Programming in Go

Matt Holiday
Christmas 2020

Control Structures

Sequence

The simplest type of program has no “control structures”

It just flows from top to bottom (sequential execution)

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

func main() {
    a, b, c := -0.5, 0.5, 5.0
    x := math.Sqrt(b*b - 4*a*c) / (2 * a)
    y1, y2 := -b + x, -b - x

    fmt.Printf("%5.4f, %5.4f\n", y1, y2)   // -3.7016, 2.7016
}
```

If-then-else

The next type of structure is a choice between alternatives

All if-then statements require braces

```
if a == b {
    fmt.Println("a equals b")
} else {
    fmt.Println("a is not equal to b")
}
```

They can start with a short declaration or statement

```
if err := doSomething(); err != nil {
    return err
}
```

For loops

The loop control structure provides automatic repetition

There is only `for` (no `do` or `while`) but with options

1. Explicit control with an index variable

```
for i := 0; i < 10; i++ {
    fmt.Printf("(%d, %d)\n", i, i*i)
}

// prints (0, 0) up to (9, 81)
```

Three parts, all optional (initialize, check, increment)

The loop ends when the explict check fails (e.g., `i == 10`)

For loops

2a. Implicit control through the `range` operator for arrays & slices

```
// one var: i is an index 0, 1, 2, ...

for i := range myArray {
    fmt.Println(i, myArray[i])
}

// two vars: i is the index, v is a value

for i, v := range myArray {
    fmt.Println(i, v)
}
```

The loop ends when the range is exhausted

For loops

2b. Implicit control through the `range` operator for maps

```
// one var: k is key

for k := range myMap {
    fmt.Println(k, myMap[k])
}

// two vars: k is the key, v is a value

for k, v := range myMap {
    fmt.Println(k, v)
}
```

The loop ends when the range is exhausted

For loops

3. An infinite loop with an explicit `break`

```
i, j := 0, 3

// this loop must be made to stop

for {
    i, j = i + 50, j * j

    fmt.Println(i, j)

    if j > i {
        break        // when i = 150, j = 6561
    }
}
```

There is also `continue` to make an iteration start over

For loops

Here's a **common mistake**

If you only want `range` values, you need the blank identifier:

```
// two vars: _ is the index (ignored),
//           v is the value

for _, v := range myArray {
    fmt.Println(v)
}
```

Sometimes you may not get a compile error for a type mismatch if you use only the one-var format (a slice of `int`s!)

The _ is an untyped, reusable "variable" placeholder

Labels and loops

Sometimes we need to break or continue the outer loop
(nested loop for quadratic search)

```
outer:
    for k := range testItemsMap {           // keys
        for _, v := range returnedData {    // values in list
            if k == v.ID {                  // found it!
                continue outer
            }
        }

        t.Errorf("key not found: %s", k)
    }
}
```

We need a label to refer to the outer loop

Switch

A switch is another choice between alternatives

It is a shortcut replacing a series of if-then statements

```
switch a := f.Get(); a {
case 0, 1, 2:
    fmt.Println("underflow possible")

case 3, 4, 5, 6, 7, 8:

default:
    fmt.Println("warning: overload")
}
```

Alternatives may be empty and **do not fall through** (`break` is not required)

Switch on true

Arbitrary comparisons may be made for an switch with no argument

```
a := f.Get()

switch {
case a <= 2:
    fmt.Println("underflow possible")

case a <= 8:
    // evaluated in order

default:
    fmt.Println("warning: overload")
}
```

Packages

Everything lives in a package

Every standalone program has a `main` package

```
package main

import "fmt"

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

Nothing is “global”; it’s either in your package or in another

It’s either at **package** scope or **function** scope

Package-level declarations

You can declare anything at *package* scope

```
package secrets

const DefaultUUID = "00000000-0000-0000-0000-000000000000"
var secretKey string

type k8secret struct {
    . . .
}

func Do(it string) error {
   . . .
}
```

But you can't use the short declaration operator `:=`

Packages control visibility

Every name that's **capitalized** is exported

```
package secrets

import . . .

type internal struct {
    . . .
}

func GetAll(space, name string) (map[string]string, error) {
    . . .
}
```

That means another package in the program can import it
(within a package, *everything* is visible even across files)

Imports

Each *source file* in your package must import what it needs

```
package secrets

import (
    "encoding/base64"
    "encoding/json"
    "fmt"
    "os"
    "strings"
)
```

It may only import what it needs; unused imports are an error

Generally, files of the same package live together in a directory

No cycles

A package “A” cannot import a package that imports A

```
package A

import "B"

//------------------

package B

import "A"    // WRONG
```

Move common dependencies to a third package, or eliminate them

Initialization

Items within a package get initialized before `main`

```
const A = 1

var B int = C
var C int = A

func Do() error {
    . . .
}

func init() {
    . . .
}
```

Only the runtime can call `init`, also before `main`

What makes a good package?

A package should embed deep functionality behind a simple API

```
package os

func Create(name string) (*File, error)
func Open(name string) (*File, error)

func (f *File) Read(b []byte) (n int, err error)
func (f *File) Write(b []byte) (n int, err error)
func (f *File) Close() error
```

The Unix file API is perhaps the best example of this model

Roughly five functions hide a lot of complexity from the user

Declarations & Compatibility

Declaration

There are six ways to introduce a name:

- Constant declaration with `const`
- Type declaration with `type`
- Variable declaration with `var`
  (must have type or initial value, sometimes both)
- Short, initialized variable declaration of any type `:=`
  only inside a function
- Function declaration with `func`
  (methods may *only* be declared at package level)
- Formal parameters and named returns of a function

Variable declarations

There are several ways to write a variable declaration:

```
var a int         // 0 by default

var b int = 1

var c = 1         // int

var d = 1.0       // float64


var (
    x, y int
    z    float64
    s    string
)
```

Short declarations

The short declaration operator `:=` has some rules:

1. It can't be used outside of a function
2. It must be used (instead of `var`) in a control statement (`if`, etc.)
3. It must declare at least one *new* variable

```
err := doSomething();

err := doSomethingElse();   // WRONG

x, err := getSomeValue();   // OK; err is not redeclared
```

4. It won't re-use an existing declaration from an outer scope

Shadowing short declarations

Short declarations with `:=` have some gotchas

```
func main() {
    n, err := fmt.Println("Hello, playground")

    if _, err := fmt.Println(n); err != nil {
        fmt.Println(err)
    }
}
```

Compile error: the first `err` is unused

This follows from the scoping rules, because `:=` is a declaration and the second `err` is in the scope of the `if` statement

Shadowing short declarations

Short declarations with `:=` have some gotchas

```
func BadRead(f *os.File, buf []byte) error {
    var err error

    for {
        n, err := f.Read(buf) // shadows 'err' above

        if err != nil {
            break             // causes return of WRONG value
        }

        foo(buf)
    }

    return err // will always be nil
}
```

Structural typing

It's the same type if it has the same *structure or behavior*

```
a := [...]int{1, 2, 3}
b := [3]int{}

a = b           // OK

c := [4]int{}

a = c           // NOT OK
```

Go uses *structural* typing in most cases

Structural typing

It’s the same type if it has the same structure or behavior:

- arrays of the same size and base type
- slices with the same base type
- maps of the same key and value types
- structs with the same sequence of field names/types
- functions with the same parameter & return types

Named typing

It's the only the same type if it has the same *declared type name*

```
type x int

func main() {
    var a x      // x is a defined type; base int

    b := 12      // b defaults to int

    a = b        // TYPE MISMATCH

    a = 12       // OK, untyped literal
    a = x(b)     // OK, type conversion
}
```

Go uses *named* typing for non-function *user-declared* types

Numeric literals

Go keeps “arbitrary” precision for literal values (256 bits or more)

- Integer literals are untyped
  - assign a literal to any size integer without conversion
  - assign an integer literal to float, complex also
- Ditto float and complex; picked by syntax of the literal
  `2.0` or `2e9` or `2.0i` or `2i3`
- Mathematical constants can be very precise

```
Pi  = 3.14159265358979323846264338327950288419716939937510582097494459
```

- Constant arithmetic done at compile time doesn’t lose precision

Operators

Basic operators

Arithmetic: numbers only except + on string

```
+    -    *    /    %    ++      --
```

Comparison: only numbers/strings support order

```
==   !=   <    >    <=   >=
```

Boolean: only booleans, with shortcut evaluation

```
!    &&   ||
```

Bitwise: operate on integers

```
&    |    ^    <<   >>   &^
```

Assignment: as above for binary operations

```
=    +=   -=   *=   /=   %=
&=   |=   ^=   <<=  >>=  &^=
```

Operator precedence

There are only five levels of precedence, otherwise left-to-right:

Operators like multiplication:

```
*    /    %    <<   >>   &    &^
```

Operators like addition:

```
+    -    |    ^
```

Comparison operators:

```
==   !=   <    <=   >    >=
```

Logical and:

```
&&
```

Logical or:

```
||
```