Quit Channel

In this lesson, we'll study a pattern related to quitting from a select statement.

You will notice that you have probably seen the quit channel pattern in the coding examples before. Let's discover more about it!

So we'll set up a car race competition. In the code below, we launch three goroutines by calling the `Race` function in a for-loop. Now we are only interested in the race until one of the cars reaches the finishing line. Let's see how it's done:

```
package main
import ( "fmt"
         "math/rand"
         "time")

func Race(channel, quit chan string, i int) {

  channel <- fmt.Sprintf("Car %d started!", i)
    for{
      rand.Seed(time.Now().UnixNano())
      time.Sleep(time.Duration(rand.Intn(500)+500) * time.Millisecond)
      quit <- fmt.Sprintf("Car %d reached the finishing line!", i)
    }

}

func main() {

  channel := make(chan string)
  quit := make(chan string)

  for i:=0; i < 3; i++{
    go Race(channel,quit,i)
  }

  for{
    select{
      case raceUpdates := <-channel:
        fmt.Println(raceUpdates)
      case winnerAnnoucement := <-quit:
        fmt.Println(winnerAnnoucement)
       return

    }
  }
}
```

After the goroutines start concurrently, we have a select statement from **line 27** to **line 34**.

```
for{
    select{
      case raceUpdates := <-channel:
        fmt.Println(raceUpdates)
      case winnerAnnoucement := <-quit:
        fmt.Println(winnerAnnoucement)
       return

    }
  }
```

The `channel` will update us when the car starts off. If any one of the cars reaches the finishing line, it means that a goroutine has reached this line:

```
quit <- fmt.Sprintf("Car %d reached the finishing line!", i)
```

Then the second case of the select statement is executed and we call `return` to break out of the `for` loop, which is supposed to be infinite.

This pattern is useful when we are supposed to break out of channel communication operations.

You can also use the quit channel methodology to communicate back and forth with the sender and inform it when you end the communication. For example:

```
package main
import ( "fmt"
         "math/rand"
         "time")

func Race(channel, quit chan string, i int) {

  channel <- fmt.Sprintf("Car %d started!", i)
    for{
      rand.Seed(time.Now().UnixNano())
      time.Sleep(time.Duration(rand.Intn(500)+500) * time.Millisecond)
      quit <- fmt.Sprintf("Car %d reached the finishing line!", i)
      fmt.Println(<-quit)
    }


}

func main() {

  channel := make(chan string)
  quit := make(chan string)

  for i:=0; i < 3; i++{
    go Race(channel,quit,i)
  }

  for{
    select{
      case raceUpdates := <-channel:
        fmt.Println(raceUpdates)
      case winnerAnnoucement := <-quit:
        fmt.Println(winnerAnnoucement)
        quit <- "You win!"
       return

    }
  }
}
```

As you can see on **line 33**, we send back the winning message to the goroutine that won, i.e. which sent the winning message to the `quit` channel first. In response, the `quit` channel also communicates to the goroutine that they have won (**line 13**).

That was all there is to quit channel pattern. Let's get familiar with the TimeOut technique using Select Statement in the next lesson.