**Declare channels:**

func main(){

var ch1 chan int

fmt.Println("channel with 'var' keyword:", ch1)

ch2 := make(chan string)

fmt.Println("channel with 'make' method:", ch2)

}

**Send operation with channels:**

func main(){

ch := make(chan int)

ch <- 8

fmt.Println(ch)

}

**Receive operation with channels:**

package main

import "fmt"

func show\_value(c chan int){

c <- 8

}

func main(){

c := make(chan int)

go show\_value(c)

a, ok := <-c

fmt.Println("value received from channel:", a)

fmt.Println("status of operation is:", ok)

}

func show\_value(c chan int){

c <- 8

}

func main(){

var c chan int

go show\_value(c)

a, ok := <-c

fmt.Println("value received from channel:", a)

fmt.Println("status of operation is:", ok)

}

**Buffered channels:**

func main(){

ch := make(chan int, 2)

ch <- 8

a := <-ch

fmt.Println(ch, a)

}

func main(){

ch := make(chan int, 2)

ch <- 8

ch <- 99

a := <-ch

b := <- ch

fmt.Println(a, b)

}

**Find capacity of channels:**

func main(){

ch := make(chan int, 2)

cap\_ch := cap(ch)

fmt.Println("Capacity of channel is:", cap\_ch)

for itr := 1; itr<=cap\_ch; itr++{

ch <- itr\*2

fmt.Println("value received from channel is:", <-ch)

}

}

**Deadlock:**

package main

import "fmt"

func show(){

fmt.Println("Hello")

}

func main(){

str\_ch := make(chan string)

go show()

str\_ch <- "Hello world"

fmt.Println("Send operation is getting blocked")

}

package main

import "fmt"

func show(){

fmt.Println("Hello world")

}

func main(){

str\_ch := make(chan string)

go show()

a := <- str\_ch

fmt.Println("Receive operation is getting blocked", a)

}

**Use multiple goroutines:**

package main

import "fmt"

func square(input chan int, output chan int) {

value := <-input

output <- value \* value

}

func cube(input chan int, output chan int) {

value := <-input

output <- value \* value \* value

}

func main(){

squareInput := make(chan int)

cubeInput := make(chan int)

squareOutput := make(chan int)

cubeOutput := make(chan int)

go square(squareInput, squareOutput)

go cube(cubeInput, cubeOutput)

squareInput <- 3 // Send to square

sqr\_val := <-squareOutput // Receive from square

cubeInput <- 3 // Send to cube

cube\_val := <-cubeOutput // Receive from cube

fmt.Println("Square of value:", sqr\_val) // 8

fmt.Println("Cube of value:", cube\_val) // 9

}

**Closing channels:**

package main

import "fmt"

func cube(value chan int){

fmt.Println("In cube goroutine")

val := <-value

value <- val\*val\*val

}

func square(value chan int){

fmt.Println("In square goroutine")

val := <-value

fmt.Println(val)

value <- val\*val

}

func main(){

num := make(chan int)

go square(num)

num <- 3

sqr\_val := <-num

go cube(num)

num <- 3

close(num)

cube\_val, ok := <-num

fmt.Println("Square of value:", sqr\_val)

fmt.Println("Cube of value:", cube\_val, ok)

}

**Direction of a channel:**

package main

import ("fmt"

"sync"

)

var wg sync.WaitGroup

func square(value chan<- int){

fmt.Println("defining send only goroutine")

val := 3

fmt.Println(val\*val)

wg.Done()

}

func main(){

send\_ch := make(chan<- int, 1)

wg.Add(1)

go square(send\_ch)

send\_ch <- 3

wg.Wait()

}

package main

import ("fmt"

"sync"

)

var wg sync.WaitGroup

func square(value chan<- int){

fmt.Println("defining send only goroutine")

val := value

fmt.Println(val)

val <- value

wg.Done()

}

func main(){

send\_ch := make(chan<- int, 1)

wg.Add(1)

go square(send\_ch)

send\_ch <- 3

wg.Wait()

}

package main

import ("fmt"

"sync"

)

var wg sync.WaitGroup

func square(value <-chan int){

fmt.Println("defining receive only goroutine")

val := <-value

fmt.Println(val)

wg.Done()

}

func main(){

receive\_ch := make(<-chan int)

wg.Add(1)

go square(receive\_ch)

wg.Wait()

}

package main

import "fmt"

func receive(receive\_ch chan<- int, val int) {

receive\_ch <- val

}

func square(receive\_ch <-chan int, send\_ch chan<- int){

fmt.Println("defining send only goroutine")

val := <- receive\_ch

send\_ch <- val

}

func main(){

send\_ch := make(chan int, 1)

receive\_ch := make(chan int,1)

receive(receive\_ch, 6)

square(receive\_ch, send\_ch)

fmt.Println(<-send\_ch)

}

**Select keyword:**

package main

import ("fmt"

"time"

)

func square(num\_ch chan int){

value := <-num\_ch

sqr\_val := value\*value

time.Sleep(5\*time.Second)

num\_ch <- sqr\_val

}

func cube(num\_ch chan int){

value := <-num\_ch

cube\_val := value\*value\*value

time.Sleep(10\*time.Second)

num\_ch <- cube\_val

}

func main(){

sqr\_ch := make(chan int)

go square(sqr\_ch)

sqr\_ch <- 3

cube\_ch := make(chan int)

go cube(cube\_ch)

cube\_ch <- 5

select{

case sqr\_val := <-sqr\_ch:

fmt.Println("square of a value is:", sqr\_val)

case cube\_val := <-cube\_ch:

fmt.Println("Cube of a value is:", cube\_val)

}

}

package main

import ("fmt"

"time"

)

func square(num\_ch chan int){

value := <-num\_ch

sqr\_val := value\*value

time.Sleep(1\*time.Second)

num\_ch <- sqr\_val

}

func cube(num\_ch chan int){

value := <-num\_ch

cube\_val := value\*value\*value

time.Sleep(1\*time.Second)

num\_ch <- cube\_val

}

func main(){

sqr\_ch := make(chan int)

go square(sqr\_ch)

sqr\_ch <- 3

cube\_ch := make(chan int)

go cube(cube\_ch)

cube\_ch <- 5

select{

case sqr\_val := <-sqr\_ch:

fmt.Println("square of a value is:", sqr\_val)

case cube\_val := <-cube\_ch:

fmt.Println("Cube of a value is:", cube\_val)

}

}

package main

import ("fmt"

"time"

)

func square(num\_ch chan int){

\_ = <-num\_ch

time.Sleep(1\*time.Second)

}

func cube(num\_ch chan int){

\_ = <-num\_ch

time.Sleep(1\*time.Second)

}

func main(){

sqr\_ch := make(chan int)

go square(sqr\_ch)

sqr\_ch <- 3

cube\_ch := make(chan int)

go cube(cube\_ch)

cube\_ch <- 5

select{

case sqr\_val := <-sqr\_ch:

fmt.Println("square of a value is:", sqr\_val)

case cube\_val := <-cube\_ch:

fmt.Println("Cube of a value is:", cube\_val)

}

}

package main

import ("fmt"

"time"

)

func square(num\_ch chan int){

\_ = <-num\_ch

time.Sleep(1\*time.Second)

}

func cube(num\_ch chan int){

\_ = <-num\_ch

time.Sleep(1\*time.Second)

}

func main(){

sqr\_ch := make(chan int)

go square(sqr\_ch)

sqr\_ch <- 3

cube\_ch := make(chan int)

go cube(cube\_ch)

cube\_ch <- 5

select{

case sqr\_val := <-sqr\_ch:

fmt.Println("square of a value is:", sqr\_val)

case cube\_val := <-cube\_ch:

fmt.Println("Cube of a value is:", cube\_val)

default:

fmt.Println("No case statement is ready to execute!")

}

}

func main(){

select{

}

}

package main

import ("fmt"

"time"

)

func square(num\_ch chan int){

value := <-num\_ch

sqr\_val := value\*value

time.Sleep(5\*time.Second)

num\_ch <- sqr\_val

}

func cube(num\_ch chan int){

value := <-num\_ch

cube\_val := value\*value\*value

time.Sleep(10\*time.Second)

num\_ch <- cube\_val

}

func main(){

sqr\_ch := make(chan int)

go square(sqr\_ch)

sqr\_ch <- 3

cube\_ch := make(chan int)

go cube(cube\_ch)

cube\_ch <- 5

for itr := 1; itr <= 2; itr++{

select{

case sqr\_val := <-sqr\_ch:

fmt.Println("square of a value is:", sqr\_val)

case cube\_val := <-cube\_ch:

fmt.Println("Cube of a value is:", cube\_val)

}

}

}

**Data race:**

package main

import ("fmt"

"sync"

)

var wg sync.WaitGroup

var value = 45

func update\_value(){

if(value==45){

value++

}

fmt.Println(value)

wg.Done()

}

func main(){

wg.Add(2)

go update\_value() //gr1

go update\_value() //gr2

wg.Wait()

}

**Mutex:**

package main

import ("fmt"

"sync"

)

type initial\_amount struct{

amount int

sync.Mutex

}

func (ia \*initial\_amount) withdraw\_amount(withdraw\_val int){

ia.Lock()

ia.amount = ia.amount - withdraw\_val

defer ia.Unlock()

}

func (ia \*initial\_amount) deposit\_amount(deposit\_val int){

ia.Lock()

ia.amount = ia.amount + deposit\_val

defer ia.Unlock()

}

func (ia \*initial\_amount) get\_balance() int {

ia.Lock()

val := ia.amount

defer ia.Unlock()

return val

}

func main(){

ia := initial\_amount{amount: 2000}

for itr:= 1; itr<=10; itr++{

go ia.deposit\_amount(500)

go ia.withdraw\_amount(300)

}

final\_amount := ia.get\_balance()

fmt.Println("Final balance is:",final\_amount)

}

package main

import ("fmt"

"sync"

)

var wg sync.WaitGroup

type initial\_amount struct{

amount int

sync.Mutex

}

func (ia \*initial\_amount) withdraw\_amount(withdraw\_val int){

ia.Lock()

ia.amount = ia.amount - withdraw\_val

defer ia.Unlock()

wg.Done()

}

func (ia \*initial\_amount) deposit\_amount(deposit\_val int){

ia.Lock()

ia.amount = ia.amount + deposit\_val

defer ia.Unlock()

wg.Done()

}

func (ia \*initial\_amount) get\_balance() int {

ia.Lock()

val := ia.amount

defer ia.Unlock()

return val

}

func main(){

ia := initial\_amount{amount: 2000}

for itr:= 1; itr<=10; itr++{

wg.Add(1)

go ia.deposit\_amount(500)

wg.Add(1)

go ia.withdraw\_amount(300)

}

wg.Wait()

final\_amount := ia.get\_balance()

fmt.Println("final balance is:", final\_amount)

}

**RWMutex:**

package main

import ("fmt"

"sync"

"time"

)

type Map\_struct struct{

map\_val map[int] int

sync.RWMutex

}

func (map\_data \*Map\_struct) update\_map(keyval int){

map\_data.Lock()

map\_data.map\_val[keyval] = keyval \* 10

map\_data.Unlock()

}

func (map\_data \*Map\_struct) read\_map(){

map\_data.RLock()

val := map\_data.map\_val

map\_data.RUnlock()

fmt.Println(val)

}

func main(){

ia := Map\_struct{map\_val:make(map[int]int)}

for itr:= 1; itr<10; itr++{

go ia.update\_map(itr)

go ia.read\_map()

}

time.Sleep(5 \* time.Second)

}