Exercise 1: An Important Message

In this exercise you will write a working program in Golang to display an important message on the screen.

If you are not used to typing detailed instructions for a computer then this could be one of the harder exercises in the book. Computers are very stupid and if you don't get ***every*** detail right, the computer won't understand your instructions. But if you can get this exercise done and working, then there is a good chance that you will be able to handle every exercise in the book as long as you work on it every day and don't quit.

Open the text editor and type the following text into a single file named FirstProg.go. Make sure to match what I have written exactly, including spacing, punctuation, and capitalization.

FirstProg.go

```
package main

import  "fmt"

func main() {
    fmt.Println("I am determined to learn how to code.")
    fmt.Println("Today's date is")
}
```

What You Should See

```
I am determined to learn how to code.
Today's date is
```

Anyway about the code, the different words may not be colored at all. Or if they are colored, they might be different colors than mine. These differences are fine.

I'm going to walk through this line-by-line, just to make sure you typed everything correctly.

So on our first line, package main is used to tells the Go compiler that the package should compile as an executable program instead of shared library,

The main function in the package “main” will be the entry point of our executable program

And then as you can see, there’s an import, so what that’s import used for? So fmt is short for format. So we can do our “Hello World” here!

Alright, maybe you familiar with “()” or we called it as “Parentheses”. But not with “{}” or we called it as Curly Brackets. Also you should remember that, we will use it very often so you should getting used to it. You can type curly brackets it’s located on Near “P”. there’s “[“, hold “SHIFT” and press “[“, ta-da it would summon our curly braces! Anyway if you use code-editor as sublime or visual studio code, it will automatically summon another one, so it would be like this “{}”.

Okay, let’s back into the code, as you can see there’s “func main()” thing, I will explain slowly.. “func” means function, we make a main function, and remember that one file “go” must have one “ func main()”.

Inside “main(){ “ as you can see, there’s “fmt” thing that we just mention about, for now we will called it as an Object, object as we known as an “item”, we can told them to do what the Object can do. But because fmt object is used to formatting-stuff, so we can make our first Text Outputting.

Notice that “fmt.Println()”? “fmt” object has an method, named Println(), and something inside “()” called as a **Parameter**, and parameter Is something that received by a method! And we fill it the parameter with “*insert a text here*” thing, anyway it’s called double quotation marks “ ”, so that method will show our output from that!.

Exercise 2: More Printing

Okay, now that we've gotten that first, hard assignment out of the way, we'll do another. The nice thing is that in this one, we still have a lot of the setup code (which will be nearly the same every time), but the ratio of set up to "useful" code is much better.

Type the following text into a single file named **GasolineReceipt.go**. Make sure to match what I have written exactly, including spacing, punctuation, and capitalization.

The little vertical bar ("|") that you see, that is called the "pipe" character, and you can type it using Shift + backslash ("\"). Assuming you are using a normal US keyboard, the pipe/backslash key is located between the Backspace and Enter keys.

GasolineReceipt.go

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("+-------------------------+")
    fmt.Println("|                         |")
    fmt.Println("|      CORNER STORE       |")
    fmt.Println("|                         |")
    fmt.Println("| 2015-03-29  04:38PM     |")
    fmt.Println("|                         |")
    fmt.Println("| Gallons:       10.870   |")
    fmt.Println("| Price/gallon: $ 2.089   |")
    fmt.Println("|                         |")
    fmt.Println("| Fuel total:  $ 22.71    |")
    fmt.Println("|                         |")
    fmt.Println("+-------------------------+")
}
```

What You Should See

```
+-----------------------------+
|                             |
|  CORNER STORE               |
|                             |
| 2015-03-29 04:38PM          |
|                             |
|        Gallons: 10.870      |
|   Price/gallon: $ 2.089     |
|                             |
|     Fuel total: $ 22.71     |
|                             |
+-----------------------------+
```

Exercise 3: Printing Choices

Java has two common commands used to display things on the screen. So far we have only looked at `Println()`, but `Print()` is sometimes used, too. This exercise will demonstrate the difference.

Type the following code into a single file. By reading the code, could you guess that the file must be called `PrintingChoices.go`? In future assignments, I may not tell you what to name the Go file.

PrintingChoices.go

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Alpha")
    fmt.Println("Bravo")

    fmt.Println("Charlie")
    fmt.Println("Delta")
    fmt.Println()

    fmt.Print("Echo")
    fmt.Print("Foxtrot")
    fmt.Println("Golf")
    fmt.Print("Hotel")
    fmt.Println()
    fmt.Println("India")
    fmt.Println()
    fmt.Println("This" + " " + "is" + " " + "a" + " test.")
}
```

What You Should See

```
Alpha
Bravo
Charlie

Delta
EchoFoxtrotGolf
Hotel
India

This is a test.
```

Can you figure out the difference?
Both `Print()` and `Println()` display on the screen whatever is between the quotation marks. But `Println()` moves to a new line after finishing printing, and `Print()` does not: it displays and then leaves the cursor right at the end of the line so that the following printing statement picks up from that same position on the line

You will also notice (On line 13) that we can have a `println()` statement with ***nothing*** between the parentheses. No quotation marks or anything. That statement instructs the computer to print ***nothing***, and then move the cursor to the beginning of the next line.

You may also notice On line 9 and 14 that this program has some lines with nothing on them. On the very first exercise, when I wrote that you must “match what I have written exactly, including spacing, punctuation, and capitalization”, I wasn’t ***quite*** being honest. Extra blank lines in your code are ignored by the Go compiler. You can put them in or remove them, and the program will work
exactly the same.

Anyway, on line 22, I did one more new thing. So far you have only been printing a single thing inside quotation marks. But it is perfectly fine to print more than one thing, as long as you combine those things before printing.
So on line 22, I have six Strings in quotation marks: the word “this”, a space, the word “is”, a space, the word “a”, and finally a space followed by “test” followed by a period. There is a plus sign (“`+`”) between each of those six Strings, so there are a total of five plus signs on line 22. When you put a plus sign between Strings, Go-lang adds them together to make one long thing-in-quotation-marks,
and then displays that all at once.

If you have an error in your code, it is probably on line 22. Remembering to start and stop all the quotes correctly and getting all those details right is tricky.

Today's lesson was hopefully ***relatively*** easy. Don't worry, I'll make up for it on the next one.

Exercise 4: Escape Sequences and Comments

Have you thought about what might happen if we wanted to display a quotation mark on the screen? Since everything we want to display is contained between quotation marks in the Println() statement, putting a quote ***inside*** the quotes would be a problem.

Most programming languages allow for “escape sequences”, where you signal with some sort of escape character that the next character you see shouldn’t be handled in the normal way.

The following code demonstrates many of Go’s escape sequences.

Call it EscapeSequencesComments.go

EscapeSequenceComments.go

EscapeSequenceComments.go

```go
package main

import (
    "fmt"
)

func main() {
    // This exercise demonstrates escape sequences & comments (like these)!
    fmt.Print("Learn\tGo\n\tthe\nHard\tWay\n\n")
    fmt.Print("\tLearn Go the \"Hard\" Way!\n")
    // System.out.frimp( "Learn Go the Hard Way" );
    fmt.Print("Hello\n")    // This line prints Hello.
    fmt.Print("Jello\by\n") // This line prints Jelly.
    /* The quick brown fox jumped over a lazy dog.
    10 Quick wafting zephyrs vex bold Jim. */
    fmt. /* testing */ Println("Hard to believe, eh?")
    fmt.Println("Surprised? /* abcde */ Or what did you expect?")
    fmt.Println("\\ // -=- \\ //")
    fmt.Println("\\\\ \\\\\\ \\\\\\\\") // it takes 2 to make 1
    fmt.Print("I hope you understand \"escape sequences\" now.\n")
    // and comments. :)
}
```

What You Should See

```
PS C:\Users\HP\Golang> go run .\EscapeSequenceComments.go
Learn   Go
        the
Hard    Way

        Learn Go the "Hard" Way!
Hello
Jelly
Hard to believe, eh?
Surprised? /* abcde */ Or what did you expect?
\ // -=- \ //
\\ \\\ \\\\
I hope you understand "escape sequences" now.
```

Go's escape character is a backslash ("\"), which is the same key you press to make a pipe ("|") show up but without holding Shift. All escape sequences in Go must be somewhere inside a set of quotes. \" represents a quotation mark. \t is a tab; it is the same as if you pressed the Tab key while typing your code. In most terminals, a tab will move the cursor enough to get to the next multiple of 8 columns.

It probably seems more complicated now because you've never seen it before, but when you're reading someone else's code a \t inside the quotes is less ambiguous than a bunch of blank spaces that might be spaces or might be a tab. Personally, I never ever press the Tab key inside quotation marks.
\n is a newline. When printing it will cause the output to move down to the beginning of the next line before continuing printing.

\\ is how you display a backslash.

On line 8 you will notice that the line begins with two slashes (or “forward slashes”, if you insist). This marks the line as a “comment”, which is in the program for the human programmers’ benefit. Comments are totally ignored by the computer
In fact, as shown in lines 12 and 13, the two slashes to mark a comment don’t have to be at the beginning of the line; we could write something like this:

```
System.out.println( "A" ); // prints an 'A' on the screen
```

…and it would totally work. Everything from the two slashes to the end of that line is ignored by the compiler

Line 13 does something funny. `\b` is the escape sequence for “backspace”, so it displays “Jello”, then emits a backspace character. That deletes the “o”, and then it displays a “y” (and then a `\n` to move to the next line).
Lines 14 and 15 are a block comment. Block comments begin with a `/*` (a slash then a star/asterisk) and end with an star and a slash (`*/`), whether they are on the same line or twenty lines later.
Everything between is considered a comment and ignored by the compiler.

You can see a surprising example of a block comment in line 16. And on line 17 you can see that Strings (things in quotes) take precedence over block comments. Block comments are also sometimes called “C-style” comments, since the C programming language was the first one to use them.

Lines 18 and 19 demonstrate that to get a backslash to show up on the screen you must escape the backslash with another backslash. This matters when you’re trying to deal with Windowsstyle paths; trying to open `"C:\Users\Graham_Mitchell\Desktop\foo.txt"` won’t work in Go because the compiler will try to interpret “U” as something. You have to double the backslashes. (`"C:\\Users\Graham_Mitchell\Desktop\\foo.txt"`)

Exercise 5: Saving Information in Variables

Programs would be pretty boring if the only thing you could do was print things on the screen. We would like our programs to be interactive.
Unfortunately, interactivity requires several different concepts working together, and explaining them all at once might be confusing. So I am going to cover them one at a time.
First up: variables! If you have ever taken an Algebra class, you are familiar with the concept of variables in mathematics. Programming languages have variables, too, and the basic concept is the same:

> *"A variable is a name that refers to a location that holds a value."*

Variables in Go have four major differences from math variables:

1. Variable names can be more than one letter long.
2. Variables can hold more than just numbers; they can hold words.
3. You have to choose what type of values the variable will hold when the variable is first created.
4. The value of a variable (but not its type) can change throughout the program. For example, the variable `score` might start out with a value of `0`, but by the end of the program, `score` might hold the value `413500` instead.

Okay, enough discussion. Let's get to the code! I'm not going to tell you what the name of the file is supposed to be. You'll have to figure it out for yourself.

```go
package main

import (
    "fmt"
)

func main() {
    var x, y int
    var seconds, e, checking float64
    var firstName, lastName, title string

    x = 10
    y = 400

    /*age = 39 (You should knew that golang has rules that every variable that
    we have been declared, it must being used, if there's a variable are
    declarated but we never use it, it will causing an error on compiling!)
    */

    seconds = 4.71
    e = 2.71828182845904523536
    checking = 1.89

    firstName = "Graham"
    lastName = "Mitchell"
    title = "Mr."

    fmt.Printf("The variable x contains %d\n", x)
    fmt.Printf("The value %d is stored in the variable y.\n", y)
    fmt.Printf("The experiment took %f seconds.\n", seconds)
    fmt.Printf("A favorite irrational # is Euler's number: %f\n", e)
    fmt.Printf("Hopefully you have more than $%f!\n", checking)
    fmt.Printf("My name's %s %s%s \n", title, firstName, lastName)
}
```

What You Should See

```
The variable x contains 10
The value 400 is stored in the variable y.
The experiment took 4.710000 seconds.
A favorite irrational # is Euler's number: 2.718282
Hopefully you have more than $1.890000!
My name's Mr. GrahamMitchell
```

Let me explain slowly.. So we will start from line 8, as you can see on line 8 there's "var x,y int", var is variable, "x,y" is name of our variable, actually you can declare one variable too, like : "var x int", but also you can declare more than one at the same lines, after "x,y" we meet "int" but we know it too as "Integer", int is a type data that can handle / contains a value of Numbers, like "1,2,3,4", how about "0.1"? No.. we have another type data for that.

The Structure of declaring variable is : var variableName typeData

Same thing as line 9 and 10, we declared a lot of variables too! But there's differences.. what is "float64"? float64 is a type data that can be filled by decimals numbers or like "0.31415" number, and next also there's string, so what is "string", string is a type data that can handle text, symbol, number, almost everything, but you still must keep them inside quotation marks.

And then, on line 12 and 13 we do "initialization a variable", so what is "initialization"? That is when we filling a variable with a value, but remember.. Every value must be match with the data type, wait let me explain like this :
Just let say the var "number" is "int" and var "word" is "string".

```
number = 2 // Works
number = "a" // error
words = "Lalala" // Works
words = 100 // error
```

Also on line 15 – 18 I make an alert just to remember for you that, if you don't use your variable that you have been declare, the compiler will show you an error

Line 20 – 26 is just same like line 12 and 13, you should make sure that the type data is match.

But, let's talk about line 28 – 33, maybe you wonder.. What is that "%d" thing? So uh, let me explain slowly :
if you read the Code slowly.. if there's %d, there will a variable that has a type data of Integer, so?

- %d is for Integer Variables.
- %f is for Float Variables.
- %s is for String Variables.

Every type data has their own "%" so the compiler knew what and which variable that must be shown or used.

Now, look at line 28 as you can see, there's “fmt.Printf” on inside of it we have our string / test like usual “the variable x contains” but after that we will see %d, just like what I have explain, we should insert a new comma “,” after the quotation mark and fill it with our variable!

```
fmt.Printf("There's %d, %f, also %s!\n", intVariable, floatVariable, stringVariable)
```

And ta-da now you can use variable and show them as an output!

Exercise 6: Mathematical Operations

Now that we know how to declare and initialize variables in Go, we can do some mathematics with those variables.

```go
package main

import "fmt"

func main() {
    var a, b, c, d, e, f, g int
    var x, y, z float32
    var one, two, both string

    a = 10
    b = 27
    fmt.Printf("a is %d, b is %d\n", a, b)

    c = a + b
    fmt.Printf("a+b is %d\n", c)
    d = a - b
    fmt.Printf("a-b is %d\n", d)
    e = a + b*3
    fmt.Printf("a+b*3 is %d\n", e)
    f = b / 2
    fmt.Printf("b/2 is %d\n", f)
    g = b % 10
    fmt.Printf("bmod10 is %d\n", g)
    x = 1.1
    fmt.Printf("\nx is %f\n", x)
    y = x * x
    fmt.Printf("x*x is %f\n", y)
    z = float32(b) / 2
    fmt.Printf("b/2 is %f\n", z)
    fmt.Println()

    one = "dog"
    two = "house"
    both = one + two
    fmt.Println(both)
}

```

What You Should See

```
a is 10, b is 27
a+b is 37
a-b is -17
a+b*3 is 91
b/2 is 13
bmod10 is 7

x is 1.100000
x*x is 1.210000
b/2 is 13.500000

doghouse
```

The plus sign (+) will add two integers or two doubles together, or one integer and one floating point value (in either order). With two Strings (like on line 34) it will concatenate the two Strings together.

The minus sign (-) will subtract one number from another. Just like addition, it works with two integers, two floating-point values, or one integer and one double (in either order).

An asterisk (*) is used to represent multiplication. You can also see on line 18 that Go knows about the correct order of operations. ***b*** is multiplied by 3 giving 81 and then ***a*** is added.

A slash (/) is used for division. Notice that when an integer is divided by another integer (like on line 20) the result is also an integer and not floating-point.
The percent sign (%) is used to mean 'modulus', which is essentially the remainder left over after dividing. On line 22, ***b*** is divided by 10 and the remainder (7) is stored into the variable ***g***. Modular arithmetic is a fairly simple mathematical operation that just isn't often taught in public school or even introductory university math curriculum. Wikipedia's example is good enough: we do modular arithmetic every

time we add times on a typical 12-hour clock. If it is 7 o'clock now, what time will it be in eight hours? Well, once we hit 12:00 we "wrap around", so it will be 3 o'clock. (8+7 = 15, 15-12 = 3)

Exercise 7: Getting Input from a Human

Now that we have practiced creating variables for a bit, we are going to look at the other part of interactive programs: letting the human who is running our program have a chance to type something.

```go
package main

import (
    "fmt"
)

func main() {
    var capital, say string
    var multiplyResult int
    var numberRange float64

    fmt.Println("What city is the capital of France?")
    fmt.Scanf("%s", &capital) // match the % with type data

    fmt.Println("What is 6 multiplied by 7?")
    fmt.Scanf("\n%d", &multiplyResult) // need a '\n' on next scanf because there's an enter when inputing a value

    fmt.Println("Enter a number between 0.0 and 1.0.")
    fmt.Scanf("\n%f", &numberRange)

    fmt.Println("Is there anything else you would like to say?")
    fmt.Scanf("\n%s", &say)
}

```

When you first run this program, it will only print the first line:

```
What city is the capital of France?
```

…and then it will blink the cursor at you, waiting for you to type in a word. When I ran the program, I typed the word “Paris”, but the program will work the same even if you type a different word.

Then after you type a word and press Enter, the program will continue, printing:

```
What is 6 multiplied by 7?
```

…and so on. Assuming you type in reasonable answers to each question, it will end up looking like this:

```
What city is the capital of France?
Paris
What is 6 multiplied by 7?
42
Enter a number between 0.0 and 1.0.
2.3
Is there anything else you would like to say?
No, there is not.
```

So let us talk about the code. On line 4, we always see this “fmt” thing, always exist in our code. Also we knew that fmt is used for formatting things, like output an Variables or texts. But, do you know that one library may contains many functions? Just like our “fmt” library that we always import, so this fmt can receive an input too! How? Let’s jump into line 13.

Alright, so on line 13, we meet a new function “Scanf” also it has different parameter (“%typeData”, &yourVariable), I know that you still remember how to show a variable depends of what the type data, we have meet that “%s” / “%f” thing on last exercise, on Scanf? It’s the same thing too! You should match with the type data like %s is for string, etc. And after the “%”, we meet “&” (And Symbol) this is for pointing what variable that will being pointed or used for this Input, also remember the type data it must be matched.

Wait, we are not done yet, there’s something different on line 16,19 and 22. Why the first Scanf it doesn’t need a “\n”? because the compiler knew which the first input will be executed! And so on.

Exercise 8: Storing the Human's Responses

In the last exercise, you learned how to pause the program and allow the human to type in something. But what happened to what was typed? When you typed in the answer "Paris" for the first question, where did that answer go? Well, it was thrown away right after it was typed because we didn't put any instructions to tell the Scanf function where to store it. So that is the topic of today's lesson.

```go
package main

import (
    "fmt"
)

func main() {
    var name string
    var age int
    var weight, income float64

    fmt.Print("Hello. What is your name? ")
    fmt.Scan(&name) // alternative of Scanf (We don't need do use "\n" anymore)

    fmt.Printf("Hi, %s! How old are you? ", name)
    fmt.Scan(&age)

    fmt.Printf("So you're %d, eh? That's not very old.\n", age)
    fmt.Printf("How much do you weigh, %s? ", name)
    fmt.Scan(&weight)

    fmt.Printf("%f! Better keep that quiet!!\n", weight)
    fmt.Printf("Finally, what's your income, %s? ", name)
    fmt.Scan(&income)

    fmt.Printf("Hopefully that is %f per hour", income)
    fmt.Println(" and not per year!")
    fmt.Print("Well, thanks for answering my rude questions, ")
    fmt.Printf("%s.", name)
}
```

Just like the last exercise, when you first run this your program will only display the first question and then pause, waiting for a response:

```
Hello what is your name?
```

What You Should See

```
Hello. What is your name? Brick
Hi, Brick! How old are you? 25
So you're 25, eh? That's not very old.
How much do you weigh, Brick? 192
192.0! Better keep that quiet!!
Finally, what's your income, Brick? 8.75
Hopefully that is 8.75 per hour and not per year!
Well, thanks for answering my rude questions, Brick.
```

At the top of the program we declared four variables: one String variable called ***name***, one integer variable called ***age***, and two floating-point variables named ***weight*** and ***income***

On line 13 we see the fmt.Scan(&targetVariable) that we know from the previous exercise will pause the program and let the human type in something it will package up in a String. So ***now*** where does the String they type go? In this case, we are storing that value into the String variable named “name”. The String value gets stored into a String variable. Nice. Still confused? Let’s breakdown it slowly..

```
fmt.Scan(&targetVariable) it just like targetVariable = “Your Input”
Also make sure that your input is match with what typeData of that variable.
```

Exercise 9: Calculations with User Input

Now that we know how to get input from the user and store it into variables and since we know how to do some basic math, we can now write our first *useful* program!

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       var m, kg, bmi float64
7
8       fmt.Print("Your height in m: ")
9       fmt.Scanf("%f", &m)
10
11      fmt.Print("Your weight in kg: ")
12      fmt.Scanf("\n%f", &kg)
13
14      bmi = kg / (m * m)
15
16      fmt.Printf("Your BMI is %f", bmi)
17  }
18
```

What You Should See

```
Your height in m: 1.75
Your weight in kg: 73
Your BMI is 23.836734693877553
```

This exercise is (hopefully) pretty straightforward. We have three variables (all - floats): ***m*** (meters), ***kg*** (kilograms) and ***bmi*** (body mass index). We read in values for ***m*** and ***kg***, but ***bmi***'s value comes not from the human but as the result of a calculation. On line 14 we compute the mass divided by the square of the height and store the result into ***bmi***. And then we print it out.

The body mass index (BMI) is commonly used by health and nutrition professionals to estimate human body fat in populations. So this result would be informative for a health professional. For now that's all we can do with it.

Eventually we will learn how to display a different message on the screen ***depending*** on what value is in the BMI variable, but for now this will have to do.

Exercise 10: Variables Only Hold Values

Okay, now that we can get input from the human and do calculations with it, I want to call attention to something that many of my students get confused about. The following code should compile, but it probably will not work the way you expect.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       // THIS CODE IS BROKEN UNTIL YOU FIX IT
7
8       var price, salesTax, total float64
9       price = 0
10
11      salesTax = price * 0.0825
12      total = price + salesTax
13
14      fmt.Print("How much is the purchase price? ")
15      fmt.Scanf("%f", &price)
16
17      fmt.Println("Item price:\t", price)
18      fmt.Println("Sales tax:\t", salesTax)
19      fmt.Println("Total cost:\t", total)
20  }
21
```

What You Should See

```
How much is the purchase price? 7.99
Item price: 7.99
Sales tax: 0.0
Total cost: 0.0
```

Are you surprised by the output? Did you expect the sales tax on $7.99 to show something like $0.66 instead of a big fat zero? And the total cost should have been something like $8.65, right? What happened?

What happened is that in Go (and most programming languages), ***variables can not hold formulas***. Variables can only hold values.
Look at line 11. My students sometimes think that line stores the ***formula*** price * 0.0825 into the variable ***salesTax*** and then later the human stores the value 7.99 into the variable ***price***. They think that on line 18 when we print out ***salesTax*** that the computer then "runs" the formula somehow

This is not what happens. In fact, this program shouldn't have even compiled. The variable ***price*** doesn't even have a proper value on line 11. The only reason it does have a value is because I did something sneaky on line 9.

Normally we have been declaring variables up at the top of our programs and then initializing them later. But on line 9 I declared ***price*** and initialized it with a value of 0. When you declare and initialize a variable at the same time, that is called "defining" the variable. ***salesTax*** and ***total*** are not defined on line 9, just declared.

So then on line 15 the value the human types in doesn't initialize ***price***; ***price*** already has an initial value (0). But the value the human types in (7.99 or whatever) ***does*** get ***stored*** into the variable ***price*** here. The 0 is replaced with 7.99.

From line 9 until 14 the variable ***price*** contains the value 0. When line 15 begins executing and while we are waiting for the human to type something, ***price*** still contains 0. But by the time line 15 has completed, whatever the human typed has been stored into ***price***, replacing the zero. Then from line 16 until the end of the program, the variable ***price*** contains the value 7.99 (or whatever was typed in).

So with this in mind, we can figure out what really happens on line 11. Line 11 does ***not*** store a formula into ***salesTax*** but it does store a value. What value? It takes the value of the variable ***price*** at this point in the code (which is 0), multiplies it by 0.0825 (which is still zero), and then stores that zero into ***salesTax***.

As line 11 is beginning, the value of *salesTax* is undefined. (*salesTax* is declared but not defined.) By the end of line 11, *salesTax* holds the value 0. There is no line of code that changes *salesTax* (there is no line of code that begins with salesTax =), so that value never changes and *salesTax* is still zero when it is displayed on line 18.
Line 12 is similar. It takes the value of *price* right then (zero) and adds it to the value of *salesTax* right then (also zero) and stores the sum (zero) into the variable *total*. And *total*'s value is never changed, and *total* does not somehow "remember" that its value came from a formula involving some variables.

So there you have it. Variables hold values, not formulas. Computer programs are not a set of rules, they are a *sequence* of instructions that the computer executes *in order*, and things later in your code depend on what happened before.

Exercise 11: Variable Modification Shortcuts

The value of a variable can change over time as your program runs. (It won't change unless you write code to change it, but it ***can*** change is what I'm saying.)
In fact, this is pretty common. Something we do pretty often is take a variable and add something to it. For example, let's say the variable ***x*** contains the value `10`. We want to add `2` to it so that ***x*** now contains `12`.

We can do this:

```
x = 10 // x is 10, tempX is undefined
tempX = x + 2 // x is still 10, tempX is now 12
x = tempX // x has been changed to 12
```

This will work, but it is annoying. If we want, we can take advantage of the fact that a variable can have one value at the beginning of a line of code and have a different value stored in it by the end. So we can write something like this:

```
y = 10    // y is 10
y = 2 + y // y *becomes* (2 plus the current value of y)
```

…which is identical to the previous example. Okay, now to the code!

```go
package main

import "fmt"

func main() {
    var i, j, k int

    i = 5
    j = 5
    k = 5
    fmt.Printf("i: %d\tj: %d\tk: %d\n", i, j, k)
    i = i + 3
    j = j - 3
    k = k * 3
    fmt.Printf("i: %d\tj: %d\tk: %d\n\n", i, j, k)

    i = 5
    j = 5
    k = 5
    fmt.Printf("i: %d\tj: %d\tk: %d\n", i, j, k)
    i += 3
    j -= 3
    k *= 3
    fmt.Printf("i: %d\tj: %d\tk: %d\n\n", i, j, k)

    i, j, k = 5, 5, 5
    fmt.Printf("i: %d\tj: %d\tk: %d\n", i, j, k)
    i += 1
    j -= 2
    k *= 3
    fmt.Printf("i: %d\tj: %d\tk: %d\n\n", i, j, k)

    i, j = 5, 5
    fmt.Printf("i: %d\tj: %d\n", i, j)
    i = +1 // Oops!
    j = -2
    fmt.Printf("i: %d\tj: %d\n\n", i, j)

    i, j = 5, 5
    fmt.Printf("i: %d\tj: %d\n", i, j)
    i++
    j--
    fmt.Printf("i: %d\tj: %d\n", i, j)
}
```

What You Should See

```
i: 5    j: 5    k: 5
i: 8    j: 2    k: 15

i: 5    j: 5    k: 5
i: 8    j: 2    k: 15

i: 5    j: 5    k: 5
i: 6    j: 3    k: 15

i: 5    j: 5
i: 1    j: -2

i: 5    j: 5
i: 6    j: 4
```

Hopefully lines 6 - 20 are nice and boring. We create three variables, give them values, display them, change their values and print them again. Then starting on line 17 we give the variables the same values they started with and print them.

On line 21 we see something new: a shortcut called a "compound assignment operator." The `i += 3` means the same as `i = i + 3`: "take the current value of *i*, add `3` to it, and store the result as the new value of *i*. When we say it out loud, we would say "i plus equals 3."

On line 22 we see `-=` ("minus equals"), which subtracts 3 from *j*, and the next line demonstrates `*=`, which multiplies. There is also `/=`, which divides whatever variable is on the left-hand side by whatever value the right-hand side ends up equaling. (" Modulus equals" (`%=`) also exists, which sets the variable on the left-hand side equal to whatever the remainder is when its previous value is divided by whatever is on the right. Whew.)

Lines 28 through 30 are basically the same as lines 21 through 23 except that we are no longer using `3` as the number to add, subtract or multiply.

Line 35 might look like a typo, and if you wrote this in your own code it probably would be. Notice that instead of `+=` I wrote `=+`. This will compile, but it is not interpreted the way you'd expect. The compiler sees `i = +1;`, that is, "Set *i* equal to positive 1." And line 36 is similar: "Set *j* equal to negative 2." So watch for that.

On line 41 we see one more shortcut: the "post-increment operator." `i++` just means "add 1 to whatever is in *i*." It's the same as writing `i = i + 1` or `i += 1`. Adding `1` to a variable is ***super*** common. (You'll see.) That's why there's a special shortcut for it.

And finally on the next line we see the "post-decrement operator": `j--`. It subtracts 1 from the value in *j*.

Today's lesson is unusual because these shortcuts are optional. You could write code your whole life and never use them. But most programmers are lazy and don't want to type any more than they have to, so if you ever read other people's code you will see these pretty often.

Especially `i++`. You will see that ***all*** the time.

Exercise 12: Boolean Expressions

So far we have only seen three types of variables:

int integers, hold numbers (positive or negative) with no fractional parts **double** "double-precision floating-point" numbers (positive or negative) that could have a fractional part **String** a string of characters, hold words, phrases, symbols, sentences, whatever But as a wise man once said, "There is another…." A "Boolean" variable (named after the mathematician George Boole) cannot hold numbers or words. It can only store one of two values: true or false. That's it. We can use them to perform logic. To the code!

```go
package main

import "fmt"

func main() {
    var a, b, c, d, e, f bool
    var x, y float64

    fmt.Print("Give me two numbers. First: ")
    fmt.Scanf("%f", &x)
    fmt.Print("Second: ")
    fmt.Scanf("\n%f", &y)

    a = (x < y)
    b = (x <= y)
    c = (x == y)
    d = (x != y)
    e = (x > y)
    f = (x >= y)

    fmt.Printf("%0.1f is LESS THAN %0.1f: %t\n", x, y, a)

    fmt.Printf("%0.1f is LESS THAN / EQUAL TO %0.1f: %t\n", x, y, b)
    fmt.Printf("%0.1f is EQUAL TO %0.1f: %t\n", x, y, c)
    fmt.Printf("%0.1f is NOT EQUAL TO %0.1f: %t\n", x, y, d)
    fmt.Printf("%0.1f is GREATER THAN %0.1f: %t\n", x, y, e)
    fmt.Printf("%0.1f is GREATER THAN / EQUAL TO %0.1f: %t\n", x, y, f)
    fmt.Println()

    fmt.Println(!(x < y), (x >= y))
    fmt.Println(!(x <= y), (x > y))
    fmt.Println(!(x == y), (x != y))
    fmt.Println(!(x != y), (x == y))
    fmt.Println(!(x > y), (x <= y))
    fmt.Println(!(x >= y), (x < y))
}
```

What You Should See

```
Give me two numbers. First: 3
```

```
Second: 4
3.0 is LESS THAN 4.0: true
3.0 is LESS THAN / EQUAL TO 4.0: true
3.0 is EQUAL TO 4.0: false
3.0 is NOT EQUAL TO 4.0: true
3.0 is GREATER THAN 4.0: false
3.0 is GREATER THAN / EQUAL TO 4.0: false
false false
false false
true true
false false
true true
true true
```

Before I start, I want to tell you that if you want to do an Output of Boolean Variable, use "%t" because it's a verbs for Boolean Expression. As you can see on line 21 I typed that for a Boolean Expression.

On line 14 the Boolean variable ***a*** is set equal to something strange: the result of a comparison. The current value in the variable ***x*** is compared to the value of the variable ***y***. If ***x***'s value is less than ***y***'s, then the comparison is true and the Boolean value `true` is stored into ***a***. If ***x*** is not less than ***y***, then the comparison is false and the Boolean value `false` is stored into ***a***. (I think that is easier to understand than it is to write.)

Line 15 is similar, except that the comparison is "less than or equal to", and the Boolean result is stored into ***b***.

Line 16 is "equal to": ***c*** will be set to the value `true` if ***x*** holds the same value as ***y***. The comparison in line 17 is "not equal to". Lines 18 and 19 are "greater than" and " greater than or equal to", respectively.

On lines 21 through 27, we display the values of all those Boolean variables on the screen.

Line 30 through line 35 introduce the “not” operator, which is an exclamation point (!). It takes the logical opposite. So on line 30 we display the logical negation of “x is less than y?”, and we also print out the truth value of “x is greater than or equal to y?”, which are equivalent. (The opposite of “less than” is “greater than or equal to”.) Lines 31 through 35 show the opposites of the remaining relational operators.

Exercise 13: Comparing Strings

In this exercise we will see something that causes trouble for beginners trying to learn Java: the regular relational operators do work too with Strings, not only numbers.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       var word string
7       var yep, nope bool
8
9       fmt.Println("Type the word \"weasel\", please.")
10      fmt.Scanf("%s", &word)
11
12      yep = word == "weasel"
13      nope = word != "weasel"
14
15      fmt.Println("You typed what was requested: ", yep)
16      fmt.Println("You ignored polite instructions: ", nope)
17  }
```

What You Should See

```
Type the word "weasel", please.
no
You typed what was requested: false
You ignored polite instructions: true
```

Before I go to new exercise, maybe you will have a question like.. Why on line 15 we didn't use "%" thing to show a variable? Because the Variable are located on very corner of the code, so it's okay to make it just like that.

Exercise 14: Compound Boolean Expressions

Sometimes we want to use logic more complicated than just “less than” or “equal to ”. Imagine a grandmother who will only approve you dating her grandchild if you are older than 25 ***and*** younger than 40 ***and*** either rich or really good looking. If that grandmother was a programmer and could convince applicants to answer honestly, her program might look a bit like this:

```go
package main

import "fmt"

func main() {
    var age int
    var income, cute float64
    var allowed bool

    fmt.Print("Enter your age: ")
    fmt.Scanf("%d", &age)

    fmt.Print("Enter your yearly income: ")
    fmt.Scanf("\n%f", &income)

    fmt.Print("How cute are you, on a scale from 0.0 to 10.0? ")
    fmt.Scanf("\n%f", &cute)

    allowed = (age > 25 && age < 40 && (income > 50000 || cute >= 8.5))

    fmt.Println("Allowed to date my grandchild? ", allowed)
}
```

What You Should See

```
Enter your age: 40
Enter your yearly income: 49000
How cute are you, on a scale from 0.0 to 10.0? 7.5
Allowed to date my grandchild? false
```

So we can see that for complicated Boolean expressions you can use parentheses to group things, and you use the symbols `&&` to mean “AND” and the symbols `||` to mean “OR”.

This next little bit is going to be a little bit weird, because I'm going to show you the "truth tables" for AND and OR, and you'll have to think of "AND" as an ***operation*** that is being performed on two values instead of a conjunction. Here is Here is the truth table for AND:

Inputs		Output
A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

You read the tables this way: let's pretend that our shallow grandmother has decided that she will only go on a cruise if it is cheap AND the alcohol is included in the price. So we will pretend that statement A is "the cruise is cheap" and statement B is "the alcohol is included". Each row in the table is a possible cruise line.

Row 1 is a cruise where both statements are true. Will grandmother be excited about cruise #1? Yes! "Cruise is cheap" is true and "alcohol is included" is true, so " grandmother will go" (A && B) is also true.

Cruise #2 is cheap, but the alcohol is ***not*** included (statement B is false). So grandmother isn't interested: (A && B) is false when A is true and B is false.
Clear? Now here is the truth table for OR:

Inputs		Output
A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

Let's say that grandmother will buy a certain used car if it is really cool-looking OR if it gets great gas mileage. Statement A is "car is cool looking", B is "good miles per gallon" and the result, A OR B, determines if it is a car grandmother would want.

Car #1 is awesome-looking and it also goes a long way on a tank of gas. Is grandmother interested? Heck, yes! We can say that the value `true` ORed with the value `true` gives a result of `true`.

In fact, the only car grandmother won't like is when both are false. An expression involving OR is only false when BOTH of its components are false.

Exercise 15: Making Decisions with If Statements

Hey! I really like this exercise. You suffered through some pretty boring ones there, so it's time to learn something that is useful and not super difficult.

We are going to learn how to write code that has decisions in it, so that the output isn't always the same. The code that gets executed changes depending on what the human enters

```go
package main

import "fmt"

func main() {
    var age int

    fmt.Print("How old are you? ")
    fmt.Scanf("%d", &age)

    fmt.Println("You are: ")
    if age < 13 {
        fmt.Println("\ttoo young to create a Facebook account")
    }
    if age < 16 {
        fmt.Println("\ttoo young to get a driver's license")
    }
    if age < 18 {
        fmt.Println("\ttoo young to get a tattoo")
    }
    if age < 21 {
        fmt.Println("\ttoo young to drink alcohol")
    }
    if age < 35 {
        fmt.Println("\ttoo young to run for President of the U.S.")
        fmt.Println("\t\t(How sad!)")
    }
}
```

What You Should See

```
How old are you? 17
You are:
too young to get a tattoo
too young to drink alcohol
too young to run for President of the U.S.
    (How sad!)
```

Okay, this is called an “if statement”. An `if` statement starts with the keyword `if`, followed by a “condition” in parentheses. The condition must be a Boolean expression that evaluates to either `true` or `false`.
When the condition of the `if` statement is true, all the code in the body of the `if` statement is executed. When the condition of the `if` statement is false, all the code in the body is skipped. You can have as many lines of code as you want inside the body of an `if` statement; they will all be executed or skipped as a group.

Notice that when I ran the code, I put in `17` for my age. Because 17 is not less than 13, the condition on line 12 is false, and so the code in the body of the first `if` statement (lines 13 and 14) was skipped.

The second `if` statement was also false because 17 is not less than 16, so the code in its body (lines 16 and 17) was skipped, too.

The condition of the third `if` statement was true: 17 ***is*** less than 18, so the body of the third `if` statement was not skipped; it was executed and the phrase “too young to get a tattoo” ***was*** printed on the screen. The remaining `if` statements in the exercise are all true

The final `if` statement contains two lines of code in its body, just to show you what it would look like.

Exercise 16: More If Statements

There is almost nothing new in this exercise. It is just more practice with if statements, because they're pretty important. It will also help you to remember the relational operators

```go
package main

import "fmt"

func main() {
    var first, second float64

    fmt.Print("Give me two numbers. First: ")
    fmt.Scanf("%f", &first)
    fmt.Print("Second: ")
    fmt.Scanf("\n%f", &second)

    if first < second {
        fmt.Printf("%f is LESS THAN %f\n", first, second)
    }
    if first <= second {
        fmt.Printf("%f is LESS THAN/EQUAL TO %f\n", first, second)
    }
    if first == second {
        fmt.Printf("%f is EQUAL TO %f\n", first, second)
    }
    if first >= second {
        fmt.Printf("%f is GREATER THAN/EQUAL TO %f\n", first, second)
    }
    if first > second {
        fmt.Printf("%f is GREATER THAN %f\n", first, second)
    }
    if first != second {
        fmt.Printf("%f is NOT EQUAL TO %f\n", first, second)
    }
}
```

What You Should See

```
Give me two numbers. First: 3
Second: 4
3.0 is LESS THAN 4.0
3.0 is LESS THAN/EQUAL TO 4.0
3.0 is NOT EQUAL TO 4.0
```

Actually nothing much we can talk about this, that actually we can have many ifs as we can.

Exercise 17: Otherwise (If Statements with Else)

So, `if` statements are pretty great. Almost every programming language has them, and you use them ALL the time. In fact, `if` statements alone are functional enough that you could do a lot just using `if` statements.
But sometimes, having something else could make things a little more convenient. Like this example:

quick! What is the logical opposite of the following expression?

```
if (onGuestList || age >= 21 || (gender == "F" && attractiveness >= 8))
```

Eh? Got it yet? Well, if you said

```
if (!(onGuestList || age >= 21 || (gender == "F" && attractiveness >= 8)))
```

…then you're right and you're my kind of person. Clever and knows when to let the machine do the work for you. If you said

```
if (!onGuestList && age < 21 && (! gender == ("F" || attractiveness < 8))
```

…then you're right and… nice job. That's actually pretty tough to do correctly. But what about the logical opposite of this:

```
if (expensiveDatabaseThatTakes45SecondsToLookup(userName, password) == true)
```

Do we really want to write

```
if (expensiveDatabaseThatTakes45SecondsToLookup(userName, password) == false)
```

…because now we're having to wait 90 seconds to do two `if` statements instead of 45 seconds. So fortunately, programming languages give us something `else`. (Yeah, sorry. Couldn't resist.)

```go
package main

import "fmt"

func main() {
    var age int = 22
    var onGuestList bool = false
    var allure float64 = 7.5
    var gender string = "F"

    if onGuestList || age >= 21 || (gender == "F" && allure >= 8) {
        fmt.Println("You are allowed to enter the club.")
    } else { // GoLang has a rules that the curly braces must be one line with the else
        fmt.Println("You are not allowed to enter the club.")
    }
}
```

What You Should See

```
You are allowed to enter the club.
```

So what the keyword else means is this: look at the preceding if statement. Was the condition of that if statement true? If so, skip. If that previous if statement did ***not*** run, however, then the body of the else statement will be executed. "If ***blah blah blah*** is true, then run this block of code. Otherwise (else), run this different block of code instead."

Else is ***super*** convenient because then we don't ***have*** to figure out the logical opposite of some complex Boolean expression. We can just say else and let the computer deal with it.

An else is ***only legal*** immediately after an if statement ends. (Technically it is only allowed after the closing of the block of code that is the body of an if statement.)

Exercise 18: If Statements with Strings

A few exercises back you learned how comparing Strings is not as easy as comparing numbers. So let's review with an example you can actually test out.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       var secret, guess string = "please", "" // Inizialiation when Declarating on GoLang has rules that must be filled
7       // every variable with the same line as declaration line
8
9       fmt.Print("What's the secret word? ")
10      fmt.Scanf("%s", &guess)
11
12      if guess == secret {
13          fmt.Println("That's correct!")
14      } else {
15          fmt.Printf("No, the secret word isn't \"%s\".", guess)
16      }
17
18  }
```

What You Should See

```
What's the secret word? abracadabra
No, the secret word isn't "abracadabra".
```

Or, when you get it right:

```
What's the secret word? please
No, the secret word isn't "abracadabra".
```

Notice that as usual I'm sneaking something ***else*** new into this exercise. On line 6 instead of just declaring ***secret*** I also gave it a value. That is, I "defined" it (declared and initialized all at once).

Exercise 19: Mutual Exclusion with Chains of If and Else

In the previous exercise, we saw how using `else` can make it easier to include a chunk of alternative code that you want to run when an `if` statement did *not* happen. But what if the alternative code is… another `if` statement?

```go
package main

import "fmt"

func main() {
    var bmi float64

    fmt.Print("Enter your BMI: ")
    fmt.Scanf("%f", &bmi)
    fmt.Print("BMI category: ")
    if bmi < 15.0 {
        fmt.Println("very severely underweight")
    } else if bmi <= 16.0 {
        fmt.Println("severely underweight")
    } else if bmi < 18.5 {
        fmt.Println("underweight")
    } else if bmi < 25.0 {
        fmt.Println("normal weight")
    } else if bmi < 30.0 {
        fmt.Println("overweight")
    } else if bmi < 35.0 {
        fmt.Println("moderately obese")
    } else if bmi < 40.0 {
        fmt.Println("severely obese")
    } else {
        fmt.Println("very severely/\"morbidly\" obese")
    }
}
```

What You Should See

```
Enter your BMI: 22.5
BMI category: normal weight
```

(*Note*: Although BMI is a very good estimate of human body fat, the formula doesn't work well for athletes with a lot of muscle, or people who are extremely short or very tall. If you are concerned about your BMI, check with a registered dietitian or your doctor.)

Notice that even though several of the `if` statements might have all been true, only the ***first*** true `if` statement printed its message on the screen. No other messages were printed: only one. That's the power of using `else` with `if`.

On line 11 there is an `if` statement that checks if your BMI is less than `15.0`, and if so, displays the proper category for that body mass index.

Line 13 begins with an `else`. That else pays attention to the preceding `if` statement – the one on line 11 – to determine if it should run its body of code or skip it automatically. Assuming you typed in a BMI of `22.5`, then the preceding `if` statement was not true and did not run. Because that `if` statement failed, the else will automatically execute its body of code.

However, that body of code starts ***right after*** the word `else` with a new `if` statement! This means that the statement `if ( bmi <= 16.0 )` will ***only*** be considered when the previous `if` statement was false.

Exercise 20: More Chains of Ifs and Else

Okay, let's look a little more at making chains of conditions using `else` and `if`.
A confession: although I did attend UT Austin I don't think this is their real admissions criteria. Don't rely on the output of this program when deciding whether or not to apply to a back-up school.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       var math int
7
8       fmt.Println("Welcome to the UT Austin College Admissions Interface!")
9       fmt.Print("Please enter your SAT math score (200-800): ")
10      fmt.Scanf("%d", &math)
11
12      fmt.Print("Admittance status: ")
13
14      if math >= 790 {
15          fmt.Print("CERTAIN ")
16      } else if math >= 710 {
17          fmt.Print("SAFE ")
18      } else if math >= 580 {
19          fmt.Print("PROBABLE ")
20      } else if math >= 500 {
21          fmt.Print("UNCERTAIN ")
22      } else if math >= 390 {
23          fmt.Print("UNLIKELY ")
24      } else { // below 390
25          fmt.Print("DENIED ")
26      }
27      fmt.Println()
28  }
```

What You Should See

```
Welcome to the UT Austin College Admissions Interface!
Please enter your SAT math score (200-800): 730
Admittance status: SAFE
```

In this exercise I omitted all the curly braces that delimit the blocks of code that are the bodies of each `if` statement. Because I only want there to be a single statement

inside the body of each `if` statement, this is okay. If I wanted there to be more than one line of code then I would have to put the curly braces back.

Anyway, in the previous exercise I wrote about how putting `else` in front of an `if` statement makes it defer to the previous `if` statement. When the previous one is true and executes the code in its body, the current one skips automatically (and all the rest of the `else if` statements in the chain will skip, too). This has the effect of making it so that only the ***first*** true value triggers the `if` statement and all the rest don't run. We sometimes say that the `if` statements are "mutually exclusive": exactly one of them will execute. Never fewer than one, never more than one.

Exercise 21: Nested If Statements

You saw a glimpse of this in the previous exercise, but you can put just about anything you like inside the body of an `if` statement including other `if` statements. This is called “nesting”, and an `if` statement which is inside another is called a “ nested if”.

Here’s an example of using that to do something useful.

```go
package main

import "fmt"

func main() {
    var title, first, last, gender, married string
    var age int

    fmt.Print("First name: ")
    fmt.Scanf("%s", &first)
    fmt.Print("Last name: ")
    fmt.Scanf("%s", &last)
    fmt.Print("Gender (M/F): ")
    fmt.Scanf("%s", &gender)
    fmt.Print("Age: ")
    fmt.Scanf("%d", &age)

    if age < 20 {
        title = first
    } else {
        if gender == "F" {
            fmt.Print("Are you married, " + first + "? (Y/N): ")
            fmt.Scanf("%s", &married)
            if married == "Y" {
                title = "Mrs."
            } else {
                title = "Ms."
            }
        } else {
            title = "Mr."
        }
    }
    fmt.Println("\n" + title + " " + last)
}

```

What You Should See

```
First name: Graham
Last name: Mitchell
Gender (M/F): M
Age: 40
Mr. Mitchell
```

You have probably figured out that I like to mix things up a little bit to keep you on your toes. Did you notice what I did differently this time?
Normally I declare all my variables at the top of the program and give them values (or "initialize" them) later. But you don't actually have to declare a variable until you're ready to use it. So this time, I declared all my variables (except ***title***) on the same line I put a value into them for the first time.

Why didn't I declare ***title*** on line 19, then? Because then it wouldn't be in "scope" later. ***Scope*** refers to the places in your program where a variable is visible. The general rule is that a variable is in scope once it is declared and from that point forward in the code until the block ends that it was declared in. Then the variable goes out of scope and can't be used any more.

Let us look at an example: on line 23 I defined (declared and initialized) a String variable called ***married***. It is declared inside the body of the female-gender `if` statement. This variable exists from line 23 down through line 32 at the close curly brace of that `if` statement's body block. The variable ***married*** is not in scope anywhere else in the program; referring to it on lines 1 through 22 or on lines 29 through 34 would give a compiler error.

This is why I had to declare ***title*** towards the beginning of the program. If I had declared it on line 19, then the variable would have gone out of scope one line later, when the close curly brace of the younger-than-20 block occurred. Because I need ***title*** to be visible all the way down through line 33, I need to make sure I declare it inside the block of code that ends on line 34.

Exercise 22: Making Decisions with a Big Switch

`if` statements aren't the only way to compare variables with a value in Java. There is also something called a `switch`. I don't use them very often, but you should be familiar with them anyway in case you read someone else's code that uses one.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       var month, days int
7       var monthName string
8
9       fmt.Print("Which month? (1-12) ")
10      fmt.Scanf("%d", &month)
11
12      switch month {
13      case 1:
14          monthName = "January"
15          break
16      case 2:
17          monthName = "February"
18          break
19      case 3:
20          monthName = "March"
21          break
22      case 4:
23          monthName = "April"
24          break
25      case 5:
26          monthName = "May"
27          break
28      case 6:
29          monthName = "June"
30          break
31      case 7:
32          monthName = "July"
33          break
34      case 8:
35          monthName = "August"
36          break
37      case 9:
38          monthName = "September"
```

```
        break
    case 10:
        monthName = "October"
        break
    case 11:
        monthName = "November"
        break
    case 12:
        monthName = "December"
        break
    default:
        monthName = "error"
    }
    /* Thirty days hath September
    April, June and November
    All the rest have thirty-one
    Except the second month alone....
    */

    switch month {
    case 9:
    case 4:
    case 6:
    case 11:
        days = 30
        break
    case 2:
        days = 28
        break
    default:
        days = 31
    }

    fmt.Printf("%d days hath %s", days, monthName)
}
```

What You Should See

```
Which month? (1-12) 4
30 days hath April
```

A switch statement starts with the keyword switch and then some parentheses. Inside the parentheses is a single variable (or an expression that simplifies to a single value). Then there's an open curly brace.

Inside the body of the switch statement are several case statements that begin with the keyword case and then a value that the variable up in the parentheses might equal. Then there's a colon (:). You don't see colons very often in Go.

After the case, the value, and the colon is some code. It can be as many lines of code as you like except that you're not allowed to declare any variables inside the switch statement. Then after all the code is the keyword break. The break marks the end of the case.

When a switch statement runs, the computer figures out the current value of the variable inside the parentheses. Then it looks through the list of cases, one at a time, looking for a match. When it finds a match it moves from the left side where the cases are to the right side and starts running code until it is stopped by a break.

If none of the cases match and there is a default case (it's optional), then the code in the default case will be run instead.

The second example starts on line 58 and demonstrates that once the switch statement finds a case that matches, it really does run code on the right side until it hits a break statement. It will even fall through from one case to another.

We can take advantage of this fall-through behavior to do clever things sometimes, like the code to figure out the number of days in a month. Since September, April, June and November all have 30 days, we can just put all their cases in a row and let it fall through for any of those to run the same thing.

Anyway, I won't use switch statements again in this book because I just virtually never find a good use for them, but it does exist and at least I can say that you saw it.

Exercise 23: More String Comparisons

Well, you have learned that you can test if Strings are the same by using ==. But I think you're finally ready to see how we can compare Strings for alphabetical ordering.

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    var name string

    fmt.Print("Make up the name of a programming language! ")
    fmt.Scanf("%s", &name)

    if strings.Compare(name, "c++") < 0 {
        fmt.Println(name + " comes BEFORE c++")
    }
    if strings.Compare(name, "c++") == 0 {
        fmt.Println("c++ isn't a made-up language!")
    }
    if strings.Compare(name, "c++") > 0 {
        fmt.Println(name + " comes AFTER c++")
    }

    if strings.Compare(name, "go") < 0 {
        fmt.Println(name + " comes BEFORE go")
    }
    if strings.Compare(name, "go") == 0 {
        fmt.Println("go isn't a made-up language!")
    }
    if strings.Compare(name, "go") > 0 {
        fmt.Println(name + " comes AFTER go")
    }
```

```
33
34      if strings.Compare(name, "java") < 0 {
35          fmt.Println(name + " comes BEFORE java")
36      }
37      if strings.Compare(name, "java") == 0 {
38          fmt.Println("java isn't a made-up language!")
39      }
40      if strings.Compare(name, "java") > 0 {
41          fmt.Println(name + " comes AFTER java")
42      }
43
44      if strings.Compare(name, "lisp") < 0 {
45          fmt.Println(name + " comes BEFORE lisp")
46      }
47      if strings.Compare(name, "lisp") == 0 {
48          fmt.Println("lisp isn't a made-up language!")
49      }
50      if strings.Compare(name, "lisp") > 0 {
51          fmt.Println(name + " comes AFTER lisp")
52      }
53
54      if strings.Compare(name, "python") < 0 {
55          fmt.Println(name + " comes BEFORE python")
56      }
57      if strings.Compare(name, "python") == 0 {
58          fmt.Println("python isn't a made-up language!")
59      }
60      if strings.Compare(name, "python") > 0 {
61          fmt.Println(name + " comes AFTER python")
62      }
63
64      if strings.Compare(name, "ruby") < 0 {
65          fmt.Println(name + " comes BEFORE ruby")
66      }
67      if strings.Compare(name, "ruby") == 0 {
68          fmt.Println("ruby isn't a made-up language!")
69      }
```

```
70      if strings.Compare(name, "ruby") > 0 {
71          fmt.Println(name + " comes AFTER ruby")
72      }
73
74      if strings.Compare(name, "visualbasic") < 0 {
75          fmt.Println(name + " comes BEFORE visualbasic")
76      }
77      if strings.Compare(name, "visualbasic") == 0 {
78          fmt.Println("visualbasic isn't a made-up language!")
79      }
80      if strings.Compare(name, "visualbasic") > 0 {
81          fmt.Println(name + " comes AFTER visualbasic")
82      }
83  }
```

What You Should See

```
Make up the name of a programming language! juniper
juniper comes AFTER c++
juniper comes AFTER go
juniper comes AFTER java
juniper comes BEFORE lisp
juniper comes BEFORE python
juniper comes BEFORE ruby
```

```
juniper comes BEFORE visualbasic
```

You compare Strings to each other using the String object's .Compare() method. The .Compare() method doesn't work the way you probably expect, but there is genius in how it works.

The comparison involves two Strings. The first String is the one to the left of the .Compare(). The second String is the one in the parentheses. And the comparison simplifies to an integer! If we call the first one ***self*** and the second one ***other*** it would look like this:

```
strings.Compare(self, other) //it will return an Integer
```

So ***self*** compares itself to ***other***. If ***self*** is identical to ***other*** (the same length and every character the same), then ***n*** would be set to 0. If ***self*** comes before ***other*** alphabetically, then ***n*** would be set to a negative number (a number less than 0). And if ***self*** comes after ***other*** alphabetically, then ***n*** would be set to a positive number (a number greater than 0).

Exercise 24: Choosing Numbers Randomly

We're going to spend a couple of exercises on something you don't always see in programming books: how to have the computer choose a "random" number within a certain range. This is because you can write a *lot* of software without needing the computer to randomly pick a number. However, having random numbers will let us make some simple interactive games, and that is easily worth the pain of this slightly weird concept.

```go
package main

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

func main() {
    var a, b, c, d int
    var r, rps float64
    rand.Seed(time.Now().UTC().UnixNano())

    rps = rand.Float64()
    if rps < 0.3333333 { // will be true 1/3 of the time
        fmt.Println("ROCK")
    } else if rps < 0.6666667 {
        fmt.Println("PAPER")
    } else {
        fmt.Println("SCISSORS")
    }

    // pick four random integers, each 1-10
    a = 1 + (int)(10*rand.Float64())
    b = 1 + (int)(10*rand.Float64())
    c = 1 + (int)(10*rand.Float64())
    d = 1 + (int)(10*rand.Float64())
    fmt.Printf("1-10:\t%d\t%d\t%d\t%d\n", a, b, c, d)

    // pick four random integers, each 1-6
    a = 1 + (int)(6*rand.Float64())
    b = 1 + (int)(6*rand.Float64())
    c = 1 + (int)(6*rand.Float64())
    d = 1 + (int)(6*rand.Float64())
    fmt.Printf("1-6:\t%d\t%d\t%d\t%d\n", a, b, c, d)
```

```
36
37      // pick a single random integer, 1-100
38      a = 1 + (int)(100*rand.Float64())
39      fmt.Printf("1-100:\t%d\t%d\t%d\t%d\n", a, a, a, a)
40
41      // pick four random integers, each 1-100
42      a = 1 + (int)(100*rand.Float64())
43      b = 1 + (int)(100*rand.Float64())
44      c = 1 + (int)(100*rand.Float64())
45      d = 1 + (int)(100*rand.Float64())
46      fmt.Printf("1-100:\t%d\t%d\t%d\t%d\n", a, b, c, d)
```

```
47
48      // pick four random integers, each 0-99
49      a = 0 + (int)(100*rand.Float64())
50      b = 0 + (int)(100*rand.Float64())
51      c = (int)(100 * rand.Float64())
52      d = (int)(100 * rand.Float64())
53      fmt.Printf("0-99:\t%d\t%d\t%d\t%d\n", a, b, c, d)
54
55      // pick four random integers, each 10-20
56      a = 10 + (int)(11*rand.Float64())
57      b = 10 + (int)(11*rand.Float64())
58      c = 10 + (int)(11*rand.Float64())
59      d = 10 + (int)(11*rand.Float64())
60      fmt.Printf("10-20:\t%d\t%d\t%d\t%d\n", a, b, c, d)
61
62      // display four random doubles, each [0-1)
63      fmt.Printf("0-1:\t%f\t%f\n", rand.Float64(), rand.Float64())
64      fmt.Printf("0-1:\t%f\t%f\n", rand.Float64(), rand.Float64())
65
66      r = 10 * rand.Float64()
67      fmt.Printf("0-9.99:\t%f\n", r)
68      fmt.Printf("0-9:\t%d\n", (int)(r))
69      fmt.Printf("1-10:\t%d", (1 + (int)(r)))
70  }
```

What You Should See

```
SCISSORS
1-10: 5 9 3 6
1-6: 4 6 6 4
1-100: 49 49 49 49
1-100: 28 88 37 3
0-99: 25 33 2 80
10-20: 13 14 17 19
0-1: 0.524564531320864 0.16490799299718129
0-1: 0.8993174099533012 0.5885409176629621
0-9.99: 1.4591530595519309
0-9: 1
1-10: 2
```

Go has a built-in function but .. we should setting it up by ourselves. As you can see there's an two of new import's, "math/rand" and "time", so we will can use the

function. So on line 12 we set a Seed for "rand" first, by inserting our time to get our values.
Every time you call the function, it will produce a new random double in the range [0.0, 1.0) (that is, it might be exactly 0.0 but will never be exactly 1.0 and will most likely be something in between. So if I write:

```
x = rand.Float64()
```

…then *x* could have a value of 0.0, or 0.123544, or 0.3, or 0.999999999 but never 1.0 and never greater than 1. So on line 14 the function rand.Float64() is called, and the result is stored into a variable named *rps*

So, about one out of every three runs, *rps*' random value should be smaller than 0.3333333. If it is, we print "ROCK". If it's smaller than 0.6666667 (but not smaller than 0.3333333 because of the else) we print out "PAPER". And the rest of the time it 'll print "SCISSORS".

If you could easily run this program 1000 times and tally up the results, you would find that each word should show up roughly 1/3 of the time.

So that's the idea. However, sometimes I don't want a random double between zero and one; I want a random *integer* between 1 and 10. Or between 1 and 100.

In most versions of Go-lang we can't control the range of the value that Math.Float64() gives us, so we will have to do some funky math to transform that random value between 0.0 and 1.0 so that it ends up being in a different range.

On lines 24 through 27 we pick four new random numbers. We multiply each one of them by 10, then we cast them all to integers, then finally add 1 to each.
This sequence of math operations turns each number into something that's always from 1 to 10.

The next chunk of code picks four more random numbers and does the same steps but ends up with random integers from 1 to 6. (That's inclusive, so both 1 and 6 are possible.) Here are the steps:

```
rand.Float64() // generates a random double from 0.0 to 0.99999999999
6* rand.Float64 () // scales it to be from 0.0 to 5.999999999999
(int)(6*rand.Float64()) // casts it to an integer, which throws aways the part af
ter the decimal point (truncates); now from 0 to 5
1 + (int)(6*rand.Float64()) //adds one, translating it to be from 1 to 6
```

Line 38 makes the computer choose a random integer from 1 to 100 and puts it into the variable ***a***. Notice that ***a*** only contains a copy of the ***value*** of the number, so printing it out more than once just displays the same random number over and over. Just printing the value of ***a*** doesn't magically make it pick again somehow.

The next chunk of code picks four random numbers from 1 to 100 and puts their values into four variables, then displays them.

The chunk of code starting on line 49 picks four more random numbers, but since we didn't add 1 to all of them, their range is 0 to 99.

Starting on line 56, there's a chunk of code that picks four random numbers from 10 to 20. Surprised?

```
rand.Float64() // generates a random double from 0.0 to 0.99999999999
11*rand.Float64() // makes it range from 0.0 to 10.999999999999
(int)(11*rand.Float64()) // casts it to an integer from 0 to 10
10 + (int)(11*rand.Float64()) // adds 10 so it's now from 10 to 20
```

In general, the number you multiply it by determines ***how many*** random integers are in the range. Multiplying by 10 gives you ten possible numbers. Mutiplying by 6 gives you six possible numbers.

Then the number you add (after truncating/casting) is called the "origin" and is the ***smallest*** random you'll end up with after all the math is done.

There are eleven numbers from 10 to 20 (if you count both 10 and 20), so that's why we multiply by `11`. And we add ten because we want them to start at `10`.
So if I wanted a random number for a card game from 2-13:

- There are 12 numbers from 2 to 13 (inclusive), and
- The smallest number we want is 2.

So, `2 + (int)( 12*rand.Float64() )` will give us a random number from 2 to 13.
In fact,

```
q = lo + (int)((hi-lo+1) * rand.Float64() ); // picks int from lo to hi
```

The number of values from `lo` to `hi` is `(hi-lo+1)`; the +1 is to account for the fact that subtracting gives you the distance between two numbers, not the count of stopping points along the way

Exercise 25: Repeating Yourself with the “While” Loop

This is one of my favorite exercises, because you are going to learn how to make chunks of code *repeat*. And if you can do that, you will be able to write all *sorts* of interesting things.

We will get back to random numbers after a we spend a few exercises learning the basics of loops.

```go
package main

import "fmt"

func main() {
    var pin, entry int

    pin = 12345

    fmt.Println("WELCOME TO THE BANK OF JAVA.")
    fmt.Print("ENTER YOUR PIN: ")
    fmt.Scanf("%d", &entry)

    for entry != pin {
        fmt.Println("\nINCORRECT PIN. TRY AGAIN.")
        fmt.Print("ENTER YOUR PIN: ")
        fmt.Scanf("\n%d", &entry)
    }

    fmt.Println("\nPIN ACCEPTED. YOUR ACCOUNT BALANCE IS $425.17")
}
```

What You Should See

```
WELCOME TO THE BANK OF JAVA.
ENTER YOUR PIN: 123
INCORRECT PIN. TRY AGAIN.
ENTER YOUR PIN: 1234
INCORRECT PIN. TRY AGAIN.
ENTER YOUR PIN: 12345
PIN ACCEPTED. YOUR ACCOUNT BALANCE IS $425.17
```

On line 14 you get your first look at the `for` loop. A `for` loop is similar to an `if` statement. They both have a condition in parentheses that is checked to see if it is true or false. If the condition is false, both `for` loops and `if` statements will skip all the code in the body. And when the condition is true, both `for` loops and `if` statements will execute all of the code inside their body one time. The only difference is that `if` statements that are true will execute all of the code in the curly braces exactly once. `For` loops that are true will execute all of the code in the curly braces once and then ***go back up and check the condition again***. If the condition is ***still*** true, the code in the body will all be executed again. Then it checks the condition ***again*** and runs the body again if the condition is still true.

Exercise 26: A Number-Guessing Game

Now that you know how to repeat something using a `for` loop we are going to write a program that another human might actually *enjoy* running! Are you as excited as I am about this?!?

We're also going to use a random number, which you learned how to do a couple of exercises back.

```go
package main

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

func main() {
    var secret, guess int
    rand.Seed(time.Now().UTC().UnixNano())

    secret = 1 + (int)(100*rand.Float64())

    fmt.Println("I'm thinking of a number between 1-100.")
    fmt.Println("Try to guess it!")
    fmt.Print("> ")
    fmt.Scanf("%d", &guess)

    for secret != guess {
        if guess < secret {
            fmt.Print("Sorry, your guess is too low.")
            fmt.Print(" Try again.\n> ")
            fmt.Scanf("\n%d", &guess)
        }
        if guess > secret {
            fmt.Print("Sorry, your guess is too high.")
            fmt.Print(" Try again.\n> ")
            fmt.Scanf("\n%d", &guess)
        }
    }

    fmt.Println("You guessed it! What are the odds?!?")
}
```

What You Should See

```
I'm thinking of a number between 1-100.
Try to guess it!
> 50
Sorry, your guess is too low. Try again.
> 75
```

```
Sorry, your guess is too low. Try again.
> 87
Sorry, your guess is too high. Try again.
> 81
Sorry, your guess is too low. Try again.
> 84
Sorry, your guess is too low. Try again.
> 86
Sorry, your guess is too high. Try again.
> 85
You guessed it! What are the odds?!?
```

So on line 13 the computer chooses a random integer from 1 to 100 and stores it into the variable ***secret***. We let the human make a guess.

Line 20 begins a `for` loop. It says "As long as the value of the variable ***secret*** is not the same as the value of the variable ***guess***… run the following chunk of code." Lines 20 through 31 are the body of the loop. Every time the condition is true, all twelve of these lines of code get executed.

Inside the body of the loop we have a couple of `if` statements. We already know that the human's guess is different from the secret number or we wouldn't be inside the `while` loop to begin with!

But we don't know if the guess is wrong because it is too low or because it is too high, so these `if` statements figure that out and display the appropriate error message.

Then after the error message is displayed, we allow them to guess again. The human (hopefully) types in a number which is then stored into the variable ***guess***, overwriting their previous guess in that variable.

Since the body of the loop has executed once all the way through, the program pops back up to line 20 and checks the condition again. If that condition is ***still*** true (their guess is still not equal to the secret number) then the whole loop body will execute

a second time. If the condition is now false (because they guessed it) then the whole loop body will be skipped and the program will skip down
to line 33.

If the loop is over, we know the condition is false. So we don't need a new `if` statement down here; it is safe to print "you guessed it."

Exercise 27: Infinite Loops

One thing which sometimes surprises students is how easy it is to make a loop that repeats *forever*. These are called “infinite loops” and we sometimes make them on purpose but usually they are the result of a logical error. Here’s an example:

```go
package main

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

func main() {
    var secret, guess int
    rand.Seed(time.Now().UTC().UnixNano())

    secret = 1 + (int)(rand.Float64()*10)

    fmt.Println("I have chosen a number between 1 and 10.")
    fmt.Println("Try to guess it.")
    fmt.Print("Your guess: ")
    fmt.Scanf("%d", &guess)

    for secret != guess {
        fmt.Println("That is incorrect. Guess again.")
        fmt.Print("Your guess: ")
    }

    fmt.Println("That's right! You're a good guesser.")
}
```

What You Should See

```
I have chosen a number between 1 and 10.
Try to guess it.
Your guess: 4
That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
Your guess: That is incorrect. Guess again.
```

```
Your guess: ^C
```

The program didn't actually stop on it's own; I had to stop it by pressing CTRL-C while the program was repeating and repeating.

This code has an infinite loop in it. Line 20 checks to see if the value of the variable ***secret*** is different than the value of the variable ***guess***. If so, it executes the body of the loop and if not it skips over the loop body down to line 24.

The problem is that once ***secret*** and ***guess*** are different the program can never reach another line of code that changes either variable, so the loop repeats lines 20 through 23 forever.

So when you are writing the condition of a while loop, try to keep in the back of your mind: "I need to make sure there's a way for this condition to eventually become false."

Exercise 28: Using Loops for Error-Checking

So far in this book we have mostly been ignoring error-checking. We have assumed that the human will follow directions, and if their lack of direction-following breaks our program, we just blame the user and don't worry about it.

This is totally fine when you are just learning. Error-checking is hard, which is why most big programs have bugs and it takes a whole army of people working really hard to make sure that software has as few bugs as possible.

But you are finally to the point where you *can* code a little bit of error-checking.

```
1   package main
2
3   import (
4       "fmt"
5       "math"
6   )
7
8   func main() {
9       var x, y float64
10
11      fmt.Print("Give me a number, and I'll find its square root. ")
12      fmt.Print("(No negatives, please.) ")
13      fmt.Scan(&x)
14
15      for x < 0 {
16          fmt.Println("I won't take the square root of a negative.")
17          fmt.Print("\nNew number: ")
18          fmt.Scan(&x)
19      }
20
21      y = math.Sqrt(x)
22
23      fmt.Printf("The square root of %f is %f\n", x, y)
24  }
```

What You Should See

```
Give me a number, and I'll find it's square root. (No negatives, please.) -8
I won't take the square root of a negative.
```

```
New number: -7
I won't take the square root of a negative.
New number: -200
I won't take the square root of a negative.
New number: 5
The square root of 5.0 is 2.23606797749979
```

Starting on line 14 is an example of what I call an “input protection loop.” On line 21 we are going to take the square root of whatever value is in the variable `x` and we would like to make sure it contains a positive number before we do that.

We could just use the built-in absolute value function `Math.abs()`, but I’m trying to demonstrate error-checking, okay?

On line 13 we let the human type in a number. We have asked them nicely to only type in a positive number, but they can type whatever they like. (They could even type “Mister Mxyzptlk”, but our error-checking skillz aren’t advanced enough to survive that, yet.)

So on line 15 we check to see if they followed directions. If the value in `x` is negative (less than zero) we print out an error message and let them try again. THEN, after they have typed their new number we ***go back up*** to line 15 and check if the condition is still true. Are they ***still*** failing to follow directions? If so, display the error message ***again*** and give them another chance.

Computers don’t get impatient or bored, so the human is **trapped** in this loop until they comply. They could type negative numbers two billion times and each time the computer would politely complain and make them type something again.

Eventually, the human will wise up and type a non-negative number. Then the condition of the `for` loop will be false (finally) and the body of the loop will be skipped (finally) and execution will pick up on line 21 where we can safely take the square root of a number that we ***know*** is positive.

Real programs have stuff like this ***all over***. You have to do it because humans are unreliable and often do unexpected things. What happens when your toddler pulls himself up to your laptop and starts mashing keys while a program is running? We would like the program to not crash.

Exercise 29: Loops That Do Then Check

In this exercise I am going to do something I normally don't do. I am going to show you ***another*** way to make loops in Golang. Since you have only been looking at for loop for four exercises, showing you a different type of loop can be confusing, we called it *do-while*. Usually I like to wait until students have been doing something a ***long*** time before I show them a new way to do the same thing

```
package main

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

func main() {
    var coin, again string
    var streak int = 0
    var gotHeads bool
    rand.Seed(time.Now().UTC().UnixNano())

    for ok := true; ok; ok = (again == "y") {
        gotHeads = rand.Float64() < 0.5

        if gotHeads {
            coin = "HEADS"
        } else {
            coin = "TAILS"
        }

        fmt.Println("You flip a coin and it is... " + coin)

        if gotHeads {
            streak++
            fmt.Println("\tThat's " + strconv.Itoa(streak) + " in a row....")
            fmt.Print("\tWould you like to flip again (y/n)? ")
            fmt.Scan(&again)
        } else {
            fmt.Println("\tYou lose everything!")
            fmt.Println("\tShould've quit while you were aHEAD....")
            streak = 0
            again = "n"
        }
    }

    fmt.Println("Final score: " + strconv.Itoa(streak))
}
```

What You Should See

```
You flip a coin and it is... HEADS
That's 1 in a row....
Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
That's 2 in a row....
Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
That's 3 in a row....
Would you like to flip again (y/n)? n
Final score: 3
```

There are only two differences between `while` loops and do-while(this exercise) loops.
1. The condition of a `while` loop is ***before*** the body, but do-while loops just have the keyword `for` before the body and the condition is at the end, just after the close curly brace.
2. `while` loops check their condition ***before*** going into the loop body, but do-while loops run the body of the loop once ***no matter what*** and only check the condition ***after*** the first time through.

In computer-science circles, the `while` loop is called a “pre-test” loop (because it checks the condition first) and the do-while is called a “post-test” loop (because it checks the condition afterward).

As you can see on Line 16, as we can see there’s another format of using “for” statement, before we learn something like “i<= 10”, but there’s another, after the “for”, there’s ok:=true, ok, ok = (again == “y”). Alright calm-down I will breakdown it one by one :
ok:=true means when “ok” is true, then if it’s match with condition, it would still loop until the condition aren’t true anymore

Then, on next semicolon. We declare the “ok”. After that there’s third semicolon, this is where we insert our condition, if the condition aren’t match.. Then it will skip the loop.

Example : for ok:=true; ok; ok = (colour == “red”)
so the program will always loop from the “for” curly brace starts until curly brace ends.

Wait, there’s a new import that we just knew! As you can see on line 6, there’s “strconv” it means string convert, then where the example if we want to convert an Integer to String? Check on line 29, as you can see we use fmt.Println() as usual, but.. Hey look! “String” + strconv.Itoa(streak) + “String”, so uh what is that Itoa? So.. Itoa is just like a method that can convert an Integer into a string, but why should I converted it first? Because of that we can show the variable without that “%s” / “%f”-thing.

But wait, let’s get back into Loop Exercise, so as you can see on line 17, we set a Boolean expression that, if our random Float is a 0.1 – 0.4 that is means the gotHeads Boolean is true, then if it’s not? It would be false!

Then, the program will check on lines 27 – 38, if our result “HEADS” that’s mean we win this round, we will get one score! Then we will get a question, want to end it or continue? If we choose continue, we will comeback again into line 17, we got a new Random Float, and then the program will check it again, and then if you won, the program will ask do you want to continue, if you’re lose on that round, Bye-bye! The program will ends.

Exercise 30: Adding Values One at a Time

This exercise will demonstrate something that you have to do a *lot*: dealing with values that you only get one at a time.

If I asked you to let the human type in three numbers and add them up, and if I promised they would only need to type in exactly three numbers (never more, never fewer), you would probably write something like this:

```
var a,b,c,total int
fmt.Scanf(&a)
fmt.Scanf(&b)
fmt.Scanf(&c)
total = a + b + c
```

But what if I told you they wanted to type in one hundred numbers? Or ten thousand? Or *maybe* three and *maybe* five, I'm not sure? Then you need a different approach. You'll need a loop (that's how we repeat things, after all). And you need a variable that will add the values one at a time as they come. A variable that starts with "nothing" in it and adds values one at a time is called an "accumulator" variable, although that's a pretty old word and so your friends who code may never have heard it if they're under the age of forty.
Anyway, the basic idea looks like this:

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var current int
    var total int = 0

    fmt.Print("Type in a bunch of values and I'll add them up. ")
    fmt.Println("I'll stop when you type a zero.")

    for current = 1; current != 0; {
        fmt.Print("Value: ")
        fmt.Scan(&current)
        total += current
        fmt.Println("The total so far is: " + strconv.Itoa(total))
    }

    fmt.Println("The final total is: " + strconv.Itoa(total))
}
```

What You Should See

```
Type in a bunch of values and I'll add them up. I'll stop when you type a zero
Value: 3
The total so far is: 3
Value: 4
The total so far is: 7
Value: 5
The total so far is: 12
Value: 6
The total so far is: 18
Value: 0
The total so far is: 18
The final total is: 18
```

We need two variables: one to hold the value they just typed in (*current*) and one to hold the running total (um… *total*). On line 10, we make sure to start by putting a zero into *total*. You'll see why soon.

At first, on line 15 we initiation variable the "current" first, After that we can set a condition / Boolean expression (Remember that Compiler know which initiation and Boolean expression). So if the value "current" still not 0 "Zero" that's mean the program will always loop.

On line 17 the human gets to type in a number. This is inside the body of a do-while loop, which runs at least once no matter what, so this code always happens. Let's pretend they type `3` at first.

On line 18 the magic happens. We add the value the human typed to ***whatever value is already in the variable*** total. There's a zero in ***total*** at first, so this line of code adds zero to ***current*** and stores that new number back into ***total***. Thus ***total*** no longer has a zero in it; it has the same value ***current*** did. So ***total*** was `0`, now it is `3`.

Then we print the subtotal and on line 19 check to see if ***current*** was zero. If not, the loop repeats back up line 16.

The human gets to type in a second number. Let's say it is a `4`. The variable ***total*** gets changed to ***current*** (`4`) plus ***total***'s existing value (`3`), so ***total*** is now `7`.

The condition is checked again, and the process continues. Eventually the human types a `0`, that `0` gets added to the total (which doesn't hurt it) and the condition is false so the do-while loop stops looping.

Exercise 31: Adding Values for a Dice Game

Pig is a simple dice game for two or more players. The basic idea is to be the first one to “bank” a score of 100 points.

When you roll a 1, your turn ends and you gain no points that turn. Any other roll adds to your score for that turn, but ***you only keep those points*** if you decide to “hold”. If you roll a 1 before you hold, all your points for that turn are lost.

You know enough to handle the code for the entire game of Pig, but it is a ***lot*** at once compared to the smaller programs you have been seeing, so I am going to break it into two lessons. Today we will write only the artificial intelligence (A.I.) code for a computer player. This computer player will utilize the “hold at 20” strategy, which means the computer keeps rolling until their score for the turn adds up to 20 or more, and then holds no matter what. This is actually not a terrible strategy, and it is easy enough to code.

```go
package main

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

func main() {
    var roll, total int
    total = 0
    rand.Seed(time.Now().UTC().UnixNano())

    for roll = 0; roll != 1 && total < 20; {
        roll = 1 + (int)(rand.Float64()*6)
        fmt.Println("Computer rolled a " + strconv.Itoa(roll) + ".")
        if roll == 1 {
            fmt.Println("\tThat ends its turn.")
            total = 0
        } else {
            total += roll
            fmt.Print("\tComputer has " + strconv.Itoa(total))
            fmt.Print(" points so far this round.\n")
            if total < 20 {
                fmt.Println("\tComputer will roll again.")
            }
        }
    }

    fmt.Println("Computer ends the round with " + strconv.Itoa(total) + " points.")
}
```

What You Should See

```
Computer rolled a 6.
	Computer has 6 points so far this round.
	Computer will roll again.
Computer rolled a 6.
	Computer has 12 points so far this round.
	Computer will roll again.
Computer rolled a 1.
	That ends its turn.
Computer ends the round with 0 points.
```

Basically the whole program is in the body of one big do-while loop that tells the computer when to stop: either it rolls a 1 or it gets a total of 20 or more. As long as the roll is not one ***and*** the total is less than 20, the condition will be true and the loop will start over from the beginning (on line 15).

And we choose a do-while loop because we want the computer to roll at least once no matter what.

The roll is made on line 16: a random number from 1-6 is a good substitute for rolling a dice.

On line 18 we check for rolling a 1. If so, all points are lost. If not (`else`), we add this roll to the running total. Notice we used “plus equals”, which we have seen before.

The `if` statement on line 25 is just so we can get a nice message that the computer is going to roll again.

Not terrible, right? So come back next lesson for the full game!

Exercise 32: The Dice Game Called ‘Pig’

In the previous lesson we wrote the computer A.I. for the dice game ***Pig***. In this lesson we will have the code for the entire game, with one human player and one computer player that take turns.

The entire program you wrote last time corresponds roughly to lines 37 through 59 in this program.

The only major difference is that instead of a single ***total*** variable we will have a ***turnTotal*** variable to hold only the points for one turn and a ***ctot*** variable that holds the computer’s overall points from round to round.

```
1   package main
2
3   import (
4       "fmt"
5       "math/rand"
6       "strconv"
7       "time"
8   )
9
10  func main() {
11      var roll, ptot, ctot, turnTotal int
12      var choice string = "roll"
13
14      ptot = 0
15      ctot = 0
16      rand.Seed(time.Now().UTC().UnixNano())
17
18      for ptot < 100 && ctot < 100 {
19          turnTotal = 0
20          fmt.Println("You have " + strconv.Itoa(ptot) + " points.")
21
22          for roll = 0; roll != 1 && choice == "roll"; {
23              roll = 1 + (int)(rand.Float64()*6)
24              fmt.Println("\tYou rolled a " + strconv.Itoa(roll) + ".")
25              if roll == 1 {
26                  fmt.Println("\tThat ends your turn.")
27                  turnTotal = 0
28              } else {
29                  turnTotal += roll
30                  fmt.Print("\tYou have " + strconv.Itoa(turnTotal) + " points")
31                  fmt.Print(" so far this round.\n")
32                  fmt.Print("\tWould you like to \"roll\" again")
33                  fmt.Print(" or \"hold\"? ")
34                  fmt.Scan(&choice)
35              }
36          }
```

```
37
38          ptot += turnTotal
39          fmt.Println("\tYou end the round with " + strconv.Itoa(ptot) + " points.")
40
41          if ptot < 100 {
42              turnTotal = 0
43              fmt.Println("Computer has " + strconv.Itoa(ctot) + " points.")
44
45              for roll = 0; roll != 1 && turnTotal < 20; {
46                  roll = 1 + (int)(rand.Float64()*6)
47                  fmt.Println("\tComputer rolled a " + strconv.Itoa(roll) + ".")
48                  if roll == 1 {
49                      fmt.Println("\tThat ends its turn.")
50                      turnTotal = 0
51                  } else {
52                      turnTotal += roll
53                      fmt.Print("\tComputer has " + strconv.Itoa(turnTotal))
54                      fmt.Print(" points so far this round.\n")
55                      if turnTotal < 20 {
56                          fmt.Println("\tComputer will roll again.")
57                      }
58                  }
59              }
60
61              ctot += turnTotal
62              fmt.Print("\tComputer ends the round with ")
63              fmt.Print(strconv.Itoa(ctot) + " points.\n")
64              choice = "roll"
65          }
66      }
67
68      if ptot > ctot {
69          fmt.Println("Humanity wins!")
70      } else {
71          fmt.Println("The computer wins.")
72      }
73  }
```

What You Should See

```
You have 0 points.
    You rolled a 6.
    You have 6 points so far this round.
    Would you like to "roll" again or "hold"? roll
    You rolled a 1.
    That ends your turn.
    You end the round with 0 points.
Computer has 0 points.
    Computer rolled a 2.
    Computer has 2 points so far this round.
    Computer will roll again.
    Computer rolled a 5.
    Computer has 7 points so far this round.
```

```
    Computer will roll again.
    Computer rolled a 6.
    Computer has 13 points so far this round.
    Computer will roll again.
    Computer rolled a 6.
    Computer has 19 points so far this round.
    Computer will roll again.
    Computer rolled a 6.
    Computer has 25 points so far this round.
    Computer ends the round with 25 points.
...skip a bit, brother

You have 70 points.
    You rolled a 1.
    That ends your turn.
    You end the round with 70 points.
Computer has 85 points.
    Computer rolled a 2.
    Computer has 2 points so far this round.
    Computer will roll again.
    Computer rolled a 5.
    Computer has 7 points so far this round.
    Computer will roll again.
    Computer rolled a 6.
    Computer has 13 points so far this round.
    Computer will roll again.
    Computer rolled a 4.
    Computer has 17 points so far this round.
    Computer will roll again.
    Computer rolled a 4.
    Computer has 21 points so far this round.
    Computer ends the round with 106 points.
The computer wins.
```

We begin the program with two variables: ***ptot*** holds the human's total and ***ctot*** holds the computer's total. Both start at 0.

Then on line 18 begins a really huge do-while(We called it as do-while so you can recognize it if you learn another Programming language) loop that basically contains the whole game and doesn't end until line 66. Scroll down and you can see that this loop repeats as long as both ***ptot*** and ***ctot*** are less than 100. When either player

reaches 100 or more, the condition is no longer true and the do-while won't repeat back up again.

Then after that do-while loop ends (starting on line 68) there is an `if` statement and an `else` to determine the winner.

Let us scroll back up and look at the human's turn, which begins on line 18. The ***turnTotal*** is the number of points the human has earned this round so far. And since it's the beginning of the round, we should start it out at 0.

Line 22 is the beginning of a do-while loop that contains the human's turn. It ends on line 36, and all the code between lines 22 and 36 will repeat as long as the human does not roll a 1 and as long as the human keeps choosing to roll again.

Each roll for the human begins just like the computer did: by choosing a random number from 1 to 6. We print this out on line 24.

Now two things could happen: either the roll is 1 – and the human loses all points earned this round – or the roll is 2-6 and the roll is added to their ***turnTotal***. We display the appropriate messages, and on lines 32-34 we give the human the choice to chance it by rolling again or play it safe by holding. Then on line 36 the condition of the do-while loop will check and repeat back up to line 22
if appropriate.

Once the player's turn ends, we add the ***turnTotal*** (which might be 0) to the player's overall total and display their current number of points.

On line 42 the computer's turn begins. However, the computer doesn't get a turn if the human has already reached 100 points: the game is over in that case. So to prevent the computer from playing we must wrap the whole computer's turn in a big `if` statement so that it is skipped if the human's total (***ptot***) is greater than or equal to 100. This `if` statement begins here on line 41 and ends on line 58.

So on line 42 the computer's turn begins for real. This is basically the same as the previous exercise, so I won't bother to explain it again. Notice that the computer is deciding whether or not to roll again based on its turn total.

Line 66 ends the do-while loop containing the whole game, and lines 68 through 69 determine and display the winner.
Hopefully you were able to follow the flow of the game well enough. It's pretty complicated

Exercise 33: Calling a Function

The previous exercise was pretty complicated. So we will relax a bit with today's exercise. We are going to learn how to write a "function" in Go and how to make it execute by "calling" it.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       fmt.Println("Here.")
7       erebor()
8       fmt.Println("Back first time.")
9       erebor()
10      fmt.Println("Back second time.")
11  }
12
13  func erebor() {
14      fmt.Println("There.")
15  }
```

What You Should See

```
Here.
There.
Back first time.
There.
Back second time.
```

So lines 6 through 10 are pretty boring, except that on lines 7 and 9 we are referring to some thing called "erebor" that you haven't seen before in Go. Do you know why you haven't seen it? Because it doesn't exist!

Skipping down to lines 13 through 15 you will notice that I added something to our program that is ***not*** inside the body of `main()`. Normally the close curly brace of `main` is almost at the end of the code, But not this time!

Lines 13 through 15 define a ***function*** named erebor(). (The word erebor() doesn't mean anything in particular to Go. We could have named it bilbo() or smaug() or anything we like.)

This function has an open curly brace on line 13 just like main() always has an open curly brace.

And on line 15 is the end of the function's body, and there's a close curly brace. So the function definition starts on line 13 and ends on line 15.

What does the function do? It prints the String "There." on the screen. So now let's go back up to main() and look at the function calls inside the body of main().

On line 6 we print the String "Here." on the screen. Then on line 7 you will see a " function call."This line of code tells the computer to jump down to the function erebor(), run through all the code in the body of that function, and to return to line 7 once that has been accomplished.

So you see that when we call the erebor() function, the String "There." gets printed on the screen right after the String "Here.". When the computer runs line 7, execution of the program pauses in main(), skips over all the rest of the code in main(), jumps down to line 13, runs all the code in the body of the function erebor() (all 1 line of it) and then once execution hits the close curly brace on line 15, it returns back up to the end of line 7 and unpauses the execution in main(). It runs line 8 next.

On line 8 it displays another message on the screen, then on line 9 there is another function call. The function ***erebor*** is called a second time. It pauses main() on line 9, jumps down and runs through the body of ***erebor*** (which prints the String "There." again), then returns back up to line 9 where execution of main() resumes.

Finally line 10 prints one last String on the screen. Execution then proceeds to the close curly brace of main() which is on line 11. When main() ends, the program ends.

That's pretty important, so I will say it again: when `main()` ends, the program ends. Even if you have a bunch of different functions inside the `class`, program execution begins with the first line of `main()`. And once the last line of `main()` has been executed, the program stops running even if there are functions that never got called. (We will see an example of this in the next exercise.)

Exercise 34: Calling Functions to Draw a Flag

Now that you understand the absolute basics about how to define a function and how to call that function, let us get some practice by defining ***eleven*** functions! There are no zeros (0) in this program. Everything that looks like an O is a capital letter O. Also notice that lines 36 and 40 feature `Print()` instead of `Println()`.

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       printTopHalf()
7
8       print48Colons()
9       print48Ohs()
10      print48Colons()
11      print48Ohs()
12      print48Colons()
13      print48Ohs()
14  }
15
16  func print48Colons() {
17      fmt.Println("|::::::::::::::::::::::::::::::::::::::::::::::::|")
18  }
19
20  func print48Ohs() {
21      fmt.Println("|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|")
22  }
23
24  func print29Colons() {
25      fmt.Println("|:::::::::::::::::::::::::::::|")
26  }
27
28  func printPledge() {
29      fmt.Println("I pledge allegiance to the flag.")
30  }
31
32  func print29Ohs() {
33      fmt.Println("|OOOOOOOOOOOOOOOOOOOOOOOOOOOO|")
34  }
```

```
35
36  func print6Stars() {
37      fmt.Print("| *  *  *  *  *  * ")
38  }
39
40  func print5Stars() {
41      fmt.Print("|   *  *  *  *  *  ")
42  }
43
44  func printSixStarLine() {
45      print6Stars()
46      print29Ohs()
47  }
48
49  func printFiveStarLine() {
50      print5Stars()
51      print29Colons()
52  }
53
54  func printTopHalf() {
55      fmt.Println(" _____________________________________________")
56      // the line above has 1 space then 48 underscores between the quotes
57      printSixStarLine()
58      printFiveStarLine()
59      printSixStarLine()
60      printFiveStarLine()
61      printSixStarLine()
62      printFiveStarLine()
63      printSixStarLine()
64  }
```

What You Should See

```
 _____________________________________________
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *  *  *  *  *  |::::::::::::::::::::::::::::|
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *  *  *  *  *  |::::::::::::::::::::::::::::|
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|   *  *  *  *  *  |::::::::::::::::::::::::::::|
| *  *  *  *  *  * |OOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|:::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|:::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
|:::::::::::::::::::::::::::::::::::::::::::::::|
|OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO|
```

This exercise is ridiculous. There is no good reason that any self-respecting programmer would ever write the code to draw a flag on the screen like this. It is okay if writing this program felt a little silly. But functions are important and I prefer

to start with silly examples that you can actually understand instead of realistic examples that are too hard to follow.
So how do we even trace through the execution of a program like this? We start at the beginning: the first line of main(). On line 6 the first thing *main* does is call the function printTopHalf(). So *main* gets put on pause and execution jumps down to the first line of the printTopHalf() function, which is on line 54.

The first thing that *printTopHalf* does is print a bunch of underscores on the screen, which we will be the top of our flag. After that execution moves on to line 57, which is *another* function call! So main() is still on pause from before, waiting for printTopHalf() to finish, and now printTopHalf() itself is on pause, waiting for printSixStarLine() to finish and return control to here. printSixStarLine() begins on line 45, where it calls the print6Stars() function. That function (thankfully) only displays something on the screen so when the close curly brace of print6Stars() comes on line 37, it returns control back to the second line (line 45) of printSixStarLine(), which then has another function call. This runs through the body of the function print29Ohs() and comes back to line 46. So then printSixStarLine() ends, which returns control to the end of line 57.

Exercise 35: Displaying Dice with Functions

The last exercise used functions in a program where functions actually made things *worse*. So today we are ready to look at a situation where using a function actually makes the program better.

Yacht is an old dice game that was modified for the commercial game Yahtzee. It involves rolling five dice at once and earning points for various combinations. The rarest combination is “The Yacht”, when all five dice show the same number.

```
1   package main
2
3   import (
4       "fmt"
5       "math/rand"
6       "time"
7   )
8
9   func main() {
10      var r1, r2, r3, r4, r5 int
11      var allSame bool
12
13      rand.Seed(time.Now().UTC().UnixNano())
14
15      for ok := true; ok; ok = !allSame {
16          r1 = 1 + (int)(rand.Float64()*6)
17          r2 = 1 + (int)(rand.Float64()*6)
18          r3 = 1 + (int)(rand.Float64()*6)
19          r4 = 1 + (int)(rand.Float64()*6)
20          r5 = 1 + (int)(rand.Float64()*6)
21          fmt.Print("\nYou rolled : ", r1, " ", r2, " ")
22          fmt.Println(r3, r4, r5)
23          showDice(r1)
24          showDice(r2)
25          showDice(r3)
26          showDice(r4)
27          showDice(r5)
28          allSame = (r1 == r2 && r2 == r3 && r3 == r4 && r4 == r5)
29      }
30      fmt.Println("The Yacht!!")
31  }
```

```
32
33  func showDice(roll int) {
34      fmt.Println("+---+")
35      if roll == 1 {
36          fmt.Println("|   |")
37          fmt.Println("| o |")
38          fmt.Println("|   |")
39      } else if roll == 2 {
40          fmt.Println("|o  |")
41          fmt.Println("|   |")
42          fmt.Println("|  o|")
43      } else if roll == 3 {
44          fmt.Println("|o  |")
45          fmt.Println("| o |")
46          fmt.Println("|  o|")
47      } else if roll == 4 {
48          fmt.Println("|o o|")
49          fmt.Println("|   |")
50          fmt.Println("|o o|")
51      } else if roll == 5 {
52          fmt.Println("|o o|")
53          fmt.Println("| o |")
54          fmt.Println("|o o|")
55      } else if roll == 6 {
56          fmt.Println("|o o|")
57          fmt.Println("|o o|")
58          fmt.Println("|o o|")
59      }
60      fmt.Println("+---+")
61  }
```

What You Should See

```
You rolled : 2 6 6 3 5
+---+
|o  |
|   |
|  o|
+---+
+---+
|o o|
|o o|
|o o|
+---+
+---+
|o o|
|o o|
|o o|
+---+
+---+
|o  |
| o |
|  o|
+---+
+---+
|o o|
| o |
|o o|
```

```
+---+

You rolled : 5 5 1 6 6
+---+
|o o|
| o |
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+
+---+
|   |
| o |
|   |
+---+
+---+
|o o|
|o o|
|o o|
+---+
+---+
|o o|
|o o|
|o o|
+---+

You rolled : 1 1 1 1 1
+---+
|   |
| o |
|   |
+---+
+---+
|   |
| o |
|   |
+---+
+---+
|   |
| o |
|   |
+---+
+---+
 |   |
| o |
|   |
+---+
+---+
|   |
| o |
|   |
+---+
```

```
The Yacht!!
```

Other than the fancy Boolean expression on line 28, the interesting thing in this exercise is a single function called showDice.

On lines 16 through 20 we choose five random numbers (each from 1 to 6) and store the results into the five integer variables *r1* through *r5*.

We want to use some if statements to display a picture of the die's value on the screen, but we don't want to have to write the same if statements five times (which we would have to do because the variables are different). The solution is to create a function that takes a parameter.

On line 33 you see the beginning of the definition of the showDice function. After the name (or "identifier") showDice there is a set of parentheses and between them a variable is declared! This variable is called a "parameter". The showDice function has one parameter. That parameter is an integer. It is named *roll*.

This means that whenever you write a function call for showDice you can *not* just write the name of the function with parentheses like showDice(). It won't compile. You must include an integer value in the parentheses (this is called an "argument"), either a variable or an expression that simplifies to an integer value.

Here are some examples.

```
showDice; // NO (without parens this refers to a variable
// rather than a function call)
showDice(); // NO (function call must have one argument, not zero)
showDice(1); // YES (one argument is just right)
showDice(4); // YES
showDice(1+2); // YES
showDice(r2); // YES
showDice(r5); // YES
showDice( (r3+r4) / 2 ); // YES (strange but legal)
showDice(17); // YES (although it won't show a proper dice picture)
showDice(3, 4); // NO (function call must have one argument, not two)
showDice(2.0); // NO (argument must be an integer, not a double)
```

```
showDice("two"); // NO (argument must be an integer, not a String)
showDice(false); // NO (argument must be an integer, not a Boolean)
```

In all cases, a copy of the argument's value is stored into the parameter. So if you call the function like so `showDice(3);` then the function is called and the value `3` is stored into the parameter ***roll***. So by line 34 the parameter variable ***roll*** has already been declared ***and*** initialized with the value `3`.

If we call the function using a variable like `showDice(r2);` then the function is called and a copy of whatever value is currently in ***r2*** will have been stored into the parameter variable ***roll*** before the body of the function is executed.

So on line 23 the `showDice` function is executed, and ***roll*** will have been set equal to whatever value is in ***r1***.

Then on line 24 `showDice` is called again, but this time ***roll*** will be set equal to whatever value is in ***r2***. Line 25 calls `showDice` while setting its parameter equal to the value of ***r3***. And so on.

In this way we basically run the same chunk of code five times, but substituting a different variable for ***roll*** each time. This saves us a lot of code.

Exercise 36: Returning a Value from a Function

Some functions have parameters and some do not. Parameters are the only way to send values *into* a function. There is also only one way to get a value *out* of a function: the return value.

This exercise gives an example of a function that has three parameters (the side lengths of a triangle) and one response: the area of that triangle using Heron's Formula.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    var a, g float64
    var tws string = "A triangle with sides "

    a = triangleArea(3, 3, 3)
    fmt.Println("A triangle with sides 3,3,3 has area ", a)

    a = triangleArea(3, 4, 5)
    fmt.Println("A triangle with sides 3,4,5 has area ", a)
    g = triangleArea(7, 8, 9)
    fmt.Println(tws, "7,8,9 has area ", g)

    fmt.Println(tws, "5,12,13 has area ", triangleArea(5, 12, 13))
    fmt.Println(tws, "10,9,11 has area ", triangleArea(10, 9, 11))
    fmt.Println(tws, "8,15,17 has area ", triangleArea(8, 15, 17))
}

// This function computes the area of a triangle with side lengths a, b, & c.
func triangleArea(a int, b int, c int) (A float64) {
    var s float64

    s = float64((a + b + c) / 2)
    A = math.Sqrt(s * (s - float64(a)) * (s - float64(b)) * (s - float64(c)))

    // After computing the area, you must "return" the computed value:
    return
}
```

What You Should See

```
A triangle with sides 3,3,3 has area  2
A triangle with sides 3,4,5 has area  6
A triangle with sides  7,8,9 has area  26.832815729997478
A triangle with sides  5,12,13 has area  30
A triangle with sides  10,9,11 has area  42.42640687119285
A triangle with sides  8,15,17 has area  60
```

You can see that the function triangleArea has three parameters. They are all integers, and they are named ***a***, ***b*** and ***c***. As you already know, this means that we cannot call the function without providing three integer values as arguments.

In addition to this, the triangleArea function ***returns*** a value. Notice that on line 26 there's two paren. The one is for a variables it says 3 integer so? We receive three values of something that integer, and then the second one is for retuning a variable(In go-lang function can return more than one variables). That means "this function returns a value, and the type of value it returns is a float64." If the function only has one "(. . .)" paren, it means "this function does not return any value." If we wanted triangleArea to return a different type of value:

```
// a function defined this way will return a value that is an int
func triangleArea(a int, b int, c int) (number int)
// this one must return a value that is a String
func triangleArea(a int, b int, c int) (words string)
// this function must return either the value true or the value false
func triangleArea(a int, b int, c int) (boolean bool)
// a function defined this way cannot return any value of any type
func triangleArea(a int, b int, c int)
```

Sometimes my students get confused about functions that return values versus functions that do not return values. An analogy is helpful.
Let us say that we are sitting in my school classroom. We hear the sound of thunder and I remember that I left my car windows down. I don't want rain to make the inside of my car wet, so I send you out into the parking lot.

> “Student, please go out into the parking lot and roll up the windows of my car.”
> “Yes, sir,” you say.
> If you need information from me about what my car looks like, then those are parameters. If you already know which one is mine, you need no parameters.
> Eventually you return and say “I completed the task.” This is like a function that does not return a value.

```
rollUpWindows(); // if you don't need parameters
rollUpWindows("Toyota", "Corolla", 2008, "blue"); // if you need parameters
```

In either case, the function is executed and goes off and does its thing, but returns no value. Now, example #2:

> Again we are in my classroom. I am online trying to update my car insurance and the web page is asking me for my car’s license plate number. I don’t remember it, so I ask you to go to the parking lot and get it for me.
>
> Eventually you return and ***tell me the license plate number***. Maybe you wrote it down on a scrap of paper or maybe you memorized it. When you give it to me, I copy it down myself. This is like a function that returns a value.

```
var plate string;
plate = retrieveLicensePlate(); // if you don't need parameters
plate = retrieveLicensePlate("Toyota", "Corolla", 2008, "blue"); // if you do
```

If I am rude, you could return to my classroom and give me the value and I could put my fingers in my ears so I don’t hear you or refuse to write it down myself so that I quickly forget it. If you call a function that returns a value, you can choose to ***not*** store the return value into a variable and just allow the value to vanish:

```
// returns a value which is lost
retrieveLicensePlate("Toyota", "Corolla", 2008, "blue");
// returns the area but we refuse to store it into a variable
triangleArea(3, 3, 3);
```

This is usually a bad idea, but maybe you have your reasons. Anyway, back to the code.

On line 9 we call the triangleArea function. We pass in 3, 4 and 5 as the three arguments. The 3 gets stored as the value of *a* (down on line 20). The 4 is stored into *b*, and 5 is put into *c*. It runs all the code on lines 23 and 24 with those values for the parameters. By the end, the variable *A* has a value stored in it.
On line 27 we *return* the value that is in the variable *A*. This value travels back up to line 9, where it is stored into the variable *a*. Notice that both main() and triangleArea() have a variable called *a*, but these variables are different from each other. They have different types and different meanings, and they hold different values. This is not a problem.

Also notice that on line 11 we store the return value from the function into a variable named *g* instead of *a*. This is also fine. The triangleArea() function does not know or care about the name of the variable where its return value is stored. It doesn't even know if you store the return value or throw it away! It returned a value, and that's enough.

Before we go into the next exercises. I want to show you how to make a function that can returning multi-values

Remember that the syntax of Function is :

```
func function_name(parameter_list)(return_type_list){
    // code...
}
```

And this is how when you want to returning a multi-values. I don't use "screen-shot" for this time, so you can try and copy it (I don't include the import, so you should type it by yourself).

```
func multiReturn(a int, b int) (add int, minus int) {
    add = a + b
    minus = a - b
```

```
    return
}

func main() {
    var number1, number2 int
    var addedNumber, minusNumber int

    number1 = 10
    number2 = 6

    addedNumber, minusNumber = multiReturn(number1, number2)
    fmt.Printf("%d is Added and Minus is %d", addedNumber, minusNumber)

}
```

What You Should See

```
16 is Added and Minus is 4
```

Also, you should remember that any function must have same sort of value / type-data

```
func anyFunction(a int, b int, greetings string) (add int, anyFloat float64, output string)
```

So? You should use it as

```
addedNumber, anyFloat, anyString = multiReturn(number1, number2, greeting)

// requested parameter : int, int, string
// returned values int, float64, string
```

Exercise 37: Areas of Shapes

Today's exercise has nothing new. It is merely additional practice with functions. This program has three functions (four if you count main) and they all have parameters and all three return values.

```
1  package main
2
3  import (
4      "fmt"
5      "math"
6  )
7
8  func main() {
9      var choice int = 0
10     var area float64 = 0
11
12     fmt.Print("Shape Area Calculator version 0.1")
13     fmt.Println(" (c) 2015 LJtHW Sample Output, inc.")
14
15     for choice = 1; choice != 4; {
16         fmt.Println("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n")
17         fmt.Println("1) Triangle")
18         fmt.Println("2) Circle")
19         fmt.Println("3) Rectangle")
20         fmt.Println("4) Quit")
21         fmt.Print("> ")
22         fmt.Scanf("\n%d", &choice)
23
24         if choice == 1 {
25             var b, h int
26             fmt.Println("\nBase : ")
27             fmt.Scanf("\n%d", &b)
28             fmt.Print("Height : ")
29             fmt.Scanf("\n%d", &h)
```

```
            area = computeTriangleArea(b, h)
            fmt.Println("The area is ", area)
        } else if choice == 2 {
            var radius int
            fmt.Println("\nRadius : ")
            fmt.Scanf("\n%d", &radius)
            area = computeCircleArea(radius)
            fmt.Println("The area is ", area)
        } else if choice == 3 {
            var q, w int
            fmt.Println("\nLength : ")
            fmt.Scanf("\n%d", &q)
            fmt.Println("\nLength : ")
            fmt.Scanf("\n%d", &w)
            fmt.Println("The area is ", computeRectangleArea(q, w))
        } else if choice != 4 {
            fmt.Println("ERROR.")
        }
    }
}

func computeTriangleArea(base int, height int) (A float64) {
    A = 0.5 * float64(base*height)
    return
}

func computeCircleArea(radius int) (A float64) {

    A = math.Pi * float64(radius*radius)
    return
}

func computeRectangleArea(length int, width int) (A int) {
    A = length * width
    return
}
```

What You Should See

```
Shape Area Calculator version 0.1 (c) 2015 LJtHW Sample Output, Inc.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
1) Triangle
2) Circle
3) Rectangle
4) Quit
> 1
Base: 3
Height: 5
The area is 7.5
-=-=-=-=-=-=-=-=-=-=-=-=-=-
1) Triangle
2) Circle
3) Rectangle
```

```
4) Quit
> 2
Radius: 3
The area is 28.274333882308138
-=-=-=-=-=-=-=-=-=-=-=-=-=-
1) Triangle
2) Circle
3) Rectangle
4) Quit
> 4
```

On line 51 we have defined a function to compute the area of a triangle (using just the base and height this time). It needs two arguments and will return a `double` value. On line 51 we used the variable named ***A***. This variable is "imported" by parameter to the function.
The value of the variable ***b*** (defined on line 25) is passed in as the initial value of the parameter ***base*** in the function call on line 30. ***b*** is stored into ***base*** because ***b*** is first, not because ***base*** starts with a ***b***. The computer doesn't care anything about that. Only the order matters.

On line 53 the value of ***A*** is returned to `main` and ends up getting stored in the variable called ***area***.

Exercise 38: Thirty Days Revisited with Documentation

In the previous exercise we wrote some functions that might have been better off omitted. In today's exercise we are going to re-do a previous exercise, making it better with functions.

And, because I always like to cover something new, I have added special comments above the class and above each function called "Documentation comments". You should type them in.

```
package main

/**
* Contains functions that make it easier to work with months.
*/
import (
    "fmt"
)

func main() {

    var month int
    fmt.Printf("Which month? (1 - 12) \n")
    fmt.Scanf("%d", &month)

    fmt.Printf("%s days hath %d", monthName(month), monthDays(month))
}

/**
* Returns the name for the given month number (1-12).
*
* @author Graham Mitchell
* @param month the month number (1-12)
* @return the English name of the month, or "error" if out of range
*/

```

```go
func monthName(month int) (monthName string) {
    monthName = "error"
    if month == 1 {
        monthName = "January"
    } else if month == 2 {
        monthName = "February"
    } else if month == 3 {
        monthName = "March"
    } else if month == 4 {
        monthName = "April"
    } else if month == 5 {
        monthName = "May"
    } else if month == 6 {
        monthName = "June"
    } else if month == 7 {
        monthName = "July"
    } else if month == 8 {
        monthName = "August"
    } else if month == 9 {
        monthName = "September"
    } else if month == 10 {
        monthName = "October"
    } else if month == 11 {
        monthName = "November"
    } else if month == 12 {
        monthName = "December"
    }

    return
}

/**
* Returns the number of days in a given month.
*
* @author Graham Mitchell
* @param month the month number (1-12)
* @return the number of days in the month, or 31 if out of range
*/

func monthDays(month int) (days int) {

    switch month {
    case 9:
    case 4:
    case 6:
    case 11:
        days = 30
        break
    case 2:
        days = 28
        break
    default:
        days = 31
    }

    return
}
```

What You Should See

```
Which month? (1-12) 9
30 days hath September
```

If you ignore the Documentation comments for now, hopefully you should see that using functions here actually improves the code. `main()` is very short, because most of the interesting work is being done in the functions.

All the code and variables pertaining to the name of the month are isolated in the `monthName()` function. And all the code to find the number of days in a month is contained inside the `monthDays()` function.

Collecting variables and code into functions like this is called “procedural programming” and it is considered a major advance over just having all your code in `main()`. It makes your code easier to debug because if you have a problem with the name of the month, you ***know*** it has to be inside the `monthName()` function.
Okay, now let’s talk about the Javadoc comments. You write documentation right in your code by doing a special sort of block comment above classes, functions or variables.

The comment begins with `/**` and ends with `*/` and every line in between starts with an asterisk (`*`) which is lined up like you see in the exercise.

The first line of the javadoc comment is a one-sentence summary of the thing (class or function). And then there are tags like `@author` or `@return` that give more detail about who wrote the code, what parameters the function expects or what value it is going to return.

Exercise 39: Importing Standard Libraries

In the last exercise you got a terrifying look at all of the built-in modules that are available in Go.

Today we will look at a “simple” program that took me about half an hour to write because I spent a lot of time searching the Internet and importing things and trying things that didn’t work.

This code works, though. It allows the human to enter a password (or anything, really) and then prints out the SHA-256 message digest of that password.

```go
package main

import (
    "crypto/sha256"
    "fmt"
)

func main() {
    var pw string

    fmt.Printf("Password : ")
    fmt.Scanf("%s", &pw)

    data := []byte(pw)
    hash := sha256.Sum256(data)

    fmt.Printf("%x", hash[:])
}
```

What You Should See

```
Password: password
5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8
```

That 64-character long string is the SHA-256 digest of the String `password`. That message digest will always be the same for that input.

If you type in a different password, you'll get a different digest, of course:

```
Password: hunter2
F52FBD32B2B3B86FF88EF6C490628285F482AF15DDCB29541F94BCF526A3F6C7
```

Back in the early days of programming, when machines first started having usernames and password, it was pretty obvious that you wouldn't want to store the passwords themselves in a database. Instead, they would store some sort of cryptographic hash of the password.

Cryptographic hashes have two useful properties:

1. They are consistent. A given input will always produce exactly the same output.
2. They are one-way. You can easily compute the output for a given input, but figuring out the input that gave you a certain output is very hard or impossible.

SHA-256 is a very good cryptographic hash function, and it produces a "digest" for a given input (or "message") that is always exactly 256 bits long. Here instead of trying to deal with bits we have printed out the base-16 representation of those bits. Since each hexadecimal (base-16) digit corresponds to 4 bits, we end up with an output 64 characters long.

Okay, enough about secure passwords, let us walk through this code. You might want to have the documentation for this one library open: "crypto/sha256"

On line 4 we import the library we will be using to do the hard parts of this exercise.

On line 14, we meet variable with type data of byte, well byte is a type data that contains with numbers that actually is a data code of strings or number. As you can see that we create a byte from our input so it can be converted into sha256.

On line 15, we use our library, “sha256” to convert this byte into a sha256-hash, so when it’s converted we can Printf it but with “%x” and also remember to put [:], so maybe you unfamiliar with this but we will talk about it later.

Exercise 40: Programs that Write to Files

We are going to take a break from focusing on functions now for a bit and learn something easy.

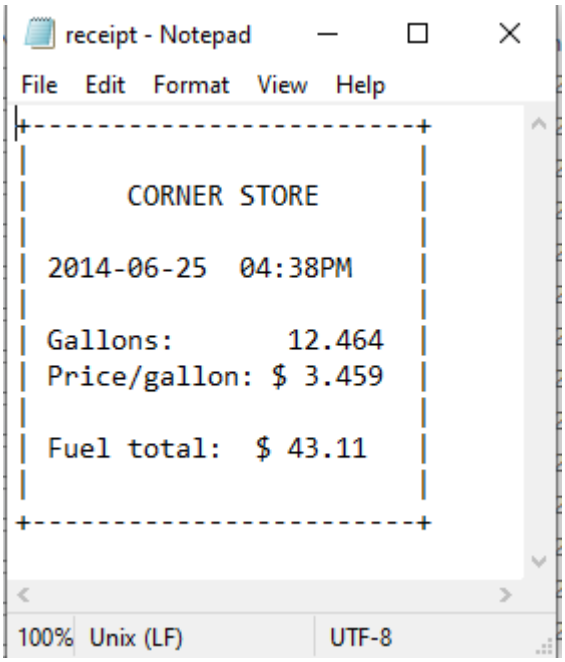
We are going to create a program that can put information into a text file instead of only being able to print things on the screen.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fileout, err := os.Create("receipt.txt")
    if err != nil {
        fmt.Println(err)
        return
    }

    fileout.WriteString("+-------------------------+\n")
    fileout.WriteString("|                         |\n")
    fileout.WriteString("|      CORNER STORE       |\n")
    fileout.WriteString("|                         |\n")
    fileout.WriteString("| 2014-06-25  04:38PM     |\n")
    fileout.WriteString("|                         |\n")
    fileout.WriteString("| Gallons:       12.464   |\n")
    fileout.WriteString("| Price/gallon: $ 3.459   |\n")
    fileout.WriteString("|                         |\n")
    fileout.WriteString("| Fuel total:  $ 43.11    |\n")
    fileout.WriteString("|                         |\n")
    fileout.WriteString("+-------------------------+\n")
    fileout.Close()
}
```

What You Should See

That's right. When you run your program, it will appear to do nothing. But if you wrote it correctly, it should have created a file called receipt.txt in the same folder your

code is in. You can view this file using the same text editor you are using to write your code.
If for some reason you are using the version of Notepad that came with Windows 10, it will look a little something like this:



a screenshot of the file “receipt.txt” opened in Notepad

On line 9 there is a new `import` statement for the Java class that will make this easy.

On line 9 we declare and initialized a variable. The variable is of type `os` and I have chosen to name it ***fileout*** (although the variable’s name doesn’t matter).

On this same line we give the import variable a value: the reference to a new import object. Creating the import object requires an argument, though. The argument we give it is a String containing the desired output filename. (The name of the file to write to.)

On line 10, we make an exception so if there’s an error it will handled by our if statement!

So on line 15 you can see that writing to the file looks very similar to printing on the screen. But the String (+-----) will ***not*** be printed on the screen. It will be stored as the first line of the file `receipt.txt`!

If a file named `receipt.txt` already exists in that folder, its contents will be overwritten without warning. If the file does not exist, it will be created.

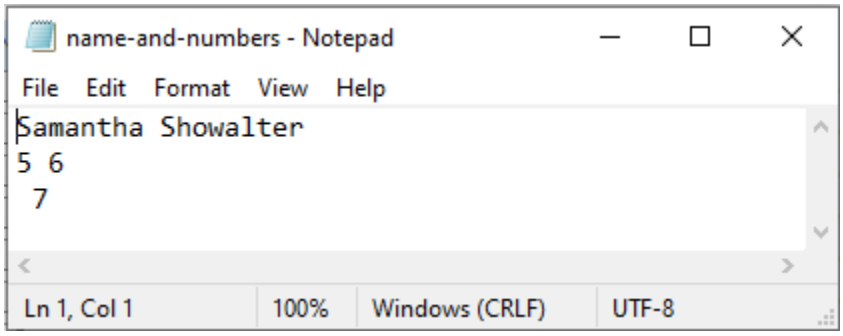
The only other important line in the exercise is line 27. This actually saves the contents of the file and closes it so your program can't write to it anymore. If you remove this line, your program will most likely create a file called `receipt.txt`, but the file will be empty.

Exercise 41: Getting Data from a File

A program that can put information into a file is only part of the story. So in this exercise you will learn how to read information that is already in a text file.

If you type up this code and compile it and run, it will blow up. This is because it is trying to read from a text file called name-and-numbers.txt, which must be in the same folder as your code. You probably don't have a file like this!

So before you even write the code, let us make a text file containing a String and three integers. My file looks like this:



a screenshot of the file "name-and-numbers.txt" opened in Notepad

Okay, to the code!

```
1  package main
2
3  import (
4      "bufio"
5      "fmt"
6      "os"
7      "strconv"
8  )
9
10 func main() {
11     var name string
12     var a, b, c, sum int
13
14     fmt.Print("Getting name and three numbers from file...")
15     fileIn, err := os.Open("name-and-numbers.txt")
16     if err != nil {
17         fmt.Println(err)
18         return
19     }
20
```

```
21      scanner := bufio.NewScanner(fileIn)
22      scanner.Split(bufio.ScanWords)
23
24      scanner.Scan()
25      name = scanner.Text()
26      scanner.Scan()
27      name += " " + scanner.Text()
28      scanner.Scan()
29      a, _ = strconv.Atoi(scanner.Text())
30      scanner.Scan()
31      b, _ = strconv.Atoi(scanner.Text())
32      scanner.Scan()
33      c, _ = strconv.Atoi(scanner.Text())
34      fileIn.Close()
35
36      fmt.Println("done.")
37      fmt.Println("Your name is " + name)
38      sum = a + b + c
39      fmt.Println(strconv.Itoa(a) + "+" + strconv.Itoa(b) + "+" + strconv.Itoa(c) + " = " + strconv.Itoa(sum))
40  }
```

What You Should See

```
Getting name and three numbers from file...done.
Your name is Samantha Showalter
5+6+7 = 18
```

We just create the Scanner object slightly differently: instead of System.in as the argument, we use os.Open("blah.txt"). This will open the text file read-only.

Line 25 looks pretty uninteresting. It “pauses” the program and reads in a String from the Scanner object, which gets it from the file. This String from the file is stored into the variable.

Lines 29 through 33 are simple, too. Except what is read from the file is converted to an integer before putting it in the variables.

What if the next thing in the file isn’t an integer? Then your program will blow up. And now you can’t blame the human anymore: you created this file. It is your job to make sure you know what values are in it, and in what order.

On line 34 the file is closed, which means your Scanner object isn’t connected to it anymore. If you ever write a program that uses the same file for both reading and

writing, then forgetting to `.Close()` the file when reading will make it so you can't write to it either. Was this easier than you expected it to be? Hopefully so.

Exercise 42: Getting ALL the Data from a File

In the previous exercise, we knew the file's contents: one line with a number and then exactly three numbers. If there had been seven numbers in the file only the first three would have been used. And if there had been only *two* numbers in the file, the program would have ended with an error when run.

So we often use a loop to get *all* the values in a file, no matter how many or how few!
Start by creating new text file called some-words.txt in the same folder as the code. My file looks like this:

some-words.txt

```
The man in black fled across the desert, and the gunslinger followed.
Ask not what your country can do for you; ask what you can do for your country.
i   will be
M o ving in the Street of her
bodyfee l inga ro undMe the traffic of
lovely;muscles-sinke x  p  I  r   i  n             g S
uddenl
Y   totouch
```

The contents don't matter too much, though, which is sort-of the whole point! Once this file exists and has some words in it you should be able to type in and run the code below.

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "strconv"
    "unicode"
)

func main() {
    fourLetter := 0
    caps := 0

    fn1 := "some-words.txt"
    fn2 := "42GettingAllTheDataFromAFile.go"

```

```
18      fileIn, err := os.Open(fn1)
19      if err != nil {
20          fmt.Println(err)
21          return
22      }
23      wordReadr := bufio.NewScanner(fileIn)
24      wordReadr.Split(bufio.ScanWords)
25
26      for wordReadr.Scan() {
27          w := wordReadr.Text()
28          if len(w) == 4 {
29              fourLetter++
30          }
31      }
32      fileIn.Close()
33
34      fileIn, err = os.Open(fn2)
35      if err != nil {
36          fmt.Println(err)
37          return
38      }
39      selfInput := bufio.NewScanner(fileIn)
40      selfInput.Split(bufio.ScanWords)
41
42      for selfInput.Scan() {
43          token := selfInput.Text()
44          for i, char := range token {
45              if i == 0 && unicode.IsUpper(char) {
46                  caps++
47              }
48          }
49      }
50      fileIn.Close()
51
52      fmt.Println(strconv.Itoa(fourLetter) + " Four-letter words in " + fn1)
53      fmt.Println(strconv.Itoa(caps) + " Words start with capitals in " + fn2)
54  }
```

What You Should See

```
9 Four-letter words in some-words.txt
2 Words start with capitals in 42GettingAllTheDataFromAFile.go
```

Just like before, we create an object and attach it to read from a file (line 18). This time, the file name is stored in a String variable first. Our Scanner is called ***wordReadr*** but we could have called it anything.

Line 26 is where the magic starts. We use a new import called “bufio” that can checks every words on the file, so it will still check it until we have check everything.

Inside the body of the loop, we read a single String into a new local variable called ***w***. And we add to a counter if ***w*** is four characters long.

Once the loop is over, we've read everything in the file so we `.Close()` it.

Exercise 43: Saving a High Score

Now that you know how to get information from files ***and*** how to put information in files, we can create a game that saves the high score!

This is the coin flipping game from a few exercises ago, but now the high score is saved from run to run.

```
package main

import (
    "bufio"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func main() {
    var coin, again, bestName string
    saveFileName := "coin-flip-score.txt"
    streak := 0
    var best int
    var gotHeads bool
    rand.Seed(time.Now().UTC().UnixNano())

    f, err := os.Open(saveFileName)
    if err != nil {
        fmt.Println(err)
        return
    }

    wordReadr := bufio.NewScanner(f)
    wordReadr.Split(bufio.ScanWords)

    wordReadr.Scan()
    if wordReadr.Text() == "" {
        fmt.Println("Save game file doesn't exist or is empty.")
        best = -1
        bestName = ""
    } else {
        bestName = wordReadr.Text()
        wordReadr.Scan()
        best, _ = strconv.Atoi(wordReadr.Text())
        fmt.Print("High score is " + strconv.Itoa(best))
        fmt.Println(" flips in a row by " + bestName)
        f.Close()
    }

    for again = "y"; again == "y"; {
        gotHeads = rand.Float64() < 0.5

        if gotHeads {
            coin = "HEADS"
        } else {
            coin = "TAILS"
        }

```

```
52          fmt.Println("You flip a coin and it is... " + coin)
53
54          if gotHeads {
55              streak++
56              fmt.Println("\tThat's " + strconv.Itoa(streak) + " in a row....")
57              fmt.Print("\tWould you like to flip again (y/n)? ")
58              fmt.Scan(&again)
```

```
59          } else {
60              streak = 0
61              again = "n"
62          }
63      }
64
65      fmt.Println("Final score: " + strconv.Itoa(streak))
66
67      if streak > best {
68          fmt.Println("That's a new high score!")
69          fmt.Print("Your name: ")
70          fmt.Scan(&bestName)
71          best = streak
72      } else if streak == best {
73          fmt.Println("That ties the high score. Cool.")
74      } else {
75          fmt.Print("You'll have to do better than ")
76          fmt.Println(strconv.Itoa(streak) + " if you want a high score.")
77      }
78
79      // Save this name and high score to the file.
80      f, err = os.Create(saveFileName)
81      if err != nil {
82          fmt.Println(err)
83          return
84      }
85      f.WriteString(bestName + "\n")
86      f.WriteString(strconv.Itoa(best))
87      f.Close()
88  }
```

What You Should See

```
Save game file doesn't exist or is empty.
You flip a coin and it is... HEADS
That's 1 in a row....
Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
That's 2 in a row....
Would you like to flip again (y/n)? y
You flip a coin and it is... HEADS
That's 3 in a row....
Would you like to flip again (y/n)? y
You flip a coin and it is... TAILS
Final score: 0
That's a new high score!
Your name: Mitchell
```

(Okay, so I cheated. It took me quite a few tries to get a streak of three in a row.)

On line 20 we create an `os` object using the filename `coin-flip-score.txt`.

On line 30 there is an `if` statement, and in the condition to check if it's empty so it will tell the save-file game is empty.

When the `if` statement is false, then, it means the save game file does exist and has *something* stored in it. (Hopefully a name and high score!) So, we use a Scanner object to get the existing name and high score out of the file.

If the file doesn't exist or doesn't have anything in it, we say so and put suitable initial values into the variables *best* and *bestName*. Cool, eh?

Lines 43 through 65 are the existing coin flip game. I didn't change any of this code at all. On line 67 we need to figure out if they beat the high score. If so, we print out a message to that effect and let them enter their name.

If they tied the high score, we say so, but they don't get any fame for that.
And on line 74 the `else` will run if they didn't beat or tie the high score. So we taunt them, of course.

On lines 80 through 87 we save the current high score along with the name of the high scorer to the file. This might be a new score, or it might be the previous value we read at the beginning of the program.

Exercise 44: Counting with a For Loop

As you have seen in previous exercises, loops can be used to make something happen more than once.

But both kinds of loops are designed to keep going *as long as* something is true. If we know in advance how many times we want to do something, Go has a special kind of loop designed just for making a variable change values: the `for` standard loop.

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var n int
    var message string

    fmt.Println("Enter a message and I'll display it five times.")
    fmt.Print("Message: ")
    fmt.Scanf("%s", &message)

    for n = 1; n <= 5; n++ {
        fmt.Println(strconv.Itoa(n) + ". " + message)
    }

    fmt.Println("\nNow I'll show it ten times and count by 5s.")
    for n = 5; n <= 50; n += 5 {
        fmt.Println(strconv.Itoa(n) + ". " + message)
    }

    fmt.Println("\nFinally, three times counting backward.")
    for n = 3; n > 0; n -= 1 {
        fmt.Println(strconv.Itoa(n) + ". " + message)
    }
}
```

What You Should See

```
Enter a message and I'll display it five times.
Message: Howdy, y'all!
1. Howdy, y'all!
2. Howdy, y'all!
3. Howdy, y'all!
```

```
4. Howdy, y'all!
5. Howdy, y'all!

Now I'll show it ten times and count by 5s.
5. Howdy, y'all!
10. Howdy, y'all!
15. Howdy, y'all!
20. Howdy, y'all!
25. Howdy, y'all!
30. Howdy, y'all!
35. Howdy, y'all!
40. Howdy, y'all!
45. Howdy, y'all!
50. Howdy, y'all!

Finally, three times counting backward.
3. Howdy, y'all!
2. Howdy, y'all!
1. Howdy, y'all!
```

Line 16 demonstrates a very basic `for` loop. Every `for` loop has three parts with semicolons between.

Most `for` loops pick a single variable and name that variable in all three parts.

1. The ***initialization statement*** tells the variable where to start.
2. The loop keeps going as long as the ***condition*** is true.
3. The ***update statement*** controls how the variable is changed each time.

The first part (`n=1`) only happens once no matter how many times the loop repeats. It happens at the very beginning of the loop and usually sets a starting value for some variable that is going to be used to control the loop. In this case, our “loop control variable” is ***n*** and it will start with a value of `1`. This first part is called the “ initialization” statement

The second part (`n <= 5`) is a condition, just like the condition of a `while` or do-while loop. The `for` loop is a pre-test loop just like a `while` loop, which means that this condition is tested before the loop starts looping. If the condition is true, the loop body will be executed one time. If the condition is false, the loop body will be skipped and the loop is over.

The third part (`n++`) runs ***after*** each iteration of the loop, just before it checks the condition again. Remember that `++` adds one to a variable. The third part is sometimes called an "update statement".

After the three parts in parentheses there are curly braces which contain the "body" of the loop.

Note that the first and last parts of the `for` loop are complete statements; they would compile if you placed them on a line by themselves in some other Go program. But the middle part (the condition) isn't a complete statement, it's just an expression that evaluates to either `true` or `false`.

So if we ***unroll*** this loop, these are the statements that will happen and their order:

```
n = 1
// check if ( n <= 5 ), which is true
fmt.Printf( 1 + "." + message )
n++  // so now n is 2
// check if ( n <= 5 ), which is true
fmt.Printf( 2 + "." + message )
n++  // so now n is 3
// check if ( n <= 5 ), which is true
fmt.Printf( 3 + "." + message )
n++  // so now n is 4
// check if ( n <= 5 ), which is true
fmt.Printf( 4 + "." + message )
n++  // so now n is 5
// check if ( n <= 5 ), which is true
fmt.Printf( 5 + "." + message )
n++  // so now n is 6
// check if ( n <= 5 ), which is false. The loop stops
```

Notice that the first part only happened once, and that the third part happened exactly as many times as the loop body did.

On line 21 there is another `for` loop. The loop control variable is still ***n***. (Notice that the loop control variable appears in all three parts of the loop. This is almost always the case.)

The first part (the "initialization" statement) sets the loop control variable to start at `5`. Then the second part checks to see if ***n*** is less than or equal to `50`. If so, the body is executed one time and then the third part is executed. The third part adds `5` to the loop control variable, and then the condition is checked again. If it is still true, the loop repeats. Once it is false, the loop stops.

On line 26 there is one final `for` loop. This time the loop control variable starts at `3` and the loop repeats as long as ***n*** is greater than zero. And after each iteration of the loop body the third part (the "update expression") ***subtracts*** `1` from the loop control variable.

So when should you use a `for` loop versus a `while` loop?
`for` loops are best when we know in advance how many times we want to do something.

- Do this ten times.
- Do this five times.
- Pick a random number, and do it that many times.
- Take this list of items, and do it one time for each item in the list.

On the other hand, `while` and do-while loops are best for repeating ***as long as*** something is true:

- Keep going as long as they haven't guessed it.

- Keep going as long as you haven’t got doubles.
- Keep going as long as they keep typing in a negative number.
- Keep going as long as they haven’t typed in a zero.

Exercise 45: Caesar Cipher (Looping Through a String)

The Caesar cipher is a very simple form of cryptography named after Julius Caesar, who used it to protect his private letters. In the cipher, each letter is shifted up or down in the alphabet by a certain amount. For example, if the shift is 2, then all As in the message are replaced with C, B is replaced with D, and so on.

This exercise is pretty complicated but it is not very important. Don't worry if you get confused

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "unicode"
)

/**
* Returns the character shifted by the given number of letters.
*/
func shiftLetter(c rune, n int) rune {
    var u int = (int)(c)

    if !unicode.IsLetter(c) {
        return c
    }

    u = u + n
    if unicode.IsUpper(c) && u > 'Z' || unicode.IsLower(c) && u > 'z' {
        u -= 26
    }
    if unicode.IsUpper(c) && u < 'A' || unicode.IsLower(c) && u < 'a' {
        u += 26
    }

    return (rune)(u)
}

func main() {
    keyboard := bufio.NewReader(os.Stdin)
    var plaintext string
    var cipher string = ""
    var shift int

    fmt.Print("Message: ")
    plaintext, _ = keyboard.ReadString('\n') // To read a line, equals to .nextLine() in java
    fmt.Print("Shift (0-26): ")
    fmt.Scan(&shift)

    for i := 0; i < len(plaintext); i++ {
        stringToRune := rune(plaintext[i])
        cipher += fmt.Sprintf("%c", shiftLetter(stringToRune, shift))
    }
    fmt.Println(cipher)
}
```

What You Should See

```
Message: This is a test. XyZaBcDeF
Shift (0-26): 2
Vjku ku c vguv. ZaBcDeFgH
```

Did you know that main() doesn't have to be the first function in the class? Well, it doesn't. Functions can appear in any order.

Also in addition to int, float, String and boolean there is a basic variable type I haven't

mentioned: `rune`. A `rune` variable can hold characters like `Strings` do, but it can only hold ***one*** character at a time. String literals in the code are enclosed in double quotes like `"Axe"`, while `rune` literals in the code are in single quotes like `'A'`.
Starting on line 13 there is a function called `shiftLetter()`. It has two parameters: ***c*** is the character to shift and ***n*** is the number of spaces to shift it. This function returns a `rune`. So `shiftLetter('A', 2)` would return the character `'C'`.

We don't want to try to shift anything that isn't a letter, so on line 16 we use the built-in Character class to tell us.

And since we are going to be doing a little math with the character, we store the character's Unicode value into an `int` on line 14 to make this easier. Then on line 20 we add the desired offset to the character

Exercise 46: Nested For Loops

In programming, the term “nested” usually means to put something inside the same thing. “Nested loops” would be two loops with one inside the other one. If you do it right this means the inner loop will repeat all its iterations every time the outer loop does one more iteration.
(One “iteration” is a single time through a repeated process.)

```go
package main

import "fmt"

func main() {
    // this is #1 - I'll call it "CN"
    for c := 'A'; c <= 'E'; c++ {
        for n := 1; n <= 3; n++ {
            fmt.Printf("%c %d\n", c, n)
        }
    }

    fmt.Println("\n")

    // this is #2 - I'll call it "AB"
    for a := 1; a <= 3; a++ {
        for b := 1; b <= 3; b++ {
            fmt.Print("(", a, ",", b, ") ")
        }
        // * You will add a line of code here.
    }

    fmt.Println("\n")
}
```

What You Should See

```
A 1
A 2
A 3
B 1
B 2
B 3
C 1
C 2
C 3
D 1
D 2
D 3
E 1
```

```
E 2
E 3
(1,1) (1,2) (1,3) (2,1) (2,2) (2,3) (3,1) (3,2) (3,3)
```

I’m not really going to explain this one. Just look closely at the output and the code.

Exercise 47: Generating and Filtering Values

Nested `for` loops are sometimes handy because they are very compact and can make some variables change through a lot of different combinations of values.
Many years ago, a student posed the following math problem to me:

> "Farmer Brown wants to spend exactly $100.00 and wants to purchase exactly 100 animals. If sheep cost $10 each, goats cost $3.50 each and chickens are $0.50 a piece, then how many of each animal should he buy? (He wants at least one of each type of animal.)"

After he left, I thought about it for a few minutes and then wrote the following program. (True story, by the way.)

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       for s := 1; s <= 100; s++ {
7           for g := 1; g <= 100; g++ {
8               for c := 1; c <= 100; c++ {
9                   if s+g+c == 100 && 10.00*float64(s)+3.50*float64(g)+0.50*float64(c) == 100.00 {
10                      fmt.Print(s, " sheep, ")
11                      fmt.Print(g, " goats, and ")
12                      fmt.Println(c, " chickens.")
13                  }
14              }
15          }
16      }
17  }
18
```

What You Should See

```
4 sheep, 4 goats, and 92 chickens.
```

This program is neat because it is very short. But an observer sitting inside the innermost loop (just in front of the `if` statement on line 9 will see one million different combinations of *s*, *g* and *c* flow by. The first combination attempted will be 1 sheep, 1

goat, 1 chicken. That will be plugged into the math equations in the `if` statement. They won't be true, and nothing will be printed.
Then the next combination will be 1 sheep, 1 goat and 2 chickens. Which will also fail. Then 1 sheep, 1 goat, 3 chickens. And so on up to 1 sheep, 1 goat and 100 chickens when the inner loop runs its last iteration.

Then the `g++` on line 7 will execute, the condition on line 7 will check to make sure *g* is still less than or equal to 100 (which it is) and the body of the middle `for` loop will execute again.

This will cause the initialization statement of the innermost loop to run again, which resets *c* to 1.

So the next combination of variables that will be tested in the `if` statement is 1 sheep, 2 goats and 1 chicken. Then 1 sheep, 2 goats, 2 chickens, then 1 sheep, 2 goats, 3 chickens. *Et cetera*.

By the end all 100 * 100 * 100 combinations have been tested and 999,999 of them failed. But because computers are very fast, the answer appears instantaneously.

Exercise 48: Arrays - Many Values in a Single Variable

In this exercise you will learn two new things. The first one is ***super*** important and the second one is just kind-of neat.

In Go, an “array” is a type of variable with one name (“identifier”) but containing more than one value. In my opinion, you’re not a Real Programmer until you can work with arrays. So, that’s good news. You’re almost there!

```
1   package main
2
3   import (
4       "fmt"
5       "strings"
6   )
7
8   func main() {
9       planets := []string{"Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"}
10
11      for _, p := range planets { // range function return 2 Values,
12                                  // the first one is number(with index+1 value)
13                                  // and the second one is the array value
14          fmt.Println(p + "\t" + strings.ToUpper(p))
15      }
16  }
```

On line 9 we declare and define a variable named ***planets***. It is not just a String: notice the square brackets. This variable is an ***array*** of Strings. That means that this one variable holds all eight of those Strings and they are separated into distinct slots so we can access them one at a time.

The curly braces on this line are used for a different purpose than usual. All these values are in quotes because they are Strings. There are commas between each value, then the whole initializer list is in curly braces.

The second new thing in this exercise is a new kind of `for` loop. (This is sometimes called a “foreach” loop, since it works a bit like a loop in another programming

language where the keyword actually is `foreach` instead of `for`. It is also sometimes called an “enhanced for loop”.)

On line 11 you will see this foreach loop in action. You read it out loud like this: “for each String ‘p’ in the array ‘planets’….”

So inside the body of this foreach loop the String variable ***p*** will take on a copy of each value in the String array ***planets***. That is, the first time through the loop, ***p*** will contain a copy of the first value in the array (`"Mercury"`). Then the second time through the loop, ***p*** will contain a copy of the second value in the array (`"Venus"`). And so on, until all the values in the array have been seen. Then the loop will automatically stop.

Inside the body of the loop (on line 14) we are just printing out the current value of ***p*** and an uppercase version of ***p***. Just for fun, I guess.

This new kind of `for` loop only works with compound variables like this: variables that have one name but contain multiple values. Arrays aren’t the only sort of compound variable in Go, but we won’t be looking at any of the others in this book.

Arrays are a big deal, so that’s enough for this exercise. I want to make ***absolutely sure*** you understand what is happening in this assignment before throwing more on your plate.

Exercise 49: Finding Things in an Array

More with arrays! In this exercise we will examine how to find a particular value. The technique we are using here is sometimes called a “linear search” because it starts with the first slot of the array and looks there, then moves to the second slot, then the third and so on down the line

```
1   package main
2
3   import (
4       "fmt"
5       "strconv"
6   )
7
8   func main() {
9       orderNumbers := []int{12345, 54321, 101010, 8675309, 31415, 271828}
10      var toFind int
11
12      fmt.Print("There are " + strconv.Itoa(len(orderNumbers)))
13      fmt.Println(" orders in the database.")
14
15      fmt.Print("Orders: ")
16      for _, num := range orderNumbers {
17          fmt.Print(strconv.Itoa(num) + " ")
18      }
19      fmt.Println()
20
21      fmt.Print("Which order to find? ")
22      fmt.Scan(&toFind)
23
24      for _, num := range orderNumbers {
25          if num == toFind {
26              fmt.Println(strconv.Itoa(num) + " found.")
27          }
28      }
29  }
```

What You Should See

```
There are 6 orders in the database.
Orders: 12345 54321 101010 8675309 31415 271828
Which order to find? 78753
```

This time the array is named *orderNumbers* and it is an array of integers. It has six slots. `12345` is the first slot, and `271828` is in the last slot of the array. Each of the six slots can hold an integer.

When we create an array Java gives us a built-in variable called .length which tells us the capacity of the array. This variable is read-only; you can retrieve its value but not change it. In this case, since the array ***orderNumbers*** has six slots, the variable orderNumbers.length is equal to 6. The len(arrayVariable) field is used on line 12.

On line 16 we have a foreach loop to display all the order numbers on the screen. " For each integer 'num' in the array 'orderNumbers'…." So inside the body of this loop, ***num*** will take on each value in the array one at a time and display them all.

On line 22 we let the human type in an order number. Then we use a loop to let ***num*** take on each order number and compare them to ***toFind*** one at a time. When we have a match, we say so.

Exercise 50: Saying Something Is NOT in an Array

In life, there is a general lack of symmetry between certain types of statements. A white crow exists.

This statement is easy enough to prove. Start looking at crows. Once you find a white one, stop. Done.

> No white crows exist.

This statement is ***much*** harder to prove because to prove it we have to gather up everything in the world that qualifies as a crow. If we have looked at them ***all*** and not found any white crows, only then can we safely say that ***none*** exist.

```go
package main

import "fmt"

func main() {
    heroes := []string{"Abderus", "Achilles", "Aeneas", "Ajax", "Amphitryon",
        "Bellerophon", "Castor", "Chrysippus", "Daedalus", "Diomedes",
        "Eleusis", "Eunostus", "Ganymede", "Hector", "Iolaus", "Jason",
        "Meleager", "Odysseus", "Orpheus", "Perseus", "Theseus"}
    var guess string
    var found bool

    fmt.Println("Pop Quiz!")
    fmt.Print("Name any *mortal* hero from Greek mythology: ")
    fmt.Scan(&guess)

    found = false
    for _, hero := range heroes {
        if guess == hero {
            fmt.Println("That's one of them!")
            found = true
        }
    }

    if found == false {
        fmt.Println("No, " + guess + " wasn't a Greek mortal hero.")
    }
}
```

What You Should See

```
Pop Quiz!
Name any *mortal* hero from Greek mythology: Hercules
No, Hercules wasn't a Greek mortal hero
```

Most students want to solve this problem by putting another `if` statement (or an `else`) inside the loop to say “not found”. But this can never work.

If I want to know if something ***is*** found, it is okay to say so as soon as I find it. But if I want to know if something was ***never*** found, you have to wait until the loop is over before you know for sure.

So in this case I use a technique called a “flag”. A flag is a variable that starts with one value and the value is changed when/if something happens. Then later in the program you can use the value of the flag to see if the thing happened or not.

My flag variable is a Boolean called ***found***, which is set to `false` on line 17. If a match is found, we say so ***and*** change the flag to `true` on line 21. Notice that inside the body of the loop there is no code that can change the flag to `false`, so once it has been flipped to `true` it will stay that way.

Then on line 25, after the loop is done, you can examine the flag. If it is still `false`, then we know the `if` statement inside the loop was never true and therefore we never found what we were looking for.

This is a pretty important trick. I use code like this ***all*** the time when I’m writing programs.

Exercise 51: Arrays Without Foreach Loops

As you might noticed by now, arrays and foreach loops are designed to work together well. But there are situations where what we have been doing won't work.

- A foreach loop can't iterate through an array ***backward***; it can only go forward.
- A foreach loop can't be used to ***change*** the values in the array. The foreach loop variable is a read-only copy of what's in the array and changing it doesn't affect the array.

In addition, we have only been putting values into an array using an initializer list (the curly braces thing), which has its own limitations:

- An initializer list only works when the array is first being declared; you can't use it elsewhere in the code.
- An initializer list is best suited for relatively small arrays, if you have 1000+ values in the array, an initializer list will be no fun.
- Initializer lists don't help us if we want the values in the array to come from a file or some other place we don't have when we are typing the code.

So there is another way to store values in an array and access them. In fact, it is more common than what you have been doing. Using square brackets and a slot number, we can access the slots of an array individually.

```go
package main

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

func main() {
    var arr [3]int
    rand.Seed(time.Now().UTC().UnixNano())

    // We almost always use a for loop to access each slot of an array.
    for i := 0; i < len(arr); i++ {
        arr[i] = 1 + (int)(rand.Float64()*100)
    }

    // Displaying all the values in an array usually looks like this
    fmt.Print("Values: ")
    for i := 0; i < len(arr); i++ {
        fmt.Print(strconv.Itoa(arr[i]) + " ")
    }
    fmt.Println()

    ////////////////////////////////////
    // But let's break this down step-by-step...
    // Put a number into each slot of the array, one at a time.
    arr[0] = 6
    arr[1] = 7
    arr[2] = 8

    // Then display the values in those slots, one at a time.
    fmt.Println("Values: " + strconv.Itoa(arr[0]) + " " + strconv.Itoa(arr[1]) + " " + strconv.Itoa(arr[2]))

    ////////////////////////////////////
    // Put a random number 1-100 into each slot of the array, one at a time.
    arr[0] = 1 + (int)(rand.Float64()*100)
    arr[1] = 1 + (int)(rand.Float64()*100)
    arr[2] = 1 + (int)(rand.Float64()*100)

    // Display them again, one at a time.
    fmt.Println("Values: " + strconv.Itoa(arr[0]) + " " + strconv.Itoa(arr[1]) + " " + strconv.Itoa(arr[2]))

    ////////////////////////////////////
    // This is a bit silly, but try to understand it.
    var m int = 0
    arr[m] = 1 + (int)(rand.Float64()*100)
    m = 1
    arr[m] = 1 + (int)(rand.Float64()*100)
    m = 2
    arr[m] = 1 + (int)(rand.Float64()*100)

    // Display them again.
    fmt.Print("Values: ")
    m = 0
    fmt.Print(strconv.Itoa(arr[m]) + " ")
    m = 1
    fmt.Print(strconv.Itoa(arr[m]) + " ")
    m = 2
    fmt.Print(strconv.Itoa(arr[m]) + " ")
    fmt.Println()

```

```
64      ////////////////////////////////////
65      // This is even more silly but it works.
66      var n int = 0
67      arr[n] = 1 + (int)(rand.Float64()*100)
68      n++
69      arr[n] = 1 + (int)(rand.Float64()*100)
70      n++
71      arr[n] = 1 + (int)(rand.Float64()*100)
72      n++
73
74      // Display them again.
75      fmt.Print("Values: ")
76      n = 0
77      fmt.Print(strconv.Itoa(arr[n]) + " ")
78      n++
79      fmt.Print(strconv.Itoa(arr[n]) + " ")
80      n++
81      fmt.Print(strconv.Itoa(arr[n]) + " ")
82      n++
83      fmt.Println()
84
85      ////////////////////////////////////
86      // Now does using a loop make more sense?
87      for q := 0; q < len(arr); q++ {
88          arr[q] = 1 + (int)(rand.Float64()*100)
89      }
90
91      // I hope so. If not, read through this code again more slowly.
92      fmt.Print("Values: ")
93      for q := 0; q < len(arr); q++ {
94          fmt.Print(strconv.Itoa(arr[q]) + " ")
95      }
96      fmt.Println()
97  }
```

What You Should See

```
Values: 82 42 59
Values: 6 7 8
Values: 13 21 41
Values: 42 44 78
Values: 83 93 78
Values: 61 74 96
```

On line 11 we are creating an array of integers ***without*** using an initializer list. The `[3]` means that the array has a capacity of 3. Since we didn't provide values, every slot in the array starts out with a value of `0` stored in it. Once an array has been created, its capacity can't be changed.

On lines 15 through 17 I skip straight to the hard part: using a for loop to make a variable whose value changes into all the ***locations*** of the values in the array. That is, the variable *i* will equal `0` then `1` then `2` instead of the ***values*** in the array.

In the body of the loop we store a random number from 1-100 into each slot in the array. I'll explain more later.

On lines 21 through 23 we use an identical loop to display all the random values from the array onto the screen. If both of these loops don't make total sense to you, you're in luck! The rest of the code explains the technique.
So let's skip down to line 30, which uses square brackets and a number to store the *value* 6 into *slot* 0 of the array.

Maybe this seems weird. But the first slot in an array is slot number `0` in most programming languages. You could also say that the first slot in an array has the index 0, because the number that refers to an array slot is called an "index".

(Collectively these ought to be called "indices" (INNduh- SEEZ) but most people just say "indexes".)

So Go – like almost all other programming languages – has "0-based array indexes." This means that though our array has room for three values, they are numbered 0-1-2.

The first slot in an array is index `0`. This array can hold three values, so the last index is `2`. There is nothing you can do about this except get used to it. So `len()` is `3`, but there is not a slot with index `3`. This will probably be a source of bugs for you at first, but eventually you will learn.

Anyway, lines 29 through 31 store values into all three slots in the array. (All the slots used to have random numbers in them, but those have now been overwritten.)

On line 34 we print out all three current values in the array so you can see that they have been changed.

On lines 38 through 40 we put random numbers into each slot of the array. And print them out again on line 43.

Starting on line 47 I have done something silly. Try to withhold judgment until the end of the exercise.

Forgetting about why you might ***want*** to do it, do you see that line 48 is essentially identical to line 38? Line 48 stores a random number into a spot in the array. Which spot? The index depends on the current value of ***m***. And ***m*** is currently `0`. So we are storing the random number into the slot with index `0`. Okay?

So on line 49 we ***change*** the value of ***m*** from `0` to `1`. Then on line 50 we store a random value in the slot indexed by the value of ***m***, so index `1`. Clear? Weird, but legal.

I have used similar shenanigans on lines 55 through 62 to display all the values on the screen again. Now, this is clearly objectively worse than what I was doing on line 34. I mean, it took me 8 lines of code to do what I had been doing in one line. (Stay with me.)

On lines 66 through 72 we do something that might even be worse than lines 47 through 52. Lines 66 and 67 are the same, but instead of putting a `1` directly into ***n*** on line 68, I just say "increase the value of ***n*** by 1." So ***n*** had contained a `0`; it contains a `1` after that statement completes.

Pretty much the only advantage to this approach is that at least copy-and-paste is easier. Lines 69 and 70 are ***literally identical*** to lines 67 and 68. And the same for lines 71 and 72. I mean, like bytefor-byte the same. We display them in a similar silly fashion on lines 75 through 83.

But then maybe it occurs to you. "Why would I bother to type the exact same lines three times in a row when I could just…." You know a thing that allows you to

repeat a chunk of code while making a single variable increase by one each time, right?

That's right: a `for` loop is just the thing. Not so silly after all, am I?

Lines 87 through 89 are the same as lines 76 through 82 except that we let the `for` loop handle the repeating and the changing of the index. The initialization statement (the first part) of the `for` loop sets ***q*** to start at `0`, which happens to be the smallest legal index for an array. The condition says "repeat this as long as ***q*** is less than `arr.length` (which is `3`)." And note that it says ***less than***, not less than or equal to, which would be too far. The update statement (the third part) just adds 1 to ***q*** each time. Lines 92 through 96 display the values on the screen.

Here's the thing: this sort of code on lines 87 through 96 might seem a little bit complex, but working with arrays in Go you end up writing code like this ***all the time***. I cannot even tell you how many times I have written a `for` loop just like that for working with an array.

In fact, if your question is "How do I _____ an array?" (Fill in the blank with any task you like.) The answer is "With a `for` loop." Pretty much guaranteed.