Sequencing

This lesson will teach you about a pattern used for sequencing in a program by sending channel over a channel.

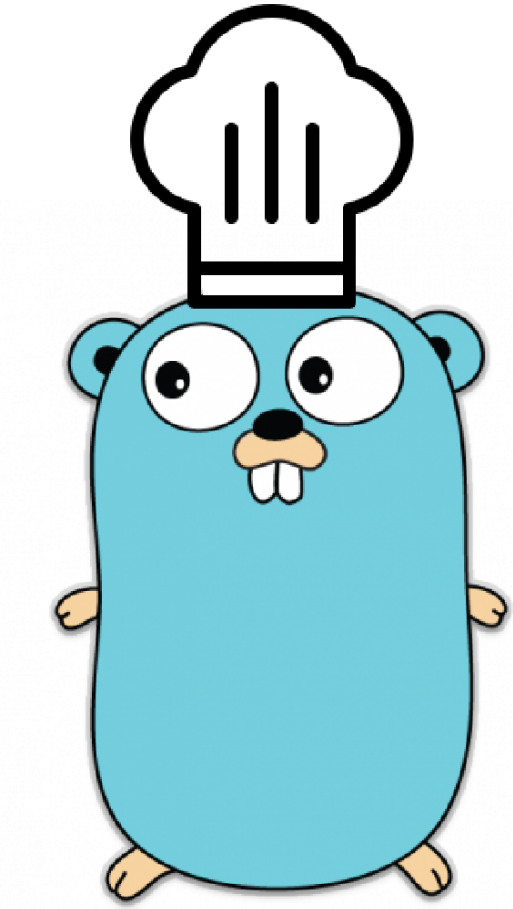
Remember the code from the last lesson where we unblock the two receive operations. Here is the code for you to recap:

```
package main

import ( "fmt"
                  "math/rand"
                  "time")

func updatePosition(name string) <-chan string {
        positionChannel := make(chan string)

        go func() {
                for i := 0; ; i++ {
                        positionChannel <- fmt.Sprintf("%s %d", name , i)
                        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
                }
        }()

        return positionChannel
}

func fanIn(mychannel1, mychannel2 <-chan string) <-chan string {
        mychannel := make(chan string)

        go func() {
                for {
                        mychannel <- <-mychannel1
                }
        }()

        go func() {
                for {
                        mychannel <- <-mychannel2
                }
        }()

        return mychannel
}


func main() {
        positionsChannel := fanIn(updatePosition("Legolas :"), updatePosition("Gandalf :"))


        for i := 0; i < 10; i++ {
```

```
        for i := 0; i < 10; i++ {
                fmt.Println(<-positionsChannel)
        }

        fmt.Println("Done with getting updates on positions.")
}
```

What if we don't want to block the code and introduce sequence to our program instead of randomness? Let's see how we approach this problem.

Imagine a cooking competition. You are participating in it with your partner.



The rules of the game are:

1. There are three rounds in the competition.
2. In each round, both partners will have to come up with their own dishes.
3. A player cannot move on to the next round until their partner is done with their dish.
4. The judge will decide the entry to the next round after tasting food from both the team members.

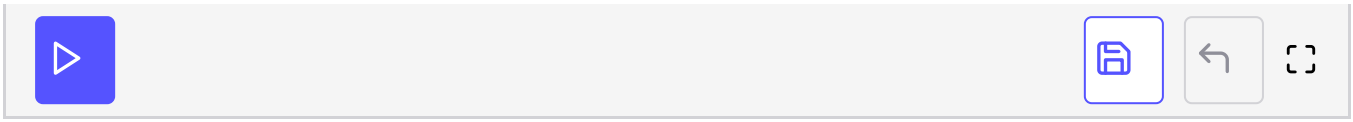
Hope the rules are clear. Now in order to achieve this scenario, we'll send a channel over a channel!

Surprised? Let's see how it's done:

```
package main

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

```
type CookInfo struct {
        foodCooked  string
        waitForPartner chan bool
}



func cookFood(name string) <-chan CookInfo {

        cookChannel := make(chan CookInfo)
        wait := make(chan bool)
        go func() {
                for i := 0; ; i++ {
                        cookChannel<- CookInfo{fmt.Sprintf("%s %s", name,"Done") , wait}
                        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)

                        <-wait
                }
        }()

        return cookChannel
}

func fanIn(mychannel1, mychannel2 <-chan CookInfo) <-chan CookInfo {
        mychannel := make(chan CookInfo)

        go func() {
                for {
                        mychannel <- <-mychannel1
                }
        }()

        go func() {
                for {
                        mychannel <- <-mychannel2
                }
        }()

        return mychannel
}


func main() {
        gameChannel := fanIn(cookFood("Player 1 : "), cookFood("Player 2 :"))


        for round := 0; round < 3; round++ {
                food1 := <-gameChannel
                fmt.Println(food1.foodCooked)

                food2 := <-gameChannel
                fmt.Println(food2.foodCooked)

                food1.waitForPartner <- true
                food2.waitForPartner <- true

                fmt.Printf("Done with round %d\n", round+1)
        }

        fmt.Println("Done with the competition")
```

```
}
```



You can see from the code above that both our players complete their dishes in each round because they wait for each other instead of proceeding to the next round.

First of all, we created:

```
type CookInfo struct {
    foodCooked  string
    waitForPartner chan bool
}
```

`CookInfo` contains a string named `foodCooked` and a channel `waitForPartner` of type `bool` which will act as a signaler.

In the main routine, on **line 51**, we call the `fanIn` function and `cookFood` functions. Let's dive into the working of the `cookFood` function.

```
func cookFood(name string) <-chan CookInfo {

    cookChannel := make(chan CookInfo)
    wait := make(chan bool)
    go func() {
        for i := 0; ; i++ {
            cookChannel<- CookInfo{fmt.Sprintf("%s %s", name,"Done") , wai
t}
            time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)

            <-wait
        }
    }()

    return cookChannel
}
```

We are sending `CookInfo` of type `struct` which also contains a channel `waitForPartner` over the `cookChannel`. After sending the struct on to the channel, we block our goroutine by using the following statement: `<-wait`.

In the main routine, when we receive the data i.e. the `struct` from these goroutines, we send back `true` on the channel inside the struct using these statements (**lines 61-62**):

```
        food1.waitForPartner <- true
        food2.waitForPartner <- true
```

Note that we only send `true` back once we have received food from both the players (**lines 55-62**):

```
    food1 := <-gameChannel
    //waiting for food1
        fmt.Println(food1.foodCooked)

        food2 := <-gameChannel
    //waiting for food2
        fmt.Println(food2.foodCooked)

        food1.waitForPartner <- true
        food2.waitForPartner <- true
   //only now can we proceed to the next round
```

The rest of the code uses the Fan-In technique that you are already familiar with.

Let's move on to the next lesson which will teach us about the range and close pattern.