**Go Pipelines** (<https://blog.golang.org/pipelines>)

**Synchronisation Use Cases**

|  |  |
| --- | --- |
| **Channel** | **Mutex** |
| passing data | state |
| communicating async results | caches |

**Basic Channel Syntax**

**func main() {  
 cWriter := make(chan int) // create a channel with the make keyword  
 cWriter <- 2 // write   
 cReader := <- cWriter // read   
 println(cReader)  
}**

**The Go Playground https://play.golang.org/**

This code fails as writing to the channel makes no sense in a single goroutine

**func main() {  
 cWriter := make(chan int)   
 // use a function literal as a goroutine channel writer  
 go func() {   
 cWriter <- 2   
 }()**

**// use the main function goroutine as a channel reader  
 cReader := <- cWriter   
 println(cReader)  
}**

* **Channels are go system types**
* **Channels are created with the make keyword**
* **There exists a channel operator ‘<-’**
* **channel operator position determines whether it will be a read or a write**

**Buffered channels**

**package main**

**import ( "fmt"**

**"time" )**

**var unbuffered = make(chan int)**

**var buffered = make(chan int, 5)**

**func main() {  
 count := 5  
 go func() {  
 for i := 0; i <= count; i++ {  
 fmt.Println("send message")  
 unbuffered <- i  
 }  
 }()  
  
 time.Sleep(time.Second \* 3)  
  
 for i := 0; i <= count; i++ {  
 fmt.Println(<- unbuffered )  
 }  
}**

// synchronous unbuffered channel

**unbuffered := make(chan int)**

// asynchronous buffered channel

**buffered := make(chan int, 5)**

* unbuffered channels block execution until sender and receiver are ready to communicate
* buffered channels read operations succeed without blocking while buffer is not empty, write operations succeed without blocking while buffer is not full.

**package main**

**import "fmt"**

**var unbuffered = make(chan int)**

**var buffered = make(chan int, 5)**

**func main() {  
 count := 5  
 go func() {  
 for i := 0; i <= count; i++ {  
 fmt.Println("send message")  
 buffered <- i  
 }**

**close(buffered )**

**}()  
  
 for num := range buffered {  
 fmt.Println(num)  
 }  
}**

* using the **range** key word we can iterate through an undefined number of messages
* using the **range** key word the channel must be closed
* to perform reads asynchronously need to know the exact number messages

**Select on Multiple channels**

**package main**

**import ( "fmt"; "time" )**

**func main() {  
 c1 := getChannel("channel one")  
 c2 := getChannel("channel two")  
 for i := 1; i <= 4; i++ {  
 select {  
 case msg := <-c1:  
 println(msg)  
 case msg := <-c2:  
 println(msg)  
 }  
 } }**

**func getChannel(msg string ) <-chan string {  
 c := make(chan string)  
 go func() {  
 for i := 1; i <= 2; i++ {  
 c <- fmt.Sprintf("%s %d", msg, i)  
 // Wait before sending next message  
 time.Sleep(150)  
 }  
 }()  
 return c  
}**

* We use a [select](http://golang.org/ref/spec#Select_statements) statement for communication operations on multiple channels
* When there is data in the channels case statement then the code block executes

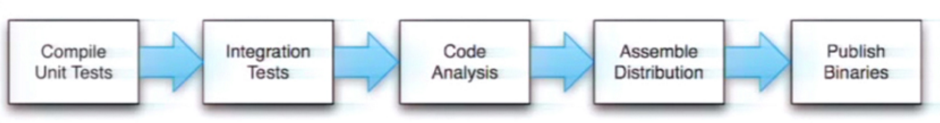
There's no formal definition of a pipeline, rather it is a design pattern we can employ to organise/optimises complex data flows in application

**Pipeline:** pipelines take inputs then connect a sequence of tools to produce outputs

To construct a pipeline you would

1. Identify the inputs and where you get them
2. Identify the tools and strategies for the steps in the sequence
3. Identify the outputs and where to find them

Example build pipeline



In go we can use channels to construct pipelines this is good software engineering practice

**import "fmt"**

**// stage 1**

**func inputStage(data ...int) <-chan int {  
 outPutData := make(chan int)  
 go func() {  
 for \_, inputData := range data {  
 outPutData <- inputData }  
 close(outPutData )  
 }()  
 return outPutData**

**}**

**// stage 2**

**func processStage(input <-chan int) <-chan int {  
 outPutData := make(chan int)  
 go func() {  
 for inputData := range input {  
 outPutData <- inputData \* inputData }  
 close(outPutData )  
 }()  
 return outPutData**

**}**

**func main() {  
 // Set up the pipeline and consume the output.**

**for transformedData := range processStage(inputStage(2, 3)) {  
 fmt.Println(transformedData )   
 }  
}**

* we can create pipelines by chaining channels
* a chained channel is a stage in the pipeline
* a stage will receive from an inbound channel until closed.
* pipelines are a design pattern not an explicit type