Buffered Channels

This lesson will introduce you to a type of channels known as Buffered Channels.

WE'LL COVER THE FOLLOWING

- Creating A Buffered Channel

If you were able to understand the previous lesson, you should be more clear on the code provided in the lesson on deadlocks.

```
package main
import "fmt"

func main() {
        mychannel := make(chan int)
        mychannel <- 10
        fmt.Println(<-mychannel)
}
```

Now you'll be fully clear on why the above code didn't work. This is because of the sending and receiving operations which are blocking the code. When we wrap one of them in a goroutine such that they are ready to unblock each other, the program executes successfully.

```
package main
import "fmt"

func main() {
        mychannel := make(chan int)
        go func(){
                mychannel <- 10
        }()
        fmt.Println(<-mychannel)
}
```

Alternatively, we can also use **buffered channels** to solve this issue.

Creating A Buffered Channel

Buffered Channels are channels with a capacity/buffer. They can be created with the following syntax:

```
channelName := make(chan datatype, capacity)
```

Initializing A Buffered Channel

Let's use them in an example:

```
package main
import "fmt"

func main() {
        mychannel := make(chan int, 2)
        mychannel <- 10
        fmt.Println(<-mychannel)
}
```

As you can see, we don't get a deadlock as before. This is because the send operation on **line 6** does not block the code until we have reached our capacity. Have a look at the code below:

```
package main
import "fmt"

func main() {
        mychannel := make(chan int, 2)
        mychannel <- 10
        mychannel <- 20
        mychannel <- 30
        fmt.Println(<-mychannel)
}
```

The code above generates an error as `mychannel` has a capacity of `2` and we

already sent the data two times over the channel. Hence `mychannel <- 30` on

line 8 blocks the code resulting in a deadlock because it is our third attempt to send data over the channel which has a capacity of `2`.

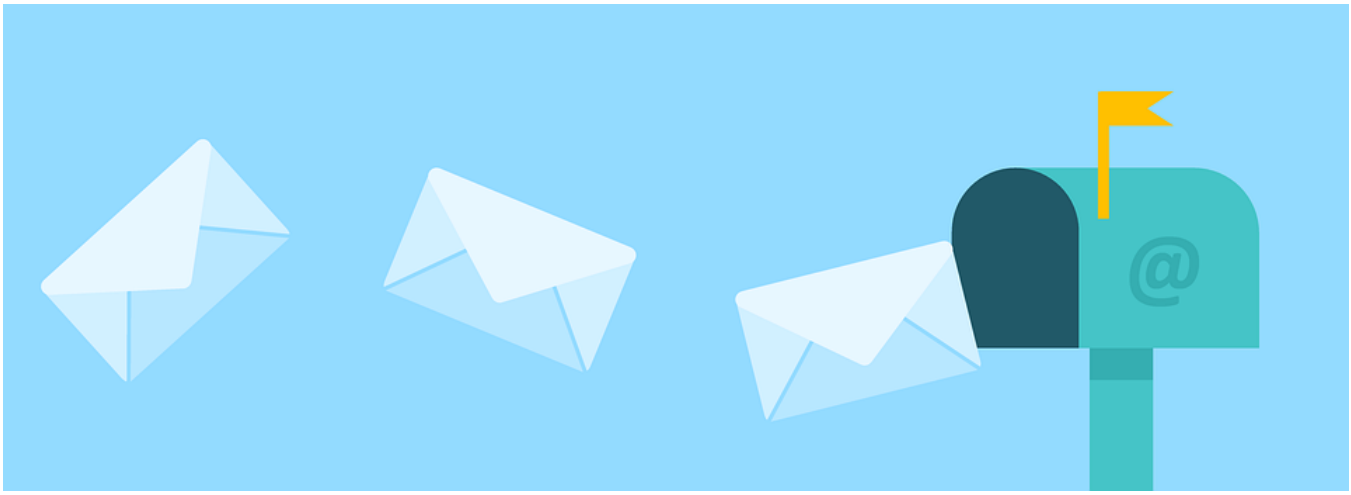
Also, note that the receive operation will block the code if it finds the buffer to be empty. Check out the code given below which also generates an error pointing to a deadlock:

```
package main
import "fmt"

func main() {
        mychannel := make(chan int, 2)

        fmt.Println(<-mychannel)
}
```

In conclusion, we can say that if there are no receive operations for a channel, a goroutine can still perform ***c*** sending operations, where ***c*** is the capacity of the buffered channel. So you can see that buffered channels remove synchronization. They work in a way similar to mailboxes and their usage depends on the type of problem you have to solve.



Now that we have learned about both the unbuffered and buffered channels, it's time for a challenge. Good luck for the next lesson!