 Anonymous Functions

It is possible to declare a function without a name. These type of functions can be used to create function closures. A closure is an anonymous function that access variables from outside its body.

Listing 5.7 : Anonymous Function (anonfunc.go)

```go
package main

import "fmt"

func main() {
     name := "Tom"
     func() {
```

```
		fmt.Println("Hello", name)
	}()
}
```

5.5 Function as Value

Function is a first class citizen in Go, so it can be passed as an argument and return as a value.

Listing 5.8 : Function as value (funcvalue.go)

```
package main

import "fmt"

func Greeting(msg string) func(name string) string {
}

func main() {
	name := "Tom"
	func() {
		fmt.Println("Hello", name)
	}()
}
```

5.6 Methods

A function can be associated with a type, that is called method. Additional methods can be added to types defined locally. However, adding additional methods for non-local type is not allowed. Here is an example program:

```
package main

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

type Number int

func (num Number) Even() bool {
	if num%2 == 0 {
```

```
                return true
        } else {
                return false
        }

}

func main() {
        i := os.Args[1]
        n, err := strconv.Atoi(i)
        if err != nil {
                fmt.Println("Not a number:", i)
                os.Exit(1)
        }
        num := Number(n)
        fmt.Println(num.Even())
}
```

In the above program, a custom type named `Number` is defined. Later a method named `Even` is defined below. To define a method for any type, the syntax is like this: `func (value CustomType) MethodName()`. You can also define input parameters and output parameters. In the above the output parameter is given as a `bool` value.

You can associate methods to structs. Consider this struct:

```
type Rectangle struct {
    Width  float64
    Height float64
}
```

If you want methods to calculate area and perimeter for this rectangle, you can define methods like this:

```
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

func (r Rectangle) Perimeter() float64 {
    return 2 * (r.Width * r.Height)
}
```

You can call these methods from the struct initialized using the `Rectangle` struct. Here is an example:

```
r := Rectangle{3.0, 5.0}
area := r.Area()
perimeter := r.Perimeter()
```

When a function is bound to a type, it is called method. The type that is bound is called receiver. A receiver could be any type with a name. When you declare a method, it is defined using receiver argument. The receiver argument points to the type where the method will be available. The receiver argument is specified between func keyword and the method name inside a bracket with a name.

Methods can be defined only on types declared in the same package. Declaring a method on built-in type is also illegal.

Here is an example:

Listing 5.9 : Method

```
package main

import "fmt"

type Circle struct {
    radius float64
}

func (c Circle) Area() float64 {
    return 3.14 * c.radius * c.radius
}

func main() {
     c := Circle{3.4}
     a := c.Area()
     fmt.Println(a)
}
```

In the above example, the method `Area` calculate area for a circle.

The receiver could be pointer also. Here is a modified example with pointer receiver:

Listing 5.10 : Method with pointer receiver

```
package main

import "fmt"

type Circle struct {
    radius float64
}

func (c *Circle) Area() float64 {
    return 3.14 * c.radius * c.radius
}
```

```
func main() {
    c1 := Circle{3.4}
    a1 := c1.Area()
    fmt.Println(a1)

    c2 := &Circle{3.4}
    a2 := c2.Area()
    fmt.Println(a2)

}
```

In the above example, the `Area` method is using a pointer receiver. When creating object, you can create a normal value or a pointer value. Calling the `Area` can use either a normal value or a pointer value.

Pointer receiver can be used when for any of these three reason:

- To modify the receiver itself by changing the value of attributes.
- The object is very large and a passing a deep copy is expensive.
- Consistency: Let all methods have pointer receivers.

You can use new function to allocate memory for *struct*:

```
type Temperature struct{
     Value float64
}

name := new(Temperature)
```

In the above example, the zero value is allocated and assigned to the variable `name`. But in some cases, zero value is not what you required. So you can use `&` to with struct syntax like this:

```
type Temperature struct{
     Value float64
}

name := &Temperature{Value: -7.6}
```

As you can see, the temperature value is set to `-7.6` and assigned to the variable.

5.7 Exercises

Exercise 1: Write a method to calculate the area of a rectangle for a given struct with width and height.

Solution:

```
type Rectangle struct {
    Width  float64
    Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a program with a function to calculate the perimeter of a circle.

Summary

This chapter explained the main features of functions in Go. It covered how to send input parameters and receive return values. It also explained about variadic functions and anonymous functions. This chapter briefly covered methods. The next chapter will cover interfaces. Along with that, we will learn more about methods.

Brief summary of key concepts introduced in this chapter:

- Functions are used to group together related code and make it easier to read and understand. They can also be used to reuse code in different parts of a program.
- Input parameters are values that are passed into a function when it is called. Return values are values that are returned by a function when it finishes executing.
- Variadic functions are functions that can accept a variable number of input parameters. Anonymous functions are functions that are defined without a name.
- Methods are functions that are associated with a specific type. They can be used to operate on objects of that type.

Chapter 6

Objects

Program to an interface, not an implementation. – Design Patterns by Gang of Four

In the chapter on functions, we have also learned about methods. A method is a function associated with a type, more specifically a concrete type. As you know, an object is an instance of a type. In general, methods define the behavior of an object.

Interfaces in Go provide a formal way to specify the behavior of an object. In layman's terms, Interface is like a blueprint which describes an object. So, Interface is considered as an abstract type, commonly referred to as interface type.

Small interfaces with one or two methods are common in Go.

Here is a `Geometry` interface which defines two methods:

```
type Geometry interface {
    Area() float64
    Perimeter() float64
}
```

If any type satisfy this interface - that is define these two methods which returns float64 - then, we can say that type is implementing this interface. One difference with many other languages with interface support and Go is that, in Go implementing an interface happens implicitly. So, no need to explicitly declare a type is implementing a particular interface.

To understand this idea, consider this `Rectangle` struct type:

```
type Rectangle struct {
    Width float64
    Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

func (r Rectangle) Perimeter() float64 {
    return 2*(r.width + r.height)
}
```

As you can see above, the above Rectangle type has two methods named Area and Perimeter which returns float64. So, we can say Rectangle is implementing the `Geometry` interface. To elaborate the example further, we will create one more implementation:

```
type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}

func (c Circle) Perimeter() float64 {
    return 2 * 3.14 * c.Radius
}
```

Now we have two separate implementations for the `Geometry` interface. So, if anywhere the `Geometry` interface type is expected, you can use any of these implementations.

Let's define a function which accepts a `Geometry` type and prints area and perimeter.

```
func Measure(g Geometry) {
    fmt.Println("Area:", g.Area())
    fmt.Println("Perimeter:", g.Perimeter())
}
```

When you call the above function, you can pass the argument as an object of `Geometry` interface type. Since both `Rectangle` and `Circle` satisfy that interface, you can use either one of them.

Here is a code which call `Measure` function with `Rectangle` and `Circle` objects:

```
r := Rectangle{Width: 2.5, Height: 4.0}
c := Circle{Radius: 6.5}
Measure(r)
Measure(c)
```

6.1 Type with Multiple Interfaces

In Go, a type can implement more than one interface. If a type has methods that satisfy different interfaces, we can say that that type is implementing those interfaces.

Consider this interface:

```
type Stringer interface {
     String() string
}
```

In previous section, there was a Rectangle type declared with two methods. In the same package, if you declare one more method like below, it makes that type implementing Stringer interface in addition to the Geometry interface.

```
func (r Rectangle) String() string {
    return fmt.Sprintf("Rectangle %vx%v", r.Width * r.Height)
}
```

Now the `Rectangle` type conforms to both `Geometry` interface and `Stringer` interface.

6.2 Empty Interface

The empty interface is the interface type that has no methods. Normally the empty interface will be used in the literal form: `interface`. All types satisfy empty interface. A function which accepts empty interface, can receive any type as the argument. Here is an example:

```
func blackHole(v interface{}) {
}

blackHole(1)
blackHole("Hello")
blackHole(struct{})
```

In the above code, the `blackHole` functions accepts an empty interface. So, when you are calling the function, any type of argument can be passed.

The `Println` function in the `fmt` package is variadic function which accepts empty interfaces. This is how the function signature looks like:

```
func Println(a ...interface{}) (n int, err error) {
```

Since the `Println` accepts empty interfaces, you could pass any type arguments like this:

```
fmt.Println(1, "Hello", struct{})
```

6.3 Pointer Receiver

In the chapter on Functions, you have seen that the methods can use a pointer receiver. Also we understood that the pointer receivers are required when the object attributes need be to modified or when passing large size data.

Consider the implementation of `Stringer` interface here:

```
type Temperature struct {
     Value float64
     Location string
}

func (t *Temperature) String() string {
     o := fmt.Sprintf("Temp: %.2f Loc: %s", t.Value, t.Location)
     return o
}
```

In the above example, the `String` method is implemented using a pointer receiver. Now if you define a function which accepts the `fmt.Stringer` interface, and want the `Temperature` object, it should be a pointer to `Temperature`.

```
func cityTemperature(v fmt.Stringer) {
    fmt.Println(v.String())
}

func main() {
    v := Temperature{35.6, "Bangalore"}
    cityTemperature(&v)
}
```

As you can see, the `cityTemperature` function is called with a pointer. If you modify the above code and pass normal value, you will get an error. The below code will produce an error as pointer is not passed.

```
func main() {
    v := Temperature{35.6, "Bangalore"}
    cityTemperature(v)
}
```

The error message will be something like this:

```
cannot use v (type Temperature) as type fmt.Stringer in argument to
cityTemperature: Temperature does not implement fmt.Stringer (String
method has pointer receiver)
```

6.4 Type Assertions

In some cases, you may want to access the underlying concrete value from the interface value. Let's say you define a function which accepts an interface value and want access attribute of the concrete value. Consider this example:

```
type Geometry interface {
    Area() float64
    Perimeter() float64
}

type Rectangle struct {
    Width float64
    Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

func (r Rectangle) Perimeter() float64 {
    return 2*(r.width + r.height)
}

func Measure(g Geometry) {
    fmt.Println("Area:", g.Area())
    fmt.Println("Perimeter:", g.Perimeter())
}
```

In the above example, if you want to print the width and and height from the `Measure` function, you can use type assertions.

Type assertion gives the underlying concrete value of an interface type. In the above example, you can access the rectangle object like this:

```
    r := g.(Rectangle)
    fmt.Println("Width:", r.Width)
    fmt.Println("Height:", r.Height)
```

If the assertion fail, it will trigger a panic.

Type assertion has an alternate syntax where it will not panic if assertion fail, but gives one more return value of boolean type. The second return value will be `true` if assertion succeeds otherwise it will give `false`.

```
r, ok := g.(Rectangle)
if ok {
    fmt.Println("Width:", r.Width)
    fmt.Println("Height:", r.Height)
}
```

If there are many types that need to be asserted like this, Go provides a type switches which is explained in the next section.

6.5 Type Switches

As you have seen in the previous section, type assertions gives access to the underlying value. But if there any many assertions need to be made, there will be lots `if` blocks. To avoid this, Go provides type switches.

```
switch v := g.(type) {
case Rectangle:
    fmt.Println("Width:", v.Width)
    fmt.Println("Height:", v.Height)
case Circle:
    fmt.Println("Width:", v.Radius)
case default:
    fmt.Println("Unknown:")
}
```

In the above example, type assertion is used with switch cases. Based on the type of `g`, the case is executed.

Note that the *fallthrough* statement does not work in type switch.

6.6 Exercises

Exercise 1: Using the `Marshaller` interface, make the marshalled output of the `Person` object given here all in upper case.

```
type Person struct {
    Name  string
    Place string
}
```

Solution:

```go
package main

import (
    "encoding/json"
    "fmt"
    "strings"
)

// Person represents a person
type Person struct {
    Name  string
    Place string
}

// MarshalJSON implements the Marshaller interface
func (p Person) MarshalJSON() ([]byte, error) {
    name := strings.ToUpper(p.Name)
    place := strings.ToUpper(p.Place)
    s := []byte(`{"NAME":"` + name + `","PLACE":"` + place + `"}`)
    return s, nil
}

func main() {
    p := Person{Name: "Baiju", Place: "Bangalore"}
    o, err := json.Marshal(p)
    if err != nil {
        panic(err)
    }
    fmt.Println(string(o))
}
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Implement the built-in `error` interface for a custom data type. This is how the `error` interface is defined:

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

Summary

This chapter explained the concept of interfaces and their uses. Interfaces are an important concept in Go. Understanding interfaces and using them properly makes the design robust. The chapter covered the empty interface, pointer receivers, and type assertions and type switches.

Brief summary of key concepts introduced in this chapter:

- An interface is a set of methods that a type must implement. A type that implements an interface can be used anywhere an interface is expected. This allows for greater flexibility and reusability in Go code.
- A pointer receiver is a method that takes a pointer to a struct as its receiver. Pointer receivers are often used to modify the state of a struct.
- A type assertion is a way of checking the type of a value at runtime. Type assertions can be used to ensure that a value is of a certain type before using it.
- A type switch is a control flow statement that allows for different code to be executed based on the type of a value. Type switches can be used to make code more robust and easier to read.

Chapter 7

Concurrency

Do not communicate by sharing memory; instead, share memory by communicating. — Effective Go

If you observe, you could see many things happening around you at any given time. This is how the world function - the train is gently moving, passengers talking each other, farmers working in the field and many other things are happening simultaneously. We can say, the world we live in function concurrently.

Go has built-in concurrency features with syntax support. The Go concurrency is inspired by a paper published in 1978 by Tony Hoare. The paper title is *Communicating sequential processes* [1].

Go has some new terminologies and keywords related to concurrent programming. The two important words are *goroutine* and *channel*. This chapter will go through these concepts and walk through some examples to further explain concurrency in Go.

The Go runtime is part of the executable binary created when compiling any Go code. The Go runtime contains a garbage collector and a scheduler to manage lightweight threads called Goroutines. Goroutine is a fundamental abstraction to support concurrency. Goroutine is an independently executing part of the program. You can invoke any number of goroutines and all of them could run concurrently.

Goroutines can communicate to each other via typed conduits called channels. Channels can be used to send and receive data.

[1] `http://usingcsp.com`

7.1 Goroutine

Goroutine is like a process running in the background. A function with *go* keyword as prefix starts the goroutine. Any function including anonymous function can be invoked with *go* keyword. In fact, the *main* function is a special goroutine invoked during the starup of any program by the Go runtime.

To understand the Goroutine better let's look at a simple program:

Listing 7.1 : Goroutine with explicit sleep

```
 1 package main

 3 import (
 4     "fmt"
 5     "time"
 6 )

 8 var msg string

10 func setMessage() {
11     msg = "Hello, World!"
12 }

14 func main() {
15     go setMessage()
16     time.Sleep(1 * time.Millisecond)
17     fmt.Println(msg)
18 }
```

In the above program, *setMessage* function is invoked as a goroutine in line no 15 using the *go* keyword. If you run this program, you will get the hello world message printed. If you change the sleep time to Zero, the message will not be printed. This is because, the program exits when main function completes execution. And in this case, since *setMessage* is called as a goroutine, it goes to background and main goroutine execution continues. In the earlier case when the time sleep was 1 second, the goroutine gets some time to execute before main completed. That's why the *msg* value is set and printed.

7.2 Channels

Multiple goroutines can communicate using channels. Channels can be used to send and receive any type of values. You can send and receive values with this channel operator: `<-`

This is how to declare a channel of *int* values:

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

To send a value to *ch* channel:

```
ch <- 4
```

To receive a value from `ch` channel and assign to a variable:

```
v := <-ch
```

You can also receive value without really assigning:

```
<-ch
```

Sending and receiving values from channels becomes a blocking operation. So, if you try to receive value from a channel, there should be some other part of the code which sends a value this channel. Until a value sends to the channel, the receiving part of the code will block the execution.

Here is an example:

Listing 7.2 : Goroutine with channels

```
 1 package main

 3 import (
 4     "fmt"
 5 )

 7 var c = make(chan int)
 8 var msg string

10 func setMessage() {
11     msg = "Hello, World!"
12     c <- 0
13 }

15 func main() {
16     go setMessage()
17     <-c
18     fmt.Println(msg)
19 }
```

In the above example, an int channel is assigned to a global variable named `c`. In line number 17, immediately after calling goroutines, channel is trying to receive a value. This becomes a blocking operation in the `main` goroutine. In line number 12, inside the `setMessage` function, after setting a value for `msg`, a value is send to the `c` channel. This will make the operation to continue in the `main` goroutine.

7.3 Waitgroups

Go standard library has a *sync* package which provides few synchronization primitives. One of the mechanism is *Waitgroups* which can be used to wait for multiple goroutines to complete. The **Add** function add the number of goroutines to wait for. At the end of these goroutines call **Done** function to indicate the task has completed. The **Wait** function call, block further operations until all goroutines are completed.

Here is a modified version of the previous example using *Waitgroups*.

Listing 7.3 : Goroutine with Waitgroups

```
 1 package main

 3 import (
 4     "fmt"
 5     "sync"
 6 )

 8 var msg string
 9 var wg sync.WaitGroup

11 func setMessage() {
12     msg = "Hello, World!"
13     wg.Done()
14 }

16 func main() {
17     wg.Add(1)
18     go setMessage()
19     wg.Wait()
20     fmt.Println(msg)
21 }
```

In the above example, the **Add** method at line number 17 make one item to wait for. The next line invoke the goroutine. The line number 19, the **Wait** method call blocks any further operations until goroutines are completed. The previous line made goroutine and inside the goroutine, at the end of that goroutine, there is a **Done** call at line number 13.

Here is another example:

Listing 7.4 : Goroutine with Waitgroups

```
 1 package main
```

```go
import (
    "fmt"
    "sync"
    "time"
)

func someWork(i int) {
    time.Sleep(time.Millisecond * 10)
    fmt.Println(i)
}

func main() {
    var wg sync.WaitGroup
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(j int) {
            defer wg.Done()
            someWork(j)
        }(i)
    }
    wg.Wait()
}
```

7.4 Select

The *select* is a statement with some similarity to *switch*, but used with channels and goroutines. The *select* statement lets a goroutine wait on multiple communication operations through channels.

Under a *select* statement, you can add multiple cases. A select statement blocks until one of its case is available for run – that is the channel has some value. If multiple channels used in cases has value readily avaibale, select chooses one at random.

Here is an example:

```go
package main

import "time"
import "fmt"

func main() {

    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(time.Second * 1)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(time.Second * 2)
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("received", msg1)
        case msg2 := <-c2:
            fmt.Println("received", msg2)
        }
    }
}
```

7.5 Buffered Channels

Buffered channels are channels with a given capacity. The capacity is the size of channel in terms of number of elements. If the capacity is zero or absent, the channel is unbuffered. For a buffered channel communication succeeds only when both a sender and receiver are ready. Whereas for a buffered channel, communication succeeds without blocking if the buffer is not full (sends) or not empty (receives).

The capacity can be given as the third argument to make function:

```
make(chan int, 100)
```

Consider the below example:

Listing 7.5 : Buffered Channel

```
package main

import "fmt"

func main() {
    ch := make(chan string, 2)
    ch <- "Hello"
    ch <- "World"
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

The *ch* channel is a buffered channel, this makes it possible to send value without any receiver present.

7.6 Channel Direction

When declaring a function with channels as input parameters, you can also specify the direction of the channel. The direction of channel declares whether it can only receive or only send values. The channel direction helps to increases the type-safety of the program.

Here is an example:

Listing 7.6 : Channel channel

```go
package main

import "fmt"

func sendOnly(name chan<- string) {
    name <- "Hi"
}

func receiveOnly(name <-chan string) {
    fmt.Println(<-name)
}

func main() {
    n := make(chan string)

    go func() {
        fmt.Println(<-n)
    }()

    sendOnly(n)

    go func() {
        n <- "Hello"
    }()

    receiveOnly(n)
}
```

In the above example, the `sendOnly` function define a channel variable which can be only used for sending data. If you tried to read from that channel within that function, it's going to be compile time error. Similary the `receiveOnly` function define a channel variable which can be only user for receive data. You cannot send any value to that channel from that function.

7.7 Lazy Initialization Using sync.Once

The sync package provide another struct called Once which is useful for lazy initialization.

Here is an example:

```
import (
    "sync"
)

type DB struct{}

var db *DB
var once sync.Once

func GetDB() *DB {
    once.Do(func() {
        db = &DB{}
    })
    return db
}
```

If the above GetDB function is called multiple times, only once the DB object will get constructed.

7.8 Exercises

Exercise 1: Write a program to download a list of web pages concurrently using Goroutines.

Hint: Use this tool for serving junk content for testing: `https://github.com/baijum/lipsum`

Solution:

```
package main

import (
        "io/ioutil"
        "log"
        "net/http"
        "net/url"
        "sync"
)

func main() {
        urls := []string{
                "http://localhost:9999/1.txt",
                "http://localhost:9999/2.txt",
                "http://localhost:9999/3.txt",
                "http://localhost:9999/4.txt",
        }
        var wg sync.WaitGroup
        for _, u := range urls {
                wg.Add(1)
                go func(u string) {
                        defer wg.Done()
                        ul, err := url.Parse(u)
                        fn := ul.Path[1:len(ul.Path)]
                        res, err := http.Get(u)
                        if err != nil {
                                log.Println(err, u)
                        }
                        content, _ := ioutil.ReadAll(res.Body)
                        ioutil.WriteFile(fn, content, 0644)
                        res.Body.Close()
                }(u)
        }
        wg.Wait()
}
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a program to watch log files and detect any entry with a particular word.

Summary

This chapter explained how to use Go's concurrency features. You can choose channels or other synchronization techniques, depending on your problem. This chapter covered goroutines and how to use channels. It also covered Waitgroups and Select statements. Additionally, it covered buffered channels and channel direction. Finally, the chapter briefly discussed the *sync.Once* function.

Chapter 8

Packages

A little copying is better than a little dependency. — Go Proverbs

Go encourages and provides mechanisms for code reusability. Packages are one of the building block for code reusability. We have seen other code reusability mechanisms like functions in the earlier chapters.

This chapter will explain everything you need to know about packages. In the previous programs, there was a clause at the beginning of every source files like this:

```
package main
```

That clause was declaring the name of the package as `main`. The package name was `main` because the program was executable. For non-executable programs, you can give a different meaningful short name as the package name.

Most of the previous programs imported other packages like this:

```
import "fmt"
```

Sometimes the import was within parenthesis (factored imports) for multiple packages:

```
import (
	"fmt"
	"time"
)
```

Importing packages gives access to functionalities provided by those packages.

Packages helps you to create modular reusable programs. Go programs are organized using packages. Every source file will be associated with a package. Basically, package is a collection of source files.

A package generally consists of elements like constants, types, variables and functions which are accessible across all the source files. The source files should should start with a statement declaring the name of the package. The package name should be a meaningful short name. This is important because normally the exported names within the package will be accessed with the package name as a prefix.

The Go standard library has many packages. Each package will be related to certain ideas.

8.1 Creating a Package

Consider this file named **circle1.go** which belongs to package **main**:

Listing 8.1 : circle1.go

```go
package main

const pi float64 = 3.14

type Circle struct {
    Radius float64
}
```

Another file named **circle2.go** in the same directory:

Listing 8.2 : circle2.go

```go
package main

func (c Circle) Area() float64 {
    return pi * c.Radius * c.Radius
}
```

Yet another file named **circle3.go**:

Listing 8.3 : circle3.go

```go
package main

import "fmt"
```

```
var c Circle

func main() {
    c = Circle{3.5}
    fmt.Println(c.Area())
}
```

As you can see above all the above source files belongs to `main` package.

You can build and run it like this:

```
$ go build -o circle
$ ./circle
38.465
```

Alternatively, you can use `go run`:

```
$ go run *.go
38.465
```

The final Go program is a created by joining multiple files. The package is a container for your code.

8.2 Package Initialization

When you run a Go program, all the associated packages initialize in a particular order. The package level variables initialize as per the order of declaration. However, if there is any dependency on initialization, that is resolved first.

Consider this example:

```
package varinit

var a = b + 2
var b = 1
```

In the above program, the variable `a` is depending on the value of variable `b`. Therefore, the variable `b` initialize first and the variable `a` is the second. If you import the above package from different other packages in the same program, the variable initialization happens only once.

If the expression for variable initialization is complicated, you can use a function to return the value. There is another approach possible for initializing variables with complex expressions, that is by using the special *init* function. The Go runtime executes the *init* function only once. Here is an example:

Listing 8.4 : Configuration initialization

```go
package config

import (
    "log"

    "github.com/kelseyhightower/envconfig"
)

type Configuration struct {
    Address        string `default:":8080"`
    TokenSecretKey string `default:"secret" split_words:"true"`
}

var Config Configuration

func init() {
    err := envconfig.Process("app", &Config)
    if err != nil {
        log.Fatal(err.Error())
    }
}
```

The *init* function cannot be called or even referred from the program – it's going to produce a compile error. Also, the *init* function should not return any value.

8.3 Documenting Packages

You can write documentation for the package in the form of comment before the declarion of package name. The comment should begin with the word *Package* followed by the name of the package. Here is an example:

```go
// Package hello gives various greeting messages
package hello
```

If the package is defined in multiple files, you may create a file named *doc.go*. The *doc.go* file is just a naming convention. You can write multi-line comment using `/* ... */` syntax. You can write source code with tab-indentation which will be highlighted and displayed using monospace font in HTML format.

Here is an example from the *http* package *doc.go* file.

```go
// Copyright 2011 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

/*
Package http provides HTTP client and server implementations.

Get, Head, Post, and PostForm make HTTP (or HTTPS) requests:

        resp, err := http.Get("http://example.com/")
        ...

<...strip lines...>

*/
package http
```

Many lines are stripped from the above example where its marked. As you can see the above documentation file starts with copyright notice. But the copyright notice is using single line comment multiple times. This notice will be ignored when generating documentation. And the documentation is written within multi-line comments. At the end of file, the package name is also declared.

8.4 Publishing Packages

Unlike other mainstream languages, Go doesn't have a central package server. You can publish your code directly to any version control system repositories. Git is most widely used VCS used to publish packages, but you can also use Mercurial or Bazaar.

Here are few example packages:

- `https://github.com/auth0/go-jwt-middleware`
- `https://github.com/blevesearch/bleve`
- `https://github.com/dgrijalva/jwt-go`
- `https://github.com/elazarl/go-bindata-assetfs`
- `https://github.com/google/jsonapi`
- `https://github.com/gorilla/mux`
- `https://github.com/jpillora/backoff`
- `https://github.com/kelseyhightower/envconfig`
- `https://github.com/lib/pq`

- `https://github.com/pkg/errors`
- `https://github.com/thoas/stats`
- `https://github.com/urfave/negroni`

You can use `go get` command to get these packages locally. Go 1.11 release introduced the module support. The modules can be used to manage external dependant packages.

8.5 Module

A *module* is a collection of Go packages with well defined name module and dependency requirements. Also, the module has reproducible builds.

Modules use semantic versioning. The format of version should vMAJOR.MINOR.PATCH. For example, v0.1.1, v1.3.1, or v2.0.0. Note that the *v* prefix is mandatory.

There should be a file named *go.mod* at the top-level directory. This file is used to declare the name of the module and list dependencies.

The minimal version selection algorithm is used to select the versions of all modules used in a build. For each module in a build, the version selected by minimal version selection is always the semantically highest of the versions explicitly listed by a require directive in the main module or one of its dependencies.

You can see an explanation of the algorithm here:
`https://research.swtch.com/vgo-mvs`

Creating a module

The Go tool has support for creating modules. You can use the *mod* command to manage module.

To initialize a new module, use *mod init* command with name of module as argument. Normally the name will be same as the publishing location. Here is an example:

```
mkdir hello
cd hello
go mod init github.com/baijum/hello
```

In the above example, the name is given as **github.com/baijum/hello**. This command is going to create file named **go.mod** and the content of that file will be like this:

```
module github.com/baijum/hello

go 1.20
```

As of now there are no dependencies. That is the reason the **go.mod** file doesn't list any dependencies.

Let's create a **hello.go** with **github.com/lib/pq** as a dependency.

```
package main

import (
    "database/sql"

    _ "github.com/lib/pq"
)

func main() {
    sql.Open("postgres",
        "host=lh port=5432 user=gt dbname=db password=secret")
}
```

Note: The imported package is the driver used by the *sql* package.

You will see the **mod.mod** updated with dependency.

```
module github.com/github.com/hello

go 1.20

require github.com/lib/pq v1.10.9
```

There will be another auto-generated file named **go.sum** which is used for validation. You can commit both these files to your version control system.

8.6 Moving Type Across Packages

When you refactoring a large package into smaller package or separating certain features into another package, moving types will be required. But moving types will be difficult as it may introduce backward compatibility and it may affect existing consumer packages. Go has support for type aliases to solve this problem. Type alias can be declared as given in this example:

```
type NewType = OldType
```

Type alias can be removed once all the dependant packages migrate to use import path.

8.7 Exercises

Exercise 1: Create a package named `rectangle` with exported functions to calculate area and perimeter for the given rectangle.

Solution:

The name of the package could be `rectangle`. Here is the program:

```
// Package rectangle provides functions to calculate area
// and perimeter for rectangle.
package rectangle

// Area calculate area of the given rectangle
func Area(width, height float64) float64 {
     return width * height
}

// Perimeter calculate area of the given rectangle
func Perimeter(width, height float64) float64 {
     return 2 * (width + height)
}
```

As you can see above, the source file starts with package documentation. Also you can see the documentation for all exported functions.

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Create a package with 3 source files and another *doc.go* for documentation. The package should provide functions to calculate areas for circle, rectangle, and triangle.

Summary

This chapter explained packages in Go programming. Packages are a collection of related Go source files that are compiled together to form a single unit. They are one of the building blocks of a reusable Go program. This chapter explained how to create packages, document packages, and publish packages. It also covered modules and their usage. Finally, it explained how to move types across packages during refactoring. By understanding packages, you can write more modular and reusable Go programs.

Chapter 9

Input/Output

for out of the abundance of the heart the mouth speaketh.
– Bible

Users interact with software systems through various input/output mechanisms. Some of the commonly used mechanisms are these:

- web browser for web applications using various controllers/widgets
- mobile for mobile applications
- shell for command line applications
- desktop application with native controllers/widgets

To provide these kind of user interactions, you will be required to use specialized libraries. If the standard library doesn't provide what you are looking for, you may need to use third party libraries. This chapter cover basic mechanisms provided at the language level which you can use for input/output.

We have already seen some of the basic input/output techniques in the last few chapters. This chapter will go though more input/output mechanisms available in Go.

9.1 Command Line Arguments

The command line arguments as user input and console for output is the way command line programs are designed. Sometimes output will be other files and devices. You can access all the command line arguments using the **Args** array/slice attribute available in **os** package. Here is an example:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println(os.Args)
    fmt.Println(os.Args[0])
    fmt.Println(os.Args[1])
    fmt.Println(os.Args[2])
}
```

You can run this program with minimum two additional arguments:

```
$ go build cmdline.go
$ ./cmdline one -two
[cmdline one -two]
./cmdline
one
-two
```

As you can see, it's difficult to parse command line arguments like this. Go standard library has a package named *flag* which helps to easily parse command line arguments. This chapter has a section to explain the *flag* package.

9.2 Files

Reading and writing data to files is a common I/O operation in computer programming. You can manipulate files using the *os* and *io* packages. It works with both text files and binary files. For the simplicity of this section, all the examples given here works with text files.

Consider there exists a file named **poem.txt** with this text:

```
I wandered lonely as a cloud
That floats on high o'er vales and hills,
When all at once I saw a crowd,
A host, of golden daffodils;
Beside the lake, beneath the trees,
Fluttering and dancing in the breeze.
```

Here is a program to read the the whole file and print:

Listing 9.1 : Read whole file

```
 1 package main

 3 import (
 4     "fmt"
 5     "io"
 6     "os"
 7 )

 9 func main() {
10     fd, err := os.Open("poem.txt")
11     if err != nil {
12         fmt.Println("Error reading file:", err)
13     }
14     for {
15         chars := make([]byte, 50)
16         n, err := fd.Read(chars)
17         if n == 0 && err == io.EOF {
18             break
19         }
20         fmt.Print(string(chars))
21     }
22     fd.Close()
23 }
```

When you run this program, you will get the whole text as output. In the line number 10, the `Open` function from the `os` package is called to open the file for reading. It returns a file descriptor[1] and error. In the line number 14, an infinite loop is stared to read the content. Line 15 initialize a slice of bytes of length 50. The `fd.Read` method reads the given length of characters and writes to the given slice. It returns the number of characters read and error. The `io.EOF` error is returned when end of file is reached. This is used as the condition to break the loop.

Here is a program to write some text to a file:

Listing 9.2 : Write to file

```
package main

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

[1]https://en.wikipedia.org/wiki/File_descriptor

```
 8 func main() {
 9     fd, err := os.Create("hello.txt")
10     if err != nil {
11         fmt.Println("Cannot write file:", err)
12         os.Exit(1)
13     }
14     fd.Write([]byte("Hello, World!\n"))
15     fd.Close()
16 }
```

In th line number 9, the *Create* function from the *os* package is called open the file for writing. It returns a file descriptor and error. In the line number 14, the *Write* method is give a slice of bytes to write. After running the program you can see the text in the `hello.txt` file.

```
$ go run writefile.go
$ cat hello.txt
Hello, World!
```

9.3 Standard Streams

Standard streams[2] are input and output communication channels between a computer program and its environment. The three input/output connections are called standard input (stdin), standard output (stdout) and standard error (stderr).

Stdin, Stdout, and Stderr are open files pointing to the standard input, standard output, and standard error file descriptors.

The *fmt* package has functions to read values interactively.

Here is an example:

Listing 9.3 : Read name and print

```
package main

import "fmt"

func main() {
    var name string
    fmt.Printf("Enter your name: ")
    fmt.Scanf("%s", &name)
    fmt.Println("Your name:", name)
}
```

[2] https://en.wikipedia.org/wiki/Standard_streams

The *Scanf* function reads the standard input. The first argument is the format and the second one is the pointer variable. The value read from standard input cab be accessed using the given variable.

You can run the above program in different ways:

```
$ go run code/io/readprint.go
Enter your name: Baiju
Your name: Baiju
$ echo "Baiju" |go run code/io/readprint.go
Enter your name: Your name: Baiju
$ go run code/io/readprint.go << EOF
> Baiju
> EOF
Enter your name: Your name: Baiju
$ echo "Baiju" > /tmp/baiju.txt
$ go run code/io/readprint.go < /tmp/baiju.txt
Enter your name: Your name: Baiju
```

As you can see from this program, the *Printf* function writes to standard output and the *Scanf* reads the standard input. Go can also writes to standard error output stream.

The *io* package provides a set of interfaces and functions that allow developers to work with different types of input and output streams.

Consider a use case to convert everything that comes to standard input to convert to upper case. This can be achieved by reading all standard input using `io.ReadAll` and converting to upper case. Here is code:

Listing 9.4 : Convert standard input to upper case

```go
package main

import (
    "fmt"
    "io"
    "os"
    "strings"
)

func main() {
    stdin, err := io.ReadAll(os.Stdin)
    if err != nil {
        panic(err)
    }
    str := string(stdin)
    newStr := strings.TrimSuffix(str, "\n")
    upper := strings.ToUpper(newStr)
    fmt.Println(upper)
}
```

You can run this program similar to how you did with the previous program.

You can use *fmt.Fprintf* with *os.Stderr* as the first argument to write to standard error.

```
fmt.Fprintf(os.Stderr, "This goes to standard error: %s", "OK")
```

Alternatively, you can call *WriteString* method of *os.Stderr*:

```
os.Stderr.WriteString("This goes to standard error")
```

9.4 Using flag Package

As you have noticed before `os.Args` attribute in the *os* package provides access to all command line arguments. The *flag* package provides an easy way to parse command line arguments.

You can define string, boolean, and integer flags among others using the *flag* package..

Here is an integer flag declaration:

```
var pageCount = flag.Int("count", 240, "number of pages")
```

The above code snippet defines an integer flag with name as `count` and it is stored in a variable with the name as `pageCount`. The type of the variable is **int*. Similar to this integer flag, you can defines flags of other types.

Once all the flags are defined, you can parse it like this:

```
flag.Parse()
```

The above `Parse` function call parse the command line arguments and store the values in the given variables.

Once the flags are parsed, you can dereference it like this:

```
fmt.Println("pageCount: ", *pageCount)
```

To access non-flag arguments:

```
flag.Args()
```

The above call returns a the arguments as a slice of strings. It contains arguments not parsed as flags.

Cobra is a third party package providing a simple interface to create command line interfaces. Cobra also helps to generate applications and command files. Many of the most widely used Go projects are built using Cobra. This is the Cobra website: `https://github.com/spf13/cobra`

9.5 String Formatting

Go supports many string format options. To get the default format of any value, you can use `%v` as the format string. Here is an example which print formatted values using `%v`:

Listing 9.5 : Default format

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Printf("Value: %v, Type: %T\n", "Baiju", "Baiju")
    fmt.Printf("Value: %v, Type: %T\n", 7, 7)
    fmt.Printf("Value: %v, Type: %T\n", uint(7), uint(7))
    fmt.Printf("Value: %v, Type: %T\n", int8(7), int8(7))
    fmt.Printf("Value: %v, Type: %T\n", true, true)
    fmt.Printf("Value: %v, Type: %T\n", 7.0, 7.0)
    fmt.Printf("Value: %v, Type: %T\n", (1 + 6i), (1 + 6i))
}
```

The `%T` shows the type of the value. The output of the above program will be like this.

```
Value: Baiju, Type: string
Value: 7, Type: int
Value: 7, Type: uint
Value: 7, Type: int8
Value: true, Type: bool
Value: 7, Type: float64
Value: (1+6i), Type: complex128
```

If you use a `%+v` as the format string for struct it shows the field names. See this example:

Listing 9.6 : Default format for struct

```go
package main

import (
    "fmt"
)

// Circle represents a circle
type Circle struct {
    radius float64
    color  string
}

func main() {
    c := Circle{radius: 76.45, color: "blue"}
    fmt.Printf("Value: %#v\n", c)
    fmt.Printf("Value with fields: %+v\n", c)
    fmt.Printf("Type: %T\n", c)
}
```

If you run the above program, the output is going to be like this:

```
Value: {76.45 blue}
Value with fields: {radius:76.45 color:blue}
Type: main.Circle
```

As you can see from the output, in the first line `%v` doesn't show the fields. But in the second line, `%+v` shows the struct fields.

The `%#v` shows the representation of the value. Here is a modified version of above program to print few values of primitive type.

Listing 9.7 : Representation format

```
package main

import (
    "fmt"
)

func main() {
    fmt.Printf("Value: %#v, Type: %T\n", "Baiju", "Baiju")
    fmt.Printf("Value: %#v, Type: %T\n", 7, 7)
    fmt.Printf("Value: %#v, Type: %T\n", uint(7), uint(7))
    fmt.Printf("Value: %#v, Type: %T\n", int8(7), int8(7))
    fmt.Printf("Value: %#v, Type: %T\n", true, true)
    fmt.Printf("Value: %#v, Type: %T\n", 7.0, 7.0)
    fmt.Printf("Value: %#v, Type: %T\n", (1 + 6i), (1 + 6i))
}
```

```
Value: "Baiju", Type: string
Value: 7, Type: int
Value: 0x7, Type: uint
Value: 7, Type: int8
Value: true, Type: bool
Value: 7, Type: float64
Value: (1+6i), Type: complex128
```

As you can see in the representation, strings are written within quotes. You can also see representation of few other primitive types.

If you want a literal `%` sign, use two `%` signs next to each other. Here is a code snippet:

```
        fmt.Println("Tom scored 92%% marks")
```

The default string representation of custom types can be changed by implementing `fmt.Stringer` interafce. The interface definition is like this:

```
type Stringer interface {
        String() string
}
```

As per the `Stringer` interface, you need to create a `String` function which return a string. Now the value printed will be whatever returned by that function. Here is an example:

Listing 9.8 : Custom representation using Stringer

```go
package main

import (
    "fmt"
    "strconv"
)

// Temperature repesent air temperature
type Temperature struct {
    Value float64
    Unit  string
}

func (t Temperature) String() string {
    f := strconv.FormatFloat(t.Value, 'f', 2, 64)
    return f + " degree " + t.Unit
}

func main() {
    temp := Temperature{30.456, "Celsius"}
    fmt.Println(temp)
    fmt.Printf("%v\n", temp)
    fmt.Printf("%+v\n", temp)
    fmt.Printf("%#v\n", temp)
}
```

The output of the above program will be like this:

```
30.46 degree Celsius
30.46 degree Celsius
30.46 degree Celsius
main.Temperature{Value:30.456, Unit:"Celsius"}
```

9.6 Exercises

Exercise 1: Write a program to read length and width of a rectangle through command line arguments and print the area. Use `-length` switch to get length and `-width` switch to get width. Represent the rectangle using a struct.

Solution:

```go
package main

import (
    "flag"
    "fmt"
    "log"
    "os"
)

// Rectangle represents a rectangle shape
type Rectangle struct {
    Length float64
    Width  float64
}

// Area return the area of a rectangle
func (r Rectangle) Area() float64 {
    return r.Length * r.Width
}

var length = flag.Float64("length", 0, "length of rectangle")
var width = flag.Float64("width", 0, "width of rectangle")

func main() {
    flag.Parse()
    if *length <= 0 {
        log.Println("Invalid length")
        os.Exit(1)
    }
    if *width <= 0 {
        log.Println("Invalid width")
        os.Exit(1)
    }
    r := Rectangle{Length: *length, Width: *width}
    a := r.Area()
    fmt.Println("Area: ", a)
}
```

You can run the program like this:

```
$ go run rectangle.go -length 2.5 -width 3.4
```

```
Area:  8.5
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a program to format a complex number as used in mathematics. Example: `2 + 5i`

Use a struct like this to define the complex number:

```
type Complex struct {
    Real float64
    Imaginary float64
}
```

Summary

This chapter discussed the input/output (I/O) features of the Go programming language. It explained how to use command line arguments and interactive input, and how to use the *flag* package to parse command line arguments. It also explained various string formatting techniques.

Chapter 10

Testing

Test as You Fly, Fly as You Test. — NASA

Writing automated tests helps you to improve the quality and reliability of software. This chapter is about writing automated tests in Go. The standard library contains a package named *testing* to write tests. Also, the built-in Go tool has a test runner to run tests.

Consider this test for a `Hello` function which takes a string as input and returns another string. The expected output string should start with ”Hello,” and ends with the input value followed by an exclamation.

Listing 10.1 : Test for Hello

```go
package hello

import "testing"

func TestHello(t *testing.T) {
    out := Hello("Tom")
    if out != "Hello, Tom!" {
        t.Fail()
    }
}
```

In the first line, the package name is `hello` which is the same as the package where the function is going to be defined. Since both test and actual code is in the same package, the test can access any name within that package irrespective whether it is an exported name or not. At the same, when the actual problem is compiled using the **go build** command, the test files are ignored. The build ignores all files with the name ending **_test.go**. Sometimes these kinds of tests are called white-box test as it is accessing both exported and unexported names within the package. If you only want to access exported names within the test, you can declare the name of the test package with **_test** suffix. In this case, that would be **hello_test**, which should work in this case as we are only testing the exported function directly. However, to access those exported names – in this case, a function – the package should import explicitly.

The line no. 5 starts the test function declaration. As per the test framework, the function name should start with **Test** prefix. The prefix is what helps the test runner to identify the tests that should be running.

The input parameter **t** is a pointer of type **testing.T**. The **testing.T** type provides functions to make the test pass or fail. It also gives functions to log messages.

In the line no. 6 the **Hello** function is called with ”Tom” as the input string. The return value is assigning to a variable named **out**.

In line no. 7 the actual output value is checked against the expected output value. If the values are not matching, the **Fail** function is getting called. The particular function state is going to be marked as a failure when the **Fail** function is getting called.

The test is going to pass or fail based on the implementation of the **Hello** function. Here is an implementation of the function to satisfy the test:

Listing 10.2 : Hello function

```go
package hello

import "fmt"

// Hello says "Hello" with name
func Hello(name string) string {
    return fmt.Sprintf("Hello, %s!", name)
}
```

As you can see in the above function definition, it takes a string as an input argument. A new string is getting constructed as per the requirement, and it returns that value.

Now you can run the test like this:

```
$ go test
PASS
ok      _/home/baiju/hello     0.001s
```

If you want to see the verbose output, use the `-v` option:

```
$ go test -v
=== RUN   TestHello
--- PASS: TestHello (0.00s)
PASS
ok      _/home/baiju/hello     0.001s
```

10.1 Failing a Test

To fail a test, you need to explicitly call `Fail` function provided by the value of `testing.T` type. As we have seen before, every test function has access to a `testing.T` object. Usually, the name of that value is going to write as `t`. To fail a test, you can call the `Fail` function like this:

```
t.Fail()
```

The test is going be a failure when the `Fail` function is getting called. However, the remaining code in the same test function continue to execute. If you want to stop executing the further lines immediately, you can call `FailNow` function.

```
t.FailNow()
```

Alternatively, there are other convenient functions which give similar behavior along with logging message. The next section discusses logging messages.

10.2 Logging Message

The `testing.T` has two functions for logging, one with default formatting and the other with the user-specified format. Both functions accept an arbitrary number of arguments.

The `Log` function formats its arguments using the default formats available for any type. The behavior is similar to `fmt.Println` function. So, you can change the formatted value by implementing the `fmt.Stringer` interface:

```
type Stringer interface {
        String() string
}
```

You need to create a method named `String` which returns a string for your custom types.

Here is an example calling `Log` with two arguments:

```
t.Log("Some message", someValue)
```

In the above function call, there are only two arguments given, but you can pass any number of arguments.

The log message going to print if the test is failing. The verbose mode, the `-v` command-line option, also log the message irrespective of whether a test fails or not.

The `Logf` function takes a string format followed by arguments expected by the given string format. Here is an example:

```
t.Logf("%d no. of lines: %s", 34, "More number of lines")
```

The `Logf` formats the values based on the given format. The `Logf` is similar to `fmt.Printf` function.

10.3 Failing with Log Message

Usually, logging and marking a test as failure happens simultaneously. The `testing.T` has two functions for logging with failing, one with default formatting and the other with the user-specified format. Both functions accept an arbitrary number of arguments.

The `Error` function is equivalent to calling `Log` followed by `Fail`. The function signature is similar to `Log` function.

Here is an example calling `Error` with two arguments:

```
t.Error("Some message", someValue)
```

Similar to `Error` function, the `Errorf` function is equivalent to calling `Logf` followed by `Fail`. The function signature is similar to `Logf` function.

The `Errorf` function takes a string format followed by arguments expected by the given string format. Here is an example:

```
t.Errorf("%d no. of lines: %s", 34, "More number of lines")
```

The `Errorf` formats the values based on the given format.

10.4 Skipping Test

When writing tests, there are situations where particular tests need not run. Some tests might have written for a specific environment. The criteria for running tests could be CPU architecture, operating system or any other parameter. The *testing* package has functions to mark test for skipping.

The `SkipNow` function call marks the test as having been skipped. It stops the current test execution. If the test has marked as failed before skipping, that particular test is yet considered to have failed. The `SkipNow` function doesn't accept any argument. Here is a simple example:

Listing 10.3 : Skipping test

```go
package hello

import (
    "runtime"
    "testing"
)

func TestHello(t *testing.T) {
    if runtime.GOOS == "linux" {
        t.SkipNow()
    }
    out := Hello("Tom")
    if out != "Hello, Tom!" {
        t.Fail()
    }
}
```

If you run the above code on a Linux system, you can see the test has skipped. The output is going to be something like this:

```
$ go test . -v
=== RUN   TestHello
--- SKIP: TestHello (0.00s)
PASS
ok      _/home/baiju/skip      0.001s
```

As you can see from the output, the test has skipped execution.

There are two more convenient functions similar to `Error` and `Errorf`. Those functions are `Skip` and `Skipf`. These functions help you to log a message before skipping. The message could be the reason for skipping the test.

Here is an example:

```
t.Skip("Some reason message", someValue)
```

The `Skipf` function takes a string format followed by arguments expected by the given string format. Here is an example:

```
t.Skipf("%d no. of lines: %s", 34, "More number of lines")
```

The `Skipf` formats the values based on the given format.

10.5 Parallel Running

You can mark a test to run in parallel. To do so, you can call the `t.Parallel` function. The test is going to run in parallel to other tests marked parallel.

10.6 Sub Tests

The Go testing package allows you to group related tests together in a hierarchical form. You can define multiple related tests under a single-parent test using the 'Run' method.

To create a subtest, you use the `t.Run()` method. The `t.Run()` method takes two arguments: the name of the subtest and the body of the subtest. The body of the subtest is a regular Go function.

For example, the following code creates a subtest called `foo`:

```
func TestBar(t *testing.T) {
  t.Run("foo", func(t *testing.T) {
    // This is the body of the subtest.
  })
}
```

Subtests are reported separately from each other. This means that if a subtest fails, the test runner will report the failure for that subtest only. The parent test will still be considered to have passed.

Subtests can be used to test different aspects of a function or method. For example, you could use subtests to test different input values, different output values, or different error conditions.

Subtests can also be used to test different implementations of a function or method. For example, you could use subtests to test a function that is implemented in Go and a function that is implemented in C.

Subtests are a powerful feature of the Go testing package. They can be used to organize your tests, make them easier to read and maintain, and test different aspects of your code.

10.7 Exercises

Exercise 1: Create a package with a function to return the sum of two integers and write a test for the same.

Solution:

sum.go:

```
package sum

// Add adds to integers
func Add(first, second int) int {
    return first + second
}
```

sum_test.go:

```
package sum

import "testing"

func TestAdd(t *testing.T) {
    out := Add(2, 3)
    if out != 5 {
        t.Error("Sum of 2 and 3:", out)
    }
}
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a test case program to fail the test and not continue with the remaining tests.

Summary

This chapter explained how to write tests using the *testing* package. It covered how to mark a test as a failure, how to log messages, how to skip tests, and how to run tests in parallel. It also briefly mentioned sub-tests.

Chapter 11

Tooling

We become what we behold. We shape our tools, and thereafter our tools shape us. — Marshall McLuhan

Good support for lots of useful tools is another strength of Go. Apart from the built-in tools, there any many other community-built tools. This chapter covers the built-in Go tools and few other external tools.

The built-in Go tools can access through the `go` command. When you install the Go compiler (`gc`); the `go` tool is available in the path. The `go` tool has many commands. You can use the `go` tool to compile Go programs, run test cases, and format source files among other things.

11.1 Getting Help

The go tool is self-documented. You can get help about any commands easily. To see the list of all commands, you can run the `"help"` command. For example, to see help for `build` command, you can run like this:

```
go help build
```

The help command also provides help for specific topics like ”buildmode”, ”cache”, ”filetype”, and ”environment” among other topics. To see help for a specific topic, you can run the command like this:

```
go help environment
```

11.2 Basic Information

Version

When reporting bugs, it is essential to specify the Go version number and environment details. The Go tool gives access to this information through the following commands.

To get version information, run this command:

```
go version
```

The output should look something like this:

```
go version go1.20.4 linux/amd64
```

As you can see, it shows the version number followed by operating system and CPU architecture.

Environment

To get environment variables, you can run this command:

```
go env
```

The output should display all the environment variables used by the Go tool when running different commands.

A typical output will look like this:

```
GO111MODULE=""
GOARCH="amd64"
GOBIN=""
GOCACHE="/home/baiju/.cache/go-build"
GOENV="/home/baiju/.config/go/env"
GOEXE=""
GOEXPERIMENT=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOINSECURE=""
GOMODCACHE="/home/baiju/go/pkg/mod"
GONOPROXY=""
GONOSUMDB=""
GOOS="linux"
GOPATH="/home/baiju/go"
GOPRIVATE=""
GOPROXY="https://proxy.golang.org,direct"
GOROOT="/usr/local/go"
```

```
GOSUMDB="sum.golang.org"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
GOVCS=""
GOVERSION="go1.20.4"
GCCGO="gccgo"
GOAMD64="v1"
AR="ar"
CC="gcc"
CXX="g++"
CGO_ENABLED="1"
GOMOD="/dev/null"
GOWORK=""
CGO_CFLAGS="-O2 -g"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-O2 -g"
CGO_FFLAGS="-O2 -g"
CGO_LDFLAGS="-O2 -g"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -Wl,--no-gc-sections -fmessage-length=0
  -fdebug-prefix-map=/tmp/go-build1378738152=/tmp/go-build
  -gno-record-gcc-switches"
```

List

The *list* command provides meta information about packages. Running without any arguments shows the current packages import path. The *-f* helps to extract more information, and it can specify a format. The *text/template* package syntax can be used to specify the format.

The struct used to format has many attributes, here is a subset:

- *Dir* – directory containing package sources
- *ImportPath* – import path of package in dir
- *ImportComment* – path in import comment on package statement
- *Name* – package name
- *Doc* – package documentation string
- *Target* – install path
- *GoFiles* – list of `.go` source files

Here is an example usage:

```
$ go list -f '{{.GoFiles}}' text/template
[doc.go exec.go funcs.go helper.go option.go template.go]
```

11.3 Building and Running

To compile a program, you can use the `build` command. To compile a package, first change to the directory where the program is located and run the `build` command:

```
go build
```

You can also compile Go programs without changing the directory. To do that, you are required to specify the package location in the command line. For example, to compile `github.com/baijum/introduction` package run the command given below:

```
go build github.com/baijum/introduction
```

If you want to set the executable binary file name, use the `-o` option:

```
go build -o myprog
```

If you want to build and run at once, you can use the `"run"` command:

```
go run program.go
```

You can specify more that one Go source file in the command line arguments:

```
go run file1.go file2.go file3.go
```

Of course, when you specify more than one file names, only one ”main” function should be defined among all of the files.

Conditional Compilation

Sometimes you need to write code specific to a particular operating system. In some other case the code for a particular CPU architecture. It could be code optimized for that particular combination. The Go build tool supports conditional compilation using build constraints. The Build constraint is also known as build tag. There is another approach for conditional compilation using a naming convention for files names. This section is going to discuss both these approaches.

The build tag should be given as comments at the top of the source code. The build tag comment should start like this:

```
// +build
```

The comment should be before package documentation and there should be a line in between.

The space is *OR* and comma is *AND*. The exclamation character is stands for negation.

Here is an example:

```
// +build linux,386
```

In the above example, the file will compile on 32-bit x86 Linux system.

```
// +build linux darwin
```

The above one compiles on Linux or Darwin (Mac OS).

```
// +build !linux
```

The above runs on anything that is not Linux.

The other approach uses file naming convention for conditional compilation. The files are ignore if it doesn't match the target OS and CPU architecture, if any.

This compiles only on Linux:

```
stat_linux.go
```

This one on 64 bit ARM linux:

```
os_linux_arm64.go
```

11.4 Running Test

The Go tool has a built-in test runner. To run tests for the current package, run this command:

```
go test
```

To demonstrate the remaining commands, consider packages organized like this:

```
.
|-- hello.go
|-- hello_test.go
|-- sub1
|   |-- sub1.go
|   `-- sub1_test.go
`-- sub2
    |-- sub2.go
    `-- sub2_test.go
```

If you run **go test** from the top-level directory, it's going to run tests in that directory, and not any sub directories. You can specify directories as command line arguments to **go test** command to run tests under multiple packages simultaneously. In the above listed case, you can run all tests like this:

```
go test . ./sub1 ./sub2
```

Instead of listing each packages separates, you can use the ellipsis syntax:

```
go test ./...
```

The above command run tests under current directory and its child directories.

By default **go test** shows very few details about the tests.

```
$ go test ./...
ok      _/home/baiju/code/mypkg   0.001s
ok      _/home/baiju/code/mypkg/sub1      0.001s
--- FAIL: TestSub (0.00s)
FAIL
FAIL    _/home/baiju/code/mypkg/sub2      0.003s
```

In the above results, it shows the name of failed test. But details about other passing tests are not available. If you want to see verbose results, use the **-v** option.

```
$ go test ./... -v
=== RUN   TestHello
--- PASS: TestHello (0.00s)
PASS
ok      _/home/baiju/code/mypkg   0.001s
=== RUN   TestSub
--- PASS: TestSub (0.00s)
PASS
ok      _/home/baiju/code/mypkg/sub1      0.001s
=== RUN   TestSub
--- FAIL: TestSub (0.00s)
FAIL
FAIL    _/home/baiju/code/mypkg/sub2      0.002s
```

If you need to filter tests based on the name, you can use the `-run` option.

```
$ go test ./... -v -run Sub
testing: warning: no tests to run
PASS
ok      _/home/baiju/code/mypkg   0.001s [no tests to run]
=== RUN   TestSub
--- PASS: TestSub (0.00s)
PASS
ok      _/home/baiju/code/mypkg/sub1      0.001s
=== RUN   TestSub
--- FAIL: TestSub (0.00s)
FAIL
FAIL    _/home/baiju/code/mypkg/sub2      0.002s
```

As you can see above, the `TestHello` test was skipped as it doesn't match "Sub" pattern.

The chapter on testing has more details about writing test cases.

golangci-lint is a handy program to run various lint tools and normalize their output. This program is useful to run through continuous integration. You can download the program from here: `https://github.com/golangci/golangci-lint`

The supported lint tools include Vet, Golint, Varcheck, Errcheck, Deadcode, Gocyclo among others. *golangci-lint* allows to enable/disable the lint tools through a configuration file.

11.5 Formatting Code

Go has a recommended source code formatting. To format any Go source file to conform to that format, it's just a matter of running one command. Normally you can integrate this command with your text editor or IDE. But if you really want to invoke this program from command line, this is how you do it:

```
go fmt myprogram.go
```

In the above command, the source file name is explicitly specified. You can also give package name:

```
go fmt github.com/baijum/introduction
```

The command will format source files and write it back to the same file. Also it will list the files that is formatted.

11.6 Generating Code

If you have use case to generate Go code from a grammar, you may consider the *go generate*. In fact, you can add any command to be run before compiling the code. You can add a special comment in your Go code with this syntax:

```
//go:generate command arguments
```

For example, if you want to use *peg* (**https://github.com/pointlander/peg**), a Parsing Expression Grammar implementation, you can add the command like this:

```
//go:generate peg -output=parser.peg.go grammar.peg
```

When you build the program, the parser will be generated and will be part of the code that compiles.

11.7 Embedding Code

Go programs are normally distributed as a single binary. What if your program need some files to run. Go has a feature to embed files in the binary. You can embed any type of files, including text files and binary files. Some of the commonly embedded files are SQL, HTML, CSS, JavaScript, and images. You can embed individual files or diectories including nested sub-directories.

You need to import the *embed* package and use the `//go:embed` compiler directive to embed. Here is an example to embed an SQL file:

```
import _ "embed"

//go:embed database-schema.sql
var dbSchema string
```

As you can see, the ”embed” package is imported with a blank identifier as it is not directly used in the code. This is required to initialize the package to embed files. The variable must be at package level and not at function or method level.

The variable could be slice of bytes. This is useful for binary files. Here is an example:

```
import _ "embed"

//go:embed logo.jpg
var logo []byte
```

If you need an entire directory, you can use the **`embed.FS`** as the type:

```
import "embed"

//go:embed static
var content embed.FS
```

11.8 Displaying Documentation

Go has good support for writing documentation along with source code. You can write documentation for packages, functions and custom defined types. The Go tool can be used to display those documentation.

To see the documentation for the current packages, run this command:

```
go doc
```

To see documentation for a specific package:

```
go doc strings
```

The above command shows documentation for the ”strings” package.

```
go doc strings
```

If you want to see documentation for a particular function within that package:

```
go doc strings.ToLower
```

or a type:

```
go doc strings.Reader
```

Or a method:

```
go doc strings.Reader.Size
```

11.9 Find Suspicious Code Using Vet

There is a handy tool named *vet* to find suspicious code in your program. Your program might compile and run. But some of the results may not be desired output.

Consider this program:

Listing 11.1 : Suspicious code

```
package main

import (
	"fmt"
)

func main() {
	v := 1
	fmt.Printf("%#v %s\n", v)
}
```

If you compile and run it. It's going to be give some output. But if you observe the code, there is an unnecessary `%s` format string.

If you run **vet** command, you can see the issue:

```
$ go vet susp.go
# command-line-arguments
./susp.go:9: Printf format %s reads arg #2,
but call has only 1 arg
```

Note: The *vet* command is automatically run along with the *test* command.

11.10 Exercises

Exercise 1: Create a program with function to return "Hello, world!" and write test and run it.

hello.go:

```
package hello

// SayHello returns a "Hello word!" message
func SayHello() string {
     return "Hello, world!"
}
```

hello_test.go:

```
package hello

import "testing"

func TestSayHello(t *testing.T) {
     out := SayHello()
     if out != "Hello, world!" {
        t.Error("Incorrect message", out)
    }
}
```

To run the test:

```
go test . -v
```

Additional Exercises

Answers to these additional exercises are given in the Appendix A.

Problem 1: Write a program with exported type and methods with documentation strings. Then print the documentation using the `go doc` command.

Summary

This chapter introduced the Go tool. It explained all the major Go commands in detail and provided practical examples for each command. It covered how to build and run programs, run tests, format code, and display documentation. It also mentioned a few other useful tools.

Appendix A: Answers

Chapter 2: Quick Start

Problem 1: Write a function to check whether the first letter in a given string is capital letters in English (A,B,C,D etc).

Solution:

```
package main

import "fmt"

func StartsCapital(s string) bool {
    for _, v := range "ABCDEFGHIJKLMNOPQRSTUVWXYZ" {
        if string(s[0]) == string(v) {
            return true
        }
    }
    return false
}

func main() {
    h := StartsCapital("Hello")
    fmt.Println(h)
    w := StartsCapital("world")
    fmt.Println(w)
}
```

Problem 2: Write a function to generate Fibonacci numbers below a given value.

Solution:

```
package main

import "fmt"

func Fib(n int) {
    for i, j := 0, 1; i < n; i, j = i+j, i {
        fmt.Println(i)
    }

}

func main() {
    Fib(200)
}
```

Chapter 3: Control Structures

Problem 1: Write a program to print greetings based on time. Possible greetings are Good morning, Good afternoon and Good evening.

Solution:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Good morning!")
    case t.Hour() < 17:
        fmt.Println("Good afternoon.")
    default:
        fmt.Println("Good evening.")
    }
}
```

Problem 2: Write a program to check if the given number is divisible by 2, 3, or 5.

Solution:

```
package main

import (
    "fmt"
)
```

```
func main() {
    n := 7
    found := false
    for _, j := range []int{2, 3, 5} {
        if n%j == 0 {
            fmt.Printf("%d is a multiple of %d\n", n, j)
            found = true
        }
    }
    if !found {
        fmt.Printf("%d is not a multiple of 2, 3, or 5\n", n)
    }
}
```

Chapter 4: Data Structures

Problem 1: Write a program to record temperatures for different locations and check if it's freezing for a given place.

Solution:

```
package main

import "fmt"

type Temperature float64

func (t Temperature) Freezing() bool {
    if t < Temperature(0.0) {
        return true
    }
    return false
}

func main() {

    temperatures := map[string]Temperature{
        "New York":  9.3,
        "London":    13.5,
        "New Delhi": 31.5,
        "Montreal":  -9.0,
    }

    location := "New Delhi"
    fmt.Println(location, temperatures[location].Freezing())

    location = "Montreal"
    fmt.Println(location, temperatures[location].Freezing())

}
```

Problem 2: Create a map of world nations and details. The key could be the country name and value could be an object with details including capital, currency, and population.

Solution:

```
package main

import "fmt"

type Country struct {
    Capital    string
    Currency   string
    Popolation int
}

func main() {
    countries := map[string]Country{}
    countries["India"] = Country{Capital: "New Delhi",
        Currency: "Indian Rupee", Popolation: 1428600000}
    fmt.Printf("%#v\n", countries)
}
```

Chapter 5: Functions

Problem 1: Write a program with a function to calculate the perimeter of a circle.

Solution:

```
package main

import "fmt"

type Circle struct {
    Radius float64
}

// Area return the area of a circle for the given radius
func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}

func main() {
    c := Circle{5.0}
    fmt.Println(c.Area())
}
```

Chapter 6: Objects

Problem 1: Implement the built-in `error` interface for a custom data type. This is how the `error` interface is defined:

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

Solution:

```
package main

import "fmt"

type UnauthorizedError struct {
    UserID string
}

func (e UnauthorizedError) Error() string {
    return "User not authorised: " + e.UserID
}

func SomeAction() error {
    return UnauthorizedError{"jack"}
}

func main() {
    err := SomeAction()
    if err != nil {
        fmt.Println(err)
    }
}
```

Chapter 7: Concurrency

Problem 1: Write a program to watch log files and detect any entry with a particular word.

Solution:

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "os/signal"
    "strings"
    "time"
)

func watch(word, fp string) error {

    f, err := os.Open(fp)
    if err != nil {
        return err
    }
    r := bufio.NewReader(f)
    defer f.Close()
    for {
        line, err := r.ReadBytes('\n')
        if err != nil {
            if err.Error() == "EOF" {
                time.Sleep(2 * time.Second)
                continue
            }
            fmt.Printf("Error: %s\n%v\n", line, err)
        }
        if strings.Contains(string(line), word) {
            fmt.Printf("%s: Matched: %s\n", fp, line)
        }
        time.Sleep(2 * time.Second)
    }
}

func main() {
    word := os.Args[1]
    files := []string{}
    for _, f := range os.Args[2:len(os.Args)] {
        files = append(files, f)
        go watch(word, f)
    }
    sig := make(chan os.Signal, 1)
    done := make(chan bool)
    signal.Notify(sig, os.Interrupt)
    go func() {
        for _ = range sig {
            done <- true
        }
    }()
    <-done
}
```

Chapter 8: Packages

Problem 1: Create a package with 3 source files and another *doc.go* for documentation. The package should provide functions to calculate areas for circle, rectangle, and triangle.

Solution:

circle.go:

```
package shape

// Circle represents a circle shape
type Circle struct {
    Radius float64
}

// Area return the area of a circle
func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}
```

rectangle.go:

```
package shape

// Rectangle represents a rectangle shape
type Rectangle struct {
    Length float64
    Width float64
}

// Area return the area of a rectangle
func (r Rectangle) Area() float64 {
    return r.Length * r.Width
}
```

triangle.go:

```
package shape

// Triangle represents a rectangle shape
type Triangle struct {
    Breadth float64
    Height float64
}

// Area return the area of a triangle
func (t Triangle) Area() float64 {
    return (t.Breadth * t.Height)/2
}
```

doc.go:

```
// Package shape provides areas for different shapes
// This includes circle, rectangle, and triangle.
```

Chapter 9: Input/Output

Problem 1: Write a program to format a complex number as used in mathematics. Example: `2 + 5i`

Use a struct like this to define the complex number:

```
type Complex struct {
    Real float64
    Imaginary float64
}
```

Solution:

```
package main

import "fmt"

type Complex struct {
    Real      float64
    Imaginary float64
}

func (c Complex) String() string {
    return fmt.Sprintf("%.02f + %.02fi", c.Real, c.Imaginary)
}

func main() {
    c1 := Complex{Real: 2.3, Imaginary: 5}
    fmt.Println(c1)
}
```

Chapter 10: Testing

Problem 1: Write a test case program to fail the test and not continue with the remaining tests.

Solution:

```
package main

import "testing"

func TestHelloWorld(t *testing.T) {
    t.Errorf("First error and continue")
    t.Fatalf("Second error and not continue")
    t.Errorf("Third error does not display")
}
```

Chapter 11: Tooling

Problem 1: Write a program with exported type and methods with documentation strings. Then print the documentation using the `go doc` command.

Solution:

Here is the package definition for a circle object:

```
// Package defines a circle object
package circle

// Circle represents a circle shape
type Circle struct {
    Radius float64
}

// Area return the area of a circle
func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}
```

The docs can be accessed like this:

```
$ go doc
package circle // import "."

Package defines a circle object

type Circle struct{ ... }
```

```
$ go doc  Circle
type Circle struct {
        Radius float64
}
    Circle represents a circle shape


func (c Circle) Area() float64

$ go doc  Circle.Area
func (c Circle) Area() float64
    Area return the area of a circle
```

Further Readings

[1] Drew Neil, *Practical Vim: Edit Text at the Speed of Thought*,The Pragmatic Bookshelf, Raleigh, 2012.

[2] Erich Gamma, Ralph Johnson, John Vlissides, and Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, Massachusetts, 1994.

[3] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.

[4] Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change (XP Series)*, Addison-Wesley Professional, Massachusetts, 2nd edition, 2004.

[5] Scott Chacon and Ben Straub, *Pro Git*, Apress, New York 2014.

Index

_, 57

array, 58

blank identifier, 57
break
    label, 35
    loop, 34
    switch, 41

channel, 94
    buffered, 98
command line, 5, 113
comment, 16
constant, 54
continue, 36
    label, 36

defer, 43

fallthrough, 40
for, 18, 33
    infinite, 37
function, 21
    anonymous, 78
    value, 79
    variadic, 77

git, 5
goroutine, 94
goto, 45

if, 19, 29
    else, 30
    inline statement, 33
input
    flag, 118
    scanf, 116
input/output, 113
install
    linux, 3
    windows, 4
interface, 85
    empty, 87

lint, 139

map, 25, 63
method, 80, 81
    pointer receiver, 88
module, 108

operators, 22

package, 103
    creating, 104
    documentation, 106
    publish, 107
panic, 44
parameter, 75
    argument, 75
pointer, 68
    receiver, 82

range, 24
    loop, 38
recover, 44
reflect, 51

return, 76
    named parameter, 77

select, 97
slice, 23, 59
slide
    append, 62
string
    format, 119
struct, 67
    embedding, 67
switch, 39
    multiple cases, 41
    type, 90
    without expression, 42
sync package, Once, 100

test, 125
tool
    build, 136
    doc, 141
    env, 134
    format, 139
    help, 133
    test, 137
    version, 134
    vet, 141
type
    alias, 109
    assertion, 89
    built-in, 13
    custom, 66
type keyword, 66

var keyword, 49
variable, 14
variadic function, 77
version, 134

waitgroup, 96

zero value, 49

Colophon

The author typeset the print version of this book in Dejavu and Source Code Pro typefaces using the TEX system. LATEX macros and XƎTEX extensions were used. Many other TEX packages were also used in the preparation of this book. The book uses two variants of the Dejavu typeface: DejaVuSerif and DejaVuSans. It also uses SourceCodePro for monospaced text.

The picture used on the book cover is taken from Wikimedia Commons. The original photo is by Mykl Roventine and it is licensed under the Creative Commons CC BY 2.0 license.

The typefaces used in the cover design include Cabin, CabinCondensed, and DejaVuSans.

Without the contributions of all the developers who work on free and open source projects, it would not be possible to publish a book like this. Thank you to all of these contributors for their hard work and dedication.

Reviews and Feedback

The author would be very grateful for your reviews on Amazon and other websites. When you tweet about the book, please include the hashtag #essentialsofgo. Your blog posts, videos, or anything else you create about this book will help to spread the word. Your reviews and social media posts can help other people find the book and learn about Go programming.

Please kindly send your feedback and suggestions to the following email address: baiju.m.mail@gmail.com. You have the option to create issues in the GitHub repository at: **`https://github.com/baijum/essential-go`**. Additionally, you can propose changes by creating pull requests.

Thank you for reviewing my book. Your feedback and suggestions are very valuable and will help me improve the book. I will carefully consider your feedback and make changes as needed. If you have any further questions or suggestions, please do not hesitate to contact me.

Why should you review the book?

1. This book will help many people learn Go programming. As a fellow Go community member, you are contributing to this effort.
2. Your feedback will influence the content of the book.
3. As you study the book with a critical mind, you are enhancing your Go knowledge. The learning is more intense than reading a book casually.
4. You will be required to refer to other works to verify the content in the text. This will help you to broaden your knowledge on the subject.
5. You are supporting a fellow Go community member. This will motivate me to contribute more.
6. Your name will be mentioned in the book at the beginning in the acknowledgments, just after the preface.
7. The book is a free or open source book licensed under the Creative Commons Attribution-ShareAlike 4.0 International license.

How to review the book?

Reflect on these questions as you read the book.

1. How is the structure of the chapters in the book?
2. How are the sections under each chapter organized?
3. Is the book consistent throughout all the pages? (Consider consistency of organization, writing style, terminology usage, etc.)
4. Is there any essential point missing when explaining a particular topic?

5. Is the logical flow of introducing the concepts easy to follow?
6. Is the narrative clear, reasonable, and engaging?
7. Are the examples and exercises short and easy to understand?
8. Did you notice any sentence or paragraph that is difficult to understand?
9. Did you see an opportunity to add a side note that would be helpful to the reader?
10. Did you notice any awkward use of the English language? (You can ignore minor language issues.)
11. How can this book be made more interesting?

When you provide feedback about a chapter, please specify the chapter title. If your feedback is about a section, please include the section number and chapter title. There is no need to mention the page number, as it may change.

If you are able to send a pull request to the Git repo with your changes, that would be preferable to sending an email. If sending a pull request is not possible, you can send an email as explained here.

About the Author

Baiju Muthukadan is a software engineer from Kerala, India. He began his software development career in 2002. Over the years, Baiju has contributed to many free and open source software projects. In 2007, he authored his first book on Python. Since 2013, he has been working on various Go projects. His Twitter handle is @baijum.