Timeout using Select Statement

This lesson will introduce you to a pattern which uses the `time.After` function in a select statement.

In the previous lesson, we learned how to break out of channel operations in the select statement. Now what if we want to break out of channel communications after a certain period of time?

This will be done using the `time.After` function which is imported from the `time` package. It returns a channel that blocks the code for the specified duration. After that duration, the channel delivers the current time but only once.

Let's have a look at an example using the `time.After` function:

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


func main() {
  dynamite := make(chan string)

  go func(){
    rand.Seed(time.Now().UnixNano())
    time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    dynamite <- "Dynamite Diffused!"
  }()

  for{
    select
    { case s := <-dynamite:
        fmt.Println(s)
        return
      case <-time.After(time.Duration(rand.Intn(500)) * time.Millisecond):
        fmt.Println("Dynamite Explodes!")
        return
    }
  }
}
```

In the above code example, we have a channel named `dynamite`. On **line 10**, we create a goroutine which sends `Dynamite Diffused!` on the `dynamite` channel after a random period of time.

```
go func(){
    rand.Seed(time.Now().UnixNano())
    time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    dynamite <- "Dynamite Diffused!"
  }()
```

From **line 17** to **line 24**, we have a select statement comprising of two cases.

```
select
  { case s := <-dynamite:
      fmt.Println(s)
    case <-time.After(time.Duration(rand.Intn(500)) * time.Millisecond):
      fmt.Println("Dynamite Explodes!")
  }
```

If the `dynamite` channel is able to deliver the `Dynamite Diffused` message before the random period of time that will execute the second case, we are safe! But if the `time.After` function delivers the current time after a random duration, the bomb will explode and we are doomed!

The same functionality as above can also be done in the following way:

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


func main() {
  dynamite := make(chan string)

  go func(){
    rand.Seed(time.Now().UnixNano())
    time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    dynamite <- "Dynamite Diffused!"
  }()

  timeout := time.After(time.Duration(rand.Intn(500)) * time.Millisecond)
  for {
    select
    { case s := <-dynamite:
        fmt.Println(s)
    fmt.Println(s)
    return
   case <-timeout:
    fmt.Println("Dynamite Explodes!")
    return
  }
 }
}
```

On **line 16**, we create the timer outside the loop to time out the entire channel communication. This is useful if we have an infinite for-loop with no return statements to break out. Then, having the following statement from the first example in this lesson will have a timeout for each message:

```
case <-time.After(time.Duration(rand.Intn(500)) * time.Millisecond)
```

Hope everything is clear up till now. We are up for a challenge in the next lesson!