Module Eleven

Concurrency in Go

[Go] is the most elegant imperative language ever (including dynamic ones like Lua, Ruby, and Python)
Quoc Anh Trinh

Four out of five language designers agree: Go sucks. The fifth was too busy [to answer] actually writing code [in Go].
aiju

11.1 Concurrency in Go

As Rob Pike pointed out in one of his lectures:

> *when people hear the word concurrency they often think of parallelism, a related but quite distinct concept. In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.*

The motivation for making concurrency an integral part of the Go language is that we live in a concurrent world. The real world is made up of independent activities that interact with each other and are coordinated to achieve some common goal. As the developers of Go point out, you can't model these real world concurrent activities in an application with just sequential logic. This real world concurrency is exactly what the developers at Google were working with in in their problem domain, so it became quite natural to think of concurrency as being a fundamental part of the language design when trying to look at ways to address these kinds of challenges

Concurrency is defined in Go as t*he composition of independently executing computations so that software can be structured to interact with real world while still adhering to the principles of clean code.* (This is a slightly paraphrased definition from Rob Pike's presentations.)

11.1.2 Parallelism versus Concurrency

Parallelism is about doing many tasks at the same time, usually in some sort of highly coordinated manner. Parallelism requires concurrency but concurrency does not require parallelism. Concurrency is a way to model a solution be breaking it down into independent parts, and a concurrent application may be able to use parallelism (eg. multi-core processors) if the facility if the resources are available to do so.

As an example of a concurrent but non-parallel situation, consider the usual way of handling user input at a standard workstation. The keyboard handler and the mouse handler are concurrent, and occasionally coordinating their activities, but we do not require any sort of parallelism to implement this concurrent solution.

Concurrent applications can be implemented in a single CPU environment but parallel applications generally cannot be. Bottom line is Go is has built in concurrency which is independent of the underlying infrastructure and architecture.

11.1.3 Concurrency In Go

The model that Go uses to implement concurrency is based on an approach called CSP (communicating sequential processes) first proposed by Tony Hoare in 1978 and later implemented in several variants in Occam (1983) and Erlang (1986). The form of concurrency that Go uses is the latest in what Pike calls the Occam lineage.

Go implements concurrency through four mechanisms, three of which are supported by basic language features while the fourth is supported through the use of specific Go idioms or programming patterns.

1. Concurrent function execution implemented with goroutines.
2. Synchronization and communication between goroutines (channels).
3. Multi-way concurrent control (**select** statement).
4. Specific Go concurrency idioms.

The fourth point is a bit different that the other three since it refers to the ways in which Go programmers compose the three language constructs to produce elegant concurrent solutions. Generally we use the term *idiom* instead of pattern when we are speaking about a programming pattern within a specific programming language and reserve the word *pattern* for design solutions that are programming language independent.

The design goals of the implementation of Go concurrency are:

1. Make it very simple to code by having a lot of the low level synchronization handled by Go in the background.
2. Keep goroutines very lightweight, lighter than threads, so they are very efficient when it comes to resource utilization.
3. Keep the garbage collection managed by Go so that memory management does not have to be dealt with at the program level.

11.2 Goroutines

A goroutine is an independently executing function. It is not as heavyweight as a thread, in fact goroutines are multiplexed into threads and Go itself does the thread management. A single thread could possibly have thousands of goroutines multiplexed into it where each goroutine has its own dynamic stack which grows and shrinks dynamically as required. Goroutines can effectively be thought of as really cheap to use threads.

In every Go program, the **main()** function is by default a goroutine.

11.2.1 Creating Goroutines

Any function can be turned into a goroutine by use of the **go** operator. Consider first the non concurrent function in example 11.01 called “service” that does nothing in particular except print some stuff out. If we call this function in the usual synchronous manner, then the main function will wait for the service function to return before proceeding. Since the service function never returns because of the infinite loop, the program runs forever. Or at least in the listing below until the program was manually halted.

```
// Example 11-01 Normal function call
package main

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

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    service("Message:")
}
```

```
[Module11]$ go run ex11-01.go
Message: 0
Message: 1
Error: process crashed or was terminated while running.
```

In example 11-02, by using the go operator during the function call, the service() function now executes independently of the main function. What we have essentially done is turned what was a synchronous function call into an asynchronous function call. The side effect though is that the main function continues to execute and exits before the service() function can actually do anything. In Go when the main function exits, any goroutines spawned since the main function started are immediately terminated.

```
// Example 12-02 Goroutine function call
package main

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

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Message:")
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-02.go
Done!
```

In example 11-03, we put the main function to sleep for a few seconds to give the service() goroutine some time to actually execute.

```
// Example 11-03 Goroutine function call
package main

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

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Message:")
    time.Sleep(2 * time.Second)
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-03.go
Message: 0
Message: 1
Message: 2
Message: 3
Message: 4
Message: 5
Done!
```

In example 11-04, we make two go calls to service() and the output shows that they are in fact executing independently of each other with the output of the two running versions interleaved.

```
// Example 11-04 Multiple Goroutine function call
package main

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

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    go service("Alpha:")
    go service("Beta:")
    time.Sleep(2 * time.Second)
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-04.go
Alpha: 0
Beta: 0
Alpha: 1
Beta: 1
Alpha: 2
Beta: 2
Alpha: 3
Beta: 3
Alpha: 4
Beta: 4
Alpha: 5
Done!
```

11.2.2 Anonymous Goroutines

Often the task that we set out to run as a goroutine is quite simple and doesn't require a formal function definition. A very common Go idiom is to define the task with an anonymous function literal and set it running as a goroutine. You can think of this as delegating a task off to a goroutine and then forgetting about it.

This is demonstrated in example 11-05 by making the service function into an anonymous function. The final effect is the same although one way may be more natural to code than the other depending on the programming context.

```
// Example 11-05 Anonymous Goroutine function call
package main

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

func main() {
    go func() {
        for i := 0; ; i++ {
            fmt.Println("Anon:", i)
            time.Sleep(time.Duration(rand.Intn(1e3)) *
                                              time.Millisecond)
         }
    }()
    time.Sleep(2 * time.Second)
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-05.go
Anon: 0
Anon: 1
Anon: 2
Anon: 3
Anon: 4
Anon: 5
Done!
```

11.3 Channels.

So far we can create goroutines that all run independently, but in order to get true concurrency, we need to be able to compose these different goroutines into a logical structure which means they need to be able to communicate with each other and to coordinate and synchronize their activities.

This is done through something called a *channel*, which is a bit like a socket or a pipe in Unix. Channels in Go are bi-directional by default so they can be thought of as both a sink and a source of data.

A channel is a first class object in Go which means that we can use them in all the ways that we use variables in Go. But because we can pass channels as parameters and use them as return values, it means that like other first class objects such as variables, pointers and functions; channels have types which describes the type of data that can be sent over the channel.

11.3.1 Creating channels

The following are two equivalent ways to create an int channel called “c” which would be used by two goroutines to send integer “messages” back and forth . Recall from using make() that the variable c is actually a pointer to a channel, which means that the cost of passing it as a parameter is very low, like passing references to maps or slices.

```
var c chan int
c = make(chan int)

or

c := make(chan int)
```

11.3.2 Using Channels

The channel operator is the left pointing arrow < - which is used for both read from and write to a channel.

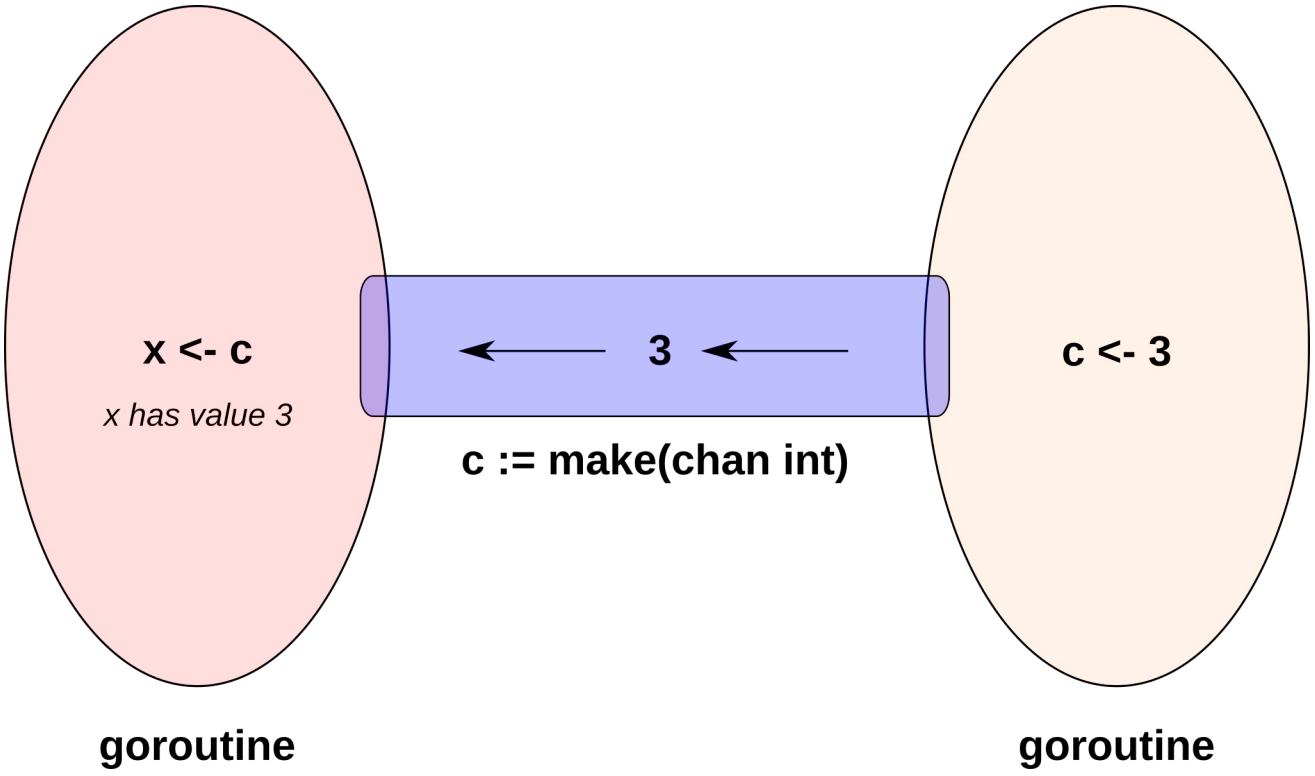
To write to the channel c we use the syntax

```
c <-  2
```

and to read from a channel c we use the syntax

```
myvariable <- c
```

In summary, when the channel appears on the left hand side of the arrow we are writing to the channel, and when it appears on the right hand side of the arrow, we are reading from the channel. This is illustrated in the diagram at the the top of the next page.



In the diagram above, a channel “c” is used to connect two goroutines. The value 3 is put onto the channel by the goroutine on the right then made available to goroutine on the left which then reads it and assigns it to the variable x.

Channels will block on either side if the channel transmission cannot be completed. In other words, if the goroutine on the right in the diagram above tries to write to the channel, it will block at the statement “c <-3 “ until the goroutine on the left executes the statement “x <- c”. And similarly if the goroutine on the left blocks when it tries to read until the go routine on the right writes to the channel. Both goroutines have to be at their corresponding channel i/o statements for the operation to complete, and each goroutine will wait for the other if necessary to get there.

This forces a synchronization to occur between the two goroutines at the point the channel is used.

The basic use of a channel is demonstrated in example 11-06. In this case the channel is created then passed as a parameter to the service() goroutine both the main and service() functions have access to the same channel.

```go
// Example 11-06 Basic channel
package main

import (
    "fmt"
    "time"
)

func service(message string, c chan string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}

func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-06.go
service:  0
service:  1
main 0 Got back: "Message:  0"
main 1 Got back: "Message:  1"
service:  2
main 2 Got back: "Message:  2"
service:  3
main 3 Got back: "Message:  3"
service:  4
main 4 Got back: "Message:  4"
service:  5
Done!
```

Ignoring the first iteration for now, notice that main function and service() function are now synchronized – the loops are iterating in lockstep with each other. One of the reasons why we are ignoring the first iteration is that the order of print statements may not be a totally accurate picture of the order of execution because of the time lag for the print statements to execute, but by slowing down the iterations with the wait function in the for loop in the main function, we can more or less corrects for the lag.

11.3.3 Deadlocks

Because of the unbuffered and synchronous behavior of channels, it is possible to deadlock a channel. Generally in Go having deadlocked channels is considered to be a symptom of poor program design.

Example 11-07 demonstrates an example of a deadlock can occur. The first action taken by both the main function and the service() function on the channel is to write to it. Both goroutines have written to the channel and are now waiting for the other goroutine to read it. At this point since no read can occur the program is deadlocked.

However Go monitors the status of all the active channels and when it detects a deadlock situation, it generates a panic an ends the program, as can be seen output of the example below.

```
// Example 11-07 Deadlocks

package main

import "fmt"

func service(c chan string) {
    c <- "Hello"     //write
    fmt.Println(<-c) // read
}

func main() {
    c := make(chan string)
    go service(c)
    c <- "ho"        // write
    fmt.Println(<-c) //read
}
```

```
[Module11]$ go run ex11-07.go
fatal error: all goroutines are asleep - deadlock!
```

11.3.4 Buffered Channels

It is possible to attach a buffer to to a channel which has the effect of creating an asynchronous channel. The value of a buffered channel comes when there is a mismatch between the rate of reading and writing of the two goroutines at either end of the channel. The use of a buffer decouples the read write operations of the goroutines when it may, from a program design perspective, be advantageous to reduce the synchronization between the goroutines.

Buffered channels are created by providing a capacity value to the make() operation. For example

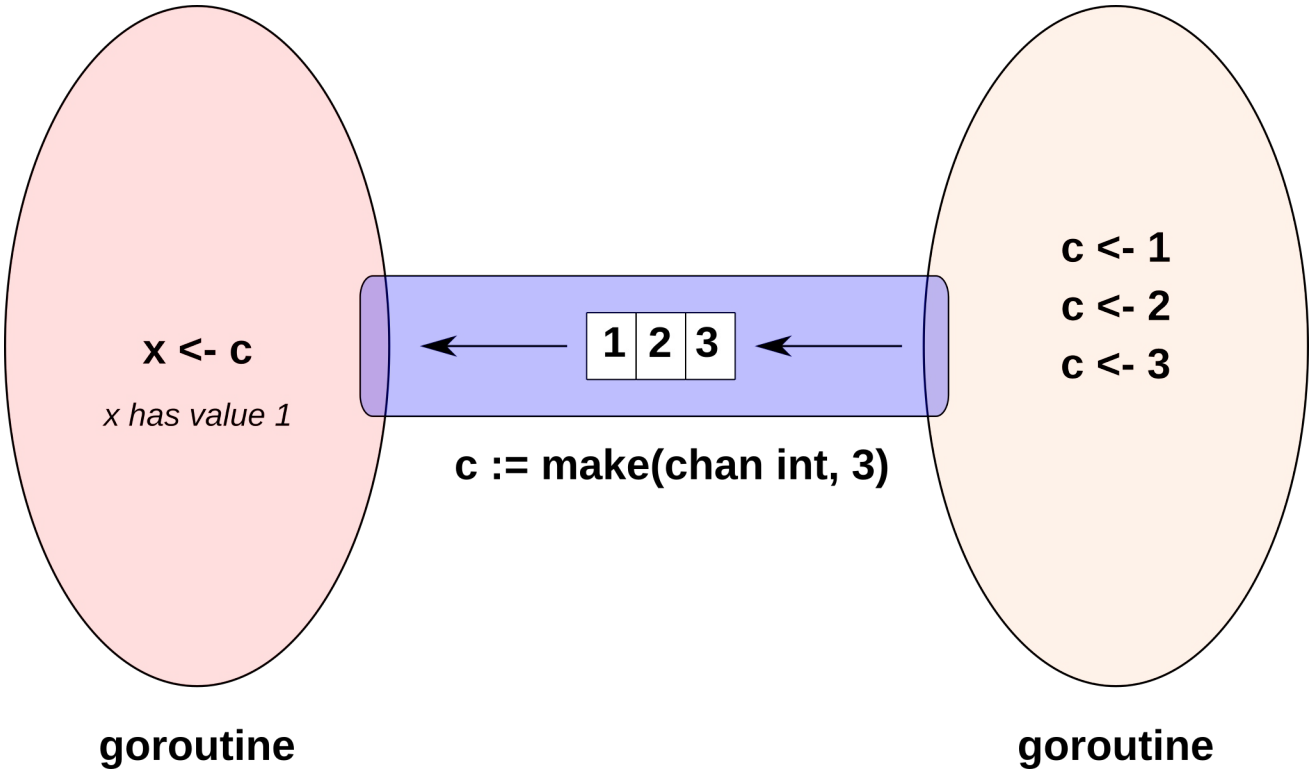
```
c: = make(chan int, 5)
```

makes an int channel with a buffer of size five.

The capacity of the buffer can be accessed with cap(buffer) and the current number of items in the buffer can accessed by using len(buffer).

Each write to a buffered channel pushes the item onto the end of the buffer unless the buffer is full, in which case it blocks until a slot in the buffer is available.

Each read from a buffered channel takes the item from the front of the buffer – the buffer is a FIFO queue. If there are no items in the buffer, then it behaves like an unbuffered channel and the attempt to read blocks until something is in the buffer.



```go
// Example 11-08 Buffered Channels

package main

import (
    "fmt"
    "time"
)

func service(message string, c chan string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i,"buffered items ",len(c)))
        c <- fmt.Sprintf("%s %d", message, i)
    }
}

func main() {
    c := make(chan string, 3)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-08.go
service:  0 buffered items  0
service:  1 buffered items  0
service:  2 buffered items  1
service:  3 buffered items  2
service:  4 buffered items  3
main 0 Got back: "Message:  0"
main 1 Got back: "Message:  1"
service:  5 buffered items  3
main 2 Got back: "Message:  2"
service:  6 buffered items  3
service:  7 buffered items  3
main 3 Got back: "Message:  3"
main 4 Got back: "Message:  4"
service:  8 buffered items  3
Done!
```

11.3.5 Unidirectional Channels

So far all of the channels we have seen have been bi-directional. However in many cases, the nature of the problem suggests that a goroutine only need to receive or send but not do both. This is very common in a pattern of goroutine composition called the pipeline where a goroutine will take inputs from one or more goroutines, perform some sort of operation on the inputs then send the results of the operation off to another set of one or more goroutines. In this case, only unidirectional message passing is required.

Channels themselves are not inherently unidirectional, obviously they cannot be since my receive channel has to be your send channel or the channel is useless. What I can specify that at your end, the channel is write only when I send the channel reference to you.

In example 11-09 the channel has been passed to the service goroutine as a write only channel The mnemonics chan<- and<-chan are used to specific that the channel is a write only or a read only channel. The only difference between this and the previous example is that the direction of the channel as passed to the goroutine is now specified.

```
// Example 11-09 Unidirectional channel
package main

import (
    "fmt"
    "time"
)

func service(message string, c chan<- string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}

func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

11.3.6 Closing a channel

The close operation on a channel is performed by a sender to indicate that no more data will be sent. If an attempt is made to write to a closed channel, then Go generates a panic. It is also a compile error to try and close a read only channel since the closing a channel indicates that there will be no more data written.

In example 11-10 we close the channel after we send on it, then because the loop continues, iteration 2 causes attempts to write on the closed channel and we get a panic.

```
// Example 11-10 Closing a channel
package main

import (
    "fmt"
    "time"
)

func service(message string, c chan<- string) {
    for i := 0; ; i++ {
        fmt.Println("service: ", i)
        c <- fmt.Sprintf("%s %d", message, i)
        close(c)  // will cause a panic next iteration
    }
}

func main() {
    c := make(chan string)
    go service("Message:", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("main %d Got back: %q\n", i, <-c)
        time.Sleep(time.Second)
    }
    fmt.Println("Done!")
}
```

```
[Module11]$ go run ex11-10.go
service:  0
service:  1
panic: send on closed channel
```

In example 11-11 we see these concepts combined in a more elegant concurrency pattern. A channel is created, it is passed to a "sender," a goroutine that generates output to the channel as a write channel, and it is also passed to a "receiver," a goroutine reads input from the same channel that the other goroutine writes to.

When the output is finished, the sender closes the output channel to indicate that they are finished sending. Meanwhile the receiver is using the range function to read continuously from the channel until the channel is closed on the other end. Once that happens, the receiver now reaches the end of the “range” of the channel and exits the for loop.

This is also an excellent example of the use of the range operation. A channel is essentially a sequence so the range operation can read from that sequence until it gets to the end of the sequence (i.e. when the channel is closed) and then we are done.

```
// Example 11-11 Unidirectional Channels
package main

import "fmt"
import "time"

func sender(output chan<- int) {
  for i := 0;    outputi < 3; i++ {
       output <- i
  }
  close(output)
}

func receiver(input <-chan int) {
  for j := range input {
       fmt.Println("Received ", j)

  }
  fmt.Println("I'm done ")
}

func main() {
  c := make(chan int)
  go receiver(c)
  go sender(c)
  time.Sleep(1 * time.Second)
}
```

```
[Module11]$ go run ex11-11.go
Received  0
Received  1
Received  2
I'm done
```

11.3.7 The Blocking Problem

In the previous example, what if the sender doesn't bother closing the channel? Then the receiver blocks. One of the best practices is to always start a goroutine using defer at the start of the goroutine that is responsible for closing a channel . Then if the goroutine exits for any reason, its output channel is closed and the rest of the goroutines that are waiting on input from it can stop waiting and proceed with their own execution.

If the range function is not the right tool for the job of checking to see if a channel is closed, then we can check explicitly to see if a channel is closed by using the comma ok idiom from before as demonstrated in example 11-12 where the receiver function is modified to use the ok comma idiom. If a channel is open, then ok is true but as soon as it is closed, then ok is false and we know not to continue listening on that channel.

```
// Example 11-12 Modification to receiver and sender

func sender(output chan<- int) {
  defer close(output)
  for i := 0; i < 3; i++ {
       output <- i
  }
}

func receiver(input <-chan int) {
  for {
       j, ok := <-input
       fmt.Println(ok)
       if !ok {
            break
       }
       fmt.Println("Received ", j)
  }
  fmt.Println("I'm done ")
}
```

```
[Module11]$ go run ex11-12.go
true
Received  0
true
Received  1
true
Received  2
false
I'm done
```

11.4 The Select Statement

The Go select statement looks like a switch statement, and in fact it is a lot like it, but it is a specialized polling mechanism that monitors a number of channels as a sort of multiplexed listener. This is critically important to developing concurrent applications because without the facility, we can find ourselves getting int all sorts of blocking issues that pretty much subvert the power of being able to use concurrency.

One of the common problems we need to solve in concurrency is the multiplex problem where a goroutine needs to read from several different sources and merge the inputs. As it stands, we cannot do this effectively since even using buffered channels our multiplexer could block if any one of the channels we are reading from blocks.

To allow effective multiplexing, Go uses the select statement which syntactically looks like a switch statement. Each case in the select statement is the input from a channel. Each time through the select statement, input is read from a channel that is ready. If more than one channel is ready, then one of those is picked at random. If none of the channels are ready then the default case, if present, executes.

In the example 11-13 there are two source goroutines to be multiplexed. One produces a stream of odd integers and the other a stream of even integers. Each one generates output on its channel at random intervals, just to make the problem interesting.

The select statement polls the two channels and when one is ready, reads from that channel and executes the case statement associated with it. If neither are ready, then the default case executes

```
// Example 11-13 Select Statement Multiplexer

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

func sourceEven(outchan chan<- int) {
  for i := 0; ; i++ {
       outchan <- i * 2
       time.Sleep(time.Duration(rand.Intn(1e3)) *
                                          time.Millisecond)
  }
}
func sourceOdd(outchan chan<- int) {
  for i := 0; ; i++ {
       outchan <- (i * 2) + 1
       time.Sleep(time.Duration(rand.Intn(1e3)) *
                                          time.Millisecond)
  }
}
func counter(ceven <-chan int, codd <-chan int) {
  for {
       select {
       case e := <-ceven:
            fmt.Println("Even:", e)
       case o := <-codd:
            fmt.Println("Odd:", o)
       default:
            fmt.Println("no one is ready")
            time.Sleep(100 * time.Millisecond)
       }
  }
}

func main() {
  codd := make(chan int)
  ceven := make(chan int)
  go sourceOdd(codd)
  go sourceEven(ceven)
  go counter(ceven, codd)
  time.Sleep(2 * time.Second)
}
```

```
[Module11]$ go run ex11-13.go
Even: 0
Odd: 1
no one is ready
Odd: 3
no one is ready
no one is ready
no one is ready
no one is ready
Even: 2
no one is ready
Odd: 5
Even: 4
no one is ready
Even: 6
no one is ready
no one is ready
no one is ready
Odd: 7
no one is ready
no one is ready
Even: 8
no one is ready
no one is ready
no one is ready
no one is ready
```

11.4 Race Conditions

All of the concurrent goroutines that have been presented so far are concurrency safe. However when two concurrent goroutines access a common resource, usually a variable or other kind of data structure, then we can have what is called a race condition. This is a common sort of problem in any concurrent system where accesses to a shared resource by concurrent processes may be interleaved in an unpredictable manner. Concurrency by its nature leads to race conditions since while we know that two concurrent goroutines will access a shared resource, we cannot predict in which order they will access it.

The classic example of a race condition is a shared variable update. The code in example 11-14 shows the sort of code that can lead to a data race condition – the data is read from a variable, updated then written back to the variable. This is a simplified version of something that can occur in a number of ways – access to a shared data structure via pointers, modifying part of a complex data structure and so on. What makes this clearly a problem is that between the original read of the data, other goroutines may change the value of the data before the the original reader does its update.

The problem with a data race condition is that it doesn't occur all of the time, it may take a particular combination of loading or timing for the race condition to occur, which means that it often will not be caught by standard testing.

The example 11-14 is rather artificial since it deliberately introduces a delay from the time the balance variable is accessed until it is updated to allow the data race condition to take place. This is our way of emulating some exceptional condition to create a data race problem. Without putting the first goroutine to sleep, the balance is updated as we expect it be, but when we add in the sleep condition then we have wrong result. If we actually ran the example without the sleep, it is doubtful that we would see an actual data race condition manifest itself.

In the real world, the sort of thing that the sleep condition represents may be anything from the time it takes data to transfer or network latency due to loading, or any other sort of condition that may affect the rate at which the concurrent routines work.

```
// Example 11-14 Race Condition
package main
import "fmt"
import "time"

var balance int

func update(amount int) {
  balance = amount
}
func query() int {
  return balance
}
func main() {
  go func() {
       bal := query()
       time.Sleep(time.Duration(time.Millisecond))
       update(bal + 100)
  }()
  go func() {
       bal := query()
       update(bal + 1)
  }()

  time.Sleep(time.Duration(time.Second))
  fmt.Println("Final Balance=", query())
}
```

```
[Module11]$ go run ex11-14.go
Final Balance= 100
```

There are two solutions to this problem, one is a design solution and the other involves introducing a lock on the variable like in other programming languages. The preferred solution is to think in terms of the concurrent design solution.

The core of the design solution is to let only one goroutine access the resource and have it take requests from other goroutines via channels to update the resource. This solution is more in keeping with the idea of concurrency through shared messaging not shared memory.

This sort of solution is intuitive since we often find that in the non-computer world we often restrict access to resources to “keepers of the resource” to whom we have to make requests. The example that comes to mind is when I was a graduate student working with historical documents. None of us were allowed to get or return the documents from the collection – we had to have the librarian do that for us. We can apply this pattern to rethink the sort of situation in example 11-14.

```go
// Example 11-15 Shared Resource CSP Style
package main

import "fmt"
import "time"

var deposit chan int
var query chan int

func accountUpdater(d <-chan int, q chan<- int) {
  balance := 0
  for {
       select {
       case amt := <-d:
            balance = balance + amt
       case q <- balance:

       }
  }
}

func accountRequester(amt int, d chan<- int) {
  for i := 0; i < 5; i++ {
       time.Sleep(time.Duration(time.Millisecond))
       d <- amt
  }

}

func main() {
  deposit := make(chan int)
  query := make(chan int)
  go accountUpdater(deposit, query)
  go accountRequester(100, deposit)
  go accountRequester(10, deposit)
  time.Sleep(time.Duration(time.Second))
  fmt.Println("Final Balance=", <-query)
}
```

```
[Module11]$ go run ex11-15.go
Final Balance= 550
```

11.4.1 Synchronization

Go also supports a variety of other more traditional methods of locking resources using mutex and semaphore types, as well as other standard concepts. For the purpose of this course, we will look at only one of these as an example of how they are used.

In example 11-16 we use some of the functionality from the sync package to lock the balance variable over a critical section of code. We start by defining a lock on the resource, in this case called bal_lock. When a goroutine locks the shared resource, no other goroutine can access the resource until the lock is released.

This is a more typical approach to synchronization as seen in many other programming languages that support concurrency, but getting into the standard sorts of models is bit out of scope of the course simply because it is a large and complex topic. While it is always preferable to design concurrency into a program like we did before because it usually tends to both produce a more elegant design but also one that does not depend on lower level constructs in the operational environment to support concurrency. However it is not always possible to build a concurrent design, or there may be other sorts of implementation issues that require us to use a more standard concurrency technique.

```go
// Example 11-16 Race Condition and Mutex
package main

import "fmt"
import "time"
import "sync"

var (
  bal_lock sync.Mutex
  balance  int
)

func update(amount int) {
  balance = amount
}
func query() int {
  return balance
}

func main() {
  go func() {
       bal_lock.Lock()
       bal := query()
       time.Sleep(time.Duration(time.Millisecond))
       update(bal + 100)
       bal_lock.Unlock()
  }()
  go func() {
       bal_lock.Lock()
       bal := query()
       update(bal + 1)
       bal_lock.Unlock()
  }()

  time.Sleep(time.Duration(time.Second))
  fmt.Println("Final Balance=", query())
}
```

```
[Module11]$ go run ex11-16.go
Final Balance= 110
```