Module Three

Control Stuctures

Most of the appeal for me is not the features that Go has, but rather the features that have been intentionally left out
txxxxd in Hacker News

I like that Go forces you to clean up little messes, like unused variables, for example.
Jamu Kakar

3.1 Introduction

This module is not deep in terms of introducing new concepts, but it does introduce a lot of detail at the nuts and bolts level of writing Go code. This module is really about getting comfortable with the general appearance, flow and feel of writing Go code in the places where it differs from other C-style languages.

I don't know if my personal experience is commonly shared by other programmers, but there are two things I find frustrating when starting to learn a new programming new language.

The first is that my code doesn't doesn't compile or execute but to my programmer's eye it should because I don't see anything obviously wrong with it. This is because I have missed some critical difference between how I normally write code and how code should be written in the new language. Of course the difference is usually so minor that it does not warrant mention in a lot of the material on the language, but it is not obvious enough to me to pick it up right away – especially since I am conditioned to look at the code in terms of other programming languages I already know.

I have been writing C, Java and C++code for decades and will often write a variable declaration and “int i” in Go because that is what I automatically do, and I then wonder for just a moment what the compiler is complaining about until I remember..oh yeah this is Go and that is not how we do it in Go.

The second thing I find frustrating is when low level details are just not mentioned in most of the material on a language because the people who write in the language regularly as either blindingly obvious or too trivial to mention. That may be true if you know the language, and after I learn the language I realize it is in fact trivial, but if you don't know the language then it's not obvious initially; but also almost impossible to find any clarification on that specific topic or point.

This section just focuses on the aspects of Go control structures that are important to point out in terms of transitioning from writing code in another language to writing code in Go. There is no exposition in this module on what conditionals are, or how they work, or what a for loop does – the focus is only how a Go conditional is written differently from conditionals in C, C++ and Java.

3.2 Mixed Mode Operations

The last thing we want to do in this course is to start going over all the operators. Given that you are an experienced programmers, then only teaching points we have to make are about the differences between operators in Go and other C-style languages – there are only a few but, based on my own personal experience, they are significant enough that they really confused me the few times I encountered them.

The most minor difference, in my opinion, is that there is no exponentiation operator. This is not a real loss since there is an exponentiation function in the math package that provides exactly the same functionality.

3.2.1 Increment Operator

The main difference that takes getting used to, or at least it was for me because I was so used to using it in other languages, was that the increment operator "i++" is actually a statement and not an expression. That means that it cannot appear on the right hand side of any sort of assignment operation.

An expression is something that can be evaluated to produce a result which can then use in some way, such as assigning it to a variable or as a value in another expression.

In Go `x++` is a short form of the statement `x = x +1` which does not return a result. A common construct I would use in C++ is j=i++ would be interpreted in Go as `j = i = i+1` which is an illegal statement in Go. In fact Go does not support this chaining of operations where more than one “=” appears in a line.

The operation i++ does not produce a result, it just has a side effect which is to change the value of the variable i. Since i++ is not an expression, the need to distinguish between incrementing before or after we use it is no longer an issue. These means that ++i and i++ are semantically identical which makes the prefix operator is redundant so Go drops its usage and only supports the postfix form.

3.2.2 No Mixed Mode Arithmetic

The mixed mode arithmetic operations rule is something that just takes a little getting used to but more because of the strictness of the rule than because of any conceptual complexity. It's easy to remember that you will get a compiler error when you add a int32 and a float32 but it's more difficult to remember that you can't add a float32 and a float64 either.

The rationale for the no mixed mode rule actually makes a lot of sense. In the section in Module 2 on variable type conversions, I presented a quote by Rob Pike about why this decision was made. It is relevant here as well.

This is illustrated in example 03-01.

```go
// Example 03-01 Operators

package main

import "fmt"

func main() {

  a, b, c := uint8(1), uint16(1), int8(1)
  //fmt.Println("1. a + b =", a+b)
  //fmt.Println("2. a + c =", a+c)
  fmt.Println("3. a + uint8(b)=", a+uint8(b))
  fmt.Println("4. uint8(a) + c=", int8(a)+c))
}
```

```
[Module03]$ go run ex03-01.go
3. a + uint8(b)= 2
4. uint8(a) + c= 2
```

3.2.3 Parallel Assignment

Parallel multiple assignment is not supported in other C-style languages so those languages tend use the the more cumbersome C-style way of using a temporary variable to swap two values. Go does all assignments in parallel as shown in example 03-02

```
// Example 03-02 parallel Assignment

package main

import "fmt"

func main() {

  first, last := "York", "New"
  fmt.Println(first, last)
  first, last = last, first
  fmt.Println(first, last)
}
```

```
[Module03]$ go run ex03-02.go
York New
New York
```

3.3 Conditionals – the if Statement

Conditionals in Go work for the most part like conditionals in other C-style languages except for the following differences:

1. No parentheses "(..)" are allowed around the test condition
2. Local variables can be defined in the if statement itself
3. Braces "{ .. }" are mandatory for all then and else blocks
4. The opening "{" for each block cannot start on a new line
5. The else keyword cannot appear at the start of a new line

Most of these differences are syntactic which is not a conceptually difficult stretch to make. The major difference between Go and other C-style conditionals is the way variables that are local to the conditional are defined.

```
// Example 03-03 Basic If Statement

package main

import "fmt"

func main() {
  x := 22
  if x == 0 {
       fmt.Printf("%d is zero\n", x)
  } else if x%2 == 0 {
       fmt.Printf("%d is even\n", x)
  } else {
       fmt.Printf("%d is odd\n", x)
  }
}
```

```
[Module03]$ go run ex03-03.go
22 is even
```

3.2.1 Local Variables in Conditionals

We can do the same in Go as in other C-style languages and define local variables in each of the blocks that make up the clauses of the if statement – but doing it this way means that variable is defined only for that block and so has to be redefined for each block that corresponds to a clause in the conditional.

However Go allows the definition of local variables at the start of the if statement which are then local to *all* of the clauses of the if statement. Since only one of the clauses will ever execute (the whole point of a conditional statement) then any interaction between these local variables across the clauses is impossible.

We can only use the short form of variable declarations here. Earlier in the class we said that we would see a reason for having the two forms of variable declaration – well here it is.

```
// Example 03-04 Local Variables

package main

import "fmt"

func main() {

  if x, y := 22, "hi"; x == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else if x % 2 == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else {
       fmt.Println("Value of x=", x, " y=", y)
  }
}
```

```
[Module03]$ go run ex03-04.go
Value of x= 22  y= hi
```

3.2.2 Using Non-local variables

We can also assign values to existing variables using the “=” operator instead of the “:=” operator.

However we cannot mix the local and non-local or the local definition will shadow the the non-local definition. This will be explored as one of the lab exercises.

```
// Example 03-05 Local and Non-Local Variables

package main

import "fmt"

var x int = 10

func main() {

  if x, y := 22, "hi"; x == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else if x % 2 == 0 {
       fmt.Println("Value of x=", x, " y=", y)
  } else {
       fmt.Println("Value of x=", x, " y=", y)
  }
  fmt.Println("Value of x=", x)
}
```

```
[Module03]$ go run ex03-05.go
Value of x= 22  y= hi
Value of x= 10
```

3.4 Loops – Differences in Go

The only loop construct in Go is the for loop. The major differences between Go for loops and other C-style language are similar to those for conditionals.

1. No parentheses "(..)" allowed in the for clause
2. Braces "{..}" are mandatory for the loop body
3. The pre and post terms in the for clause can be empty

The last point means that a for loop can look like:

```
for  ; text==true ; {...}
```

which can be also written as if it were a while loop

```
for text==true { ...}
```

Like the if statement in Go, the opening brace "{" of the loop body must appear on the same line as the for clause.

Also notice in the example we cannot define "total" in the for clause because that would make it local to the loop body.

```go
// Example 03-06 Basic for loop

package main

import "fmt"

func main() {
  var total = 0
  for count := 0; count < 100; count++ {
       total += count
  }
  fmt.Println("total = ", total)
}
```

```
[Module03]$ go run ex03-06.go
total =  4950
```

3.4.1 Multiple Loop Variables

Example 03-07 shows how we can define multiple loop variables in the for loop. The example also illustrates the use of the break statement – the break and continue statements work the same way in Go as they do other C-style languages.

```
// Example 03-07 Multiple Declarations

package main

import "fmt"

func main() {
  var total = 0
  for i, m := 0, "abort"; i < 100; i++ {
       total += i
       if total > 100 {
            fmt.Println(m)
            break
       }
  }
  fmt.Println("total = ", total)
}
```

```
[Module03]$ go run ex03-07.go
abort
total =  105
```

3.4.2 Non-local Variables

Just a reworking of the previous example to show how non-local variables are used in the loop. The dangers of mixing the two types – local and non-local – are explored in the lab. The example also shows the use of the continue statement, used the same as in other C-style languages.

```
// Example 03-08 Non Local Variables

package main

import "fmt"

func main() {
  var total, i int = 1000, 1000
  for i, total = 0, 0; i < 100; i++ {
       if total > 200 {
            continue
       } else {
            total += i
       }
  }
  fmt.Println("total = ", total)
}
```

```
[Module03]$ go run ex03-08.go
total =  210
```

3.4.3 For Loop as a While Loop

A while loop in C-style languages is equivalent to a for loop with an empty clause usually written and "for(;;)" In Go we can do that as well, and by dropping off the extraneous semicolons we get something that looks just like a while loop.

```
// Example 03-09 For loop as a while loop

package main

import "fmt"

func main() {

  count := 0

  for count < 2 {
       count++
  }

  fmt.Println("count = ", count)
}
```

```
[Module03]$ go run ex03-09.go
count =  2
```

3.4.4 Looping Using Ranges

This loop form is functionally equivalent to a for-each loop. In the example, we are using the range function, which we will see a lot in this course, to iterate over a string.

The example also demonstrates the multiple return value feature of Go functions. During each iteration, the range function returns both the current index as an integer and the current contents at that index, which in this example is the letter at that position.

```
// Example 03-10 Looping with range

package main

import "fmt"

func main() {

  test := "Hi!"

  for index, letter := range test {
       fmt.Printf("Letter %d is %#U\n", index, letter)
  }
}
```

```
[Module03]$ go run ex03-10.go
Letter 0 is U+0048 'H'
Letter 1 is U+0069 'i'
Letter 2 is U+0021 '!'
```

3.5 Switch Statements

One small change which I think makes the Go version of the switch statement cleaner than in other languages is revising the default behavior of the case construct so that the flow of control breaks by default at the end of the case rather than falling through to the next case. In Go, the fallthrough statement is now required to fall through to the next case rather than needing to use the break statement to prevent a fall through. The break statement is still available but now it is used only when you want to break prematurely out of a case instead of executing the whole case.

In C-style languages, a test value must be provided and it usually must be some sort of integral type. In Go both of these requirements are dropped which, again in my opinion, make the switch in Go more powerful. One of the more useful roles of the switch statement that we will explore in a later module is the ability to switch on the basis of the type of variable.

In the example things to note are the lack of break statements and the use of the fallthrough statement to get the default clause to execute right after the '1' case when the switch test value is 1.

```
// Example 03-11 Simple Switch

package main

import "fmt"

func main() {

  for i := 0; i < 3; i++ {
       switch i {
       case 0:
            fmt.Println("Case 0")
       case 1:
            fmt.Println("Case 1")
            fallthrough
       default:
            fmt.Println("Default")
       }
  }
}
```

```
[Module03]$ go run ex03-11.go
Case 0
Case 1
Default
Default
```

3.5.1 Break Statements

A sort of contrived example to show the use of the break statement.

```
// Example 03-12 Switch Statement Break

package main

import "fmt"

func main() {

  for i := 0; i < 2; i++ {
       switch i {
       case 0:
            fmt.Println("Case 0")
       case 1:
            fmt.Println("Case 1")
            break
            fmt.Println("After break")
       default:
            fmt.Println("Default")
       }
  }
}
```

```
[Module03]$ go run ex03-12.go
Case 0
Case 1
```

3.5.2 Non-integral Test Value

Simple example where the test value is a string rather than a numeric value. Obviously the restriction is that whatever type the switch test value is, it must be something that has the notion of equality defined. In this example using more than one matching value for a test case, a feature of other C-style switch statements, is also demonstrated.

```
// Example 03-13 Non-integral test

package main

import "fmt"

func main() {

  os := "fedora"
  switch os {
  case "fedora", "redhat":
       fmt.Println("Open Source")
  case "Windows":
       fmt.Println("Proprietary")
  default:
       fmt.Println("unknown")
  }
}
```

```
[Module03]$ go run ex03-12.go
Open Source
```

3.5.3 Switch – No Test Value

In this variant, the cases are evaluated as usual from start to finish. However each test case now has to be a predicate (ie. an expression that evaluates to true or false). The first test case that evaluates to true is the case that is executed.

The developers of Go note that the switch statement in this form is equivalent to a series of else-if statements. Aside from the fact it might be an easier way to write complex switch logic, this would seem to be useful when the test value is not a variable but a condition or combination of conditions that cannot be represented as a single variable of a specific type.

```
// Example 03-14 Switch with no test value

package main

import "fmt"

func main() {
  x := 22

  switch {
  case x == 0:
       fmt.Println("zero")
  case x % 2 == 0:
       fmt.Println("even")
  default:
       fmt.Println("odd")
  }
}
```

```
[Module03]$ go run ex03-14.go
a=int b=int c=int
a=0 b=2 c=4
```