Golang Session

- Harsh Dusane

Topic : Working with Channels

Why channel is needed?

- 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 conduit(pipe) between the goroutines and provide a mechanism that guarantees a synchronous exchange.

Types of Channels

- 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.

```
Unbuffered := make(chan int) // Unbuffered channel of integer type
buffered := make(chan int, 10)  // Buffered channel of integer type
```

Sending and Receiving data from channel

- goroutine1 := make(chan string, 5) // Buffered channel of strings.
- goroutine1 <- "Australia" // Send a string through the channel.
- data := <-goroutine1 // Receive a string from the channel.

Unbuffered channels

- In unbuffered channel there is no capacity to hold any value before it's received. In this type of channels both a sending and receiving goroutine to be ready at the same instant before any send or receive operation can complete. If the two goroutines aren't ready at the same instant, the channel makes the goroutine that performs its respective send or receive operation first wait. Synchronization is fundamental in the interaction between the send and receive on the channel. One can't happen without the other.

Buffered channels

- In buffered channel there is a capacity to hold one or more values before they're received. In this types of channels don't force goroutines to be ready at the same instant to perform sends and receives. There are also different conditions for when a send or receive does block. A receive will block only if there's no value in the channel to receive. A send will block only if there's no available buffer to place the value being sent.

Sample code

```
// Simple program to demonstrate use of Buffered Channel
package main
import (
	"fmt"
	"math/rand"
	"sync"
	"time"
)
var goRoutine sync.WaitGroup
func main(){
	rand.Seed(time.Now().Unix())
	// Create a buffered channel to manage the
employee vs project load.
	projects := make(chan string,10)
	// Launch 5 goroutines to handle the projects.
	goRoutine.Add(5)
	for i :=1; i <= 5; i++ {
		go employee(projects, i)
	}
	for j :=1; j <= 10; j++ {
		projects <- fmt.Sprintf("Project :%d", j)
	}
	// Close the channel so the goroutines will quit

	close(projects)
	goRoutine.Wait()
}
```

```
func employee(projects chan string, employee int) {
	defer goRoutine.Done()
	for {
		// Wait for project to be assigned.
		project, result := <-projects
		if result==false {
			// This means the channel is empty and
closed.
			fmt.Printf("Employee : %d : Exit\n",
employee)
			return
		}
		fmt.Printf("Employee : %d : Started   %s\n",
employee, project)
		// Randomly wait to simulate work time.
		sleep := rand.Int63n(50)
		time.Sleep(time.Duration(sleep) *
time.Millisecond)
		// Display time to wait
		fmt.Println("\nTime to sleep",sleep,"ms\n")
		// Display project completed by employee.
		fmt.Printf("Employee : %d : Completed %s\n",
employee, project)
	}
}
```