Constants

This lesson discusses how constants are used to store data values in Go.

WE'LL COVER THE FOLLOWING

- Introduction
- Explicit and implicit typing
- Typed and untyped constants
- Compilation
- Overflow
- Multiple assignments
- Enumerations

Introduction

A value that *cannot* be changed by the program is called a **constant**. This data can only be of type *boolean*, *number* (integer, float, or complex) or *string*.

Explicit and implicit typing

In Go, a constant can be defined using the keyword **const** as:

```
const identifier [type] = value
```

Here, `identifier` is the name, and `type` is the type of constant. Following is an example of a declaration:

```
const PI = 3.14159
```

You may have noticed that we didn't specify the type of constant `PI` here. It's perfectly fine because the type specifier [type] is *optional* because the compiler can implicitly derive the type from the value. Let's look at another example of implicit typing:

```
const B = "hello"
```

The compiler knows that the constant `B` is a string by looking at its value. However, you can also write the above declaration with explicit typing as:

```
const B string = "hello"
```

> **Remark**: There is a convention to name constant identifiers with all uppercase letters, e.g., const INCHTOCM = 2.54. This improves readability.

Typed and untyped constants

Constants declared through explicit typing are called *typed constants*, and constants declared through implicit typing are called *untyped constants*. A value derived from an untyped constant becomes typed when it is used within a context that requires a typed value. For example:

```
var n int
f(n + 5)   // untyped numeric constant "5" becomes typed as int, because n was int.
```

Compilation

Constants must be evaluated at compile-time. A const can be defined as a calculation, but all the values necessary for the calculation must be available at compile time. See the case below:

```
const C1 = 2/3 //okay
```

Here, the value of `c1` was available at compile time. But the following will give an error:

```
const C2 = getNumber() //not okay
```

Because the function `getNumber()` can't provide the value at compile-time. A constant’s value should be known at compile time according to the design

principles where the function’s value is computed at run time. So, it will give the build error: `getNumber() used as value`.

Overflow

Numeric constants have no size or sign. They can be of *arbitrarily high precision* and do not overflow:

```
const Ln2= 0.693147180559945309417232121458\
176568075500134360255254120680009
const Log2E= 1/Ln2 // this is a precise reciprocal
const BILLION = 1e9 // float constant
const HARD_EIGHT = (1 << 100) >> 97
```

We used \ (backslash) in declaring constant `Ln2`. It can be used as a *continuation character* in a constant.

Multiple assignments

The assignments made in one single assignment statement are called multiple assignments. Go allows different ways of multiple assignments. Let’s start with a simple example:

```
const BEEF, TWO, C = "meat", 2, "veg"
```

As you can see, we made **3** constants. All of them are *untyped* constants. Let’s look at another method where all the constants are named first, and then their values are written if needed. For example:

```
const MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY int= 1, 2, 3,
 4, 5, 6
```

As you can see, the constants, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY` and `SATURDAY` are *typed* constants because their type (**int**) is mentioned explicitly, and they have the values **1**, **2**, **3**, **4**, **5** and **6** respectively.

Enumerations

Listing of all elements of a set is called *enumeration*. Constants can be used for enumerations. For example:

```
const (
  UNKNOWN = 0
  FEMALE = 1
  MALE = 2
)
```

`UNKNOWN`, `FEMALE` and `MALE` are now aliases for **0**, **1** and **2**. Interestingly value **iota** can be used to enumerate the values. Let's enumerate the above example with iota:

```
const (
  UNKNOWN = iota
  FEMALE = iota
  MALE = iota
)
```

The first use of iota gives **0**. Whenever iota is used again on a new line, its value is incremented by **1**; so `UNKNOWN` gets **0**, `FEMALE` gets **1** and `MALE` gets **2**. Remember that a new const block or declaration initializes iota back to 0. The above notation can be shortened, making no difference as:

```
const (
  UNKNOWN = iota
  FEMALE
  MALE
)
```

You can give enumeration a type name. For example, `FEMALE`, `MALE` and `UNKNOWN` are categories of **Gender**. Let's give them `Gender` as the type name:

```
type Gender int
const (
  UNKNOWN = iota
  FEMALE
  MALE
)
```

Now you are familiar with constants. In the next lesson, you'll study variables.