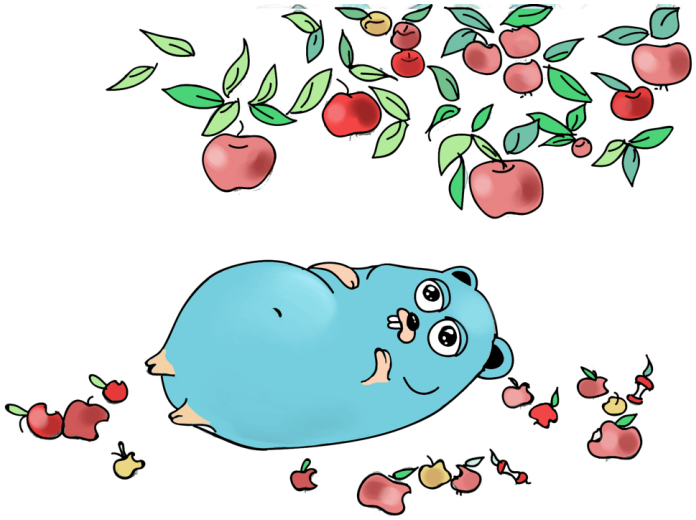

Golang concurrency

```
 8    type counter int64
 9
10    func (c *counter) Add1(x int64) {
11        *c++
12    }
13
14    func (c *counter) Add2(x int64) {
15        *c++
16    }
```

Suppose we define a new type of counter. And two functions that increment the instance of "c counter", which is a pointer. The functions Add1 and Add2 are executed in parallel. Suppose we run a cycle up to 100000 (inclusive). Thus, in theory, we should get a result of 200000. But we get a completely different result. So we got the race condition.

Data race is when one parallel operation tries to read a variable, while at some indefinite time another parallel operation tries to write to the same variable.

A **race condition** occurs when two or more operations must be performed in the correct order, but they do not because of how the program is written.

Code

```
package main

import (
   "fmt"
)

type counter int64

func (c *counter) Add1(x int64) {
   *c++
}

func (c *counter) Add2(x int64) {
   *c++
}

func (c *counter) Value() (x int64) {
   return int64(*c)
}

func main() {
   number := counter(0)

   for i := 0; i < 100000; i++ {
         go number.Add1(1)
         go number.Add2(1)
      }

  fmt.Println(number.Value())

}
```