Why is always a very important question for any concept. Same is true for go lang channels. A question comes to mind: Why should we use go lang channels? How are channels helpful? REMEMBER .. we had discussed "messaging". Functions or Classes need to communicate with each other. Processes have to communicate with each other, even computers need to communicate. If classes or functions communicate within the process, we call it "calling a function, calling a method or message passing" depending on the language. If processes are communicating then it is called Inter-Process Communications. If computers are communicating, then its called networking. In go lang, when one go routine calls the CHANNEL, it uses a SEND or a RECEIVE operation. Channel can be one way or bi-directional. Channels can help block execution or can be asynchronous. So answering the original question - Why channels? Channels are used in Golang to communicate between goroutines. They are a powerful tool that can be used to implement a variety of concurrent programming patterns. **Channels help with asynchronous programming in a few ways. First,** they allow goroutines to communicate with each other without blocking. This means that goroutines can continue executing even if they are waiting for data from another goroutine. **Second,** channels can be used to synchronize the execution of goroutines. For example, a goroutine can only continue executing if it receives a value from a channel. This can be used to ensure that goroutines do not access shared data in an unsafe way. Here are some examples of how channels can be used to implement asynchronous programming patterns: **Producer-consumer pattern:** This pattern is used to implement a system where one goroutine (the producer) produces data, and another goroutine (the consumer) consumes the data. The producer and consumer can communicate with each other using a channel. **Worker pool pattern:** This pattern is used to implement a system where a pool of goroutines (workers) execute tasks. The tasks are submitted to the pool using a channel. The workers then receive tasks from the channel and execute them. **Pipeline pattern:** This pattern is used to implement a system where a series of goroutines (stages) process data. The data is passed from one stage to the next using a channel. Each stage performs a specific operation on the data, and the final stage produces the output. Channels are a powerful tool that can be used to implement a variety of concurrent programming patterns. They can help to make asynchronous programming easier and more reliable.

Some things to keep in mind when dealing with go channels. **Channel type:** The type of channel determines the type of data that can be sent and received on the channel. For example, if you want to send and receive strings on the channel, you would create a channel of type chan string. **Channel**

capacity: The capacity of a channel determines how many messages can be buffered on the channel before a send operation blocks. If the capacity of a channel is zero, then the channel is unbuffered. This means that a send operation will block if the channel is full. **Channel closing:** A channel can be closed using the close() function. Once a channel is closed, no more messages can be sent on the channel. If a goroutine tries to send a message on a closed channel, it will panic. **Channel deadlock:** A channel deadlock can occur if a goroutine is trying to send a message on a channel that is full, and another goroutine is trying to receive a message on the same channel that is empty. To avoid this, it is important to make sure that the capacity of a channel is large enough to handle the number of messages that are being sent on the channel.

Finally, understanding the scope of channels: In this example I have created a channel in func main(). Then I want to access it from go routine foobar(). Then I also want to access it from foobarchild() (which is the child go routine created in foobar() go routine). In the example below I want to access the channel from both foobar() and foobarchild(). **NOTE: We use channels to pass messages (data) between go routines. Imagine you have a program that performs tasks T1, T2, T3. You define 3 go routines - one for each task but all three go routines have to operate on the same data. This means you will have to pass data around. Channels can be very helpful in this scenario.** The scope of a channel is determined by where you declare it, just like any other variable in Go. If you declare a channel in `func main()`, it can be accessed within any goroutines that are spawned from `func main()`, provided you pass the channel to those goroutines as arguments. Similarly, a channel can also be passed from a goroutine to another goroutine spawned within it. Below is an example demonstrating this. In this example, `main` spawns a goroutine `fooBar` and passes a channel to it. Then `fooBar` spawns `fooBarChild` and passes the same channel to it:

```
package main import ( "fmt" "time" )
// fooBarChild is a goroutine function which will be called inside fooBar
func fooBarChild(ch chan int) {
ch <- 2
// Send value '2' to the channel
fmt.Println("fooBarChild: Sent data to channel")
}
// fooBar is a goroutine function which will be called in main and it calls fooBarChild
func fooBar(ch chan int) {
go fooBarChild(ch)
fmt.Println("fooBar: Sent data to fooBarChild goroutine")
}
```

```
func main() {
ch := make(chan int)
// Create a channel of type int
go fooBar(ch)
// Pass the channel as an argument // Receive value from the channel fmt.Println("main: Received data from
channel:",
<-ch)
// Give goroutines some time to finish
time.Sleep(time.Second)
}
```

In this example, the goroutine fooBarChild sends a value to the channel, and main reads that value from the channel. There's a Sleep call at the end of main to give the goroutines some time to finish, but in real-world programs, you'd use a more robust way to wait for goroutines to finish (like using a WaitGroup from the sync package). Remember that the channels block the goroutines when they are trying to send/receive and there's no corresponding receive/send operation in another goroutine. This is why main can receive from ch after fooBar is started - it will block until fooBarChild sends a value to ch.

The above example shows how messaging can be done using CHANNELS ... but messaging can also be done by passing functions around. If we were to write the same program using functional programming technique, then it would look something like this: **Functional programming with Go:** In the context of Go, this might mean passing functions as arguments or returning functions from functions. In this example fooBar and fooBarChild functions have been replaced with higher-order functions:

```
package main
import ( "fmt" "time" )
// fooBarChild is a function which returns a function that can be run as a goroutine.
func fooBarChild(ch chan int) func() {
return func() {
ch <- 2
// Send value '2' to the channel
fmt.Println("fooBarChild: Sent data to channel") } }
// fooBar is a function which returns a function that can be run as a goroutine.
func fooBar(ch chan int) func() {
return func() {
go fooBarChild(ch)()
// Run the goroutine returned by fooBarChild
fmt.Println("fooBar: Sent data to fooBarChild goroutine")
}
}
func main() {
```

```
ch := make(chan int) // Create a channel of type int
go fooBar(ch)()
// Run the goroutine returned by fooBar
// Receive value from the channel
fmt.Println("main: Received data from channel:", <-ch) // Give goroutines some time to finish
time.Sleep(time.Second) }
```

This program works in a similar way to the previous one. The only difference is that the `fooBar` and `fooBarChild` functions **return functions** that can be run as goroutines, instead of running the goroutines themselves. **This approach gives you more flexibility, because you can create a goroutine with specific parameters and run it later.**

NOTE: Go isn't a functional programming language in the same sense as languages like Haskell or Lisp, but it still has first-class functions, and you can use them to apply some techniques of functional programming.