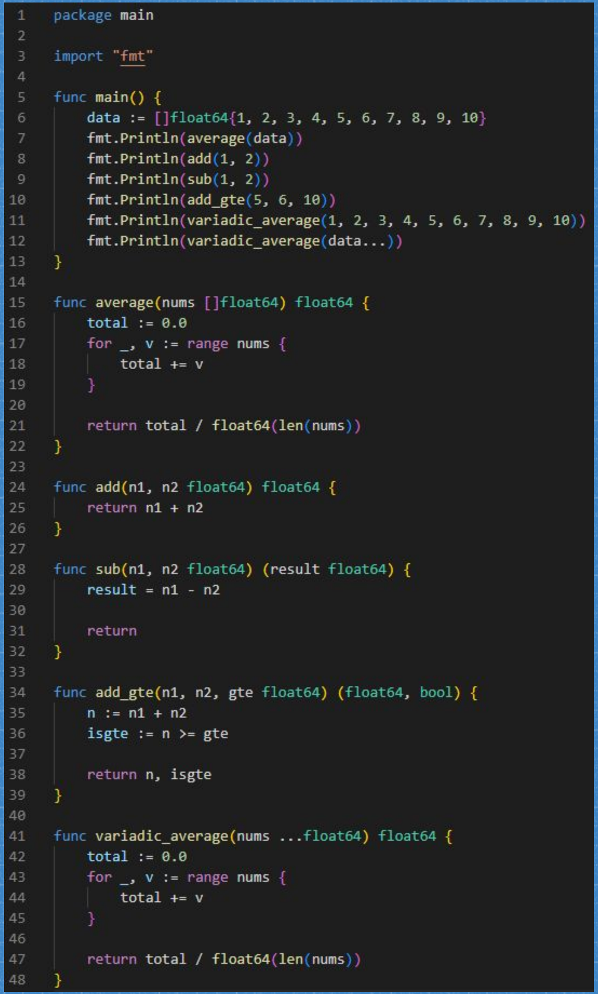
Golang
Session 2

In this session

We’ll learn about:

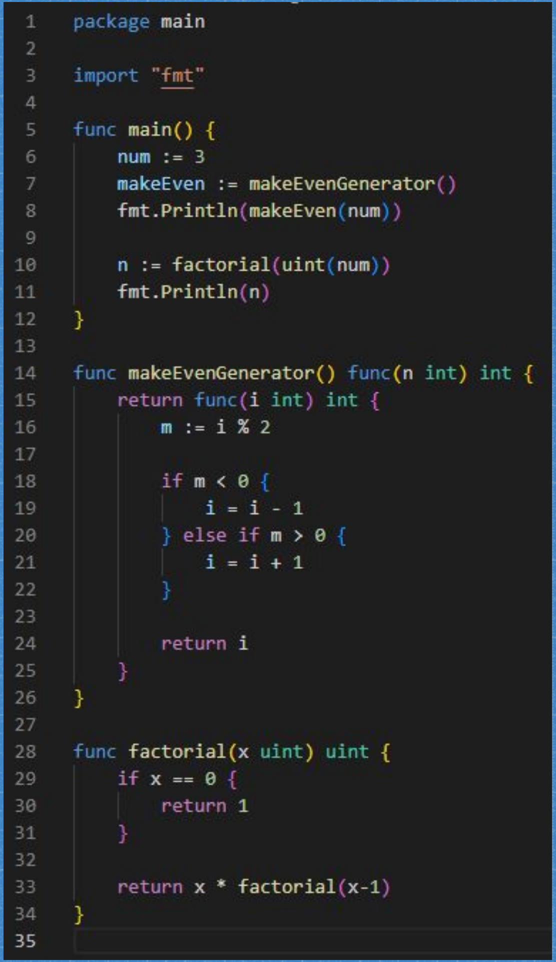
- Functions
- Pointers
- Structs & Interfaces
- Concurrency
- Packages
- Tests

Functions



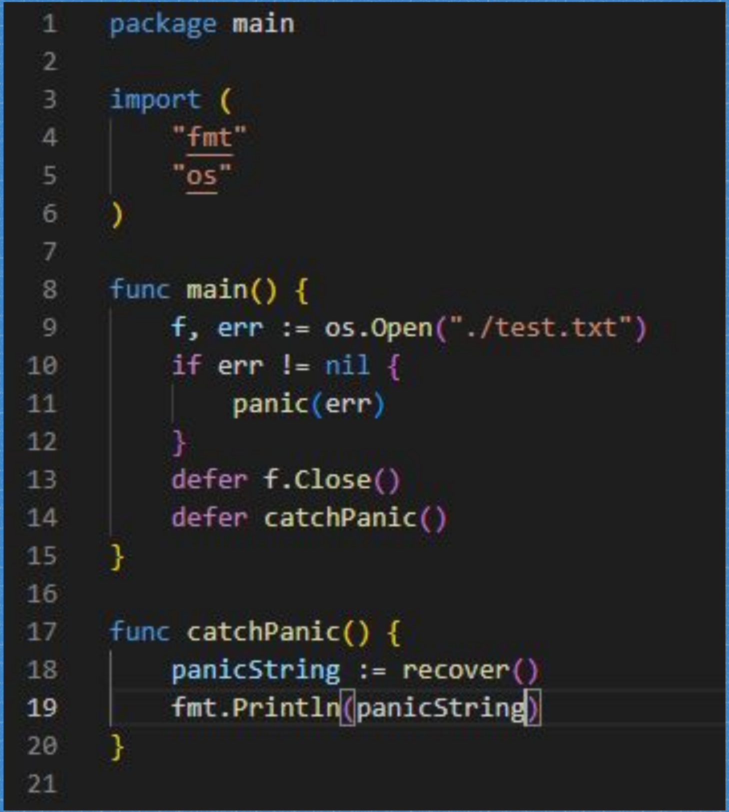
```go
package main

import "fmt"

func main() {
    data := []float64{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    fmt.Println(average(data))
    fmt.Println(add(1, 2))
    fmt.Println(sub(1, 2))
    fmt.Println(add_gte(5, 6, 10))
    fmt.Println(variadic_average(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
    fmt.Println(variadic_average(data...))
}

func average(nums []float64) float64 {
    total := 0.0
    for _, v := range nums {
        total += v
    }

    return total / float64(len(nums))
}

func add(n1, n2 float64) float64 {
    return n1 + n2
}

func sub(n1, n2 float64) (result float64) {
    result = n1 - n2

    return
}

func add_gte(n1, n2, gte float64) (float64, bool) {
    n := n1 + n2
    isgte := n >= gte

    return n, isgte
}

func variadic_average(nums ...float64) float64 {
    total := 0.0
    for _, v := range nums {
        total += v
    }

    return total / float64(len(nums))
}
```

Functions: Closure & Recursion



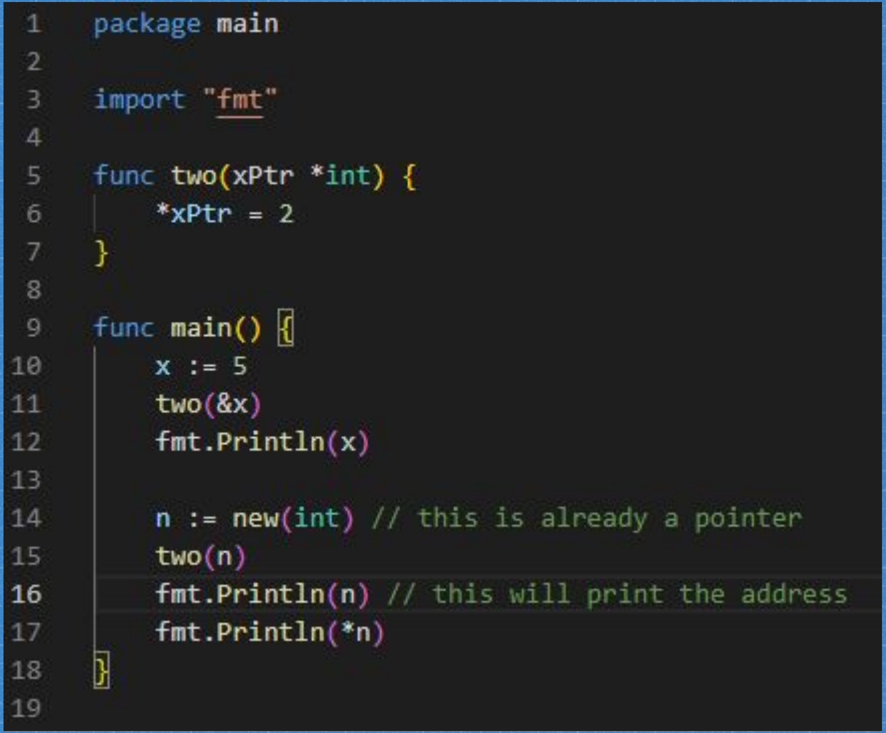
```go
package main

import "fmt"

func main() {
	num := 3
	makeEven := makeEvenGenerator()
	fmt.Println(makeEven(num))

	n := factorial(uint(num))
	fmt.Println(n)
}

func makeEvenGenerator() func(n int) int {
	return func(i int) int {
		m := i % 2

		if m < 0 {
			i = i - 1
		} else if m > 0 {
			i = i + 1
		}

		return i
	}
}

func factorial(x uint) uint {
	if x == 0 {
		return 1
	}

	return x * factorial(x-1)
}
```

Functions: Defer, Panic, & Recover



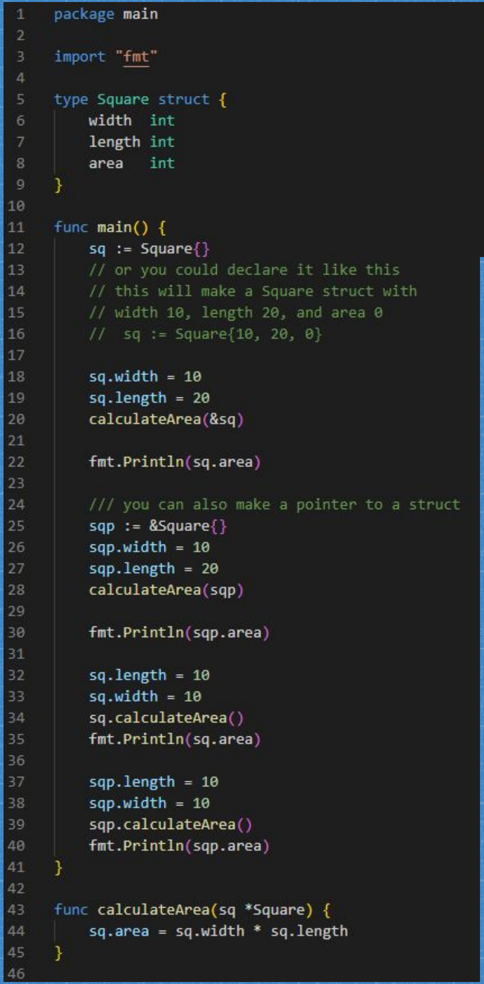
```go
package main

import (
	"fmt"
	"os"
)

func main() {
	f, err := os.Open("./test.txt")
	if err != nil {
		panic(err)
	}
	defer f.Close()
	defer catchPanic()
}

func catchPanic() {
	panicString := recover()
	fmt.Println(panicString)
}

```

Pointers

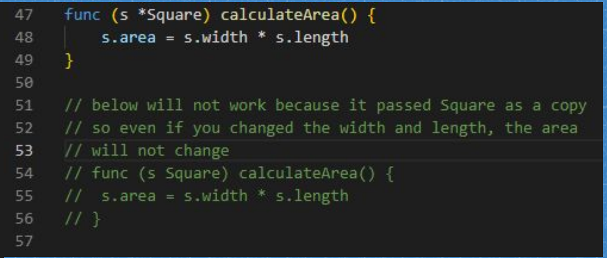


```go
package main

import "fmt"

func two(xPtr *int) {
	*xPtr = 2
}

func main() {
	x := 5
	two(&x)
	fmt.Println(x)

	n := new(int) // this is already a pointer
	two(n)
	fmt.Println(n) // this will print the address
	fmt.Println(*n)
}

```

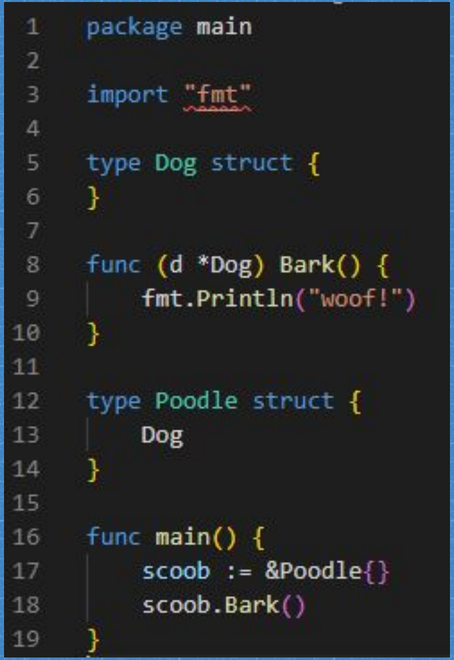
Structs



```go
package main

import "fmt"

type Square struct {
	width  int
	length int
	area   int
}

func main() {
	sq := Square{}
	// or you could declare it like this
	// this will make a Square struct with
	// width 10, length 20, and area 0
	//  sq := Square{10, 20, 0}

	sq.width = 10
	sq.length = 20
	calculateArea(&sq)

	fmt.Println(sq.area)

	/// you can also make a pointer to a struct
	sqp := &Square{}
	sqp.width = 8
	sqp.length = 20
	calculateArea(sqp)

	fmt.Println(sqp.area)

	sq.length = 10
	sq.width = 10
	sq.calculateArea()
	fmt.Println(sq.area)

	sqp.length = 10
	sqp.width = 10
	sqp.calculateArea()
	fmt.Println(sqp.area)
}

func calculateArea(sq *Square) {
	sq.area = sq.width * sq.length
}

```

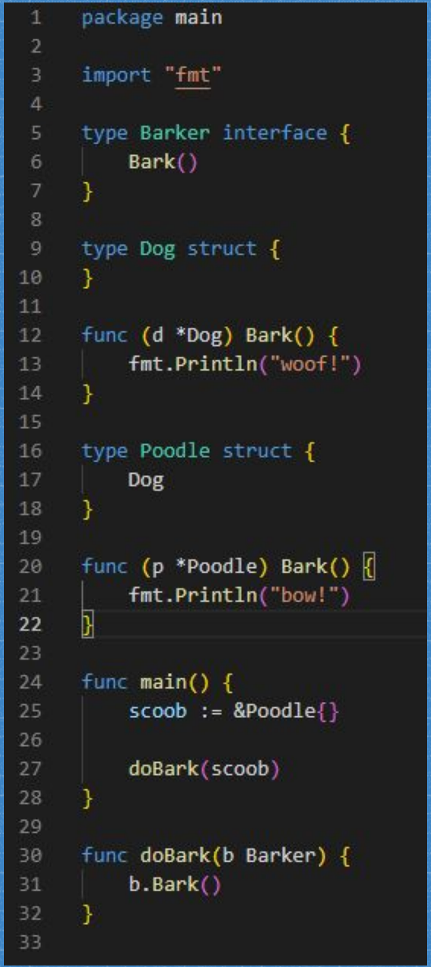


```go
func (s *Square) calculateArea() {
	s.area = s.width * s.length
}

// below will not work because it passed Square as a copy
// so even if you changed the width and length, the area
// will not change
// func (s Square) calculateArea() {
//  s.area = s.width * s.length
// }

```



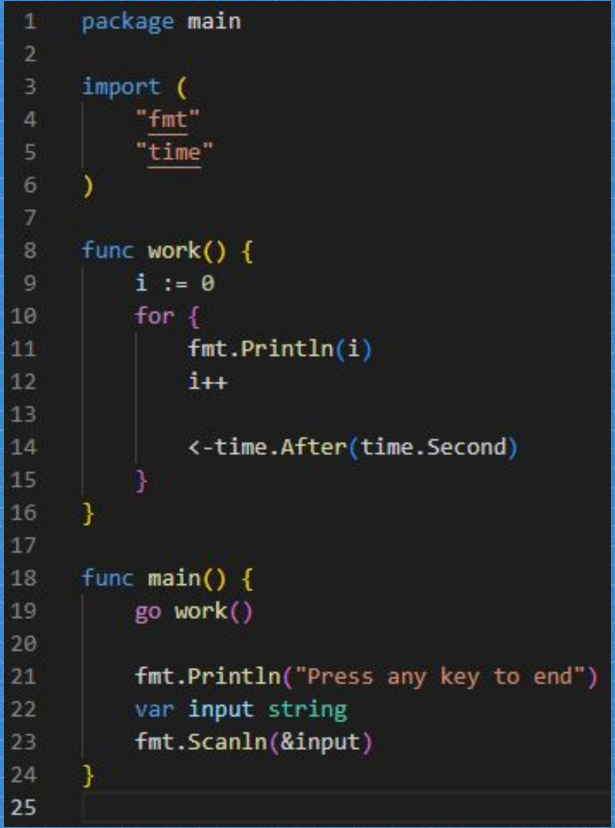
```go
package main

import "fmt"

type Dog struct {
}

func (d *Dog) Bark() {
	fmt.Println("woof!")
}

type Poodle struct {
	Dog
}

func main() {
	scoob := &Poodle{}
	scoob.Bark()
}
```

Interfaces



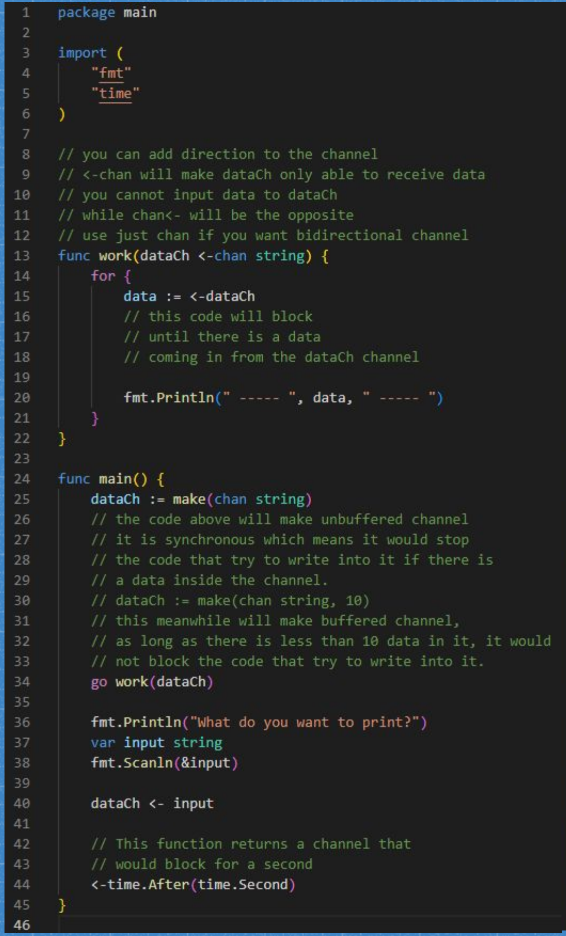
```go
package main

import "fmt"

type Barker interface {
    Bark()
}

type Dog struct {
}

func (d *Dog) Bark() {
    fmt.Println("woof!")
}

type Poodle struct {
    Dog
}

func (p *Poodle) Bark() {
    fmt.Println("bow!")
}

func main() {
    scoob := &Poodle{}

    doBark(scoob)
}

func doBark(b Barker) {
    b.Bark()
}

```

Concurrency - Goroutines

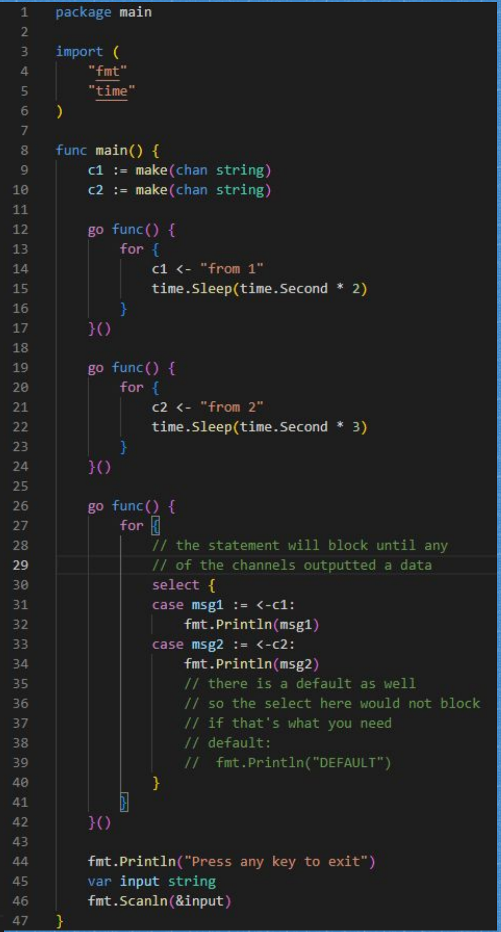


```go
package main

import (
	"fmt"
	"time"
)

func work() {
	i := 0
	for {
		fmt.Println(i)
		i++

		<-time.After(time.Second)
	}
}

func main() {
	go work()

	fmt.Println("Press any key to end")
	var input string
	fmt.Scanln(&input)
}

```

Concurrency - Channels

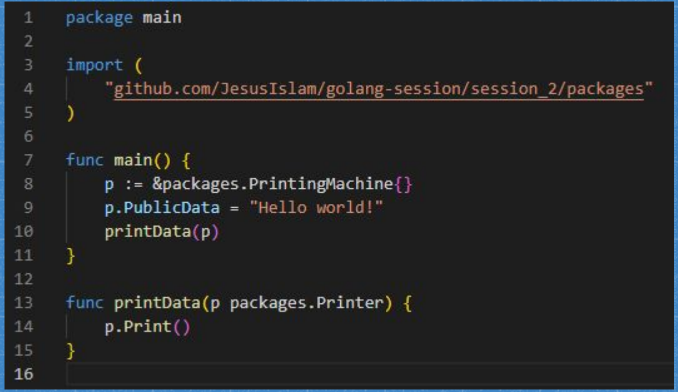


```go
package main

import (
    "fmt"
    "time"
)

// you can add direction to the channel
// <-chan will make dataCh only able to receive data
// you cannot input data to dataCh
// while chan<- will be the opposite
// use just chan if you want bidirectional channel
func work(dataCh <-chan string) {
    for {
        data := <-dataCh
        // this code will block
        // until there is a data
        // coming in from the dataCh channel

        fmt.Println(" ----- ", data, " ----- ")
    }
}

func main() {
    dataCh := make(chan string)
    // the code above will make unbuffered channel
    // it is synchronous which means it would stop
    // the code that try to write into it if there is
    // a data inside the channel.
    // dataCh := make(chan string, 10)
    // this meanwhile will make buffered channel,
    // as long as there is less than 10 data in it, it would
    // not block the code that try to write into it.
    go work(dataCh)

    fmt.Println("What do you want to print?")
    var input string
    fmt.Scanln(&input)

    dataCh <- input

    // This function returns a channel that
    // would block for a second
    <-time.After(time.Second)
}

```

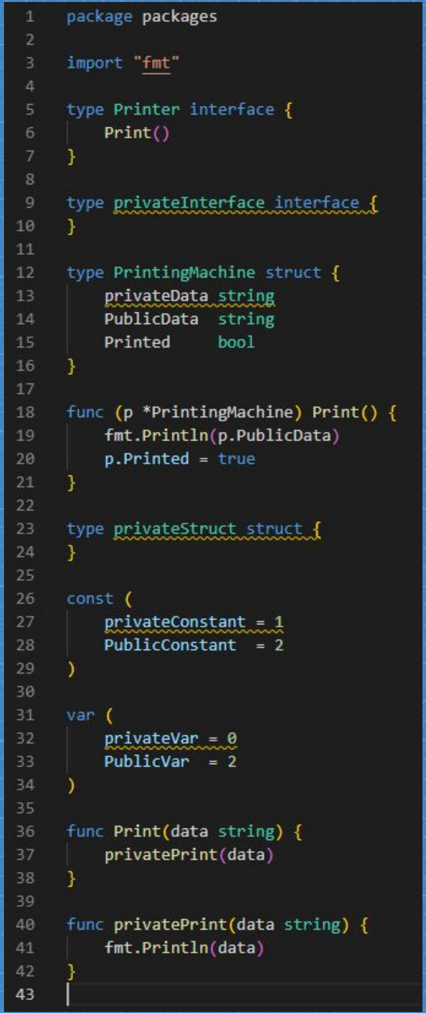


```go
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        for {
            c1 <- "from 1"
            time.Sleep(time.Second * 2)
        }
    }()

    go func() {
        for {
            c2 <- "from 2"
            time.Sleep(time.Second * 3)
        }
    }()

    go func() {
        for {
            // the statement will block until any
            // of the channels outputted a data
            select {
            case msg1 := <-c1:
                fmt.Println(msg1)
            case msg2 := <-c2:
                fmt.Println(msg2)
                // there is a default as well
                // so the select here would not block
                // if that's what you need
                // default:
                //  fmt.Println("DEFAULT")
            }
        }
    }()

    fmt.Println("Press any key to exit")
    var input string
    fmt.Scanln(&input)
}
```

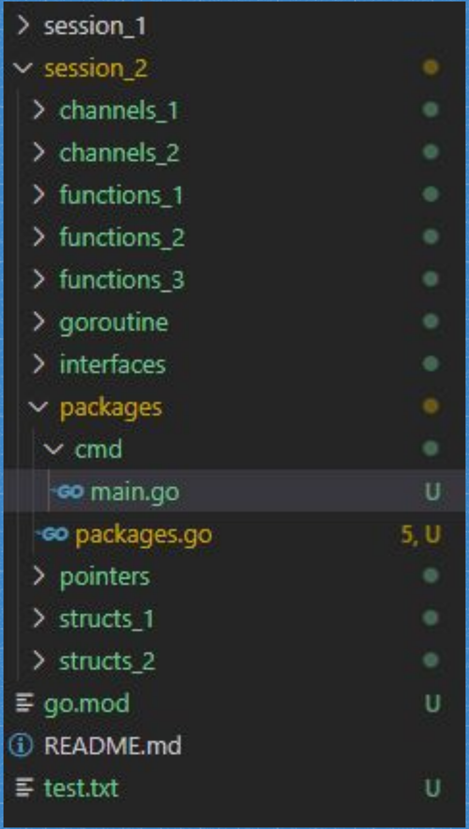
Packages



```go
package main

import (
	"github.com/JesusIslam/golang-session/session_2/packages"
)

func main() {
	p := &packages.PrintingMachine{}
	p.PublicData = "Hello world!"
	printData(p)
}

func printData(p packages.Printer) {
	p.Print()
}

```

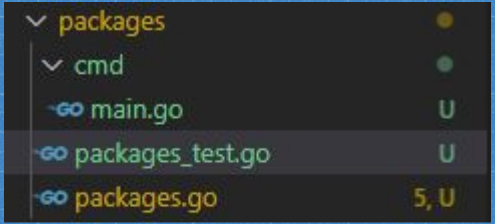


```go
package packages

import "fmt"

type Printer interface {
	Print()
}

type privateInterface interface {
}

type PrintingMachine struct {
	privateData string
	PublicData  string
	Printed     bool
}

func (p *PrintingMachine) Print() {
	fmt.Println(p.PublicData)
	p.Printed = true
}

type privateStruct struct {
}

const (
	privateConstant = 1
	PublicConstant  = 2
)

var (
	privateVar = 0
	PublicVar  = 2
)

func Print(data string) {
	privatePrint(data)
}

func privatePrint(data string) {
	fmt.Println(data)
}

```

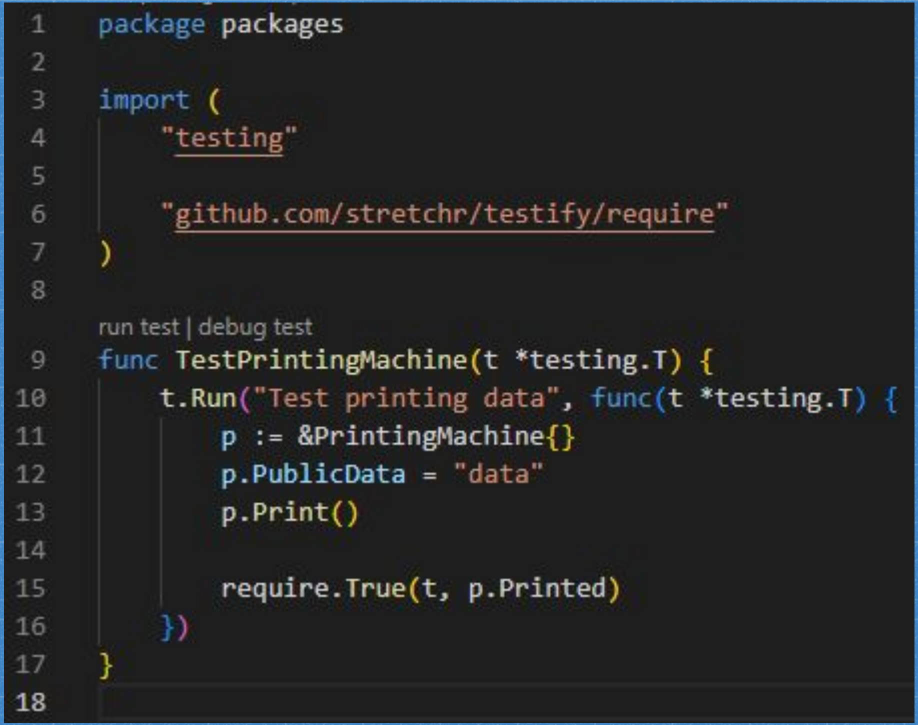


```
> session_1
v session_2
  > channels_1
  > channels_2
  > functions_1
  > functions_2
  > functions_3
  > goroutine
  > interfaces
  v packages
    v cmd
      main.go                U
    packages.go           5, U
  > pointers
  > structs_1
  > structs_2
go.mod                       U
README.md
test.txt                     U
```

Tests



```
packages
  cmd
    main.go                U
  packages_test.go         U
  packages.go           5, U
```



```go
package packages

import (
	"testing"

	"github.com/stretchr/testify/require"
)

run test | debug test
func TestPrintingMachine(t *testing.T) {
	t.Run("Test printing data", func(t *testing.T) {
		p := &PrintingMachine{}
		p.PublicData = "data"
		p.Print()

		require.True(t, p.Printed)
	})
}

```

Thanks!

ANY QUESTIONS?

You can find me at:

andputra@alamisharia.co.id

github.com/JesusIslam