Accurate Computations and the big Package

This lesson discusses how Go ensures accurate computations in its program via the big package.

WE'LL COVER THE FOLLOWING

- A major problem
- Solution provided by Go

A major problem

We know that programmatically performed floating-point computations are sometimes not accurate. If you use Go's *float64* type in floating-point numbers computation, the results are accurate to about **15** decimal digits, enough for most tasks. When computing with very big whole numbers, the range of the types *int64* or *uint64* might also be too small. In that case, *float32* or *float64* can be used if accuracy is not a concern, but if it is, we cannot use floating-point numbers because they are only represented in an approximated way in memory.

Solution provided by Go

For performing perfectly accurate computations with integer values Go provides the `math/big` package contained in `math`: `big.Int` for integers, `big.Rat` for rational numbers (these are numbers than can be represented by a fraction like 2/5 or 3.1416 but not irrational numbers like `e` or `π`) and `big.Float`. These types can hold an arbitrary number of digits, only limited by the machine's available memory. The downside is the bigger memory usage and the processing overhead, which means they are a lot slower to process than built-in integers.

A big integer is constructed with the function `big.NewInt(n)`, where `n` is an *int64*. A big rational number with `big.NewRat(n, d)`, where both `n` (the numerator) and `d` (the denominator) are of type *int64*. A big float number is constructed with the function `big.NewFloat(x)`, where `x` is a *float64*.

Because Go does not support operator overloading, all the methods of the `big` types have names like `Add()` for addition and `Mul()` for multiplication. They are methods (see Chapter 8) acting on the integer, rational or float as a receiver. In most cases, they modify their receiver and return the receiver as a result, so that the operations can be chained. This saves memory because no temporary `big.Int` variables have to be created to hold intermediate results.

We see this in action in the following program:

```go
package main
import (
  "fmt"
  "math"
  "math/big"
)

func main() {
  // Here are some calculations with bigInts:
  im := big.NewInt(math.MaxInt64)
  in := im
  io := big.NewInt(1956)
  ip := big.NewInt(1)
  ip.Mul(im, in).Add(ip, im).Div(ip, io)
  fmt.Printf("Big Int: %v\n", ip)
  // Here are some calculations with big rationals:
  rm := big.NewRat(math.MaxInt64, 1956)
  rn := big.NewRat(-1956, math.MaxInt64)
  ro := big.NewRat(19, 56)
  rp := big.NewRat(1111, 2222)
  rq := big.NewRat(1, 1)
  rq.Mul(rm, rn).Add(rq, ro).Mul(rq, rp)
  fmt.Printf("Big Rat: %v\n", rq)
}
```

big Package

In the program above, outside the `main` function at **line 5**, we import the `big` package for accurate computations. In `main`, we perform some calculations with `big` integers and then `big` rationals. At **line 10**, we are returning an ***Int** set to the value of the `math.MaxInt64`, which is equal to **9223372036854775807** and storing it in `im`. In the next line, we are declaring a new variable `in` and initializing it with `im`. At **line 12**, we are returning an ***Int** set to the value of the **1956** and storing it in `io`. At **line 13**, we are returning an ***Int** set to the value of the **1** and storing it in `ip`.

Now at **line 14**, we are applying arithmetic functions as: `ip.Mul(im, in).Add(ip, im).Div(ip, io)`. First, the values of `im` and `in` will be multiplied and stored in `ip`. Then the value of `im` and updated value of `ip` will be added together, and the result will be stored again in `ip`. Now the updated value of `ip` will be divided with the value of `io`, and the final result will again be stored in `ip`. In the next line, we are printing `ip` to check the final value.

The rest of the program is for rational numbers. At **line 17**, we are returning ***Rat** set to the value of the division of *numerator* `math.MaxInt64` and *denominator* of **1956** and storing it in `rm`. In the next line, we are returning ***Rat** set to the value of the division of *numerator* **-1956** and *denominator* of **`math.MaxInt64`**, and storing it in `rn`. At **line 19**, we are returning ***Rat** set to the value of the division of *numerator* **19** and *denominator* of **56** and storing it in `ro`. At **line 20**, we are returning ***Rat** set to the value of the division of *numerator* **1111** and *denominator* of **2222** and storing it in `rp`. In the next line, we are returning ***Rat** set to the value of the division of *numerator* **1** and *denominator* of **1** and storing it in `rq`.

Now at **line 22**, we are applying arithmetic functions on them as: `rq.Mul(rm, rn).Add(rq, ro).Mul(rq, rp)`. First, the values of `rm` and `rn` will be multiplied and stored in `rq`. Then the value of `ro` and updated value of `rq` will be added together, and the result will be stored again in `rq`. Now the updated value of `rq` will be multiplied with the value of `rp`, and the final result will again be stored in `rq`. In the last line, we are printing `rq` to check the final value.

That's it about the `big` package and its working. In the next lesson, you'll see how you can make a package of your own.