Data Races and Race Conditions

In this lesson, you will learn about data races which in most cases lead to race conditions.

One of the reasons why concurrency is hard to achieve is because of **data races**.

Data Race

> A **data race** happens when processes have to access the same variable concurrently i.e. one process reads from a memory location while another simultaneously writes to the exact same memory location.

The following function is an example of a data race:

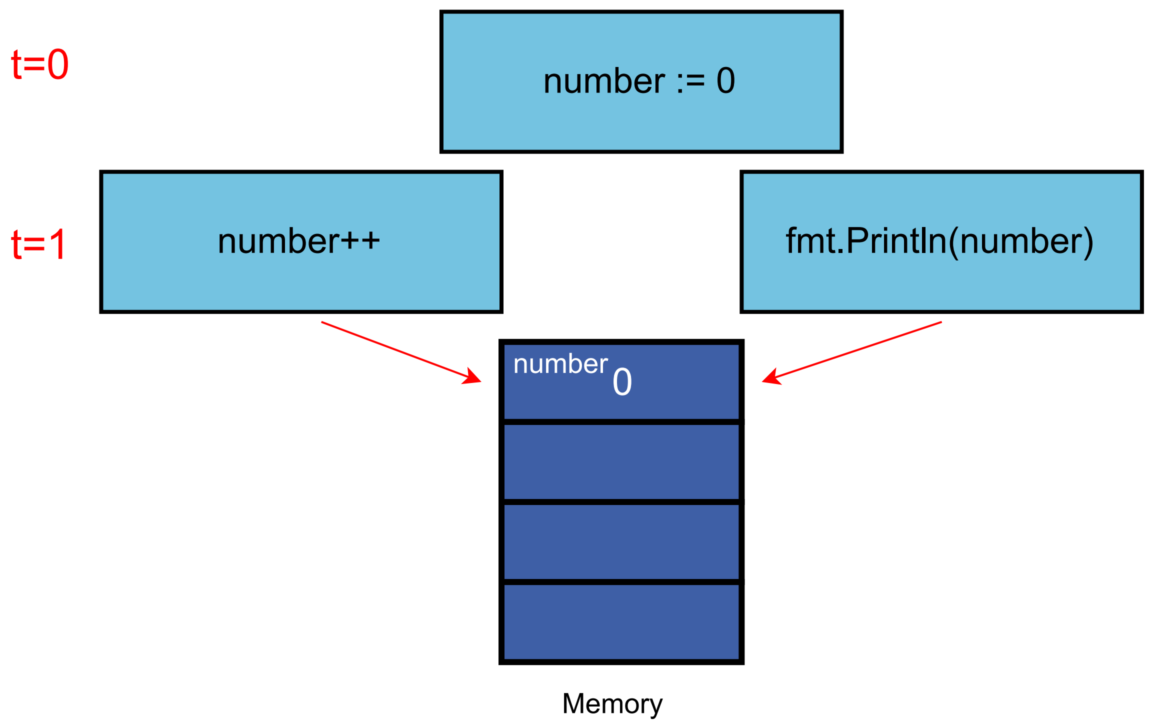
```
package main
import "fmt"

func main() {
    number := 0;

    go func(){
      number++ //reading and modifying the value of 'number'
    }()

    fmt.Println(number) //reading the value of 'number'


}
```

Data Race Example

We increment the value of the variable `number` i.e. we first access the value, add `1` to it and then write the new value back to the memory. `number++` takes place using an anonymous **go routine**. In the next step, we read the value of `number` and print it onto the console. However, the output of the code above turns out to be `0` because the main routine finishes itself before the goroutine has a chance to execute itself completely. We'll explore more about this concept in the second chapter.

The point to note in the above example is that `number++` and `fmt.Println(number)` are participating in a data race as `number++` is reading from and writing to the same memory location that `fmt.Println(number)` is reading from.

This will get us in trouble if these operations execute at the same time which is a possibility when it comes to executing code with goroutines as they are concurrent operations.



Data Race Explained

While writing concurrent code, you need to be careful about data races and avoid the temptation to think sequentially. Instead, keep in mind all possible timings of your concurrent operations while designing your solution. This is because some data races are benign whereas some lead to *race conditions*.

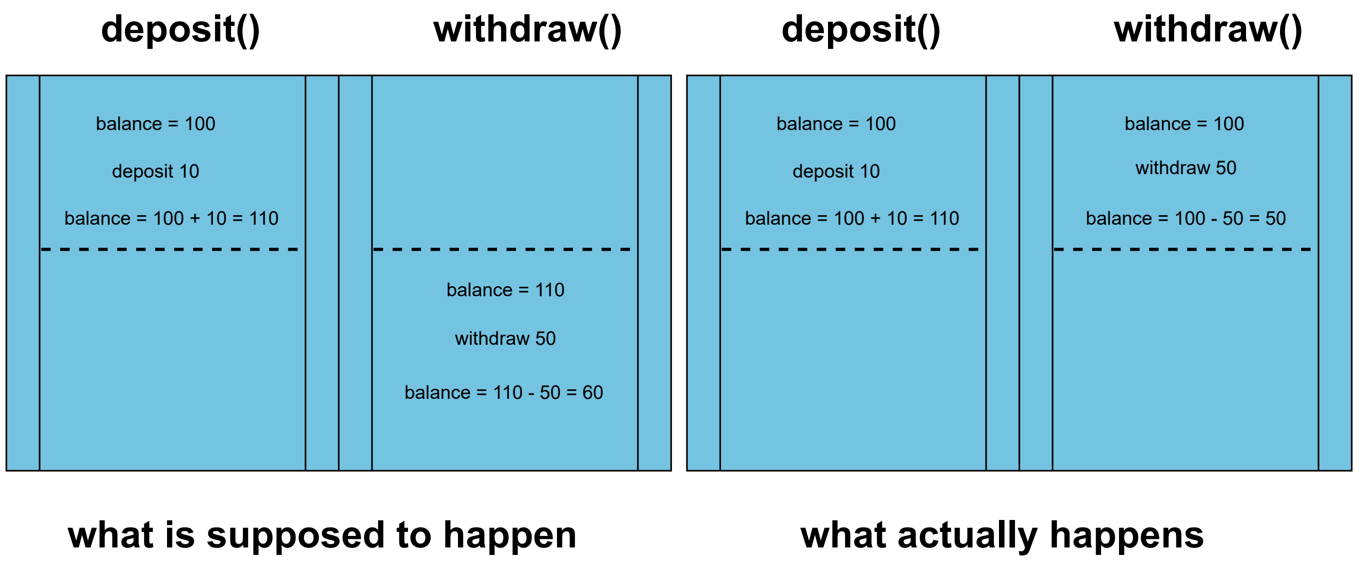
Race Condition

> A race condition is a flaw in a program regarding the timing/ordering of operations which disrupts the logic of the program and leads to erroneous results.

Let’s try to understand it using an example:

```
package main
import "fmt"

func deposit(balance *int,amount int){
    *balance += amount //add amount to balance
}

func withdraw(balance *int, amount int){
    *balance -= amount //subtract amount from balance
}

func main() {

    balance := 100

    go deposit(&balance,10) //depositing 10

    withdraw(&balance, 50) //withdrawing 50

    fmt.Println(balance)


}
```

In the code above, we have a `balance` of `100`. We first execute the `deposit()` operation using a goroutine and deposit `10` to our `balance`. Next, we withdraw `50` from our `balance` which makes the final `balance` `60`. However, if you run the code, the final value of `balance` will come out to be `50`. This is as a result of a race condition which has compromised the *correctness* of the program due to an incorrect order of execution of operations.



Race Condition Explained

Data Race Detector

Amazingly, Go has its own in-built data race detector which you can learn more about here. The code below runs the data race detector on our previous example. Let's see how it works:

Environment Variables

Key:	Value:
PATH	/go/bin:/usr/local/go/bin:/usr/local/sbi...

```
package main
import "fmt"

func deposit(balance *int,amount int){
    *balance += amount //add amount to balance
}

func withdraw(balance *int, amount int){
    *balance -= amount //subtract amount from balance
}

func main() {

    balance := 100

    go deposit(&balance,10) //depositing 10

    withdraw(&balance, 50) //withdrawing 50

    fmt.Println(balance)

}
```

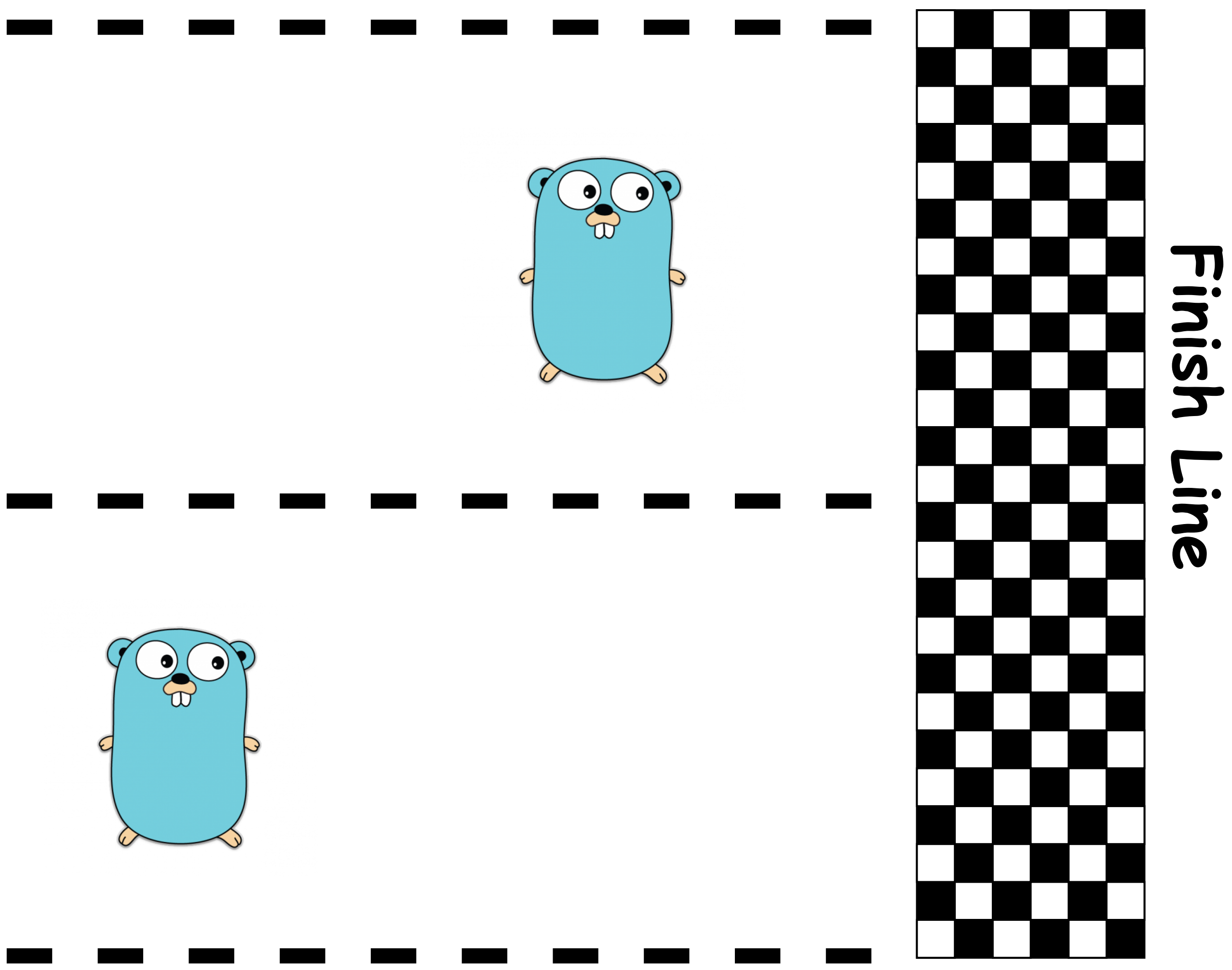
The code above gives the following output:

```
==================
WARNING: DATA RACE
Read at 0x00c42005e168 by goroutine 6:
  main.deposit()
      /usercode/main.go:5 +0x3b

Previous write at 0x00c42005e168 by main goroutine:
  main.main()
      /usercode/main.go:18 +0xb0

Goroutine 6 (running) created at:
  main.main()
      /usercode/main.go:16 +0x86
==================
Found 1 data race(s)
exit status 66
```

You can see that it has detected the data race in our code!

How to avoid data races?

In Go, we can avoid data races by using **channels** or **locks**. They will allow us to synchronize memory access to all shared mutable data. You haven't been introduced to channels yet but we'll explore them in detail in the next chapter. Wait until then!


Finish Line