12.15.2022 Go Lesson Instruction

1. **Goroutines**  
   Concurrency in Golang is the ability for functions to run independent of each other. Goroutines are functions that are run concurrently. Golang provides Goroutines as a way to handle operations concurrently.  
   New goroutines are created by the go statement.  
   To run a function as a goroutine, call that function prefixed with the go statement. Here is the example code block:  
     
   sum() // *A normal function call that executes sum synchrounsly and waits for completing it*

go sum() // *A goroutine that executes sum aschnronously and doesn’t wait for completing it.*

The go keyword makes the function call to return immediately, while the function starts running in the background as a goroutine and the rest of the program continues its execution. The **main** function of every Golang program is starts using a goroutine, so every Golang program runs as least one goroutine.

1. **Creating Goroutines**Added the gokeyword before each call of function **responseSize**. The three **responseSize** goroutines starts up concurrently and three calls to *http*. Get are made concurrently as well. The program doesn’t wait until one response comes back before sending out the next request. As a result the three response sizes are printed much sooner using goroutines.   
     
     
   After running below code. This instruction will be good.   
   We have added a call to *time.Sleep* in the **main** function which prevents the main goroutine from exiting before **responseSize** goroutines can finish. Calling time.Sleep(10\*time.Second) will make the **main** goroutine to sleep for 10 seconds.
2. package main
3. import (
4. "fmt"
5. "io/ioutil"
6. "log"
7. "net/http"
8. "time"
9. )
10. func responseSize(url string) {
11. fmt.Println("Step1: ", url)
12. fmt.Println("Step2: ", url)
13. fmt.Println("Step3: ", url)
14. fmt.Println("Step4: ", len(body))
15. }
16. func main() {
17. go responseSize("https://www.golangprograms.com")
18. go responseSize("https://coderwall.com")
19. go responseSize("https://stackoverflow.com")
20. time.Sleep(10 \* time.Second)
21. }

3. **Waiting for Goroutines to Finish Execution**

The WaitGroup type of sync package, is used to wait for the program to finish all goroutines launched from the main function. It uses a counter that specifies the number of goroutines, and Wait block the execution of the program until the WaitGroup counter is zero.  
  
The **Add** method is used to add a counter to the WaitGroup.  
The **Done** method of WaitGroup is scheduled using a defer statement to decrement the WaitGroup counter  
The **Wait** method of the WaitGroup type waits for the program to finish all gorutines.

The Wait method is called inside the main function, which blocks execution until the WaitGroup counter reaches the value of zero and ensures that all goroutines are executed.

package main

import (

    "fmt"

    "log"

    "net/http"

    "sync"

)

// WaitGroup is used to wait for the program to finish goroutines.

var wg sync.WaitGroup

func responseSize(url string) {

    // Schedule the call to WaitGroup's Done to tell goroutine is completed.

    defer wg.Done()

    fmt.Println("Step1: ", url)

    fmt.Println("Step2: ", url)

    fmt.Println("Step3: ", url)

    fmt.Println("Step4: ", len(body))

}

func main() {

    // Add a count of three, one for each goroutine.

    wg.Add(3)

    fmt.Println("Start Goroutines")

    go responseSize("https://www.golangprograms.com")

    go responseSize("https://stackoverflow.com")

    go responseSize("https://coderwall.com")

    // Wait for the goroutines to finish.

    wg.Wait()

    fmt.Println("Terminating Program")

}

4. **Working with Channels**Go provides a mechanism called a channel that is used to share data between goroutines. When you execute a concurrent activity as a goroutine a resource or data needs to be shared between goroutines, channels act as a pipe between the goroutines and provide a mechanism that guarantees a synchronous exchange.  
  
Data type need to be specified at the time of declaration of channel. We can share Values and pointer of built-in, named, struct, and reference types. Data are passed around on channels: only one goroutines has access to a data item at any given time: so data races cannot occur, by design.  
  
There are two types of channels based on their behavior of data exchange: **unbuffered** channels and **buffered** channels. An unbuffered channel is used to perform synchronous communication between goroutines while a buffered channel is used for perform asynchronous communication. An unbuffered channel provides a guarantee that an exchange between two goroutines is performed at the instant the send and receive take place. A buffered channel has no such guarantee.   
  
**A channel is created by the make function, which specifies the chan keyword and a channel’s element type.**Here is the code block that creates an unbuffered and buffered channel:

Unbuffered := make(chan int)

Buffered := make(chan int, 10)