**12.22.2022 Go Lesson Instruction**

1. **Directional channel**

*The optional* <- *operator specifies the channel direction, send or receive.*

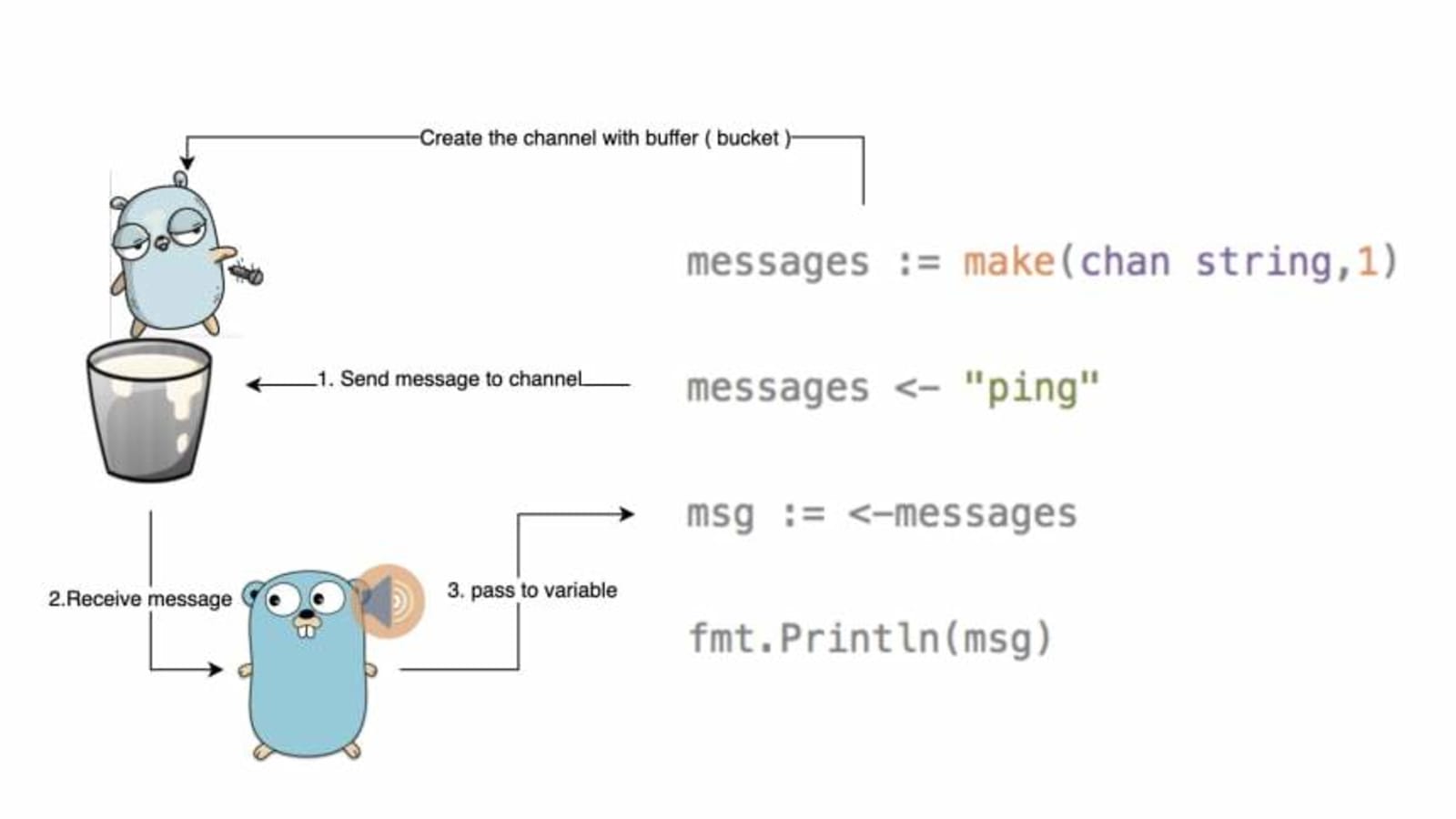
This means channel can actually be declared in one of three ways, depending on whether we want to be bidirectional, receive-only or send-only.

* ***var bidirectional chan string //*** *can read from, write to and close()*
* ***var receiveonlychan <-chan string //*** *can read from, but cannot write to or close().*
* ***var sendonlychan chan<- string //*** *cannot read from, but can write to and close().*

A good way to remember how this works is that, in declarations, the arrow indicates how the channel is allowed to be used.

<-chan //data only comes out

chan<- //data only goes on



2. **Using channel**

* Create general channels
* In funcs you can specify
* Receive channel
  + You can receive values from the channel
  + A receive channel parameter
  + Ing the func, you can only pull values from the channel
  + You can’t close a receive channel
* Send channel
  + You can push values to the channel
  + You can’t receive/pull/read from the channel
  + You can only push values to the channel

3. **Range channel**

package main

import "fmt"

func main() {

    c := make(chan int)

    go func(){

        for i := 0; i < 10; i++ {

            c <- i

        }

        close(c)

    }()

    fmt.Println(<-c)

    for v := range c {

        fmt.Println(v)

    }

    fmt.Println("Exited")

}

The **range** keyword can also be used on a channel.

By doing so, it will iterate over every item that’s send on the channel. You can iterate on both buffered and unbuffered channels, but buffered channels need to be closed before iterating over them.

**Close function**

A sender can **close** a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression: after

*V, ok := <-ch*

**ok** is **false**  if there are no more values to receive and the channel is closed.

The loop **for := range c** reveives values from the channel repeatedly until it is closed

**Note:** Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic

**Another note:** Channels aren’t like files; you don’t usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a **range** loop.

4. **Select channel**

Go’s select lets you wait on multiple channel operations. Combining goroutines and channels with select is a powerful feature of Go.

package main

import "fmt"

func main() {

    eve := make(chan int)

    odd := make(chan int)

    quit := make(chan int)

    //send

    go send(eve, odd, quit)

    //receive

    receive(eve, odd, quit)

}

func receive(e, o, q <-chan int) {

    for {

        select {

            case v := <- e:

                fmt.Println("from the eve: ", v)

            case v := <- o:

                fmt.Println("from the odd", v)

            case v := <- q:

                fmt.Println("from the quit", v)

            return

        }

    }

}

func send(e, o, q chan<- int) {

    for i := 0; i < 100; i++ {

        if i % 2 == 0 {

            e <- i

        } else {

            o <- i

        }

    }

    q <- 0

}

5. **Ok Idiom**

Go is very good language when it comes to testing different features or data in Go programs. Unlike other languages like Python that have the try catch error handling pattern, Go uses the famous **comma ok pattern.** The comma ok pattern is used to test certain statements in our code so that we can minimize developing programs that do things they are nor intended to do.

package main

import "fmt"

func main() {

    c := make(chan int)

    go func(){

        c<- 34

        close(c)

    }()

    v, ok := <-c

    fmt.Println(v, ok)

    v, ok = <-c

    fmt.Println(v, ok)

}

Commo ok is like that try catch. We can use it in every coming and sending data process. Because some problems are in transaction.

FOR FULLY UNDERSTANDING GOROUTINES AND CHANNEL -🡪  
<https://medium.com/nerd-for-tech/learning-go-concurrency-goroutines-channels-8836b3c34152>

6. **Random Number**

Golang provides a package math/rand for generating pseudorandom. This package basically uses a single source that causes the production of a deterministic sequence of values each time a program is executed. Here, if you need different output or outcome for each execution, you can use the seed function to initialize the default source which is safe for concurrent use by multiple goroutines. It generates an integer in the interval of 0 and n. It takes only argument, the n or the upper bound & throws error if argument is less than zero.

RandomInteger := rand.Int() // generates a random integer

func main() {

    fmt.Print(rand.Intn(100))

    fmt.Print(rand.Intn(100))

    fmt.Print(rand.Intn(100))

    fmt.Println()

}

The above method is not safe if the user wants to keep the random numbers secret. That’s why Golang provides Crypto rand to varies the level of randomness of numbers to come. It is used for generating passkeys, CSRF tokens or anything related to security.

func main(){

    randomCrypto, \_ := rand.Prime(rand.Reader, 128)

    fmt.Println(randomCrypto)

}