



GOLANG WORKSHOP

The Go Workshop will take the pain out of learning the Go programming language (also known as Golang). It is designed to teach you to be productive in building real-world software.

Table of Contents

Introduction	4
1. Variables and Operators.....	5
What Does Go Look Like?.....	6
Exercise 1.01: Using Variables, Packages, and Functions to Print Stars	14
Activity 1.01 Defining and Printing	15
Declaring Variables	16
Declaring a Variable Using var	16
Exercise 1.02: Declaring a Variable Using var.....	17
Declaring Multiple Variables at Once with var	18
Exercise 1.03: Declaring Multiple Variables at Once with var	19
Skipping the Type or Value When Declaring Variables	20
Exercise 1.04: Skipping the Type or Value When Declaring Variables	20
Type Inference Gone Wrong.....	22
Short Variable Declaration	23
Exercise 1.05: Implementing Short Variable Declaration	23
Declaring Multiple Variables with a Short Variable Declaration	24
Exercise 1.06: Declaring Multiple Variables from a Function.....	25
Using var to Declare Multiple Variables in One Line	27
Non-English Variable Names	28
Changing the Value of a Variable	30
Exercise 1.07: Changing the Value of a Variable.....	30
Changing Multiple Values at Once	31
Exercise 1.08: Changing Multiple Values at Once	31
Operators	33
Exercise 1.09 Using Operators with Numbers.....	33
Shorthand Operator.....	37
Exercise 1.10: Implementing Shorthand Operators	37
Comparing Values.....	39
Exercise 1.11: Comparing Values	40
Zero Values	42
Exercise 1.12 Zero Values.....	42
Value versus Pointer	44
Getting a Pointer	46
Exercise 1.13: Getting a Pointer	47

Getting a Value from a Pointer	48
Exercise 1.14: Getting a Value from a Pointer.....	49
Function Design with Pointers.....	51
Exercise 1.15: Function Design with Pointers.....	51
Activity 1.02: Pointer Value Swap	54
Constants.....	54
Exercise 1.16: Constants.....	55
Enums	59
Scope.....	60
Activity 1.03: Message Bug	63
Activity 1.04: Bad Count Bug	64
<i>Summary</i>	65
2. Logic and Loops	67
<i>Introduction</i>	67
if Statements.....	68
Exercise 2.01: A Simple if Statement.....	68
if else Statements	70
Exercise 2.02: Using an if else Statement	70
else if Statements	71
Exercise 2.03: Using an else if Statement	72
The Initial if Statement	73
Exercise 2.04: Implementing the Initial if Statements.....	74
Activity 2.01: Implementing FizzBuzz.....	76
Expression switch Statements	78
Exercise 2.05: Using a switch Statement	80
Exercise 2.06: switch Statements and Multiple case Values	82
Exercise 2.07: Expressionless switch Statements	84
Loops	85
Exercise 2.08: Using the for i Loop.....	87
Exercise 2.09: Looping Over Arrays and Slices	88
The range Loop	90
Exercise 2.10: Looping Over a Map.....	90
Activity 2.02: Looping Over Map Data Using range.....	92
break and continue.....	93
Exercise 2.11: Using break and continue to Control Loops.....	93
Activity 2.03: Bubble Sort.....	96

Summary	97
3. Core Types	98
Introduction	98
True and False.....	99
Exercise 3.01: Program to Measure Password Complexity	100
Numbers.....	104
Integer	104
Floating Point.....	107
Exercise 3.02: Floating-Point Number Accuracy.....	107
Overflow and Wraparound	110
Exercise 3.03: Triggering Number Wraparound.....	110
Big Numbers.....	112
Exercise 3.04: Big Numbers.....	112
Byte.....	114
Text	114
Rune.....	116
Exercise 3.05: Safely Looping over a String.....	119
The nil Value	122
Activity 3.01: Sales Tax Calculator	122
Activity 3.02: Loan Calculator.....	123
Summary	124

Introduction

The Go Workshop will take the pain out of learning the Go programming language (also known as Golang). It is designed to teach you to be productive in building real-world software. Presented in an engaging, hands-on way, this book focuses on the features of Go that are used by professionals in their everyday work.

Each concept is broken down, clearly explained, and followed up with activities to test your knowledge and build your practical skills.

Your first steps will involve mastering Go syntax, working with variables and operators, and using core and complex types to hold data. Moving ahead, you will build your understanding of programming logic and implement Go algorithms to construct useful functions.

As you progress, you'll discover how to handle errors, debug code to troubleshoot your applications, and implement polymorphism using interfaces. The later chapters will then teach you how to manage files, connect to a database, work with HTTP servers and REST APIs, and make use of concurrent programming.

Throughout this Workshop, you'll work on a series of mini projects, including a shopping cart, a loan calculator, a working hours tracker, a web page counter, a code checker, and a user authentication system.

By the end of this book, you'll have the knowledge and confidence to tackle your own ambitious projects with Go.

1. Variables and Operators

Overview

In this chapter, you will be introduced to features of Go and will gain a basic understanding of what Go code looks like. You will also be provided with a deep understanding of how variables work and will perform exercises and activities to get hands-on and get going.

By the end of this chapter, you will be able to use variables, packages, and functions in Go. You will learn to change variable values in Go. Later in the chapter you will use operators with numbers and design functions using pointers.

Go (or golang as it's often called) is a programming language popular with developers because of how rewarding it is to use to develop software. It's also popular with companies because teams of all sizes can be productive with it. Go has also earned a reputation for consistently delivering software with exceptionally high performance.

Go has an impressive pedigree since it was created by a team from Google with a long history of building great programming languages and operating systems. They created a language that has the feel of a dynamic language such as JavaScript or PHP but with the performance and efficiency of strongly typed languages such as C++ and Java. They wanted a language that was engaging for the programmer but practical in projects with hundreds of developers.

Go is packed with interesting and unique features, such as being compiled with memory safety and channel-based concurrency. We'll explore these features in this chapter. By doing so, you'll see that their unique implementation within Go is what makes Go truly special.

Go is written in text files that are then compiled down to machine code and packaged into a single, standalone executable file. The executable is self-contained, with nothing needed to be installed first to allow it to run. Having a single file makes deploying and distributing Go software hassle-free. When compiling, you can pick one of several target operating systems, including but not limited to Windows, Linux, macOS, and Android. With Go, you write your code once and run it anywhere. Compiled languages fell out of favor because programmers hated long waits for their code to compile. The Go team knew this and built a lightning-fast compiler that remains fast as projects grow.

Go has a statically typed and type-safe memory model with a garbage collector. This combination protects developers from creating many of the most common bugs and security flaws found in software while still providing excellent performance and efficiency. Dynamically typed languages such as Ruby and Python have become popular in part because programmers felt they could be more productive if they didn't have to worry about types and memory. The downside of these languages is that they gave up performance and memory efficiency and can be more prone to type-mismatch bugs. Go has the same levels of productivity as dynamically typed languages while not giving up performance and efficiency.

A massive shift in computer performance has taken place. Going fast now means you need to be able to do as much work parallel or concurrently as possible. This change is due to the design of modern CPUs, which emphasize more cores over high clock speed. None of the currently popular programming languages have been designed to take advantage of this fact, which makes writing parallel and concurrent code in them error-prone. Go is designed to take advantage of multiple CPU cores, and it removes all the frustration and bug-filled code. Go is designed to allow any developer to easily and safely write parallel and concurrent code that enables them to take advantage of modern multicore CPUs and cloud computing – unlocking high-performance processing and massive scalability without the drama.

What Does Go Look Like?

Let's take our first look at some Go code. This code randomly prints a message to the console from a pre-defined list of messages:

```
package main

// Import extra functionality from packages
import (
    "errors"
    "fmt"
    "log"
    "math/rand"
    "strconv"
```

```
"time"
```

```
)// Taken from:
```

```
https://en.wiktionary.org/wiki/Hello\_World#Translations
```

```
var helloList = []string{
```

```
    "Hello, world",
```

```
    "Καλημέρα κόσμε",
```

```
    "こんにちは世界",
```

```
    "سلام دنیا",
```

```
    "Привет, мир",
```

```
}
```

The **main()** function is defined as:

```
func main() {
```

```
    // Seed random number generator using the current time
```

```
    rand.Seed(time.Now().UnixNano())
```

```
    // Generate a random number in the range of out list
```

```
    index := rand.Intn(len(helloList))
```

```
    // Call a function and receive multiple return values
```

```
    msg, err := hello(index)
```

```
    // Handle any errors
```

```
    if err != nil {
```

```
        log.Fatal(err)
```

```
    }
```

```
    // Print our message to the console
```

```
    fmt.Println(msg)
```



```
}
```

Let's consider the **hello()** function:

```
func hello(index int) (string, error) {  
    if index < 0 || index > len(helloList)-1 {  
        // Create an error, convert the int type to a string  
        return "", errors.New("out of range: " + strconv.Itoa(index))  
    }  
    return helloList[index], nil  
}
```

Now, let's step through this code piece by piece.

At the top of our script is the following:

```
package main
```

This code is our package declaration. All Go files must start with one of these. If you want to run the code directly, you'll need to name it **main**. If you don't name it **main**, then you can use it as a library and import it into other Go code. When creating an importable package, you can give it any name. All Go files in the same directory are considered part of the same package, which means all the files must have the same package name.

In the following code, we're importing code from packages:

```
// Import extra functionality from packages  
  
import (  
    "errors"  
    "fmt"  
    "log"  
    "math/rand"
```

```
"strconv"
```

```
"time"
```

```
)
```

In this example, the packages are all from Go's standard library. Go's standard library is very high-quality and comprehensive. You are strongly recommended to maximize your use of it. You can tell if a package isn't from the standard library because it'll look like a URL, for example, **github.com/fatih/color**.

Go has a module system that makes using external packages easy. To use a new module, add it to your import path. Go will automatically download it for you the next time you build code.

Imports only apply to the file they're declared in, which means you must declare the same imports over and over in the same package and project. Fear not, though you don't need to do this by hand. There are many tools and Go editors that automatically add and remove the imports for you:

```
// Taken from:
```

```
https://en.wiktionary.org/wiki/Hello\_World#Translations
```

```
var helloList = []string{
```

```
    "Hello, world",
```

```
    "Καλημέρα κόσμε",
```

```
    "こんにちは世界",
```

```
    "سلام دنیا",
```

```
    "Привет, мир",
```

```
}
```

Here, we're declaring a global variable, which is a list of strings, and initializing it with data. The text or strings in Go support multi-byte UTF-8 encoding, making them safe for any language. The type of list we're using here is called a slice. There are three types of lists in Go: slices, arrays, and maps. All three are collections of keys and values, where you use the key to get a value from the collection. Slice and array collections use a number as the key. The first key is always 0 in slices and

arrays. Also, in slices and arrays, the numbers are contiguous, which means there is never a break in the sequence of numbers. With the **map** type, you get to choose the **key** type. You use this when you want to use some other data to look up the value in the map. For example, you could use a book's ISBN to look up its title and author:

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

Here, we're declaring a function. A function is some code that runs when called. You can pass data in the form of one or more variables to a function and optionally receive one or more variables back from it. The **main()** function in Go is special. The **main()** function is the entry point of your Go code. When your code runs, Go automatically calls **main** to get things started:

```
// Seed random number generator using the current time  
  
rand.Seed(time.Now().UnixNano())  
  
// Generate a random number in the range of our list  
  
index := rand.Intn(len(helloList))
```

In the preceding code, we are generating a random number. The first thing we need to do is ensure it's a good random number, so to do that, we must "seed" the random number generator. We seed it using the current time formatted to a Unix timestamp with nanoseconds. To get the time, we call the **Now** function in the **time** package.

The **Now** function returns a struct type variable. Structs are a collection of properties and functions, a little like objects in other languages. In this case, we are calling the **UnixNano** function on that struct straight away. The **UnixNano** function returns a variable of the **int64** type, which is a 64-bit integer or, more simply, a number. This number is passed into **rand.Seed**. The **rand.Seed** function accepts an **int64** variable as its input. Note that the type of the variable from **time.UnixNano** and **rand.Seed** must be the same. Now, we've successfully seeded the random number generator.

What we want is a number we can use to get a random message. We'll use **rand.Intn** for this job. This function gives us a random number between 0 and 1, minus the number you pass in. This may sound a bit

strange, but it works out perfectly for what we're trying to do. This is because our list is a slice where the keys start from 0 and increment by 1 for each value. This means the last index is 1 less than the length of the slice.

To show you what this means, here is some simple code:

```
package main

import (
    "fmt"
)

func main() {
    helloList := []string{
        "Hello, world",
        "Καλημέρα κόσμε",
        "こんにちは世界",
        "سلام دنیا",
        "Привет, мир",
    }

    fmt.Println(len(helloList))

    fmt.Println(helloList[len(helloList)-1])

    fmt.Println(helloList[len(helloList)])
}
```

This code prints the length of the list and then uses that length to print the last element. To do that, we must subtract 1, otherwise, we'd get an error, which is what the last line causes:

```
~/src/The-Go-Workshop/Chapter01/Example01.01 go run .
5
Привет, мир
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
    /home/sam/src/The-Go-Workshop/Chapter01/Example01.01/main.go:17 +0x12c
exit status 2
```

Figure 1.01: Output displaying an error

Once we've generated our random number, we assign it to a variable. We do this with the `:=` notation, which is a very popular shortcut in Go. It tells the compiler to go ahead and assign that value to my variable and select the appropriate type for that value. This shortcut is one of the many things that makes Go feel like a dynamically typed language:

```
// Call a function and receive multiple return values
```

```
msg, err := hello(index)
```

We then use that variable to call a function named **hello**. We'll look at **hello** in just a moment. The important thing to note is that we're receiving two values back from the function and we're able to assign them to two new variables, **msg** and **err**, using the `:=` notation:

```
func hello(index int) (string, error) {
...
}
```

This code is the definition of the **hello** function; we're not showing the body for now. A function acts as a unit of logic that's called when and as often as is needed. When calling a function, the code that calls it stops running and waits for the function to finish running. Functions are a great tool for keeping your code organized and understandable. In the signature of **hello**, we've defined that it accepts a single **int** value and that it returns a **string** and an **error** value. Having an **error** as your last return value is a very common thing to have in Go. The code between the `{}` is the body of the function. The following code is what's run when the function's called:

```
if index < 0 || index > len(helloList)-1 {
```

```
// Create an error, convert the int type to a string
```

```
return "", errors.New("out of range: " + strconv.Itoa(index))
}

return helloList[index], nil
```

Here, we are inside the function; the first line of the body is an **if** statement. An **if** statement runs the code inside its **{}** if its Boolean expression is true. The Boolean expression is the logic between the **if** and the **{**. In this case, we're testing to see if the passed **index** variable is greater than 0 or less than the largest possible slice index key.

If the Boolean expression were to be true, then our code would return an empty **string** and an **error**. At this point, the function would stop running, and the code that called the function would continue to run. If the Boolean expression were not true, its code would be skipped over, and our function would return a value from **helloList** and **nil**. In Go, **nil** represents something with no value and no type:

```
// Handle any errors
if err != nil {
    log.Fatal(err)
}
```

After we've run **hello**, the first thing we need to do is check to see if it ran successfully. We do this by checking the **error** value stored in **err**. If **err** is not equal to **nil**, then we know we have an error. Then, we call **log.Fatal**, which writes out a logging message and kills our app. Once the app's been killed, no more code runs:

```
// Print our message to the console
fmt.Println(msg)
```

If there is no error, then we know that **hello** ran successfully and that the value of **msg** can be trusted to hold a valid value. The final thing we need to do is print the message to the screen via the Terminal.

Here's how that looks:

```
~/src/Th...op/Ch...01/Example01.02 go run .  
سلام دنيا  
~/src/Th...op/Ch...01/Example01.02 go run .  
Привет, мир  
~/src/Th...op/Ch...01/Example01.02 go run .  
Καλημέρα κόσμε  
~/src/Th...op/Ch...01/Example01.02 go run .  
こんにちは世界
```

Figure 1.02: Output displaying valid values

In this simple Go program, we've been able to cover a lot of key concepts that we'll explore in full in the coming chapters.

Exercise 1.01: Using Variables, Packages, and Functions to Print Stars

In this exercise, we'll use some of what we learned about in the preceding example to print a random number, between 1 and 5, of stars (*) to the console. This exercise will give you a feel of what working with Go is like and some practice with using the features of Go we'll need going forward. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Now, add the imports we'll use in this file:

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

4. Create a **main()** function:

```
func main() {
```

5. Seed the random number generator:

```
rand.Seed(time.Now().UnixNano())
```

6. Generate a random number between 0 and then add 1 to get a number between 1 and 5:

```
r := rand.Intn(5) + 1
```

7. Use the string repeater to create a string with the number of stars we need:

```
stars := strings.Repeat("*", r)
```

8. Print the string with the stars to the console with a new line character at the end and close the **main()** function:

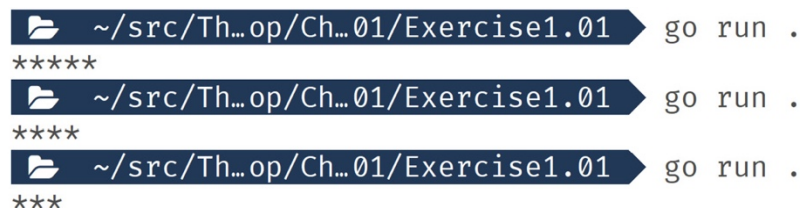
```
fmt.Println(stars)
```

```
}
```

9. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:



```
~/src/Th...op/Ch...01/Exercise1.01 go run .  
*****  
~/src/Th...op/Ch...01/Exercise1.01 go run .  
****  
~/src/Th...op/Ch...01/Exercise1.01 go run .  
***
```

Figure 1.03: Output displaying stars

In this exercise, we created a runnable Go program by defining the **main** package with a **main()** function in it. We used the standard library by adding imports to packages. Those packages helped us generate a random number, repeat strings, and write to the console.

Activity 1.01 Defining and Printing

In this activity, we are going to create a medical form for a doctor's office to capture a patient's name, age, and whether they have a peanut allergy:

1. Create a variable for the following:

1. First name as a string
 2. Family name as a string
 3. Age as an **int**
 4. Peanut allergy as a **bool**
2. Ensure they have an initial value.
3. Print the values to the console.

The following is the expected output:

```
~/src/Th...op/Ch...01/Activity01.01 go run .  
Bob  
Smith  
34  
false
```

Figure 1.04: Expected output after assigning the variables

Note

The solution for this activity can be found via [this link](#).

Next, we'll start going into detail about what we've covered so far, so don't worry if you are confused or have a question about what you've seen so far.

Declaring Variables

Now that you've had an overview of Go and completed your first exercise, we're going to dive deep. Our first stop on the journey is variables.

A variable holds data for you temporarily so you can work with it. When you declare a variable, it needs four things: a statement that you are declaring a variable, a name for the variable, the type of data it can hold, and an initial value for it. Fortunately, some of the parts are optional, but that also means there's more than one way of defining a variable.

We'll now cover all the ways you can declare a variable.

Declaring a Variable Using var

Using **var** is the foundational way to declare a variable. Every other way we'll cover is a variation of this approach, typically by omitting parts of this definition. A full **var** definition with everything in place looks like this:

```
var foo string = "bar"
```

The key parts are **var**, **foo**, **string**, and **= "bar"**:

- **var** is our declaration that we are defining a variable.
- **foo** is the name of the variable.
- **string** is the type of the variable.
- **= "bar"** is its initial value.

Exercise 1.02: Declaring a Variable Using var

In this exercise, we'll declare two variables using the full **var** notation. Then, we'll print them to the console. You'll see that you can use the var notation anywhere in your code, which isn't true for all variable declaration notations. Let's get started:

1. Create a new folder and add a **main.go** file to it:
2. In **main.go**, add the main package name to the top of the file:

```
package main
```

3. Add the imports:

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

4. Declare a variable at the package-level scope. We'll cover what scopes are in detail later:

```
var foo string = "bar"
```

5. Create the **main()** function:

```
func main() {
```

6. Declare another variable using **var** in our function:

```
    var baz string = "qux"
```

7. Print both variables to the console:

```
    fmt.Println(foo, baz)
```

8. Close the **main()** function:

```
}
```

9. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

```
bar qux
```

In this example, **foo** is declared at the package level while **baz** is declared at the function level. Where a variable is declared is important because where you declare a variable also limits what notation you can use to declare it.

Next, we'll look at another way to use the **var** notation.

Declaring Multiple Variables at Once with var

We can use a single **var** declaration to define more than one variable. Using this method is common when declaring package-level variables. The variables don't need to be of the same type, and they can all have their own initial values. The notation looks like this:

```
Var (
```

```
<name1> <type1> = <value1>
```

```
<name2> <type2> = <value2>
```

```
...
```

```
<nameN> <typeN> = <valueN>
```

```
)
```

You can have multiple of these types of declaration, which is a nice way to group related variables, thereby making your code more readable. You can use this notation in functions, but it's rare to see it used there.

Exercise 1.03: Declaring Multiple Variables at Once with var

In this exercise, we'll declare multiple variables using one `var` statement, each with a different type and initial value. Then, we'll print the value of each variable to the console. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Add the imports:

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

4. Start the **var** declaration:

```
var (
```

5. Define three variables:

```
    Debug bool = false  
    LogLevel string = "info"  
    startUpTime time.Time = time.Now()
```

6. Close the **var** declaration:

```
)
```

7. In the **main()** function, print each variable to the console:

```
func main() {  
    fmt.Println(Debug, LogLevel, startUpTime)  
}
```

8. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

```
~/src/Th...op/Ch...01/Exercise1.03 go run .  
false info 2019-11-16 14:08:51.1632775 +0000 GMT m=+0.000162601
```

Figure 1.05: Output displaying three variable values

In this exercise, we declared three variables using a single var statement. Your output looks different for the **time.Time** variable, but that's correct. The format is the same, but the time itself is different.

Using the var notation like this is a good way to keep your code well organized and to save you some typing.

Next, we'll start removing some of the optional parts of the var notation.

Skipping the Type or Value When Declaring Variables

In real-world code, it's not common to use the full var notation. There are a few cases where you need to define a package-level variable with an initial value and tightly control its type. In those cases, you need the full notation. It'll be obvious when this is needed as you'll have a type mismatch of some kind, so don't worry too much about this for now. The rest of the time, you'll remove an optional part or use the short variable declaration.

You don't need to include both the type and the initial value when declaring a variable. You can use just one or the other; Go works out the rest. If you have a type in the declaration but no initial value, Go uses the zero value for the type you picked. We'll talk more about what a zero value is in a later chapter. On the other hand, if you have an initial value and no type, Go has a ruleset for how to infer the types that are needed from the literal value you use.

Exercise 1.04: Skipping the Type or Value When Declaring Variables

In this exercise, we'll update our previous exercise to skip the optional initial values or type declarations from our variable declaration. Then, we'll print the values to the console, as we did previously, to show that the result is the same. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

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

4. Start the multi-variable declaration:

```
var (
```

5. The **bool** in the first exercise has an initial value of false. That's a **bool**'s zero value, so we'll drop the initial value from its declaration:

```
    Debug bool
```

6. The next two variables both have a non-zero value for their type, so we'll drop their type declaration:

```
    LogLevel = "info"  
    startUpTime = time.Now()
```

7. Close the var declaration:

```
)
```

8. In the **main()** function, print out each variable:

```
func main() {  
    fmt.Println(Debug, LogLevel, startUpTime)  
}
```

9. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

```
~/src/Th...op/Ch...01/Exercise1.04 go run .  
false info 2019-11-16 14:51:16.3478841 +0000 GMT m=+0.000197801
```

Figure 1.06: Output displaying variable values despite not mentioning the type while declaring the variables

In this exercise, we were able to update the previous code to use a much more compact variable declaration. Declaring variables is something you'll have to do a lot, and not having to use the notation makes for a better experience when writing code.

Next, we'll look at a situation where you can't skip any of the parts.

Type Inference Gone Wrong

There are times when you'll need to use all the parts of the declaration, for example, when Go isn't able to guess the correct type you need. Let's take a look at an example of this:

```
package main  
  
import "math/rand"  
  
func main() {  
  
    var seed = 1234456789  
  
    rand.Seed(seed)  
  
}
```

The following is the output:

```
~/src/Th...op/Ch...01/Example01.03 go run .  
# github.com/PacktWorkshops/The-Go-Workshop/Chapter01/Example01.03  
./main.go:7:11: cannot use seed (type int) as type int64 in argument to rand.Seed
```

Figure 1.07: Output showing an error

The issue here is that **rand.Seed** requires a variable of the **int64** type. Go's type inference rules interoperate a whole number, such as the one we used as an **int**. We'll look at the difference between them in more detail in a later chapter. To resolve this, we will add **int64** to the declaration. Here's how that looks:

```
package main
```

```
import "math/rand"

func main() {

    var seed int64 = 1234456789

    rand.Seed(seed)

}
```

Next, we'll look at an even quicker way to declare variables.

Short Variable Declaration

When declaring variables in functions and functions only, we can use the `:=` shorthand. This shorthand allows us to make our declarations even shorter. It does this by allowing us to not have to use the **var** keyword and by always inferring the type from a required initial value.

Exercise 1.05: Implementing Short Variable Declaration

In this exercise, we'll update our previous exercise to use a short variable declaration. Since you can only use a short variable declaration in a function, we'll move our variable out of the package scope. Where before **Debug** had a type but no initial value, we'll switch it back so that it has an initial value since that's required when using a short variable declaration. Finally, we'll print it to the console. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

```
import (

    "fmt"

    "time"

)
```

4. Create the **main()** function:


```
func main() {
```

5. Declare each variable using the short variable declaration notation:

```
    Debug := false
```

```
    LogLevel := "info"
```

```
    startUpTime := time.Now()
```

6. Print the variables to the console:

```
    fmt.Println(Debug, LogLevel, startUpTime)
```

```
}
```

7. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

```
~/src/Th...op/Ch...01/Exercise1.05 go run .  
false info 2019-11-16 15:35:17.2406485 +0000 GMT m=+0.000170701
```

Figure 1.08: Output displaying the variable values that were printed after using short variable declaration notation

In this exercise, we updated our previous code to use a very compact way to declare variables when we have an initial value to use.

The `:=` shorthand is very popular with Go developers and the most common way in which variables get defined in real-world Go code. Developers like how it makes their code concise and compact while still being clear as to what's happening.

Another shortcut is declaring multiple variables on the same line.

Declaring Multiple Variables with a Short Variable Declaration

It's possible to declare multiple variables at the same time using a short variable declaration. They must all be on the same line, and each variable must have a corresponding initial value. The notation looks like `<var1>, <var2>, ..., <varN> := <val1>, <val2>, ..., <valN>`. The variable names are on the left-hand side of the `:=`, separated by a `,`. The initial values are on the right-hand side of the `:=` again, each separated by a `,`.

The leftmost variable name gets the leftmost value. There must be an equal number of names and values.

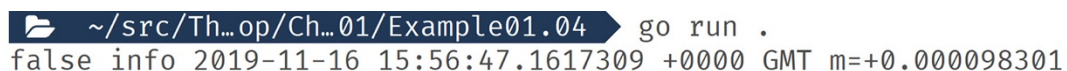
Here is an example that uses our previous exercise's code:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    Debug, LogLevel, startUpTime := false, "info", time.Now()
    fmt.Println(Debug, LogLevel, startUpTime)
}
```

The following is the output:



```
~/src/Th...op/Ch...01/Example01.04 go run .
false info 2019-11-16 15:56:47.1617309 +0000 GMT m=+0.000098301
```

Figure 1.09: Example output displaying the variable values for the program with a variable declaring function

Sometimes, you do see real-world code like this. It's a little hard to read, so it's not common to see it in terms of literal values. This doesn't mean this isn't common since it's very common when calling functions that return multiple values. We'll cover this in detail when we look at functions in a later chapter.

Exercise 1.06: Declaring Multiple Variables from a Function

In this exercise, we'll call a function that returns multiple values, and we'll assign each value to a new variable. Then, we'll print the values to the console. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

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

4. Create a function that returns three values:

```
func getConfig() (bool, string, time.Time) {
```

5. In the function, return three literal values, each separated by a comma:

```
    return false, "info", time.Now()
```

6. Close the function:

```
}
```

7. Create the **main()** function:

```
func main() {
```

8. Using a short variable declaration, capture the values returned from the function's three new variables:

```
    Debug, LogLevel, startUpTime := getConfig()
```

9. Print the three variables to the console:

```
    fmt.Println(Debug, LogLevel, startUpTime)
```

10. Close the **main()** function:

```
}
```

11. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

```
~/src/Th...op/Ch...01/Exercise01.06 go run .  
false info 2019-11-16 16:38:29.2623608 +0000 GMT m=+0.000117201
```

Figure 1.10: Output displaying the variable values for the program with the variable declaring function

In this exercise, we were able to call a function that returned multiple values and capture them using a short variable declaration in one line. If we used the **var** notation, it would look like this:

```
var (  
    Debug bool  
    LogLevel string  
    startUpTime time.Time  
)  
Debug, LogLevel, startUpTime = getConfig()
```

Short variable notation is a big part of how Go has the feel of a dynamic language.

We're not quite done with **var** yet, though. It still has a useful trick up its sleeve.

Using var to Declare Multiple Variables in One Line

While it's more common to use a short variable declaration, you can use **var** to define multiple variables on a single line. One limitation of this is that, when declaring the type, all the values must have the same type. If you use an initial value, then each value infers its type from the literal value so that they can differ. Here's an example:

```
package main  
  
import (  
    "fmt"  
    "time"  
)
```

```

func getConfig() (bool, string, time.Time) {
    return false, "info", time.Now()
}

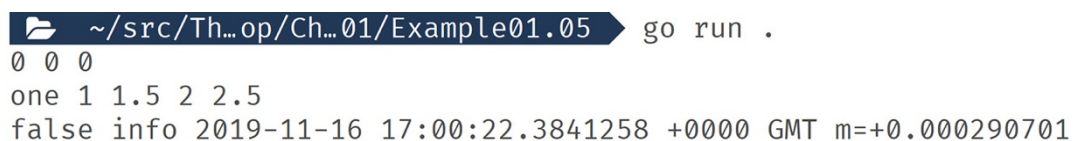
func main() {
    // Type only
    var start, middle, end float32
    fmt.Println(start, middle, end)

    // Initial value mixed type
    var name, left, right, top, bottom = "one", 1, 1.5, 2, 2.5
    fmt.Println(name, left, right, top, bottom)

    // works with functions also
    var Debug, LogLevel, startUpTime = getConfig()
    fmt.Println(Debug, LogLevel, startUpTime)
}

```

The following is the output:



```

~/.src/Th...op/Ch...01/Example01.05 go run .
0 0 0
one 1 1.5 2 2.5
false info 2019-11-16 17:00:22.3841258 +0000 GMT m=+0.000290701

```

Figure 1.11: Output displaying variable values

Most of these are more compact when using a short variable declaration. This fact means they don't come up in real-world code much. The exception is the type-only example. This notation can be useful when you need many variables of the same type, and you need to control that type carefully.

Non-English Variable Names

Go is a UTF-8 compliant language, which means you can define variables' names using alphabets other than the Latin alphabet that, for

example, English uses. There are some limitations regarding what the name of a variable can be. The first character of the name must be a letter or `_`. The rest can be a mixture of letters, numbers, and `_`. Let's have a look at what this looks like:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    デバッグ := false

    日志级别 := "info"

    现在时间 := time.Now()

    _A1_Μεϊγμα := "

    fmt.Println(デバッグ, 日志级别, 现在时间, _A1_Μεϊγμα)
}
```

The following is the output:

```
~/src/Th...op/Ch...01/Example01.06 go run .
false info 2019-11-16 17:17:19.963412 +0000 GMT m=+0.000095801 ✓
```

Figure 1.12: Output showing variable values

Note

Languages and Language: *Not all programming languages allow you to use UTF-8 characters as variables and function names. This feature could be one of the reasons why Go has become so popular in Asian countries, particularly in China.*

Changing the Value of a Variable

Now that we've defined our variables, let's see what we can do with them. First, let's change the value from its initial value. To do that, we use similar notation to when we set an initial value. This looks like **<variable> = <value>**.

Exercise 1.07: Changing the Value of a Variable

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

```
import "fmt"
```

4. Create the **main()** function:

```
func main() {
```

5. Declare a variable:

```
    offset := 5
```

6. Print the variable to the console:

```
    fmt.Println(offset)
```

7. Change the value of the variable:

```
    offset = 10
```

8. Print it to the console again and close the **main()** function:

```
    fmt.Println(offset)
```

```
}
```

9. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output before changing the variable's value:

5

10

In this example, we've changed the value of `offset` from its initial value of **5** to **10**. Anywhere you use a raw value, such as **5** and **10** in our example, you can use a variable. Here's how that looks:

```
package main

import "fmt"
var defaultOffset = 10

func main() {
    offset := defaultOffset

    fmt.Println(offset)

    offset = offset + defaultOffset

    fmt.Println(offset)
}
```

The following is the output after changing the variable's value:

10

20

Next, we'll look at how we can change multiple variables in a one-line statement.

Changing Multiple Values at Once

In the same way that you can declare multiple variables in one line, you can also change the value of more than one variable at a time. The syntax is similar, too; it looks like `<var1>, <var2>, ..., <varN> = <val1>, <val2>, ..., <valN>`.

Exercise 1.08: Changing Multiple Values at Once

In this exercise, we'll define some variables and use a one-line statement to change their values. Then, we'll print their new values to the console. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:


```
package main
```

3. Import the packages we'll need:

```
import "fmt"
```

4. Create the **main()** function:

```
func main() {
```

5. Declare our variables with an initial value:

```
    query, limit, offset := "bat", 10, 0
```

6. Change each variable's values using a one-line statement:

```
    query, limit, offset = "ball", offset, 20
```

7. Print the values to the console and close the **main()** function:

```
    fmt.Println(query, limit, offset)
```

```
}
```

8. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output showing the changed variable values using a single statement:

```
ball 0 20
```

In this exercise, we were able to change multiple variables in a single line. This approach would also work when calling functions, just as it does with a variable declaration. You need to be careful with a feature like this to ensure that, first and foremost, your code is easy to read and understand. If using a one-line statement like this makes it hard to know what the code is doing, then it's better to take up more lines to write the code.

Next, we'll look at what operators are and how they can be used to change your variables in interesting ways.

Operators

While variables hold the data for your application, they become truly useful when you start using them to build the logic of your software. Operators are the tools you use to work with your software's data. With operators, you can compare data to other data. For example, you can check whether a price is too low or too high in a trading application. You can also use operators to manipulate data. For example, you can use operators to add the costs of all the items in a shopping cart to get the total price.

The following list mentions groups of operators:

- Arithmetic operators

Used for math-related tasks such as addition, subtraction, and multiplication.

- Comparison operators

Used to compare two values; for example, are they are equal, not equal, less than, or greater than each other.

- Logical operators

Used with Boolean values to see whether they are both true, only one is true, or whether a **bool** is false.

- Address operators

We'll cover these in detail soon when we look at pointers. These are used to work with them.

- Receive operators

Used when working with Go channels, which we'll cover in a later chapter.

Exercise 1.09 Using Operators with Numbers

In this exercise, we are going to simulate a restaurant bill. To build our simulation, we'll need to use mathematic and comparison operators. We'll start by exploring all the major uses for operators.

In our simulation, we'll sum everything together and work out the tip based on a percentage. Then, we'll use a comparison operator to see whether the customer gets a reward. Let's get started:

Note

We have considered US Dollar as the currency for this exercise. You may consider any currency of your choice; the main focus here is the operations.

1. Create a new folder and add a **main.go** file to it:
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages you'll need:

```
import "fmt"
```

4. Create the **main()** function:

```
func main() {
```

5. Create a variable to hold the total. For this item on the bill, the customer purchased 2 items that cost 13 USD. We use `*` to do the multiplication. Then, we print a subtotal:

```
// Main course
```

```
var total float64 = 2 * 13
```

```
fmt.Println("Sub :", total)
```

6. Here, they purchased 4 items that cost 2.25 USD. We use multiplication to get the total of these items and then use `+` to add it to the previous total value and then assign that back to the total:

```
// Drinks
```

```
total = total + (4 * 2.25)
```

```
fmt.Println("Sub :", total)
```

7. This customer is getting a discount of 5 USD. Here, we use the `-` to subtract 5 USD from the total:

```
// Discount
```

```
total = total - 5
```

```
fmt.Println("Sub :", total)
```

8. Then, we use multiplication to calculate a 10% tip:

```
// 10% Tip
```

```
tip := total * 0.1
```

```
fmt.Println("Tip :", tip)
```

9. Finally, we add the tip to the total:

```
total = total + tip
```

```
fmt.Println("Total:", total)
```

10. The bill will be split between two people. Use / to divide the total into two parts:

```
// Split bill
```

```
split := total / 2
```

```
fmt.Println("Split:", split)
```

11. Here, we'll calculate whether the customer gets a reward. First, we'll set the **visitCount** and then add 1 USD to this visit:

```
// Reward every 5th visit
```

```
visitCount := 24
```

```
visitCount = visitCount + 1
```

12. Then, we'll use % to give us any remainder after dividing the **visitCount** by 5 USD:

```
remainder := visitCount % 5
```

13. The customer gets a reward on every fifth visit. If the remainder is 0, then this is one of those visits. Use the == operator to check whether the remainder is 0:

```
if remainder == 0 {
```

14.If it is, print a message that they get a reward:

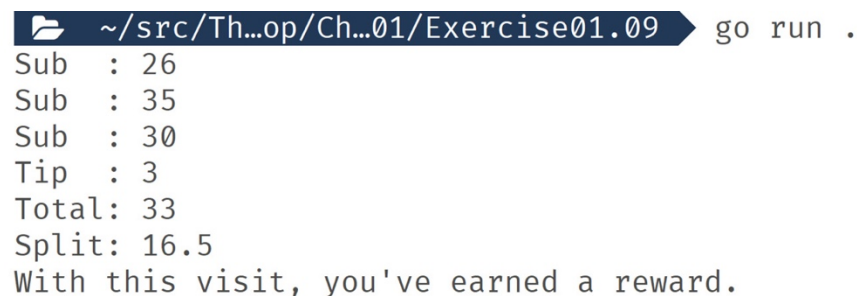
```
    fmt.Println("With this visit, you've earned a reward.")
}

}
```

15.Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:



```
~/src/Th...op/Ch...01/Exercise01.09 go run .
Sub : 26
Sub : 35
Sub : 30
Tip : 3
Total: 33
Split: 16.5
With this visit, you've earned a reward.
```

Figure 1.13: Output of operators used with numbers

In this exercise, we used the math and comparison operators with numbers. They allowed us to model a complex situation – calculating a restaurant bill. There are lots of operators and which ones you can use vary with the different types of values. For example, as well as there being an addition operator for numbers, you can use the + symbol to join strings together. Here's this in action:

```
package main

import "fmt"

func main() {

    givenName := "John"

    familyName := "Smith"

    fullName := givenName + " " + familyName

    fmt.Println("Hello,", fullName)
```

```
}
```

The following is the output:

Hello, John Smith

For some situations, there are some shortcuts we can make with operators. We'll go over this in the next section.

Note

Bitwise Operators: *Go has all the familiar bitwise operators you'd find in programming languages. If you know what bitwise operators are, then there will be no surprises here for you. If you don't know what bitwise operators are, don't worry – they aren't common in real-world code.*

Shorthand Operator

There are a few shorthand assignment operators when you want to perform operations to an existing value with its own value. For example:

- --: Reduce a number by 1
- ++: Increase a number by 1
- +=: Add and assign
- -=: Subtract and assign

Exercise 1.10: Implementing Shorthand Operators

In this exercise, we'll use some examples of operator shorthand to show how they can make your code more compact and easier to write. We'll create some variables then use shorthand to change them, printing them out as we go. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

```
import "fmt"
```

4. Create the **main()** function:

```
func main() {
```

5. Create a variable with an initial value:

```
count := 5
```

6. We'll add to it and then assign the result back to itself. Then, we'll print it out:

```
count += 5
```

```
fmt.Println(count)
```

7. Increment the value by 1 and then print it out:

```
count++
```

```
fmt.Println(count)
```

8. Decrement it by 1 and then print it out:

```
count--
```

```
fmt.Println(count)
```

9. Subtract and assign the result back to itself. Print out the new value:

```
count -= 5
```

```
fmt.Println(count)
```

10. There is also a shorthand that works with strings. Define a string:

```
name := "John"
```

11. Next, we'll append another string to the end of it and then print it out:

```
name += " Smith"
```

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

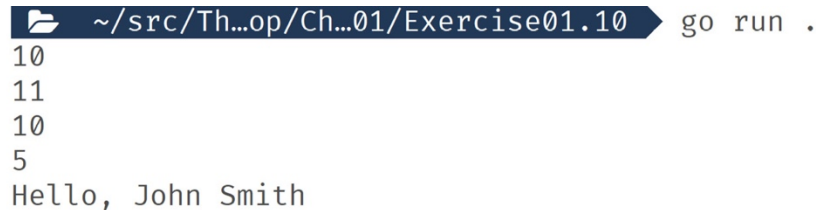
12. Close the **main()** function:

```
}
```

13. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:



```
~/src/Th...op/Ch...01/Exercise01.10 go run .
10
11
10
5
Hello, John Smith
```

Figure 1.14: Output using shorthand operators

In this exercise, we used some shorthand operators. One set focused on modification and then assignment. This type of operation is common, and having these shortcuts makes coding more engaging. The other operators are increment and decrement. These are useful in loops when you need to step over data one at a time. These shortcuts make it clear what you're doing to anyone who reads your code.

Next, we'll look at comparing values to each other in detail.

Comparing Values

Logic in applications is a matter of having your code make a decision. These decisions get made by comparing the values of variables to the rules you define. These rules come in the form of comparisons. We use another set of operators to make these comparisons. The result of these comparisons is always true or false. You'll also often need to make multiples of these comparisons to make a single decision. To help with that, we have logical operators.

These operators, for the most part, work with two values and always result in a Boolean value. You can only use logical operators with Boolean values. Let's take a look at comparison operators and logical operators in more detail:

Comparison Operators

- `==` True if two values are the same
- `!=` True if two values are not the same
- `<` True if the left value is less than the right value

- `<=` True if the left value is less or equal to the right value
- `>` True if the left value is greater than the right value
- `>=` True if the left value is greater than or equal to the right value

Logical Operators

- `&&` True if the left and right values are both true
- `||` True if one or both the left and right values are true
- `!` This operator only works with a single value and results in true if the value is false

Exercise 1.11: Comparing Values

In this exercise, we'll use comparison and logical operators to see what Boolean results we get when testing different conditions. We are testing to see what level of membership a user has based on the number of visits they've had.

Our membership levels are as follows:

- Sliver: Between 10 and 20 visits inclusively
- Gold: Between 21 and 30 visits inclusively
- Platinum: Over 30 visits

Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

```
import "fmt"
```

4. Create the **main()** function:

```
func main() {
```

5. Define our **visits** variable and initialize it with a value:

```
    visits := 15
```

6. Use the equals operator to see if this is their first visit. Then, print the result to the console:

```
fmt.Println("First visit :", visits == 1)
```

7. Use the not equal operator to see if they are a returning visitor:

```
fmt.Println("Return visit :", visits != 1)
```

8. Let's check whether they are a Silver member using the following code:

```
fmt.Println("Silver member :", visits >= 10 && visits < 21)
```

9. Let's check whether they are a Gold member using the following code:

```
fmt.Println("Gold member :", visits > 20 && visits <= 30)
```

10. Let's check whether they are a Platinum member using the following code:

```
fmt.Println("Platinum member :", visits > 30)
```

11. Close the **main()** function:

```
}
```

12. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

```
~/src/Th...op/Ch...01/Exercise01.11 go run .
First visit      : false
Return visit     : true
Silver member    : true
Gold member      : false
Platinum member  : false
```

Figure 1.15: Output displaying the comparison result

In this exercise, we used comparison and logical operators to make decisions about data. You can combine these operators in an unlimited number of ways to express almost any type of logic your software needs to make.

Next, we'll look at what happens when you don't give a variable an initial value.

Zero Values

The zero value of a variable is the empty or default value for that variable's type. Go has a set of rules stating that the zero values are for all the core types. Let's take a look:

Type	Zero Value
bool	false
Numbers (integers and floats)	0
String	"" (empty string)
pointers, functions, interfaces, slices, channels, and maps	nil (covered in detail in later chapters)

Figure 1.16: Variable types and their zero values

There are other types, but they are all derived from these core types, so the same rules still apply.

We'll look at the zero values of some types in the upcoming exercise.

Exercise 1.12 Zero Values

In this example, we'll define some variables without an initial value. Then, we'll print out their values. We're using **fmt.Printf** to help us in this exercise as we can get more detail about a value's type. **fmt.Printf** uses a template language that allows us to transform passed values. The substitution we're using is **%#v**. This transformation is a useful tool for showing a variable's value and type. Some other common substitutions you can try are as follows:

Substitution	Formatting
%v	Any value. Use this if you don't care about the type you're printing.
%+v	Values with extra information, such as struct field names.
%#v	Go syntax, such as %+v with the addition of the name of the type of the variable.
%T	Print the variable's type.
%d	Decimal (base 10).
%s	String.

Figure 1.17: Table on substitutions

When using **fmt.Printf**, you need to add the new line symbol yourself, which you do by adding **\n** at the end of the string. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

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

4. Create the **main()** function:

```
func main() {
```

5. Declare and print an integer:

```
    var count int  
  
    fmt.Printf("Count : %#v \n", count)
```

6. Declare and print a **float**:

```
    var discount float64  
  
    fmt.Printf("Discount : %#v \n", discount)
```

7. Declare and print a Boolean:

```
    var debug bool  
  
    fmt.Printf("Debug : %#v \n", debug)
```

8. Declare and print a **string**:

```
    var message string  
  
    fmt.Printf("Message : %#v \n", message)
```

9. Declare and print a collection of strings:

```
var emails []string

fmt.Printf("Emails : %#v \n", emails)
```

10. Declare and print a struct (a type composed of other types; we will cover this in a later chapter):

```
var startTime time.Time

fmt.Printf("Start : %#v \n", startTime)
```

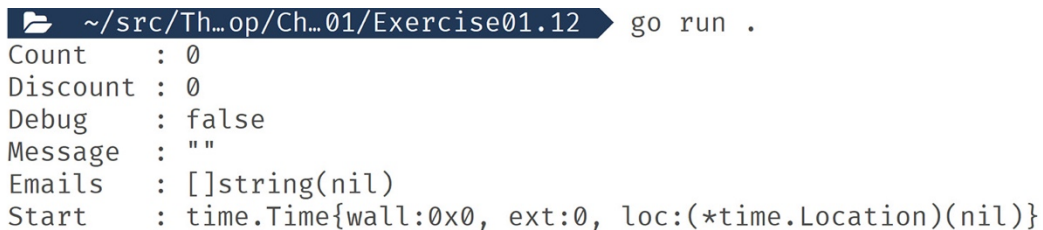
11. Close the **main()** function:

```
}
```

12. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

A terminal window showing the execution of a Go program. The command 'go run .' is entered at the prompt. The output displays the values of several variables: Count (0), Discount (0), Debug (false), Message (empty string), Emails (empty slice), and Start (a time.Time struct with zeroed-out fields).

```
~/src/Th...op/Ch...01/Exercise01.12 go run .
Count      : 0
Discount   : 0
Debug      : false
Message     : ""
Emails      : []string(nil)
Start       : time.Time{wall:0x0, ext:0, loc:(*time.Location)(nil)}
```

Figure 1.18: Output displaying zero values

In this exercise, we defined a variety of variable types without an initial value. Then, we printed them out using **fmt.Printf** to expose more detail about the values. Knowing what the zero values are and how Go controls them allows you to avoid bugs and write concise code.

Next, we'll look at what pointers are and how they can enable you to write efficient software.

Value versus Pointer

With values such as **int**, **bool**, and **string**, when you pass them to a function, Go makes a copy of the value, and it's the copy that's used in the function. This copying means that a change that's made to the value

in the function doesn't affect the value that you used when calling the function.

Passing values by copying tends to end up with code that has fewer bugs. With this method of passing values, Go can use its simple memory management system called the stack. The downside is that copying uses up more and more memory as values get passed from function to function. In real-world code, functions tend to be small, and values get passed to lots of functions, so copying by value can sometimes end up using much more memory than is needed.

There is an alternative to copying that uses less memory. Instead of passing a value, we create something called a pointer and then pass that to functions. A pointer is not a value itself, and you can't do anything useful with a pointer other than getting a value using it. You can think of a pointer as directions to a value you want, and to get to the value, you must follow the directions. If you use a pointer, Go won't make a copy of the value when passing a pointer to a function.

When creating a pointer to a value, Go can't manage the value's memory using the stack. This is because the stack relies on simple scope logic to know when it can reclaim the memory that's used by a value, and having a pointer to a variable means these rules don't work. Instead, Go puts the value on the heap. The heap allows the value to exist until no part of your software has a pointer to it anymore. Go reclaims these values in what it calls its garbage collection process. This garbage collection happens periodically in the background, and you don't need to worry about it.

Having a pointer to a value means that a value is put on the heap, but that's not the only reason that happens. Working out whether a value needs to be put on the heap is called escape analysis. There are times when a value with no pointers is put on the heap, and it's not always clear why.

You have no direct control over whether a value is put on the stack or the heap. Memory management is not part of Go's language specification. Memory management is considered an internal implementation detail. This means it could be changed at any time, and that what we've spoken about are only general guidelines and not fixed rules and could change at a later date.

While the benefits of using a pointer over a value that gets passed to lots of functions are clear for memory usage, it's not so clear for CPU usage. When a value gets copied, Go needs CPU cycles to get that memory and

then release it later. Using a pointer avoids this CPU usage when passing it to a function. On the other hand, having a value on the heap means that it then needs to be managed by the complex garbage collection process. This process can become a CPU bottleneck in certain situations, for example, if there are lots of values on the heap. When this happens, the garbage collector has to do lots of checking, which uses up CPU cycles. There is no correct answer here, and the best approach is the classic performance optimization one. First, don't prematurely optimize. When you do have a performance problem, measure before you make a change, and then measure after you've made a change.

Beyond performance, you can use pointers to change your code's design. Sometimes, using pointers allows a cleaner interface and simplifies your code. For example, if you need to know whether a value is present or not, a non-pointer value always has at least its zero value, which could be valid in your logic. You can use a pointer to allow for an **is not set** state as well as holding a value. This is because pointers, as well as holding the address to a value, can also be **nil**, which means there is no value. In Go, **nil** is a special type that represents something not having a value.

The ability for a pointer to be nil also means that it's possible to get the value of a pointer when it doesn't have a value associated with it, which means you'll get a runtime error. To prevent runtime errors, you can compare a pointer to nil before trying to get its value. This looks like **<pointer> != nil**. You can compare pointers with other pointers of the same type, but they only result in true if you are comparing a pointer to itself. No comparison of the associated values gets made.

As a beginner in the language, I suggest avoiding pointers until they become necessary, either because you have a performance problem or because having a pointer makes your code cleaner.

Getting a Pointer

To get a pointer, you have a few options. You can declare a variable as being a pointer type using a **var** statement. You can do this by adding an ***** at the front of most types. This notation looks like **var <name> *<type>**. The initial value of a variable that uses this method is **nil**. You can use the built-in **new** function for this. This function is intended to be used to get some memory for a type and return a pointer to that address. The notation looks like **<name> := new(<type>)**. The **new** function can be used with **var** too. You can also get a pointer from an existing variable using **&**. This looks like **<var1> := &<var2>**.

Exercise 1.13: Getting a Pointer

In this exercise, we'll use each of the methods we can use to get a pointer variable. Then, we'll print them to the console using **fmt.Printf** to see what their type and value is. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

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

4. Create the **main()** function:

```
func main() {
```

5. Declare a pointer using a **var** statement:

```
    var count1 *int
```

6. Create a variable using **new**:

```
    count2 := new(int)
```

7. You can't take the address of a literal number. Create a temporary variable to hold a number:

```
    countTemp := 5
```

8. Using **&**, create a pointer from the existing variable:

```
    count3 := &countTemp
```

9. It's possible to create a pointer from some types without a temporary variable. Here, we're using our trusty **time** struct:

```
    t := &time.Time{}
```


10. Print each out using **fmt.Printf**:

```
fmt.Printf("count1: %#v\n", count1)

fmt.Printf("count2: %#v\n", count2)

fmt.Printf("count3: %#v\n", count3)

fmt.Printf("time : %#v\n", t)
```

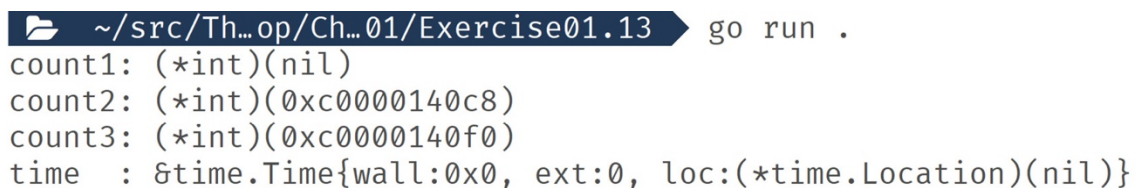
11. Close the **main()** function:

```
}
```

12. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

A terminal window with a dark background. The prompt is a file icon followed by the path ~/src/Th...op/Ch...01/Exercise01.13. The command 'go run .' has been executed. The output shows four lines: 'count1: (*int)(nil)', 'count2: (*int)(0xc0000140c8)', 'count3: (*int)(0xc0000140f0)', and 'time : &time.Time{wall:0x0, ext:0, loc:(*time.Location)(nil)}'.

```
~/src/Th...op/Ch...01/Exercise01.13 go run .
count1: (*int)(nil)
count2: (*int)(0xc0000140c8)
count3: (*int)(0xc0000140f0)
time : &time.Time{wall:0x0, ext:0, loc:(*time.Location)(nil)}
```

Figure 1.19: Output after creating a pointer

In this exercise, we looked at three different ways of creating a pointer. Each one is useful, depending on what your code needs. With the **var** statement, the pointer has a value of nil, while the others already have a value address associated with them. For the **time** variable, we can see the value, but we can tell it's a pointer because its output starts with an **&**.

Next, we'll see how we can get a value from a pointer.

Getting a Value from a Pointer

In the previous exercise, when we printed out the pointer variables for the **int** pointers to the console, we either got nil or saw a memory address. To get to the value a pointer is associated with, you dereference the value using ***** in front of the variable name. This looks like **fmt.Println(<*>)**.

Dereferencing a zero or **nil** pointer is a common bug in Go software as the compiler can't warn you about it, and it happens when the app is running. Therefore, it's always best practice to check that a pointer is not **nil** before dereferencing it unless you are certain it's not **nil**.

You don't always need to dereference; for example, when a property or function is on a struct. Don't worry too much about when you shouldn't be dereferencing as Go gives you clear errors regarding when you can and can't dereference a value.

Exercise 1.14: Getting a Value from a Pointer

In this exercise, we'll update our previous exercise to dereference the values from the pointers. We'll also add **nil** checks to prevent us from getting any errors. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

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

4. Create the **main()** function:

```
func main() {
```

5. Our pointers are declared in the same way as they were previously:

```
    var count1 *int  
    count2 := new(int)  
    countTemp := 5  
    count3 := &countTemp  
    t := &time.Time{}
```

6. For count 1, 2, and 3, we need to add a **nil** check and add * in front of the variable name:

```
if count1 != nil {  
    fmt.Printf("count1: %#v\n", *count1)  
}  
  
if count2 != nil {  
    fmt.Printf("count2: %#v\n", *count2)  
}  
  
if count3 != nil {  
    fmt.Printf("count3: %#v\n", *count3)  
}
```

7. We'll also add a **nil** check for our **time** variable:

```
if t != nil {
```

8. We'll dereference the variable using *, just like we did with the **count** variables:

```
    fmt.Printf("time : %#v\n", *t)
```

9. Here, we're calling a function on our **time** variable. This time, we don't need to dereference it:

```
    fmt.Printf("time : %#v\n", t.String())
```

10. Close the **nil** check:

```
}
```

11. Close the **main()** function:

```
}
```

12. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:

```
~/src/Th...op/Ch...01/Exercise01.14 go run .
count2: 0
count3: 5
time : time.Time{wall:0x0, ext:0, loc:(*time.Location)(nil)}
time : "0001-01-01 00:00:00 +0000 UTC"
```

Figure 1.20: Output displaying the values that were obtained using pointers

In this exercise, we used dereferencing to get the values from our pointers. We also used nil checks to prevent dereferencing errors. From the output of this exercise, we can see that **count1** was a nil value and that we'd have gotten an error if we tried to dereference. **count2** was created using **new**, and its value is a zero value for its type. **count3** also had a value that matches the value of the variable we got the pointer from. With our **time** variable, we were able to dereference the whole struct, which is why our output doesn't start with an **&**.

Next, we'll look at how using a pointer allows us to change the design of our code.

Function Design with Pointers

We'll cover functions in more detail in a later chapter, but you know enough from what we've done so far to see how using a pointer can change how you use a function. A function must be coded to accept pointers, and it's not something that you can choose whether to do or not. If you have a pointer variable or have passed a pointer of a variable to a function, any changes that are made to the value of the variable in the function also affect the value of the variable outside of the function.

Exercise 1.15: Function Design with Pointers

In this exercise, we'll create two functions: one that accepts a number by value, adds 5 to it, and then prints the number to the console; and another function that accepts a number as a pointer, adds 5 to it, and then prints the number out. We'll also print the number out after calling each function to assess what effect it has on the variable that was passed to the function. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```

3. Import the packages we'll need:

```
import "fmt"
```

4. Create a function that takes an **int** as an argument:

```
func add5Value(count int) {
```

5. Add **5** to the passed number:

```
    count += 5
```

6. Print the updated number to the console:

```
    fmt.Println("add5Value :", count)
```

7. Close the function:

```
}
```

8. Create another function that takes an **int** pointer:

```
func add5Point(count *int) {
```

9. Dereference the value and add **5** to it:

```
    *count += 5
```

10. Print out the updated value of **count** and dereference it:

```
    fmt.Println("add5Point :", *count)
```

11. Close the function:

```
}
```

12. Create the **main()** function:

```
func main() {
```

13. Declare an **int** variable:

```
    var count int
```

14. Call the first function with the variable:

```
add5Value(count)
```

15. Print the current value of the variable:

```
fmt.Println("add5Value post:", count)
```

16. Call the second function. This time, you'll need to use **&** to pass a pointer to the variable:

```
add5Point(&count)
```

17. Print the current value of the variable:

```
fmt.Println("add5Point post:", count)
```

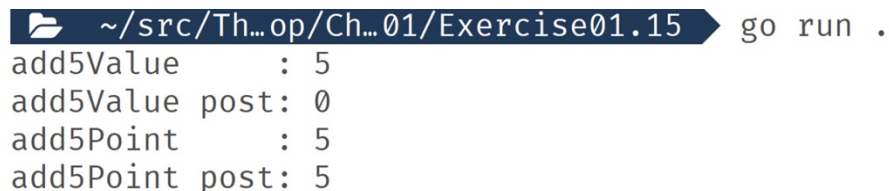
18. Close the **main()** function:

```
}
```

19. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:



```
~/src/Th...op/Ch...01/Exercise01.15 go run .
add5Value      : 5
add5Value post: 0
add5Point      : 5
add5Point post: 5
```

Figure 1.21: Output displaying the current value of the variable

In this exercise, we showed you how passing values by a pointer can affect the value variables that are passed to them. We saw that, when passing by value, the changes you make to the value in a function do not affect the value of the variable that's passed to the function, while passing a pointer to a value does change the value of the variable passed to the function.

You can use this fact to overcome awkward design problems and sometimes simplify the design of your code. Passing values by a pointer has traditionally been shown to be more error-prone, so use this design sparingly. It's also common to use pointers in functions to create more efficient code, which Go's standard library does a lot.

Activity 1.02: Pointer Value Swap

In this activity, your job is to finish some code a co-worker started. Here, we have some unfinished code for you to complete. Your task is to fill in the missing code, where the comments are to swap the values of **a** and **b**. The **swap** function only accepts pointers and doesn't return anything:

```
package main

import "fmt"

func main() {

    a, b := 5, 10

    // call swap here

    fmt.Println(a == 10, b == 5)

}

func swap(a *int, b *int) {

    // swap the values here

}
```

1. Call the **swap** function, ensuring you are passing a pointer.
2. In the **swap** function, assign the values to the other pointer, ensuring you dereference the values.

The following is the expected output:

```
true true
```

Note

The solution for this activity can be found via [this link](#).

Next, we'll look at how we can create variables with a fixed value.

Constants

Constants are like variables, but you can't change their initial value. These are useful for situations where the value of a constant doesn't need to or shouldn't change when your code is running. You could make the

argument that you could hardcode those values into the code and it would have a similar effect. Experience has shown us that while these values don't need to change at runtime, they may need to change later. If that happens, it can be an arduous and error-prone task to track down and fix all the hardcoded values. Using a constant is a tiny amount of work now that can save you a great deal of effort later.

Constant declarations are similar to **var** statements. With a constant, the initial value is required. Types are optional and inferred if left out. The initial value can be a literal or a simple statement and can use the values of other constants. Like **var**, you can declare multiple constants in one statement. Here are the notations:

```
constant <name> <type> = <value>
```

```
constant (
```

```
    <name1> <type1> = <value1>
```

```
    <name2> <type2> = <value3>
```

```
...
```

```
    <nameN> <typeN> = <valueN>
```

```
)
```

Exercise 1.16: Constants

In this exercise, we have a performance problem. Our database server is too slow. We are going to create a custom memory cache. We'll use Go's **map** collection type, which will act as the cache. There is a global limit on the number of items that can be in the cache. We'll use one **map** to help keep track of the number of items in the cache. We have two types of data we need to cache: books and CDs. Both use the ID, so we need a way to separate the two types of items in the shared cache. We need a way to set and get items from the cache.

We're going to set the maximum number of items in the cache. We'll also use constants to add a prefix to differentiate between books and CDs. Let's get started:

1. Create a new folder and add a **main.go** file to it.
2. In **main.go**, add the **main** package name to the top of the file:

```
package main
```


3. Import the packages we'll need:

```
import "fmt"
```

4. Create a constant that's our global limit size:

```
const GlobalLimit = 100
```

5. Create a **MaxCacheSize** that is 10 times the global limit size:

```
const MaxCacheSize int = 10 * GlobalLimit
```

6. Create our cache prefixes:

```
const (  
  
    CacheKeyBook = "book_"  
  
    CacheKeyCD = "cd_"  
  
)
```

7. Declare a **map** that has a **string** for a key and a **string** for its values as our cache:

```
var cache map[string]string
```

8. Create a function to get items from the cache:

```
func cacheGet(key string) string {  
  
    return cache[key]  
  
}
```

9. Create a function that sets items in the cache:

```
func cacheSet(key, val string) {
```

10. In this function, check out the **MaxCacheSize** constant to stop the cache going over that size:

```
    if len(cache)+1 >= MaxCacheSize {  
  
        return
```

```
}  
  
    cache[key] = val  
  
}
```

11. Create a function to get a book from the cache:

```
func GetBook(isbn string) string {
```

12. Use the book cache prefix to create a unique key:

```
    return cacheGet(CacheKeyBook + isbn)  
  
}
```

13. Create a function to add a book to the cache:

```
func SetBook(isbn string, name string) {
```

14. Use the book cache prefix to create a unique key:

```
    cacheSet(CacheKeyBook+isbn, name)  
  
}
```

15. Create a function to get CD data from the cache:

```
func GetCD(sku string) string {
```

16. Use the **CD** cache prefix to create a unique key:

```
    return cacheGet(CacheKeyCD + sku)  
  
}
```

17. Create a function to add CDs to the shared cache:

```
func SetCD(sku string, title string) {
```

18. Use the **CD** cache prefix constant to build a unique key for the shared cache:

```
    cacheSet(CacheKeyCD+sku, title)  
  
}
```

19. Create the **main()** function:

```
func main() {
```

20. Initialize our cache by creating a **map**:

```
cache = make(map[string]string)
```

21. Add a book to the cache:

```
SetBook("1234-5678", "Get Ready To Go")
```

22. Add a **CD** to the cache:

```
SetCD("1234-5678", "Get Ready To Go Audio Book")
```

23. Get and print that **Book** from the cache:

```
fmt.Println("Book :", GetBook("1234-5678"))
```

24. Get and print that **CD** from the cache:

```
fmt.Println("CD :", GetCD("1234-5678"))
```

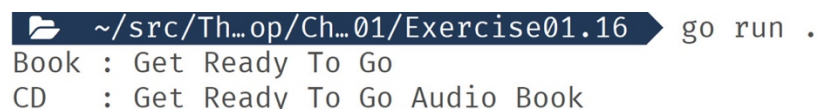
25. Close the **main()** function:

```
}
```

26. Save the file. Then, in the new folder, run the following:

```
go run .
```

The following is the output:



```
~/src/Th...op/Ch...01/Exercise01.16 go run .  
Book : Get Ready To Go  
CD : Get Ready To Go Audio Book
```

Figure 1.22: Output displaying the Book and CD caches

In this exercise, we used constants to define values that don't need to change while the code is running. We declared then using a variety of notation options, some with the typeset and some without. We declared a single constant and multiple constants in a single statement.

Next, we'll look at a variation of constants for values that are more closely related.

Enums

Enums are a way of defining a fixed list of values that are all related. Go doesn't have a built-in type for enums, but it does provide tools such as **iota** to let you define your own using constants, which we'll explore now.

For example, in the following code, we have the days of the week defined as constants. This code is a good candidate for Go's **iota** feature:

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

With **iota**, Go helps us manage lists just like this. Using **iota**, the following code is equal to the preceding code:

```
...  
  
const (  
  
    Sunday = iota  
  
    Monday
```

Tuesday

Wednesday

Thursday

Friday

Saturday

)

...

Now, we have **iota** assigning the numbers for us. Using **iota** makes enums easier to create and maintain, especially if you need to add a new value to the middle of the code later.

Next, we'll take a detailed look at Go's variable scoping rules and how they affect how you write code.

Scope

All the variables in Go live in a scope. The top-level scope is the package scope. A scope can have child scopes within it. There are a few ways a child scope gets defined; the easiest way to think about this is that when you see `{`, you are starting a new child scope, and that child scope ends when you get to a matching `}`. The parent-child relationship is defined when the code compiles, not when the code runs. When accessing a variable, Go looks at the scope the code was defined in. If it can't find a variable with that name, it looks in the parent scope, then the grandparent scope, all the way until it gets to the package scope. It stops looking once it finds a variable with a matching name or raises an error if it can't find a match.

To put it another way, when your code uses a variable, Go needs to work out where that variable was defined. It starts its search in the scope of the code using the variable it's currently running in. If a variable definition using that name is in that scope, then it stops looking and uses the variable definition to complete its work. If it can't find a variable definition, then it starts walking up the stack of scopes, stopping as soon as it finds a variable with that name. This searching is all done based on a variable name. If a variable with that name is found but is of the wrong type, Go raises an error.

In this example, we have four different scopes, but we define the **level** variable once. This fact means that no matter where you use **level**, the same variable is used:

```
package main

import "fmt"

var level = "pkg"

func main() {

    fmt.Println("Main start :", level)

    if true {

        fmt.Println("Block start :", level)

        funcA()

    }

}

func funcA() {

    fmt.Println("funcA start :", level)

}
```

The following is the output displaying variables using level:

Main start : pkg

Block start : pkg

funcA start : pkg

In this example, we've shadowed the **level** variable. This new **level** variable is not related to the **level** variable in the package scope. When we print **level** in the block, the Go runtime stops looking for variables called **level** as soon as it finds the one defined in **main**. This logic results in a different value getting printed out once that new variable shadows the package variable. You can also see that it's a different variable because it's a different type, and a variable can't have its type changed in Go:

```
package main

import "fmt"

var level = "pkg"

func main() {

    fmt.Println("Main start :", level)

    // Create a shadow variable

    level := 42

    if true {

        fmt.Println("Block start :", level)

        funcA()

    }

    fmt.Println("Main end :", level)

}

func funcA() {

    fmt.Println("funcA start :", level)

}
```

The following is the output:

Main start : pkg

Block start : 42

funcA start : pkg

Main end : 42

Go's static scope resolution comes into play when we call **funcA**. That's why, when **funcA** runs, it still sees the package scope **level** variable. The scope resolution doesn't pay attention to where **funcA** gets called.

You can't access variables defined in a child scope:

```
package main

import "fmt"

func main() {
    {
        level := "Nest 1"

        fmt.Println("Block end :", level)
    }

    // Error: undefined: level

    //fmt.Println("Main end :", level)
}
```

The following is the output:

```
 ~/src/Th...op/Ch...01/Example01.11 go run .
# github.com/PacktWorkshops/The-Go-Workshop/Chapter01/Example01.11
./main.go:11:31: undefined: level
```

Figure 1.23: Output displaying an error

Activity 1.03: Message Bug

The following code doesn't work. The person who wrote it can't fix it, and they've asked you to help them. Can you get it to work?

```
package main

import "fmt"

func main() {
    count := 5

    if count > 5 {
        message := "Greater than 5"
```



```

    } else {

        message := "Not greater than 5"

    }

    fmt.Println(message)

}

```

1. Run the code and see what the output is.
2. The problem is with the **message**; make a change to the code.
3. Rerun the code and see what difference it makes.
4. Repeat this process until you see the expected output.

The following is the expected output:

Not greater than 5

Note

The solution for this activity can be found via [this link](#).

In this activity, we saw that where you define your variables has a big impact on the code. Always think about the scope you need your variables to be in when defining them.

In the next activity, we are going to look at a similar problem that is a bit trickier.

Activity 1.04: Bad Count Bug

Your friend is back, and they have another bug in their code. This code should print **true**, but it's printing **false**. Can you help them fix the bug?

```
package main
```

```
import "fmt"
```

```
func main() {
```

```
    count := 0
```

```
    if count < 5 {
```

```
        count := 10
```

```
    count++  
}  
  
fmt.Println(count == 11)  
}
```

1. Run the code and see what the output is.
2. The problem is with **count**; make a change to the code.
3. Rerun the code and see what difference it makes.
4. Repeat this process until you see the expected output.

The following is the expected output:

True

Note

The solution for this activity can be found via [this link](#).

Summary

In this chapter, we got into the nitty-gritty of variables, including how variables are declared, and all the different notations you can use to declare them. This variety of notation gives you a nice compact notation to use for 90% of your work, while still giving you the power to be very specific when you need to the other 10% of the time. We looked at how to change and update the value of variables after you've declared them. Again, Go gives you some great shorthand to help in the most common use cases to make your life easier. All your data ends up in some form of variable. Data is what makes code dynamic and responsive. Without data, your code could only ever do exactly one thing; data unleashes the true power of software.

Now that your application has data, it needs to make choices based on that data. That's where variable comparison comes in. This helps us see whether something is true or false, bigger or smaller, and to make choices based on the results of those comparisons.

We explored how Go decided to implement their variable system by looking at zero values, pointers, and scope logic. Now, we know that these are the details that can be the difference between delivering bug-free efficient software and not doing so.

We also took a look at how we can declare immutable variables by using constants and how **iota** can help manage lists or related constants to work, such as enums.

In the next chapter, we'll start to put our variables to work by defining logic and looping over collections of variables.

2. Logic and Loops

Overview

In this chapter, we'll use branching logic and loops to demonstrate how logic can be controlled and selectively run. With these tools, you'll have control of what you do and don't want to run based on the values of variables.

*By the end of this chapter, you will be able to implement branching logic using **if**, **else**, and **else if**; use **switch** statements to simplify complex branching logic; create looping logic using a **for** loop; loop over complex data collections using **range**; and use **continue** and **break** to take control of the flow of loops.*

Introduction

In the previous chapter, we looked at variables and values and how we can temporarily store data in a variable and make changes to that data. We're now going to look at how we can use that data to run logic, or not, selectively. This logic allows you to control how data flows through your software. You can react to and perform different operations based on the values in your variables.

The logic could be for validating your user's inputs. If we were writing code to manage a bank account, and the user asked to withdraw some money, we could check that they asked for a valid amount of money. We would check that they had enough money in their account. If the validation was successful, we would use logic to update their balance, transfer the money, and show a success message. If the validation failed, we'd show a message explaining what went wrong.

If your software is a virtual world, then logic is the physical law of that world. Like the physical laws of our world, those laws must be followed and can't be broken. If you create a law with a flaw in it, then your virtual world won't run smoothly and could even explode.

Another form of logic is a loop; using loops allows you to execute the same logic multiple times. A common way to use loops is to iterate over a collection of data. For our imaginary banking software, we would use a loop to step over a user's transactions to display them to the user on request.

Loops and logic allow the software to have complex behavior that responds to changing and dynamic data.

if Statements

An **if** statement is the most basic form of logic in Go. An **if** statement either will or won't run a block of logic based on a Boolean expression. The notation looks like this: **if <boolean expression> { <code block> }**.

The Boolean expression can be a simple code that results in a Boolean value. The code block can be any logic that you could also put in a function. The code block runs when the Boolean expression is true. You can only use **if** statements in the function scope.

Exercise 2.01: A Simple if Statement

In this exercise, we'll use an **if** statement to control whether logic will or won't run. We'll define an **int** value to check it's hardcoded, but in a real-world application, this could be user input. We'll then check whether the value is an odd or even number using **%** operator, also known as a modulus expression on the variable. The modulus gives you the amount remaining after division. We'll use the modulus to get the remainder after dividing by 2. If we get a remainder of 0, we know the number is even. If the remainder is 1, we know the number is odd. The modulus results in an **int**, so we use **==** to get a Boolean value:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add the package and imports:

```
package main
```

```
import "fmt"
```

3. Create the **main** function:

```
func main() {
```

4. Define an **int** variable with an initial value. We are setting it to 5 here, which is an odd number but we could also set it to 6, which is an even number:

```
    input := 5
```

5. Create an **if** statement that uses a modulus expression; then, check whether the result is equal to 0:

```
if input%2 == 0 {
```

6. When the Boolean expression results in **true**, that means the number is even. We then print that it's even to the console using the format package:

```
    fmt.Println(input, "is even")
```

7. Close the code block:

```
}
```

8. Now do the same for odd numbers:

```
if input%2 == 1 {
```

```
    fmt.Println(input, "is odd")
```

```
}
```

9. Close **main**:

```
}
```

10. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output:

```
5 is odd
```

In this exercise, we used logic to run code selectively. Using logic to control what code runs, let's you create flows through your code. This allows you to have code that reacts to its data. These flows allow you to be able to reason about what the code is doing with your data, making it easier to understand and maintain.

Try changing the value of the input to 6 to see how the even block gets executed instead of the odd block.

In the next topic, we'll explore how we can improve this code and make it more efficient.

if else Statements

In the previous exercise, we did two evaluations. One evaluation was to check whether the number was even and the other was to see whether it was odd. As we know, a number can only ever be odd or even. With this knowledge, we can use deduction to know that if a number is not even, then it must be odd.

Using deductive logic like this is common in programming in order to make programs more efficient by not having to do unnecessary work.

We can represent this kind of logic using an **if else** statement. The notation looks like this: **if <boolean expression> { <code block> } else { <code block> }**. The **if else** statement builds on the **if** statement and gives us a second block. The second block only runs if the first block doesn't run; both blocks can't run.

Exercise 2.02: Using an if else Statement

In this exercise, we'll update our previous exercise to use an **if else** statement:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add **package** and **import**:

```
package main
```

```
import "fmt"
```

3. Create the **main** function:

```
func main() {
```

4. Define an **int** variable with an initial value, and we'll give it a different value this time:

```
    input := 4
```

5. Create an **if** statement that uses a modulus expression, and then check whether the result is equal to 0:

```
    if input%2 == 0 {
```

```
fmt.Println(input, "is even")
```

6. This time, we are not closing the code block but starting a new **else** code block:

```
} else {  
  
    fmt.Println(input, "is odd")  
  
}
```

7. Close **main**:

```
}
```

8. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output:

```
4 is even
```

In this exercise, we were able to simplify our previous code by using an **if else** statement. As well as making the code more efficient, it also makes the code easier to understand and maintain.

In the next topic, we'll demonstrate how we can add as many code blocks as you want while still only letting one execute.

else if Statements

The **if else** solves the problem of running code for only one or two possible logical outcomes. With that covered, what if our preceding exercise's code was intended to only work for non-negative numbers? We need something that can evaluate more than one Boolean expression but only execute one of the code blocks, that is, the code block for negative numbers, even numbers, or odd numbers.

In that case, we can't use an **if else** statement on its own; however, we could cover it with another extension to **if** statements. In this extension, you can give the **else** statement its own Boolean expression. This is how the notation looks: **if <boolean expression> { <code block> } else if <boolean expression> { <code block> }**. You can also combine it

with a final **else** statement at the end, which would look like this: **if** **<boolean expression> { <code block> } else if <boolean expression> { <code block> } else { <code block> }**. After the initial **if** statement, you can have as many **else if** statements as you need. Go evaluates the Boolean expressions from the top of the statements and works its way through each Boolean expression until one results in **true** or finds an **else**. If there is no **else** and none of the Boolean expressions results in true, then no block is executed and Go moves on. When Go gets a Boolean true result, it executes the code block for that statement only and it then stops evaluating any Boolean expressions of the **if** statement.

Exercise 2.03: Using an else if Statement

In this exercise, we'll update our previous exercise. We're going to add a check for negative numbers. This check must run before the even and odd checks, as only one of the code blocks can run:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add **package** and **import**:

```
package main

import "fmt"
```

3. Create the **main** function:

```
func main() {
```

4. Define an **int** variable with an initial value, and we'll give it a negative value:

```
    input := -10
```

5. Our first Boolean expression is to check for negative numbers. If we find a negative number, we'll print a message saying that they are not allowed:

```
    if input < 0 {

        fmt.Println("input can't be a negative number")
```

6. We need to move our even check to an **else if** statement:

```
    } else if input%2 == 0 {
```

```
fmt.Println(input, "is even")
```

7. The **else** statement stays the same, and we then close **main**:

```
} else {  
  
    fmt.Println(input, "is odd")  
  
}  
  
}
```

8. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output:

```
input can't be a negative number
```

In this exercise, we added even more complex logic to our **if** statement. We added an **else if** statement to it, which allowed complex evaluation. This addition took what is usually a simple fork in the road that gives you many roads to go down but still with the restriction of only going down one of them.

In the next topic, we'll use a subtle but powerful feature of **if** statements that lets you keep your code nice and tidy.

The Initial if Statement

It's common to need to call a function but not care too much about the returned value. Often, you'll want to check that it executed correctly and then discard the returned value. For example, sending an email, writing to a file, or inserting data into a database; most of the time, if these types of operations execute successfully, you don't need to worry about the variables they return. Unfortunately, the variables don't go anywhere as they are still in scope.

To stop these unwanted variables from hanging around, we can use what we know about scope rules to get rid of them. The best way to check for errors is to use "initial" statements on **if** statements. The notation looks like this: **if <initial statement>; <boolean expression> { <code**

block> }. The initial statement is in the same section as the Boolean expression, with ; to divide them.

Go only allows what it calls simple statements in the initial statement section, including:

- Assignment and short variable assignments:

E.g.: `i := 0`

- Expressions such as math or logic expressions:

E.g.: `i = (j * 10) == 40`

- Sending statements for working with channels, which we'll cover later.
- Increment and decrement expressions:

E.g.: `i++`

A common mistake is trying to define a variable using **var**. That's not allowed; you can use the short assignment in its place.

Exercise 2.04: Implementing the Initial if Statements

In this exercise, we're going to continue to build on our previous exercises. We're going to add even more rules about what numbers can be checked as to whether they are odd or even. With so many rules, putting them all in a single Boolean expression is hard to understand. We'll move all the validation logic to a function that returns an **error**. This is a built-in Go type used for errors. If the value of the error is **nil**, then everything is okay. If not, you have an error, and you need to deal with it. We'll call the function in our initial statement and then check for errors:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add **package** and **import**:

```
package main
```

```
import (
```

```
    "errors"
```

```
    "fmt"
```

)

3. Create a function to do the validation. This function takes a single integer and returns **error**:

```
func validate(input int) error {
```

4. We define some rules, and if any are true, we return a new **error** using the **New** function in the **errors** package:

```
    if input < 0 {
```

```
        return errors.New("input can't be a negative number")
```

```
    } else if input > 100 {
```

```
        return errors.New("input can't be over 100")
```

```
    } else if input%7 == 0 {
```

```
        return errors.New("input can't be divisible by 7")
```

5. If the input passes all the checks, return **nil**:

```
    } else {
```

```
        return nil
```

```
    }
```

```
}
```

6. Create our **main** function:

```
func main() {
```

7. Define a variable with a value of **21**:

```
    input := 21
```

8. Call the function using the initial statement; use the short variable assignment to capture the returned error. In the Boolean expression, check that the error is not equal to **nil** using **!=**:

```
    if err := validate(input); err != nil {
```

```
    fmt.Println(err)
}
```

9. The rest is the same as before:

```
else if input%2 == 0 {
    fmt.Println(input, "is even")
} else {
    fmt.Println(input, "is odd")
}
}
```

10. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output which displays an error statement:

```
input can't be divisible by 7
```

In this exercise, we used an initial statement to define and initialize a variable. That variable can be used in the Boolean expression and the related code block. Once the **if** statement completes, the variable goes out of scope and is reclaimed by Go's memory management system.

Activity 2.01: Implementing FizzBuzz

When interviewing for a programming job, you'll be asked to do some coding exercises. These questions have you writing something from scratch and will have several rules to follow. To give you an idea of what that looks like, we'll run you through a classic one, "FizzBuzz."

The rules are as follows:

- Write a program that prints out the numbers from 1 to 100.
- If the number is a multiple of 3, print "Fizz."
- If the number is a multiple of 5, print "Buzz."

- If the number is a multiple of 3 and 5, print "FizzBuzz."

Here are some tips:

- You can convert a number to a string using **strconv.Itoa()**.
- The first number to evaluate must be 1 and the last number to evaluate must be 100.

These steps will help you to complete the activity:

1. Create a loop that does 100 iterations.

Hint

You'll learn about loops in detail later on in the chapter (in a section titled Loops). If you're not sure how to create a loop, use the following block of code as a hint:

```
for i := 1; i <= 100; i++{  
  
<code>  
  
}
```

*The preceding code should create a **for** loop that starts at 1 and loops until i gets to 100.*

2. Have a variable that keeps count of the number of loops so far.
3. In the loop, use that count and check whether it's divisible by 3 or 5 using %.
4. Think carefully about how you'll deal with the "FizzBuzz" case.

The following screenshot shows the expected output:

Note

Considering that the output is too big to be displayed here, only a part of it will be visible in Figure 2.01.

```
~/src/Th...op/Ch...02/Activity02.01 go run .  
1  
2  
Fizz  
4  
Buzz  
Fizz  
7  
8  
Fizz  
Buzz  
11  
Fizz  
13  
14  
FizzBuzz  
16  
17  
Fizz  
19  
Buzz  
Fizz  
22  
23  
Fizz  
Buzz  
26  
Fizz  
28  
29  
FizzBuzz  
31
```

Figure 2.01: The FizzBuzz output

Note

The solution for this activity can be found via [this link](#).

In the next topic, we'll see how we can tame **if else** statements that start to get too big.

Expression switch Statements

While it's possible to add as many **else if** statements to an **if** as you want, at some point, it'll get hard to read.

When this happens, you can use Go's logic alternative: **switch**. For situations where you would need a big **if** statement, **switch** can be a more compact alternative.

The notation for **switch** is shown in the following code snippet:

```

switch <initial statement>; <expresion> {

case <expresion>:

    <statements>

case <expresion>, <expresion>:

    <statements>

default:

    <statements>

}

```

The "initial" statement works the same in **switch** as it does in the preceding **if** statements. The expression is not the same because the **if** is a Boolean expression. You can have more than just a Boolean in this expression. The cases are where you check to see whether the statements get executed. Statements are like code blocks in **if** statements, but with no need for the curly brackets here.

Both the initial statement and expression are optional. To have just the expression, it would look like this: **switch <expresion> {...** To have only the initial statement, you would write **switch <initial statment>; {...** You can leave them both off, and you'll end up with **switch {...** When the expression is missing, it's as if you put the value of **true** there.

There are two main ways of using case expressions. They can be used just like **if** statements or Boolean expressions where you use logic to control whether the statements get executed. The alternative is to put a literal value there. In this case, the value is compared to the value in the **switch** expression. If they match, then the statements run. You can have as many case expressions as you want by separating them with a,. The case expressions get checked from the top case and then from left to right if a case has multiple expressions.

When a case matches, only its statements are run, which is different from many other languages. To get the fallthrough behavior found in those languages, a **fallthrough** statement must be added to the end of each case where you want that behavior. If you call **fallthrough** before the end of the case, it will fall through at that moment and move on to the next case.

An optional **default** case can be added anywhere in the **switch** statement, but it's best practice to add it to the end. The **default** case works just like using an **else** statement in an **if** statement.

This form of switch statement is called an "expression **switch**" statement. There is also another form of **switch** statement, called a "type **switch**" statement, which we'll look at in a later chapter.

Exercise 2.05: Using a switch Statement

In this exercise, we need to create a program that prints a particular message based on the day someone was born. We are using the **time** package for the set of days of the week constants. We'll use a **switch** statement to make a more compact logic structure:

1. Load the **main** package:

```
package main
```

2. Import the **fmt** and **time** packages:

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

3. Define the **main** function:

```
func main() {
```

4. Define a variable that is the day of the week someone was born. Use the constants from the **time** package to do it. We'll set it to Monday, but it could be any day:

```
    dayBorn := time.Monday
```

5. Create a **switch** statement that uses the variable as its expression:

```
    switch dayBorn {
```

6. Each **case** will try to match its expression value against the switch expression value:

```
case time.Monday:
```

```
    fmt.Println("Monday's child is fair of face")
```

```
case time.Tuesday:
```

```
    fmt.Println("Tuesday's child is full of grace")
```

```
case time.Wednesday:
```

```
    fmt.Println("Wednesday's child is full of woe")
```

```
case time.Thursday:
```

```
    fmt.Println("Thursday's child has far to go")
```

```
case time.Friday:
```

```
    fmt.Println("Friday's child is loving and giving")
```

```
case time.Saturday:
```

```
    fmt.Println("Saturday's child works hard for a living")
```

```
case time.Sunday:
```

```
    fmt.Println("Sunday's child is bonny and blithe")
```

7. We'll use the **default** case here as a form of validation:

```
default:
```

```
    fmt.Println("Error, day born not valid")
```

```
}
```

8. Close the **main** function:

```
}
```

9. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output:

```
Monday's child is fair of face
```

In this exercise, we used **switch** to create a compact logic structure that matches lots of different possible values to give a specific message to our users. It's quite common to see **switch** statements used with a constant as we did here, using the day of the week constants from the **time** package.

Next, we'll use the **case** feature that let's us match multiple values.

Exercise 2.06: switch Statements and Multiple case Values

In this exercise, we're going to print out a message that tells us whether the day someone was born was a weekday or the weekend. We only need two cases as each case can support checking multiple values:

1. Load the **main** package:

```
package main
```

2. Import the **fmt** and **time** packages:

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

3. Define the **main** function:

```
func main() {
```

4. Define our **dayBorn** variable using one of the **time** package's constants:

```
    dayBorn := time.Sunday
```

5. **switch** starts the same by using the variable as the expression:

```
    switch dayBorn {
```

6. This time, for **case**, we have weekday constants. Go checks each one against the **switch** expression, starting from the left, and sweeps through each one by one. Once Go gets a match, it stops evaluating and runs the statements for that case only:

```
case time.Monday, time.Tuesday, time.Wednesday,  
time.Thursday, time.Friday:
```

```
    fmt.Println("Born on a weekday")
```

7. Then, it does the same for weekend days:

```
case time.Saturday, time.Sunday:
```

```
    fmt.Println("Born on the weekend")
```

8. We use **default** for validation again and close out the **switch** statement:

```
default:
```

```
    fmt.Println("Error, day born not valid")
```

```
}
```

9. Close the **main** function:

```
}
```

10. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output:

```
Born on the weekend
```

In this exercise, we used cases with multiple values. This allowed a very compact logic structure that could evaluate 7 days of the week with validation checking in a few lines of code. It makes the intention of the logic clear, which, in turn, makes it easier to change and maintain.

Next, we'll look at using more complex logic in **case** expressions.

Sometimes, you'll see code that doesn't evaluate anything in the **switch** statement but does checks in the **case** expression.

Exercise 2.07: Expressionless switch Statements

It's not always possible to be able to match values using the value of the **switch** expression. Sometimes, you'll need to match on multiple variables. Sometimes, you'll need to match on something more complicated than an equality check. For example, you may need to check whether a number is in a specific range. In these cases, **switch** is still helpful in building compact logic statements, as **case** allows the same range of expressions that you have in **if** Boolean expressions.

In this exercise, let's build a simple **switch** expression that checks whether a day is a weekend to show what can be done in **case**:

1. Load the **main** package:

```
package main
```

2. Import the **fmt** and **time** packages:

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

3. Define the **main** function:

```
func main() {
```

4. Our **switch** expression is using the initial statement to define our variable. The expression is left empty as we'll not be using it:

```
switch dayBorn := time.Sunday; {
```

5. **case** is using some complex logic to check whether the day is at the weekend:

```
    case dayBorn == time.Sunday || dayBorn == time.Saturday:  
        fmt.Println("Born on the weekend")
```

6. Add a **default** statement and close the **switch** expression:

default:

```
fmt.Println("Born some other day")
```

```
}
```

7. Close the **main** function:

```
}
```

8. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following screenshot shows the expected output:

```
Born on the weekend
```

In this exercise, we learned that you can use complex logic in the **case** expression when a simple **switch** statement match is not enough. This still offers a more compact and easier way to manage a logic statement than **if**, if you have more than a couple of cases.

Next, we'll leave logic structures behind and start to look at ways in which we can run the same statements multiple times to make processing data easier.

Loops

In real-world applications, you're often going to need to run the same logic repeatedly. It's common to have to deal with multiple inputs and give multiple outputs. Loops are the simplest way of repeating your logic.

Go only has one looping statement, **for**, but it's a flexible one. There are two distinct forms: the first is used a lot for ordered collections such as arrays and slices, which we'll cover more later. The sort of loop used for ordered collections looks as follows:

```
for <initial statement>; <condition>; <post statement> {
```

```
<statements>
```

```
}
```

The **initial** statement is just like the one found in **if** and **switch** statements. **initial** statement runs before everything else and allows the same simple statements that we defined before. The condition is checked before each loop to see whether the statements should be run or whether the loop should stop.

Like **initial** statement, **condition** also allows simple statements.

The **post** statement is run after the statements are run at the end of each loop and allow you to run simple statements. The **post** statement is mostly used for incrementing things such as loop counters, which get evaluated on the next loop by **condition**. The statements are any Go code you want to run as part of the loop.

The **initial**, **condition**, and **post** statements are all optional, and it's possible to write a **for** loop like this:

```
for {  
  
    <statements>  
  
}
```

This form would result in a loop that would run forever, also known as an infinite loop, unless the **break** statement is used to stop the loop manually. In addition to **break**, there is also a **continue** statement that can be used to skip the remainder of an individual run of a loop but doesn't stop the whole loop.

Another form the **for** loop can take is when reading from a source of data that returns a Boolean when there is more data to read. Examples of this include when reading from databases, files, command-line inputs, and network sockets. This form looks like this:

```
for <condition> {  
  
    <statements>  
  
}
```

This form is just a simplified version of the form used to read from an ordered list but without the logic needed to control the loop yourself, as the source you're using is built to work easily in **for** loops.

The other form that the **for** loop takes is when looping over unordered data collections such as maps. We'll cover what maps are in more detail in a later chapter. When looping over these, you'll use the **range** statement in your loop. With maps, the form looks like this:

```
for <key>, <value> := range <map> {  
    <statements>  
}
```

Exercise 2.08: Using the for i Loop

In this exercise, we'll use the three parts of the **for** loop to create a variable and use a variable in the loop. We'll be able to see how the variable changes after each iteration of the loop by printing out its value to the console:

1. Define **package** as **main** and add imports:

```
package main  
  
import "fmt"
```

2. Create the **main** function:

```
func main() {
```

3. Define a **for** loop that defines the **i** variable with an initial value of **0** in the **initial** statement section. In the clause, check that **i** is less than **5**. In for the post statement, increment **i** by **1**:

```
    for i := 0; i < 5; i++ {
```

4. In the body of the loop, print out the value of **i**:

```
        fmt.Println(i)
```

5. Close the loop:

```
    }
```

6. Close **main**:

```
}
```


7. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output:

0

1

2

3

4

In this exercise, we used a variable that only exists in the **for** loop. We set up the variable, checked its value, modified it, and output it. Using a loop like this is very common when working with ordered, numerically indexed collections such as arrays and slices. In this instance, we hardcoded the value for when to stop looping; however, when looking over arrays and slices, that value would be determined dynamically from the size of the collection.

Next, we'll use a **for i** loop to work with a slice.

Exercise 2.09: Looping Over Arrays and Slices

In this exercise, we'll loop over a collection of strings. We'll be using a slice, but the loop logic will also be the same set of arrays. We'll define the collection; we'll then create a loop that uses the collection to control when to stop looping and a variable to keep track of where we are in the collection.

The way the index of arrays and slices works means that there are never any gaps in the number, and the first number is always 0. The built-in function, **len**, is used to get the length of any collection. We'll use it as part of the condition to check when we've reached the end of the collection:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add **package** and **import**:

```
package main
```

```
import "fmt"
```

3. Create the **main** function:

```
func main() {
```

4. Define a variable which is a slice of "strings" and initialize it with data:

```
names := []string{"Jim", "Jane", "Joe", "June"}
```

We will cover **collection** and **string** in more detail in the next chapter.

5. The **initial** and **post** statements for the loop are the same as before; the difference is in the **condition**, where we use **len** to check whether we are at the end of the collection:

```
for i := 0; i < len(names); i++ {
```

6. The rest is the same as before:

```
    fmt.Println(names[i])  
  
}  
  
}
```

7. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output:

Jim

Jane

Joe

June

The range Loop

The **array** and **slice** types always have the number of an index, and that number always starts at **0**. The **for i** loop we've seen so far is the most common choice you'll see in real-world code for these types.

The other collection type, **map**, doesn't give the same guarantee. That means you need to use **range**. You'll use **range** instead of the **condition** of a **for** loop, and, on each loop, **range** yields both a key and a value of an element in the collection, then, moves on to the next element.

With a **range** loop, you don't need to define a condition to stop the loop as **range** takes care of that for us.

Note

Callout map Order: *The order of items is randomized to stop developers relying on the order of the elements in a map, which means you can use it as a form of pseudo data randomization if needed.*

Exercise 2.10: Looping Over a Map

In this exercise, we're going to create a **map** that has a string for its key and a string for the values. We'll cover **map** types in more detail in a later chapter, so don't worry if you don't quite get what **map** types are yet. We'll then use **range** in the **for** loop to iterate over the map. We'll then write out the key and value data to the console:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add the **package** and **import**:

```
package main
```

```
import "fmt"
```

3. Create the **main** function:

```
func main() {
```

4. Define a **map** with a **string** key and a **string** value of strings variable and initialize it with the data:

```
    config := map[string]string{
```

```
"debug": "1",  
"logLevel": "warn",  
"version": "1.2.1",  
}
```

5. Use **range** to get the **key** and **value** for an array element and assign them to variables:

```
for key, value := range config {
```

6. Print out the **key** and **value** variables:

```
    fmt.Println(key, "=", value)
```

7. Close the loop and **main**:

```
    }  
}
```

8. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output displaying a map that has a string for its key and a string for the values:

```
debug = 1
```

```
logLevel = warn
```

```
version = 1.2.1
```

In this exercise, we used **range** in a **for** loop to allow us to read out all the data from a **map** collection. Even though we used an integer for the **map** variable's key, **map** types don't give guarantees like arrays and slices do about starting at zero and having no gaps. **range** creates an integer variable which is the **key**, and the **value** is referenced by the **key**. **range** also controls when to stop the loop.

If you don't need the **key** or the **value** variable, you can use `_` as the variable name to tell the compiler you don't want it.

Activity 2.02: Looping Over Map Data Using range

Suppose you have been provided with the data in the following table. You have to find the word with the maximum count and print the word and its count using the following data:

Word	Count
Gonna	3
You	3
Give	2
Never	1
Up	4

Figure 2.02: Word and count data to perform the activity

Note

The preceding words are from the song Never Gonna Give You Up, sung by Rick Astley.

The steps to solve the activity are as follows:

1. Put the words into a map like this:

```
words := map[string]int{  
    "Gonna": 3,  
    "You": 3,  
    "Give": 2,  
    "Never": 1,  
    "Up": 4,  
}
```

2. Create a loop and use **range** to capture the word and the count.
3. Keep track of the word with the highest count using a variable for what the highest count is and its associated word.
4. Print the variables out.

The following is the expected output displaying the most popular word with its count value:

Most popular word: Up

With a count of : 4

Note

The solution for this activity can be found via [this link](#).

Next, we'll look at how we can take manual control of the loop by skipping iterations or stopping the loop.

break and continue

There are going to be times when you need to skip a single loop or stop the loop from running altogether. It's possible to do this with variables and **if** statements, but there is an easier way.

The **continue** keyword stops the execution of the current loop and starts a new loop. The **post** loop logic runs, and the loop **condition** gets evaluated.

The **break** keyword also stops the execution of the current loop and it stops any new loops from running.

Use **continue** when you want to skip a single item in a collection; for instance, perhaps it's okay if one of the items in a collection is invalid, but the rest may be okay to process. Use **break** when you need to stop processing when there are any errors in the data and there's no value in processing the rest of the collection.

Here, we have an example that generates a random number between **0** and **8**. The loop skips on a number divisible by **3** and stops on a number divisible by **2**. It also prints out the **i** variable for each loop to help us see that **continue** and **break** are stopping the execution of the rest of the loop.

Exercise 2.11: Using break and continue to Control Loops

In this exercise, we'll use **continue** and **break** in a loop to show you how you can take control of it. We're going to create a loop that keeps going forever. This means we have to stop it with **break** manually. We'll also randomly skip loops with **continue**. We'll do this skipping by generating a random number, and if that number is divisible by 3, we'll skip the rest of the loop:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add **package** and **import**:

```
package main

import (

    "fmt"

    "math/rand"

)
```

3. Create the **main** function:

```
func main() {
```

4. Create an empty **for** loop. This will loop forever if you don't stop it:

```
    for {
```

5. Use **Intn** from the **rand** package to pick a random number between 0 and 8:

```
        r := rand.Intn(8)
```

6. If the random number is divisible by 3, print "**Skip**" and skip the rest of the loop using **continue**:

```
        if r%3 == 0 {

            fmt.Println("Skip")

            continue
```

7. If the random number is divisible by 2, then print "**Stop**" and stop the loop using **break**:

```
        } else if r%2 == 0 {

            fmt.Println("Stop")

            break

        }
```

8. If the number is neither of those things, then print the number:

```
fmt.Println(r)
```

9. Close the loop and **main**:

```
}
```

```
}
```

10. Save the file, and, in the new folder, **run** the following code snippet:

```
go run main.go
```

The following is the expected output displaying random numbers, **Skip**, and **Stop**:

```
1
```

```
7
```

```
7
```

```
Skip
```

```
1
```

```
Skip
```

```
1
```

```
Stop
```

In this exercise, we created a **for** loop that would loop forever, and we then used **continue** and **break** to override normal loop behavior to take control of it ourselves. The ability to do this can allow us to reduce the number of nested **if** statements and variables needed to prevent logic from running when it shouldn't. Using **break** and **continue** help to clean up your code and make it easier to work on.

If you use an empty **for** loop like this, the loop continues forever, and you must use **break** to prevent an infinite loop. An infinite loop is a loop in your code that never stops. Once you get an infinite loop, you'll need a way to kill your application; how you do that will depend on your

operating system. If you are running your app in a terminal, closing the terminal normally does the trick. Don't panic – it happens to us all – your system may slow down, but it won't do it any harm.

Next, we'll work on some activities to test out all your new knowledge about logic and loops.

Activity 2.03: Bubble Sort

In this activity, we'll sort a given slice of numbers by swapping the values. This technique of sorting is known as the **bubble sort** technique. Go has built-in sorting algorithms in the **sort** package but I don't want you to use them; we want you to use the logic and loops you've just learned.

Steps:

1. Define a slice with unsorted numbers in it.
2. Print this slice to the console.
3. Sort the values using swapping.
4. Once done, print the now sorted numbers to the console.

Tips:

- You can do an in-place swap in Go using:

```
nums[i], nums[i-1] = nums[i-1], nums[i]
```

- You can create a new slice using:

```
var nums2 []int
```

- You can add to the end of a slice using:

```
nums2 = append(nums2, 1)
```

The following is the expected output:

Before: [5, 8, 2, 4, 0, 1, 3, 7, 9, 6]

After : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Note

The solution for this activity can be found via [this link](#).

Summary

In this chapter, we discussed logic and loops. These are the foundational building blocks to build complex software. They allow you to have data flow through your code. They let you deal with collections of data by letting you execute the same logic on every element of the data.

Being able to define the rules and laws of your code are the starting points of codifying the real world in software. If you are creating banking software and the bank has rules about what you can and can't do with money, then you can also define those rules in your code.

Logic and loops are the essential tools that you'll use to build all your software.

In the next chapter, we'll look at Go's type system and the core types it has available.

3. Core Types

Overview

This chapter aims to show you how to use Go's basic core types to design your software's data. We'll work through each type to show what they are useful for and how to use them in your software. Understanding these core types provides you with the foundation required to learn how to create complex data designs.

By the end of this chapter, you will be able to create variables of different types for Go programs and assign values to variables of different types. You will learn to identify and pick a suitable type for any programming situation. You will also write a program to measure password complexity and implement empty value types.

Introduction

In the previous chapter, we learned how to use **if**, **if-else**, **switch**, **continue**, and **break** in Go.

Go is a strongly typed language, and all data is assigned a type. That type is fixed and can't be changed. What you can and can't do with your data is constrained by the types you assign. Understanding exactly what defines every one of Go's core types is critical to success with the Go language.

In later chapters, we'll talk about Go's more complex types, but those types are built on the core types defined in this chapter.

Go's core types are well-thought-out and easy to understand once you understand the details. Having to understand the details means Go's type system is not always intuitive. For example, Go's most common number type, **int**, may be either 32 bits or 64 bits in size depending on the computer used to compile the code.

Types are needed to make data easier for humans to work with. Computers only think about data in binary. Binary is hard for people to work with. By adding a layer of abstraction to binary data and labeling it as a number or some text, humans have an easier time reasoning about it. Reducing the cognitive load allows people to build more complex

software because they're not overwhelmed by managing the details of the binary data.

Programming languages need to define what a number is or what a text is for. A programming language defines what you can call a number, and it defines what operations you can use on a number. For example, can a whole number such as 10 and a floating-point number such as 3.14 both be stored as the same type? While it seems obvious that you can multiply numbers, can you multiply text? As we progress through this chapter, we'll clearly define what the rules are for each type and what operations you can use with each of them.

The way data is stored is also a large part of what defines a **type**. To allow for the building of efficient software, Go places limits on how large some of its types can be. For example, the largest amount of storage for a number in Go's core types is 64 bits of memory. This allows for any number up to 18,446,744,073,709,551,615. Understanding these type limitations is critical in building bug-free code.

The things that define a type are:

- The kind of data that you can store in it
- What operations you can use with it
- What those operations do to it
- How much memory it can use

This chapter gives you the knowledge and confidence to use Go's types system correctly in your code.

True and False

True and false logic is represented using the Boolean type, **bool**. Use this type when you need an on/off switch in your code. The value of a **bool** can only ever be **true** or **false**. The zero value of a **bool** is **false**.

When using a comparison operator such as `==` or `>`, the result of that comparison is a **bool** value.

In this code example, we use comparison operators on two numbers. You'll see that the result is a **bool**:

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println(10 > 5)  
    fmt.Println(10 == 5)  
}
```

Running the preceding code shows the following output:

true

false

Exercise 3.01: Program to Measure Password Complexity

An online portal creates user accounts for its users and accepts passwords that are only 8-15 characters long. In this exercise, we write a program for the portal to display whether the password entered meets the character requirements. The character requirements are as follows:

- Have a lowercase letter
- Have an uppercase letter
- Have a number
- Have a symbol
- Be 8 or more characters long

To do this exercise, we're going to use a few new features. Don't worry if you don't quite understand what they are doing; we'll cover them in detail in the next chapter. Consider this a sneak peek. We'll explain what everything is as we go, but your main focus should be on the Boolean logic:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add the main package name to the top of the file:

```
package main
```

3. Now add the imports we'll use in this file:

```
import (  
    "fmt"  
    "unicode"
```

)

4. Create a function that takes a string argument and returns a **bool**:

```
func passwordChecker(pw string) bool {
```

5. Convert the password string into **rune**, which is a type that is safe for multi-byte (UTF-8) characters:

```
    pwR := []rune(pw)
```

We'll talk more about rune later in this chapter.

6. Count the number of multi-byte characters using **len**. This code results in a **bool** result that can be used in the **if** statement:

```
    if len(pwR) < 8 {
```

```
        return false
```

```
    }
```

```
    if len(pwR) > 15 {
```

```
        return false
```

```
    }
```

7. Define some **bool** variables. We'll check these at the end:

```
    hasUpper := false
```

```
    hasLower := false
```

```
    hasNumber := false
```

```
    hasSymbol := false
```

8. Loop over the multi-byte characters one at a time:

```
    for _, v := range pwR {
```

9. Using the **unicode** package, check whether this character is uppercase. This function returns a **bool** that we can use directly in the **if** statement:

```
    if unicode.IsUpper(v) {
```

10. If it is, we'll set the **hasUpper bool** variable to **true**:

```
        hasUpper = true
    }
```

11. Do the same thing for lowercase letters:

```
    if unicode.IsLower(v) {
        hasLower = true
    }
```

12. Also do it for numbers:

```
    if unicode.IsNumber(v) {
        hasNumber = true
    }
```

13. For symbols, we'll also accept punctuation. Use the **or** operator, which works with **Booleans**, to result in **true** if either of these functions returns **true**:

```
    if unicode.IsPunct(v) || unicode.IsSymbol(v) {
        hasSymbol = true
    }
}
```

14. To pass all our checks, all our variables must be **true**. Here, we combine multiple **and** operators to create a one-line statement that checks all four variables:

```
    return hasUpper && hasLower && hasNumber && hasSymbol
```

15. Close the function:

```
}
```

16. Create the **main()** function:

```
func main() {
```

17. Call the **passwordChecker()** function with an invalid password. As this returns a **bool**, it can be used directly in an **if** statement:

```
    if passwordChecker("") {  
  
        fmt.Println("password good")  
  
    } else {  
  
        fmt.Println("password bad")  
  
    }
```

18. Now, call the function with a valid password:

```
    if passwordChecker("This!I5A") {  
  
        fmt.Println("password good")  
  
    } else {  
  
        fmt.Println("password bad")  
  
    }
```

19. Close the **main()** function:

```
}
```

20. Save the file and in the new folder and then run the following:

```
go run main.go
```

Running the preceding code shows the following output:

```
password bad
```

```
password good
```

In this exercise, we highlighted a variety of ways that **bool** values manifest themselves in the code. **Bool** values are critical to giving your

code the ability to make a choice and be dynamic and responsive. Without **bool**, your code would have a hard time doing anything.

Next, we'll take a look at numbers and how Go categorizes them.

Numbers

Go has two distinct number types: integers, also known as whole numbers, and floating-point numbers. A floating-point number allows a number with whole numbers and fractions of a whole number.

1, 54, and 5,436 are examples of whole numbers. 1.5, 52.25, 33.333, and 64,567.00001 are all examples of floating-point numbers.

Note

The default and empty values for all number types is 0.

Next, we'll start our number journey by looking at integers.

Integer

Integer types are classified in two ways, based on the following conditions:

- Whether or not they can store negative numbers
- The smallest and largest numbers they can store

Types that can store negative numbers are called signed integers. Types that can't store negative numbers are called unsigned integers. How big and small a number each type can store is expressed by how many bytes of internal storage they have.

Here is an excerpt from the Go language specification with all the relevant integer types:

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)
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)
byte	alias for uint8
rune	alias for int32

Figure 3.01: Go language specification with relevant integer types

There are also special integer types as follows:

uint	either 32 or 64 bits
int	same size as uint

Figure 3.02: Special integer types

uint and **int** are either 32 or 64 bits depending on whether you compile your code for a 32-bit system or a 64-bit system. It's rare nowadays to run applications on a 32-bit system, systems so most of the time they are 64 bits.

An **int** on a 64-bit system is not an **int64**. While these two types are identical, they are not the same integer type, and you can't use them together. If Go did allow this, there would be problems when the same code gets compiled for a 32-bit machine, so keeping them separate ensures that the code is reliable.

This incompatibility is not just an **int** thing; you can't use any of the integer types together.

Picking the correct integer type to use when defining a variable is easy: use **int**. When writing code for an application, **int** does the job the majority of the time. Only think about using the other types when an **int** is causing a problem. The sorts of problems you see with **int** tend to be related to memory usage.

For example, let's say you have an app that's running out of memory. The app uses a massive number of integers, but these integers are never

negative and won't go over 255. One possible fix is to switch from using **int** to using **uint8**. Doing this cuts its memory usage from 64 bits (8 bytes) per number to 8 bits (1 byte) per number.

We can show this by creating a collection of both kinds of type then asking Go how much heap memory it is using. The output may vary on your computer, but the effect should be similar. This code creates a collection of **int** or **int8**. It then adds 10 million values to the collection. Once that's done, it uses the runtime package to give us a reading of how much heap memory is being used. We convert that reading to MB and then print it out:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    var list []int
    //var list []int8

    for i := 0; i < 10000000; i++ {
        list = append(list, 100)
    }

    var m runtime.MemStats
    runtime.ReadMemStats(&m)

    fmt.Printf("TotalAlloc (Heap) = %v MiB\n", m.TotalAlloc/1024/1024)
}
```

Here's the output using **int**:

TotalAlloc (Heap) = 403 MiB

And here's the output using **int8**:

TotalAlloc (Heap) = 54 MiB

We saved a good amount of memory here, but we need 10 million variables to make it worthwhile. Hopefully, now you are convinced that it's okay to start with **int** and only worry about performance when it's a problem, not before.

Next, we'll look at floating-point numbers.

Floating Point

Go has two floating-point number types, **float32** and **float64**. The bigger **float64** allows for more precision in the numbers. **float32** has 32 bits of storage and **float64** has 64 bits of storage. Floats split their storage between whole numbers (everything to the left of the decimal point) and decimal numbers (everything to the right of the decimal point). How much space is used for the whole number or the decimal numbers, varies by the number being stored. For example, 9,999.9 would use more storage for the whole numbers while 9.9999 would use more storage for the decimal numbers. With **float64**'s bigger space for storage, it can store more whole numbers and/or more decimal numbers than **float32** can.

Exercise 3.02: Floating-Point Number Accuracy

In this exercise, we're going to compare what happens when we do some divisions on numbers that don't divide equally. We'll be dividing 100 by 3. One way of representing the result is $33 \frac{1}{3}$. Computers, for the most part, can't compute fractions like this. Instead, they use a decimal representation, which is 33.3 recurring, where the 3 after the decimal point repeats forever. If we let the computer do that it uses up all the memory, which is not very helpful.

Luckily for us, we don't need to worry about this happening as the floating-point types have storage limits. The downside is that this leads to a number that doesn't reflect the true result; the result has a certain amount of inaccuracy. Your tolerance for inaccuracy needs and how much storage space you want to give to your floating-point numbers must be balanced out:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add the main package name to the top of the file:

```
package main
```

3. Now add the imports we'll use in this file:

```
import "fmt"
```

4. Create the **main()** function:

```
func main() {
```

5. Declare an **int** and initialize it with a value of 100:

```
var a int = 100
```

6. Declare a **float32** and initialize it with a value of 100:

```
var b float32 = 100
```

7. Declare a **float64** and initialize it with a value of 100:

```
var c float64 = 100
```

8. Divide each variable by 3 and print the result to the console:

```
fmt.Println(a / 3)
```

```
fmt.Println(b / 3)
```

```
fmt.Println(c / 3)
```

```
}
```

9. Save the file and in the new folder run the following:

```
go run main.go
```

Running the preceding code shows the following output displaying **int**, **float32**, and **float64** values:

```
33
```

```
33.333332
```

```
33.333333333333336
```

In this exercise, we can see that the computer is not able to give perfect answers to this sort of division. You can also see that when doing this sort of math on integers, you don't get an error. Go ignores any fractional part of the number, which is usually not what you want. We can also see that the **float64** gives a much more accurate answer than **float32**.

While this limit seems like it would lead to problems with inaccuracy, for real-world business work, it does get the job done well enough the vast majority of the time.

Let's see what happens if we try to get our number back to 100 by multiplying it by 3:

```
package main

import "fmt"

func main() {

    var a int = 100

    var b float32 = 100

    var c float64 = 100

    fmt.Println((a / 3) * 3)

    fmt.Println((b / 3) * 3)

    fmt.Println((c / 3) * 3)

}
```

Running the preceding code shows the following output:

```
99
100
100
```

In this example, we saw that the accuracy is not as impacted as much as you'd expect. At first glance, floating-point math can seem simple, but it gets complicated quickly. When defining your floating-point variables,

typically **float64** should be your first choice unless you need to be more memory efficient.

Next, we'll look at what happens when you go beyond the limits of a number type.

Overflow and Wraparound

When you try to initialize a number with a value that's too big for the type we are using, you get an overflow error. The highest number you can have in an **int8** is 127. In the following code, we'll try to initialize it with 128 and see what happens:

```
package main

import "fmt"

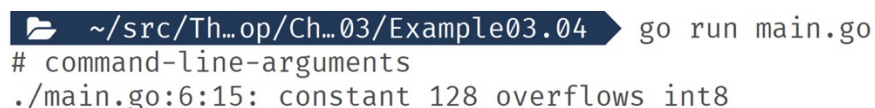
func main() {

    var a int8 = 128

    fmt.Println(a)

}
```

Running the preceding code gives the following output:

A terminal window showing the execution of a Go program. The command 'go run main.go' is entered. The output shows the file path, the command-line arguments, and an error message: './main.go:6:15: constant 128 overflows int8'.

```
~/src/Th...op/Ch...03/Example03.04 go run main.go
# command-line-arguments
./main.go:6:15: constant 128 overflows int8
```

Figure 3.03: Output after initializing with 128

This error is easy to fix and can't cause any hidden problems. The real problem is when the compiler can't catch it. When this happens, the number will ".wraparound". Wraparound means the number goes from its highest possible value to its smallest possible value. Wraparound can be easy to miss when developing your code and can cause significant problems to your users.

Exercise 3.03: Triggering Number Wraparound

In this exercise, we'll declare two small integer types: **int8** and **uint8**. We'll initialize them near their highest possible value. Then we'll use a loop statement to increment them by 1 per loop then print their value to the console. We'll be able to see exactly when they wraparound.

1. Create a new folder and add a **main.go** file.
2. In **main.go** add the main package name to the top of the file:

```
package main
```

3. Now add the imports we'll use in this file:

```
import "fmt"
```

4. Create the main function:

```
func main() {
```

5. Declare an **int8** variable with an initial value of 125:

```
var a int8 = 125
```

6. Declare an **uint8** variable with an initial value of 253:

```
var b uint8 = 253
```

7. Create a **for i** loop that runs five times:

```
for i := 0; i < 5; i++ {
```

8. Increment the two variables by 1:

```
    a++
```

```
    b++
```

9. Print the variables' values to the console:

```
    fmt.Println(i, ")", "int8", a, "uint8", b)
```

10. Close the loop:

```
    }
```

11. Close the **main()** function:

```
}
```

12. Save the file, and in the new folder run the following:

```
go run main.go
```


Running the preceding code shows the following output:

```
~/src/Th...op/Ch...03/Exercise03.03 go run main.go
0 ) int8 126 uint8 254
1 ) int8 127 uint8 255
2 ) int8 -128 uint8 0
3 ) int8 -127 uint8 1
4 ) int8 -126 uint8 2
```

Figure 3.04: Output after wraparound

In this exercise, we saw that, for signed integers, you'd end up with a negative number and for unsigned integers, it wraps around to 0. You must always consider the maximum possible number for your variable and be sure to have the appropriate type to support that number.

Next, we'll look at what you can do when you need a number that's bigger than the core types can give you.

Big Numbers

If you need a number higher or lower than **int64** or **uint64** can give, you can use the **math/big** package. This package feels a little awkward to use compared to dealing with integer types, but you'll be able to do everything you can generally do with integers using its API.

Exercise 3.04: Big Numbers

In this exercise, we're going to create a number that's larger than what is possible with Go's core number types. To show that, we'll use an addition operation. We'll also do the same to an **int** to show the difference. Then, we'll print the result to the console:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add the main package name to the top of the file:

```
package main
```

3. Now add the imports we'll use in this file:

```
import (
    "fmt"
    "math"
    "math/big"
```

)

4. Create the **main()** function:

```
func main() {
```

5. Declare an **int** and initialize with **math.MaxInt64**, which is the highest possible value for an **int64** in Go, which is defined as a constant:

```
    intA := math.MaxInt64
```

6. Add 1 to the **int**:

```
    intA = intA + 1
```

7. Now we'll create a **big int**. This is a custom type and is not based on Go's **int** type. We'll also initialize it with Go's highest possible number value:

```
    bigA := big.NewInt(math.MaxInt64)
```

8. We'll add 1 to our **big int**. You can see that this feels clumsy:

```
    bigA.Add(bigA, big.NewInt(1))
```

9. Print out the max **int** size and the values for our Go **int** and our **big int**:

```
    fmt.Println("MaxInt64: ", math.MaxInt64)
```

```
    fmt.Println("Int  :", intA)
```

```
    fmt.Println("Big Int : ", bigA.String())
```

10. Close the **main()** function:

```
}
```

11. Save the file, and in the new folder run the following:

```
go run main.go
```

Running the preceding code shows the following output:

```
~/src/Th...op/Ch...03/Exercise03.04 go run main.go
MaxInt64: 9223372036854775807
Int      : -9223372036854775808
Big Int  : 9223372036854775808
```

Figure 3.5: Output displaying large numbers with Go's number types

In this exercise, we saw that while **int** has wrapped around, **big.Int** has added the number correctly.

If you have a situation where you have a number whose value is higher than Go can manage, then the **big** package from the standard library is what you need. Next, we'll look at a special Go number type used to represent raw data.

Byte

The **byte** type in Go is just an alias for **uint8**, which is a number that has 8 bits of storage. In reality, **byte** is a significant type, and you'll see it in lots of places. A bit is a single binary value, a single on/off switch. Grouping bits into groups of 8 was a common standard in early computing and became a near-universal way to encode data. 8 bits have 256 possible combinations of "off" and "on," **uint8** has 256 possible integer values from 0 to 255. All combinations of on and off can be represented with this type.

You'll see **byte** used when reading and writing data to and from a network connection and when reading and writing data to files.

With this, we're all done with numbers. Now, let's look at how Go stores and manages text.

Text

Go has a single type to represent some text, **string**.

When you are writing some text for a **string**, it's called a string literal. There are two kinds of string literals in Go:

- Raw – defined by wrapping text in a pair of ```
- Interpreted – defined by surrounding the text in a pair of `"`

With raw, what ends up in your variable is precisely the text that you see on the screen. With interpreted, Go scans what you've written and then applies transformations based on its own set of rules.

Here's what that looks like:

```
package main

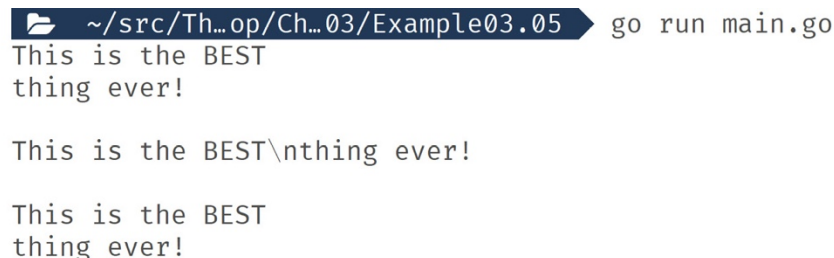
import "fmt"

func main() {
    comment1 := `This is the BEST
thing ever!`

    comment2 := `This is the BEST\nthing ever!`
    comment3 := "This is the BEST\nthing ever!"

    fmt.Print(comment1, "\n\n")
    fmt.Print(comment2, "\n\n")
    fmt.Print(comment3, "\n")
}
```

Running the preceding code gives the following output:



```
~/src/Th...op/Ch...03/Example03.05 go run main.go
This is the BEST
thing ever!

This is the BEST\nthing ever!

This is the BEST
thing ever!
```

Figure 3.6: Output printing texts

In an interpreted string, `\n` represented a new line. In our raw string, `\n` doesn't do anything. To get a new line in the raw string, we must add an actual new line in our code. The interpreted string must use `\n` to get a new line as having a real new line in an interpreted string is not allowed.

While there are a lot of things you can do with an interpreted string literal, in real-world code, the two you'll see more commonly are `\n` for a new line and occasionally `\t` for a tab.

Interpreted string literals are the most common kind in real-world code, but raw literals have their place. If you wanted to copy and paste some text that contains a lot of new lines, " or \, in it, it's easier to use **raw**.

In the following example, you can see how using raw makes the code more readable:

```
package main

import "fmt"

func main() {

    comment1 := `In "Windows" the user directory is "C:\Users\ "`

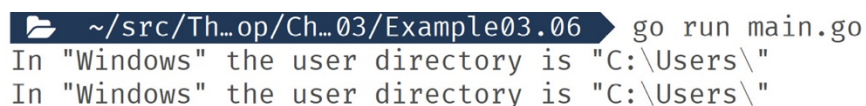
    comment2 := "In \"Windows\" the user directory is \"C:\\Users\\\\"

    fmt.Println(comment1)

    fmt.Println(comment2)

}
```

Running the preceding code shows the following output:



```
~/src/Th...op/Ch...03/Example03.06 go run main.go
In \"Windows\" the user directory is \"C:\\Users\\\"
In \"Windows\" the user directory is \"C:\\Users\\\"
```

Figure 3.7: Output for more readable code

One thing you can't have in a raw literal is a ```. If you need a literal with a ``` in it, you must use an interpreted string literal.

String literals are just ways of getting some text into a **string** type variable. Once you have the value in the variable, there are no differences.

Next, we'll look at how to work safely with multi-byte strings.

Rune

A **rune** is a type with enough storage to store a single UTF-8 multi-byte character. String literals are encoded using UTF-8. UTF-8 is a massively popular and common multi-byte text encoding standard.

The **string** type itself is not limited to UTF-8 as Go needs to also support

text encoding types other than UTF-8. **string** not being limited to UTF-8 means there is often an extra step you need to take when working with your strings to prevent bugs.

The different encodings use a different number of bytes to encode text. Legacy standards use one byte to encode a single character. UTF-8 uses up to four bytes to encode a single character. When text is in the **string** type, to allow for this variability, Go stores all strings as a **byte** collection. To be able to safely perform operations with text of any kind of encoding, single- or multi-byte, it should be converted from a **byte** collection to a **rune** collection.

Note

If you don't know the encoding of the text, it's usually safe to convert it to UTF-8. Also, UTF-8 is backward-compatible with single-byte encoded text.

Go makes it easy to access the individual bytes of a string, as shown in the following example:

1. First, we define the package, import our needed libraries, and create the **main()** function:

```
package main

import "fmt"

func main() {
```

2. We'll create a string that contains a multi-byte character:

```
    username := "Sir_King_Über"
```

3. We are going to use a **for i** loop to print out each byte of our string:

```
    for i := 0; i < len(username); i++ {

        fmt.Print(username[i], " ")

    }
```

4. Then we will close the **main()** function:

```
    }
```

Running the preceding code gives the following output:

```
~/src/Th...op/Ch...03/Example03.07 go run main.go
83 105 114 95 75 105 110 103 95 195 156 98 101 114
```

Figure 3.8: Output displaying bytes with respect to input length

The numbers printed out are the byte values of the string. There are only 13 letters in our string. However, it contained a multi-byte character, so we printed out 14 byte values.

Let's convert our bytes back to strings. This conversion uses type conversion, which we'll cover in detail soon:

```
package main

import "fmt"

func main() {
    username := "Sir_King_Über"
    for i := 0; i < len(username); i++ {
        fmt.Print(string(username[i]), " ")
    }
}
```

Running the preceding code gives the following output:

```
~/src/Th...op/Ch...03/Example03.08 go run main.go
S i r _ K i n g _ Ä b e r
```

Figure 3.9: Output displaying bytes converted as strings

The output is as expected until we get to the "Ü." That's because the "Ü" was encoded using more than one byte, and each byte on its own no longer makes sense.

To safely work with interindividual characters of a multi-byte string, you first must convert the strings slice of **byte** types to a slice of **rune** types.

Consider the following example:

```

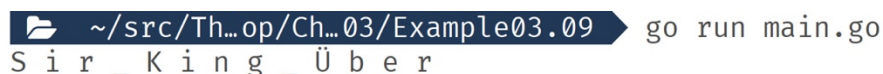
package main

import "fmt"

func main() {
    username := "Sir_King_Über"
    runes := []rune(username)
    for i := 0; i < len(runes); i++ {
        fmt.Print(string(runes[i]), " ")
    }
}

```

Running the preceding code gives the following output:



```

~/src/Th...op/Ch...03/Example03.09 go run main.go
S i r _ K i n g _ Ü b e r

```

Figure 3.10: Output displaying strings

If we do wish to work with each character in a loop like this, then using a **range** would be a better choice. When using **range**, instead of going one **byte** at a time, it moves along the string one **rune** at a time. The index is the byte offset, and the value is a **rune**.

Exercise 3.05: Safely Looping over a String

In this exercise, we'll declare a string and initialize it with a multi-byte string value. We'll then loop over the string using **range** to give us each character, one at a time. We'll then print out the byte index and the character to the console:

1. Create a new folder and add a **main.go** file.
2. In **main.go**, add the main package name to the top of the file:

```
package main
```

3. Now add the imports we'll use in this file:

```
import "fmt"
```


4. Create the **main()** function:

```
func main() {
```

5. Declare the string with a multi-byte string value:

```
    logLevel := "デバッグ"
```

6. Create a **range** loop that loops over the string, then capture the **index** and **rune** in variables:

```
    for index, runeVal := range logLevel {
```

7. Print the **index** and **rune** to the console, casting the rune to a string:

```
        fmt.Println(index, string(runeVal))
```

8. Close the loop:

```
    }
```

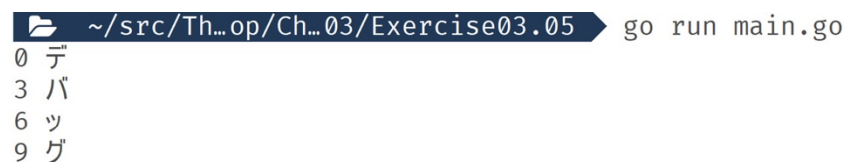
9. Close the **main()** function:

```
}
```

10. Save the file and in the new folder run the following:

```
go run main.go
```

Running the preceding code gives the following output:



```
~/src/Th...op/Ch...03/Exercise03.05 go run main.go
0 デ
3 バ
6 ツ
9 グ
```

Figure 3.11: Output after safely looping over a string

In this exercise, we demonstrated that looping over a string in a multi-byte safe way is baked right into the language. Using this method prevents you from getting invalid string data.

Another common way to find bugs is to check how many characters a string has by using **len** directly on it. Here is an example of some common ways multi-byte strings can get mishandled:

```

package main

import "fmt"

func main() {
    username := "Sir_King_Über"

    // Length of a string
    fmt.Println("Bytes:", len(username))

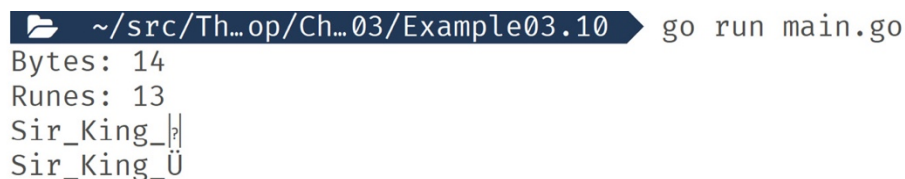
    fmt.Println("Runes:", len([]rune(username)))

    // Limit to 10 characters
    fmt.Println(string(username[:10]))

    fmt.Println(string([]rune(username)[:10]))
}

```

Running the preceding code gives the following output:



```

~/src/Th...op/Ch...03/Example03.10 go run main.go
Bytes: 14
Runes: 13
Sir_King_?
Sir_King_Ü

```

Figure 3.12: Output displaying bugs after using the `len` function

You can see that when using **len** directly on a string, you get the wrong answer. Checking the length of data input using **len** in this way would end up with invalid data. For example, if we needed the input to be exactly 8 characters long and somebody entered a multi-byte character, using **len** directly on that input would allow them to enter less than 8 characters.

When working with strings, be sure to check the **strings** package first. It's filled with useful tools that may already do what you need.

Next, let's take a close look at Go's special **nil** value.

The nil Value

nil is not a type but a special value in Go. It represents an empty value of no type. When working with pointers, maps, and interfaces (we'll cover these in the next chapter), you need to be sure they are not **nil**. If you try to interact with a **nil** value, your code will crash.

If you can't be sure whether a value is **nil** or not, you can check it like this:

```
package main

import "fmt"

func main() {

    var message *string

    if message == nil {

        fmt.Println("error, unexpected nil value")

        return

    }

    fmt.Println(&message)

}
```

Running the preceding code shows the following output:

```
error, unexpected nil value
Activity 3.01: Sales Tax Calculator
```

In this activity, we create a shopping cart application, where sales tax must be added to calculate the total:

1. Create a calculator that calculates the sales tax for a single item.
2. The calculator must take the items cost and its sales tax rate.
3. Sum the sales tax and print the total amount of sales tax required for the following items:

Item	Cost	Sales Tax Rate
Cake	0.99	7.5%
Milk	2.75	1.5%
Butter	0.87	2%

Figure 3.13:List of items with the sales tax rates

Your output should look like this:

Sales Tax Total: 0.1329

Note

The solution for this activity can be found via [this link](#).

Activity 3.02: Loan Calculator

In this activity, we must create a loan calculator for an online financial advisor platform. Our calculator should have the following rules:

1. A good credit score is a score of 450 or above.
2. For a good credit score, your interest rate is 15%.
3. For a less than good score, your interest rate is 20%.
4. For a good credit score, your monthly payment must be no more than 20% of your monthly income.
5. For a less than good credit score, your monthly payment must be no more than 10% of your monthly income.
6. If a credit score, monthly income, loan amount, or loan term is less than 0, return an error.
7. If the term of the loan is not divisible by 12 months, return an error.
8. The interest payment will be a simple calculation of loan amount * interest rate * loan term.
9. After doing these calculations, display the following details to the user:

Applicant X

Credit Score : X

Income : X

Loan Amount : X

Loan Term : X

Monthly Payment : X

Rate : X

Total Cost : X

Approved : X

This is the expected output:

```
~/src/Th...op/Ch...03/Activity03.02 go run main.go
Applicant 1
-----
Credit Score      : 500
Income            : 1000
Loan Amount       : 1000
Loan Term         : 24
Monthly Payment   : 47.916666666666664
Rate              : 15
Total Cost        : 150
Approved          : true

Applicant 2
-----
Credit Score      : 350
Income            : 1000
Loan Amount       : 10000
Loan Term         : 12
Monthly Payment   : 1000
Rate              : 20
Total Cost        : 2000
Approved          : false
```

Figure 3.14: Output of loan calculator

Note

The solution for this activity can be found via [this link](#).

Summary

In this chapter, we took a big step in working with Go's type system. We took the time to define what types are and why they are needed. We then explored each of the core types in Go. We started with the simple **bool** type, and we were able to show how critical it is to everything we do in our code. We then moved on to the number types. Go has lots of types for numbers, reflecting the control that Go likes to give developers when it comes to memory usage and accuracy. After

numbers, we looked at how strings work and how they are closely related to the rune type. With the advent of multi-byte characters, it's easy to make a mess of your text data. Go has provided power built-in features to help you get it right. Lastly, we looked at **nil** and how you use it within Go.

The concepts you've learned in this chapter have armed you with the knowledge needed to tackle Go's more complex types, such as collections and structs. We'll be looking at these complex types in the next chapter.