Scope of Variables

In this lesson, you'll study the different data types in detail.

WE'LL COVER THE FOLLOWING

The three main elementary types in Go are:

svg viewer

- Boolean
- Numeric
- Character

Let’s discuss them in detail one by one.

Boolean type

The possible values of this type are the predefined constants **true** and **false**. For example:

```
var b bool = true
```

Numerical type

Integers and floating-point numbers

Go has architecture-dependent types such as **int**, **uint**, and **uintptr**. They have the appropriate length for the machine on which a program runs.

An *int* is a default signed type, which means it takes a size of 32 bit (4 bytes) on a 32-bit machine and 64 bit (8 bytes) on a 64-bit machine, and the same goes for *unit* (unsigned int). Whereas *uintptr* is an unsigned integer large enough to store a bit pattern of any pointer.

The architecture independent types have a fixed size (in bits) indicated by their names. For integers:

- **int8** (-128 *to* 127)
- **int16** (-32768 *to* 32767)
- **int32** (– 2,147,483,648 *to* 2,147,483,647)
- **int64** (– 9,223,372,036,854,775,808 *to* 9,223,372,036,854,775,807)

For unsigned integers:

- **uint8** (with the alias byte, 0 *to* 255)
- **uint16** (0 *to* 65,535)
- **uint32** (0 *to* 4,294,967,295)
- **uint64** (0 *to* 18,446,744,073,709,551,615)

For floats:

- **float32** (± $1O^{-45}$ *to* ± $3.4 * 1O^{38}$)
- **float64** (± $5 * 1O^{-324}$ *to* $1.7 * 1O^{308}$)

> **Note:** Unlike other languages, a `float` type on its own does not exist in Golang. We have to specify the bits. For example `float32` or `float64` .

int is the integer type, which offers the fastest processing speeds. The initial (default) value for integers is 0, and for floats, this is 0.0. A float32 is reliably accurate to about 7 decimal places, and a float64 to about 15 decimal places. Use float64 whenever possible, because all the functions of the math package

Use float64 whenever possible, because all the functions of the math package expect that type.

Numbers may be denoted in:

- **Octal notation** with a *prefix of 0*: *63* can be written as 077.
- **Hexadecimal notation** with a *prefix of 0x*: *255* can be written as *0xFF*.
- **Scientific notation** with *e*, which represents the *power of 10*: *1000* can be written as $1e^3$ or $6.022 * 10^{23}$ can be written as $6.022e^{23}$.

As Go is strongly typed, the mixing of types is not allowed, as in the following program. But constants are considered to have no type in this respect, so with constants, mixing is allowed. Let's do some type-mixing.

```
package main

func main() {
    var a int
    var b int32
    a = 15
    b = a + a // compiler error
    b = b + 5 // ok: 5 is a constant
}
```

Type Mixing

The program will give a compiler error: `cannot use a + a (type int) as type int32 in assignment`. **Line 8** will work fine because **5** is a constant not a variable.

Similarly, if we declare two variables as:

```
var n int16 = 34
var m int32
```

And we do:

```
m = n
```

It will give a compiler error: `cannot use n (type int16) as type int32 in assignment`. Because an int16 cannot be assigned to an int32, there is no implicit casting. In the following program, an explicit conversion is done to avoid this.

```go
package main
import "fmt"

func main() {
    var n int16 = 34    // int16 variable
    var m int32         // int32 variable

    m = int32(n)        // explicit typing
    fmt.Printf("32 bit int is: %d\n", m)
    fmt.Printf("16 bit int is: %d\n", n)
}
```

Casting

In the above code, `n` is an `int16` variable and `m` is an `int32` variable. To set the value of `m` equals to `n`, we need explicit type casting because these variables have different types. At **line 8** doing `m = int32(n)` cast `n` with type of `int32` as `m` type is `int32` not `int16`. Then the results are printed at **line 9** and **line 10**.

Format specifiers

- In format-strings **%d** is used as a format specifier for integers (**%x** or **%X** can be used for a hexadecimal representation).
- The **%g** is used for float types (**%f** gives a floating-point, **%e** gives a scientific notation).
- The **%0n***d* shows an integer with **n** digits, a *leading 0* is necessary.
- The %**n.m***g* represents the number with **m** digits after the decimal sign, and **n** before it, instead of *g* also *e* and *f* can be used. For example, the *%5.2e* formatting of the value 3.4 gives *3.40e+00*.

Complex numbers

A *complex number* is written in the form of:

```
re + imi
```

where `re` is the real part, and `im` is the imaginary part, and `i` is the √ -1. For these data we have the following types:

- **complex64** (with a 32 bit real and imaginary part each)

- **complex128** (with a 64 bit real and imaginary part each)

Consider the following program:

```
package main
import "fmt"

func main(){
    var c1 complex64 = 5 + 10i         // Declaring complex num (real +imaginary(¡))
    fmt.Printf("The value is: %v", c1)
}
```

Complex Number

You can see that in the above code, we declare and initialize a complex variable `c1` with type `complex 64`. In this number, **5** is the real part, and **10i** is an imaginary part. Then at **line 6**, its value is printed using a general format specifier **%v**. In format-strings the default format specifier **%v** can be used for complex numbers; otherwise, use **%f** for both constituent parts (real and imaginary separate).

If `re` and `im` are of type *float32*, a variable `c` of type complex64 can be made with the function `complex`:

```
c = complex(re, im)
```

We can also get the parts of a complex number through built-in functions. The functions `real(c)` and `imag(c)` give the real and imaginary part of `c`, respectively.

Random numbers

Some programs, like games or statistical applications, need random numbers. The package **math/rand** implements pseudo-random number generators. For a simple example, see the following program that prints a random number:

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

```
"math/rand"
)

func main(){
    a := rand.Int()              // generates a random number
    b := rand.Intn(8)            // generates a random number in [0, n)
    fmt.Printf("a is: %d\n", a)
    fmt.Printf("b is: %d\n", b)
}
```

Random Number

In the above program, you can see we import a package `math/rand` at **line 4** to generate random numbers. We declare a variable `a` at **line 8**. In `a` a random value is placed. But if we want to create a random number with a range we can use `Intn(n)` function. Random values of range $[0, n)$ (starting from **0** to **n-1**) can be generated. So at **line 9** we made a variable `b` and set it as a random number from 0 to 7. Then at **line 10** and **line 11**, `a` and `b` are printed respectively.

Character type

Strictly speaking, this is not a type in Go. The characters are a *special* case of *integers*. The **byte** type is an *alias* for **uint8**, and this is okay for the traditional ASCII-encoding for characters (1 byte). A byte type variable is declared as:

```
var ch byte = 'A'
```

Single quotes ‘’ surround a character. In the ASCII-table the decimal value for A is **65**, and the hexadecimal value is **41**, so the following are also declarations for the character A:

```
var ch byte = 65
```

or

```
var ch byte = '\x41'
```

`\x` is always followed by exactly **two** hexadecimal digits. Another possible notation is a \ followed by exactly 3 octal digits, e.g., \377.

But there is also support for **Unicode (UTF-8)**. Characters are also called *Unicode code points*, and a Unicode character is represented by an *int* in memory. In the documentation, they are commonly represented as **U+hhhh**, where *h* is a hexadecimal digit. In fact, the type **rune** exists in Go and is an *alias* for type **int32**. To write a Unicode-character in code, preface the hexadecimal value with **\u** or **\U**. If 4 bytes are needed for the character, **\U** is used. Where **\u** is always followed by exactly *four* hexadecimal digits and **\U** by *eight*. Run the following program to see how *Unicode character* type works.

```
package main
import "fmt"

func main(){
    var ch1 int = '\u0041'
    var ch2 int = '\u03B2'
    var ch3 int = '\U00101234'
    fmt.Printf("%d - %d - %d\n", ch1, ch2, ch3)  // integer
    fmt.Printf("%c - %c - %c\n", ch1, ch2, ch3)  // character
    fmt.Printf("%X - %X - %X\n", ch1, ch2, ch3)  // UTF-8 bytes
    fmt.Printf("%U - %U - %U", ch1, ch2, ch3)    // UTF-8 code point
}
```

Characters

At **line 5** and **line 6**, the characters `ch1` and `ch2` that are declared are represented by four bytes because we used `\u`. Where `ch3` is represented with eight bytes using `\U`. You may have noticed that we print these characters using four different format specifiers: **%d**, **%c**, **%X**, **%U** from **line 8** to **line 11**. In format-strings %c is used as a format specifier to show the character, format-specifiers %v or %d show the integer representing the character, and %U outputs the U+hhhh notation.

Unicode package

The package *unicode* has some useful functions for testing characters. Suppose if we have a character named *ch*. Following are some main functions of this package:

- Testing for a letter

```
unicode.IsLetter(ch)
```

- Testing for a digit

```
unicode.IsDigit(ch)
```

- Testing for a whiteSpace character

```
unicode.IsSpace(ch)
```

These functions return a *bool* value. The `utf8` package further contains functions to work with *runes*.

Now you are familiar with data types in detail; it’s time to study the different operations that are possible on data.