Lesson 6

Controlling Program Flow with Loops

As you learned in the previous lesson, without specific instructions, a program executes statements linearly until the program ends or until an error causes the program to fail. You also saw how to change the flow of your programs based on conditions.

In this lesson, you will learn how to repeat a line of code or set of instructions. We will look at the use of looping to control the flow of a program during runtime as well as cover a few related items for adjusting the flow of a program from within a loop.

LEARNING OBJECTIVES

By the end of this lesson, you will be able to:

- Learn how to repeat sections of code.
- Explore the `for` keyword.

- Understand how to stop or exit a loop.
- Discover how to loop through a string.
- Explore the concept of a never-ending infinite loop.

LOOPING STATEMENTS

In every programming language, it is important to be able to repeat statements without having to write the statements over and over. For example, Listing 6.1 uses only what you've learned so far to print the values 1 to 10 using a single variable.

LISTING 6.1

Counting to 10 linearly

```
package main

import "fmt"

func main() {
   var ctr int = 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
   ctr += 1
   fmt.Println(ctr)
}
```

When you run this listing, you create a variable called `ctr` and assign it a starting value of 1. You then print the value. After printing, you add 1 to `ctr` and print its value again. You then add 1 to `ctr` and print it again. You continue to repeat this process until you've printed the numbers up to 10 as shown:

```
1
2
3
4
5
6
7
8
9
10
```

As you can see by looking at the listing, it is rather long and contains a lot of redundant code. If you were to change the listing to print to 100 instead of just 10, the code would grow to be roughly 10 times longer. Additionally, you'd have to keep track of how many times you add and print to determine when you get to 100.

There is, of course, a better way. That is the purpose of *looping statements*. In general, you will use the `for` statement to accomplish looping in your programs.

FOR LOOPS

In any programming or scripting language, it is important to be able to repeat a block of code several times based on some condition. This is done using repetition statements. Go supports only one repetition statement, which is the `for` loop.

The basic structure of a `for` loop is to use the `for` keyword followed by an initialization expression, a condition expression, and a post-argument expression:

Initialization Expression This is the starting point of your loop. It is executed once at the beginning of the loop.

Condition Expression This condition is tested every time the loop iterates. If the condition evaluates to false, then the loop will stop. If the condition evaluates to true, the loop will continue iterating.

Post Expression The post-expression is an expression that gets executed at the end of every iteration. It can contain an increment expression, which when executed can trigger the condition expression to evaluate to false and terminate the loop.

These are then followed by a block of code that will be repeated with the looping:

```
for initialization; condition; post_expression {
   // Instruction set
}
```

Listing 6.2 shows the basic structure of a `for` loop. This listing does the same task as the previous listing, but with the use of a loop.

LISTING 6.2

A basic `for` loop

```
package main

import "fmt"

func main() {
   for ctr := 1; ctr <= 10; ctr++ {
         fmt.Println(ctr)
   }
}
```

When you run this listing, we see that it indeed prints the same output as the previous listing, the numbers 1 to 10; however, it accomplishes this task with substantially fewer lines of code. If you change the value of 10 in the listing to 100, you see that the loop is repeated 100 times. Unlike Listing 6.1, we don't need additional lines of code to count to a higher number.

Let's look more closely at the code to see how it is accomplishing the looping and counting. In Listing 6.2, the `for` statement takes three arguments:

- In the first argument (the *`initialization`* argument), we initialize the `ctr` variable to 0. We use dynamic typing so that Go treats it as a number.
- In the second argument (the *`condition`*), we compare `ctr` to a fixed value, in this case, 10. If this statement is true, Go will execute the instruction set in the `for` loop.
- In the third argument (the *`post_expression`*), we increment the `ctr` by 1.

The *`initialization`* argument is only executed when the `for` loop initially starts. After the *`initialization`*, Go will check the *`condition`*. If at that time the *`condition`* is true, then the loop will begin, and the code in the body of the `for` statement will be executed. If the *`condition`* is not true at this time, then the body of the `for` statement will not be executed.

Assuming the *condition* was true, then after executing the body, Go will execute the *post_expression*. The *condition* is then checked again to see if it is still true. The looping will then continue executing the statement in the body of the `for` statement until the *condition* statement is no longer true. In the case of Listing 6.2, it will continue until `ctr` is no longer less than or equal to 10.

Because `for` loops are so important, it is worth showing a second example. Listing 6.3 is another basic `for` loop following the same structure as the previous listing. This time the code uses a loop and an `if` statement to print only even numbers.

LISTING 6.3

A basic `for` loop that prints even numbers

```
package main

// we import the strconv package which allows us to parse data
// between different data types
import (
   "fmt"
   "strconv"
)

func main() {
   for a := 0; a < 10; a++ {
      if (a % 2 == 0){
         // the Itoa function converts the int into its
         // equivalent string (UTF-8) value
         fmt.Println(strconv.Itoa(a) + " is an even number")
      }
   }
}
```

When you execute this listing, you see the following printed:

```
0 is an even number
2 is an even number
4 is an even number
6 is an even number
8 is an even number
```

Let's look more closely at the code. As noted in the comments, you use the `strconv` package here. This will allow you to compare a number variable to another number and then include the number value in a string statement. Specifically, you use the `Itoa` function, which converts an `int` into its corresponding UTF-8 code.

In this example, the `for` statement also takes three arguments:

- In the first argument, you initialize the variable (called `a`) to 0 again using dynamic typing so that Go treats it as a number.
- In the second argument, you compare the variable called `a` to the fixed value of 10. You check to see if `a` is less than 10. As long as this expression is true, Go will execute the instruction set in the `for` loop.
- In the third argument, you increment the variable `a` by 1.

You then use an `if` statement to determine if the variable's value is even. If so, Go will execute the instructions in the `if` block. If not, Go ends the `if` statement and returns to the `for` statement. Go stops looping through the `for` statement when the conditional statement `a < 10` is no longer true.

Optional Items in a *for* Loop

Both the initialization and post-argument expression are optional in a `for` loop. Either or both can be left off, as shown in Listing 6.4.

LISTING 6.4

Dropping the initialization and post-argument expression

```
package main

import (
   "fmt"
   "strconv"
)

func main() {
   var a int = 0

   for ; a < 10; {
      if (a % 2 == 0){
         fmt.Println(strconv.Itoa(a) + " is an even number")
      }

      a++ // we have to increment manually in this case
   }
}
```

The output from this listing matches that of the previous listing. The code in this listing is also very similar to the code in the previous listing, with the following differences:

- You initialize the variable called `a` as an `int` with the value 0 at the start of the program, rather than doing this in the `for` block.
- You increment the variable's value after completing the `if` block and before restarting the `for` block, rather than incrementing it before entering the `if` block.
- Note the use of the semicolon after the `for` keyword. This informs Go that the *`initialization`* statement is not included (because we initialized the variable earlier in the program).

Go's *while* Is *for*

In most other programming languages, there is a `while` loop that can be used that takes a condition and repeats the loop until the condition is true. Go does not use the `while` keyword; however, we can perform the same type of loop using `for`.

As you saw in the previous listing, the only argument a `for` statement must include is the conditional statement. You had to include a semicolon in the previous example only because you put a semicolon after the conditional statement. However, you can create something akin to a `while` statement by removing the semicolons completely, as shown in Listing 6.5.

LISTING 6.5

Go's equivalent of a while loop

```
package main

import (
   "fmt"
   "strconv"
)

func main() {
   var a int = 0

   for a < 10 {   // remove the semicolons here
      if (a % 2 == 0){
```

continues

continued

```
         fmt.Println(strconv.Itoa(a) + " is an even number")
         }

      a++
   }
}
```

This listing operates exactly like the previous listing. This includes displaying the same output:

```
0 is an even number
2 is an even number
4 is an even number
6 is an even number
8 is an even number
```

Let's look at another example using `for` to create a `while` loop. The code in Listing 6.6 calculates the powers of 2 for numbers between 1 and 10.

LISTING 6.6

Powers of 2

```
package main

import (
   "fmt"
   "strconv"
)

func main() {
   var power2 int64 = 1
   var a int64 = 1
   for a < 10 {
      fmt.Println("2 to the power of " + strconv.FormatInt(a,10) +
                  " is equal to " + strconv.FormatInt(power2,10))
      power2 += power2
      a++
   }
}
```

When you execute this listing, the following output is displayed:

```
2 to the power of 1 is equal to 1
2 to the power of 2 is equal to 2
2 to the power of 3 is equal to 4
2 to the power of 4 is equal to 8
2 to the power of 5 is equal to 16
2 to the power of 6 is equal to 32
2 to the power of 7 is equal to 64
2 to the power of 8 is equal to 128
2 to the power of 9 is equal to 256
```

Of note here is that we again use the `for` keyword and include only a comparison statement. The variables are initialized before the `for` loop and incremented at the end of the `for` loop. In this case, we use `int64` instead of `int` for the variables. This means we must use the `FormatInt` function to convert the number to a string for the output statements:

```
strconv.FormatInt(power2,10)
```

The `FormatInt` function takes the value you want to convert to an integer and the base of that number. Since you are working with standard numbers, the base is base 10. You specify that the number is base 10 in the second parameter.

NOTE When using `for` loops, it's easy to get carried away and overflow the variable. Update the code in Listing 6.6 to use `a < 100` instead of `a < 10`. Run the program to see what happens.

Infinite Loops

A potential problem that can happen when looping is the creation of an infinite loop: a `for` statement whose conditional statement can never be true. Take a look at Listing 6.7. Can you identify the problem in the code?

LISTING 6.7

Code with a problem

```
package main

import (
```

continues

continued

```
   "fmt"
   "strconv"
)

func main() {
   var power2 int64 = 1
   var a int64 = 1

   for {
      fmt.Println("2 to the power of " + strconv.FormatInt(a,10) +
                  " is equal to " + strconv.FormatInt(power2,10))

      power2 += power2
      a++
   }
}
```

Create the program in Listing 6.7 and run it to see what happens. You will find that it does, indeed, run without a compiler error. In fact, it will run, and run, and run….

NOTE You can stop an executing program using Ctrl+C.

If you hadn't noticed, there is no condition in the code to stop the loop, so the `for` loop will continue running. It is imperative that if you don't want the loop to be infinite, you include a condition that will end the loop.

LOOPING THROUGH A STRING

There are a multitude of uses for loops. One use is to loop through a string and remembering that a string is really a set of individual characters. The example in Listing 6.8 shows the most basic way to iterate through a string one byte/character at a time.

LISTING 6.8

Looping through a string one character at a time

```
package main

import "fmt"

func main() {
    var message string = "HELLO WORLD"

    fmt.Println(message)

    for idx := 0; idx < len(message); idx ++ {
       fmt.Println(string(message[idx]))
    }
}
```

In this example, you treat the message as a collection of characters, using the length of the collection as the stopping point for the loop. The start of the string (the first character) is in position 0, and the last character of the string is in the position that is 1 less than the length of the string.

You get the length of the string by using the `len` method. In this case, `len(message)` will return the length of your message string, which is 11. The space also counts as a character.

To get to each letter, you use the offset (which is an index value) within square brackets after the name of the string:

```
message[idx]
```

In this case, `message[0]` would be the first character, which is H. The second character (E) would be in `message[1]`. Using the `for` statement, the listing loops through the index values to retrieve and display each of the individual letters. The result is the printing of the characters in the `message` variable. Because we are using `Println` to print, each character is displayed on its own line:

```
H
E
L
L
O

W
O
R
L
D
```

NOTE In Lesson 9, "Accessing Arrays," we present this topic in more detail as we cover arrays and indexing.

THE *RANGE* FUNCTION

You can also leverage the `range` keyword to create indexes on a string so that you can iterate through it. Listing 6.9 gives a brief example of using `range`. We'll cover this keyword in more detail in Lesson 9, when arrays are presented.

In Listing 6.9, we convert the message into a range and then iterate through the message, where we print each index value and each character in the range.

LISTING 6.9

Using `range`

```
package main

import "fmt"

func main() {
   var message string = "HELLO WORLD"
   fmt.Println(message)

   for idx, c := range message {
      fmt.Println(idx) //index
      fmt.Println(string(c)) //value
   }
}
```

In this listing, you again create a string variable called `message` and assign it the text "HELLO WORLD". You print the value of `message` so that it can be seen.

You then use the range version of a `for` loop:

```
for idx, c := range message {
```

The first expression in the `for` loop is an index value that will be used in looping. The second expression is assigning a value to a variable (in this case called `c`). What is being assigned is the value within `message` at the location of the index (`idx`). In this case, it will be the character at that index location within the string.

There are two `Println` statements in the loop. The first one prints out the index value (`idx`). This will be the counter for the loop. The second line prints the value that is at the offset at that location. Because `Println` will treat a character as a numeric value, you use the `string` function to convert the character to a string before printing. When you run this listing, you will see each index value printed followed by the character:

```
HELLO WORLD
0
H
1
E
2
L
3
L
4
O
5
 
6
W
7
O
8
R
9
L
10
D
```

Note that if you comment out the first `Println`, then the output will match the previous listing.

LOOP CONTROL STATEMENTS

Go supports three different statements that allow you to deviate the flow of execution within a loop. For instance, you might want to search for a word in a list of words. Once you find the first occurrence, you do not want to continue iterating through the list. Rather, you might simply want to stop the loop and display that the word was found. There is no need to continue the loop and the search once the word has been found. The three options Go provides are:

- `break` statements
- `continue` statements
- `goto` statements

Let's look at an example of each.

The *break* Statement

Go provides the `break` keyword to end a loop. Once a `break` statement is encountered, program flow goes to the first statement after the loop. In the example in Listing 6.10, you will use the powers of 2 program again.

LISTING 6.10

Using `break`

```
package main

import (
   "fmt"
   "strconv"
)

func main() {
   var power2 int64  = 1
   var  a int64 = 1

   for {
      if (a >= 10){
         break // exit the loop when we reach 10
      }

      fmt.Println("2 to the power of " + strconv.FormatInt(a,10) +
                  " is equal to "+ strconv.FormatInt(power2,10))

      power2 += power2
      a++
   }
}
```

In this version, you use a `break` statement to stop the `for` loop when the value of `a` reaches 10. Other than the `if` statement that checks to see if `a` is greater than 10, this listing is just like Listing 6.6. The output is as follows:

```
2 to the power of 1 is equal to 1
2 to the power of 2 is equal to 2
2 to the power of 3 is equal to 4
2 to the power of 4 is equal to 8
2 to the power of 5 is equal to 16
```

```
2 to the power of 6 is equal to 32
2 to the power of 7 is equal to 64
2 to the power of 8 is equal to 128
2 to the power of 9 is equal to 256
```

The *continue* Statement

The `continue` keyword is used to immediately return control back to the beginning of a loop. This doesn't restart the loop from the beginning, but rather starts the next iteration of the loop. Listing 6.11 prints odd numbers between 0 and 10 by using `continue` to tell Go to skip even numbers.

LISTING 6.11

Using `continue` to print odd numbers

```
package main

import (
   "fmt"
   "strconv"
)

func main() {
   for ctr := 0; ctr < 10; ctr ++{

      if (ctr % 2 == 0){
         continue // continue to next iteration; i.e., ignore even values
      }

      fmt.Println(strconv.Itoa(ctr) + " is an odd number")

   }
}
```

Looking at the code, you can see that a simple `for` loop is used. `ctr` is set to 0 at the beginning of the loop and incremented with each iteration until `ctr` is no longer less than 10. Within the body of the `for` statement, the value of `ctr` is checked to see if it is even. If dividing `ctr` by 2 returns a value of 0, then you know it is even, so the `continue` statement is called to immediately start the next iteration of the loop. When the `if` statement fails, then the rest of the statements in the body of the `for` loop are executed, which in this case is just the print statement to print the number. The final output is as follows:

```
1 is an odd number
3 is an odd number
5 is an odd number
7 is an odd number
9 is an odd number
```

The *goto* Statement

Another loop control statement supported by Go is `goto`. The `goto` keyword sends the program flow to a different location identified by a label. This jump is done without any conditions. While a `goto` statement can be used anywhere, Listing 6.12 presents an example within an `if` statement.

LISTING 6.12

Using goto

```
package main

import (
    "fmt"
    "strconv"
  )

func main() {
   var a int = 20
   var b int = 30
   fmt.Println("a = " + strconv.Itoa(a))
   fmt.Println("b = " + strconv.Itoa(b))

   if (a > b){
      goto MESSAGE1 //this will jump the execution to where MESSAGE1 is defined
   } else {
      goto MESSAGE2
   }
```

```
    MESSAGE1: // We define a label that we can use in a goto statement
        fmt.Println("a is greater than b")

    MESSAGE2:
        fmt.Println("b is greater than a")
}
```

In this example, you use two goto statements. Each statement references a different labeled code block. You then use if-else to define which code block should run, depending on whether the initial condition is true or false. In this case, when the program runs, a is not greater than b, so the else statement executes, which uses a goto statement to send the program flow to MESSAGE2 where a message is printed:

```
b is greater than a
```

Although goto statements can be useful, they should generally be avoided. If they are not used properly, they can cause issues with the flow of the program, and because the executed code is separate from the loop itself, troubleshooting is more complicated.

> **NOTE** If you change the code in Listing 6.12 so that it checks for `a < b` instead of `a > b`, then the goto statement will go to MESSAGE1 instead of MESSAGE2. In this case, you will see that both messages will be printed, which might not be what you would expect. Such possibly unexpected results are a reason the goto statement is generally avoided by programmers.

SUMMARY

We've now covered the primary program control keywords in Go. In the previous lesson, you learned how to control flow based on conditions. In this lesson, you expanded your knowledge to understand how to repeat lines of code via looping using for statements.

You also went further by learning additional keywords that allow for controlling loops, which include the continue and break commands. You also learned about the goto keyword that lets you jump to a new location unconditionally. Of course, goto should be used with caution as it can lead to hard-to-find errors in code.

EXERCISES

The following exercises are provided to allow you to experiment with the tools and concepts presented in this lesson. For each exercise, write a program that meets the specified requirements and verify that the program runs as expected. The exercises are:

Exercise 6.1: The Alphabet

Exercise 6.2: Sum It Up

Exercise 6.3: Fifty

Exercise 6.4: Numeric Breakout

Exercise 6.5: Reverse It

Exercise 6.6: Length Without `len`

Exercise 6.7: Guessing Game

Exercise 6.8: URL Shortener

Exercise 6.9: Validating Phone Numbers

Exercise 6.10: Validating Email Addresses

Exercise 6.11: Fizz Buzz

NOTE The exercises are for your benefit. The exercises help you apply what you learn in the lessons. You are also encouraged to experiment with the code as you complete the exercises.

You should notice that this lesson has more exercises than previous lessons. It is important to understand the Go keywords you've learned so far along with those that control program flow using conditions and loops. Using what you've learned so far, you can do quite a bit. These exercises not only help demonstrate and confirm your learning, but also show you some of the things you can already do with what you've learned in six lessons. Of course, there is still a lot to learn.

Exercise 6.1: The Alphabet

Using `for` loops and string functions, create a computer program that displays the alphabet from A to Z.

Exercise 6.2: Sum It Up

Create a program that computes the sum of all numbers between 0 and 100.

Exercise 6.3: Fifty

Write two programs, each of which displays all numbers divisible by 50 between 100 and 1,000 (inclusive). Both programs should have identical output.

- Use the `range` keyword with `for` in one program.
- Use `for` without `range` in the other program.

Exercise 6.4: Numeric Breakout

Create a program that prompts the user for an integer and then displays the following information about that number:

- The number of digits in the value entered
- The first and last digits of the number
- The sum of the digits in the number
- The product of the digits in the number
- Whether or not the number is prime
- The factorial of the number

Exercise 6.5: Reverse It

Write a program that asks the user for an input integer and then computes the reverse of that number. For example:

- Input: 12456
- Output: 65421

Exercise 6.6: Length Without *len*

Write a program that computes the length of a string without using the `len` function.

Exercise 6.7: Guessing Game

Write a program that generates a random integer between 0 and 10 and asks the user to guess what the number is.

- If the user's guess is higher than the random number, the program should display "Too high, try again."
- If the user's guess is lower than the random number, the program should display "Too low, try again."
- If the user's number is the same as the random number, the program should display "That's right. You guessed the number!"

The program should use a loop that repeats until the user correctly guesses the random number.

Exercise 6.8: URL Shortener

Create a program that mimics a URL shortener. The program will take a URL as input from the user, and it will return a short version of the URL. For example, if the user inputs the URL

```
http://www.thisisalongurl.com/somedirectory/somepage
```

The program will generate a short URL, like this:

```
http://surl.com/se04
```

The requirements for this program are the following:

- Use your own imaginary domain but keep it short.
- Generate a four-character identifier for the page:
  - Use the first and last character of the original page name.
  - Assign a random two-digit number.
- Include comments to explain the logic of how to shorten the URL.

Exercise 6.9: Validating Phone Numbers

Create a program that validates a phone number. The program should satisfy the following requirements:

- The program should accept a string that the user inputs as a phone number.
- It should check if the input value is appropriate for a U.S. phone number.

    - If it is not valid, it should display a message with information about the problem it found.
    - If it is valid, it should display the phone number in a normalized format, using the representation *999-999-9999*.

Exercise 6.10: Validating Email Addresses

Create a program that validates an email address. The program should satisfy the following requirements:

- The program should accept an email address as string input.
- The program should check that the address is formatted as a valid email address and provide appropriate feedback.
- After accepting the email, the program should output the following (with appropriate output messages):
    - The domain of the email address.
    - The identifier of the email address.

For example, given the input `someperson@somedomain.com`, the output should look something like:

```
Valid format: True
Domain: somedomain
Identifier: someperson
```

Challenge

Set up the program so that if the user enters an invalid string as an email address, the program informs the user of the problem and prompts them to reenter the address before continuing.

Exercise 6.11: Fizz Buzz

Write a program that loops through a series of values and uses those values to determine the output shown to the user. The program should perform the following steps:

1. Ask the user for a number.
2. Output a count starting with 0.
    - Display the count number if it is not divisible by 3 or 5.
    - Replace every multiple of 3 with the word "fizz."

- Replace every multiple of 5 with the word "buzz."
- Replace multiples of both 3 and 5 with "fizz buzz."

3. Continue counting until the number of integers replaced with "fizz," "buzz," or "fizz buzz" reaches the input number.
4. The last output line should read "TRADITION!!"

For example:

```
How many fizzing and buzzing units do you need in your life? 7
0
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizz buzz
TRADITION!!
```