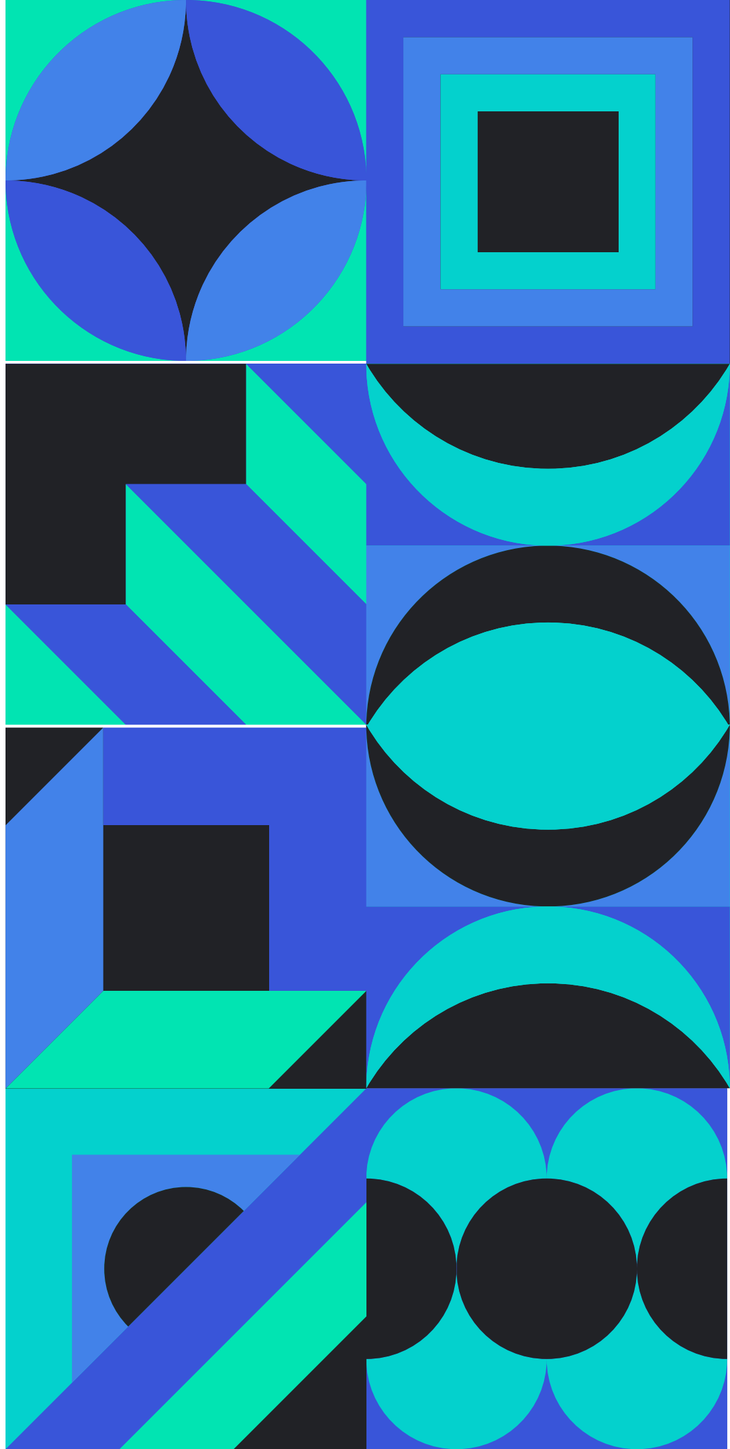
GOLANG: CONCURRENCY

By Abhiram Siddanthi

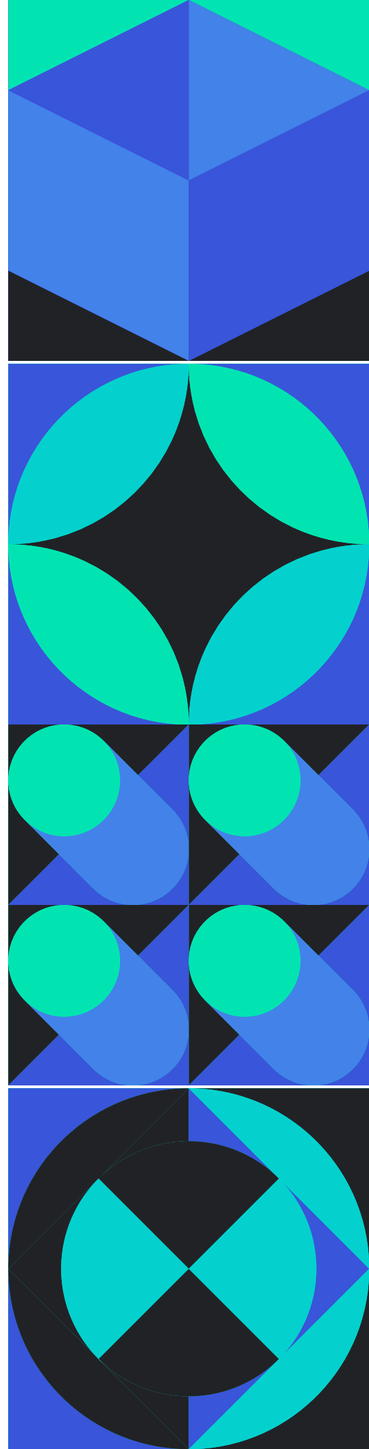


WHAT IS CONCURRENCY

- Concurrency is the composition of independently executing computations
- Way to structure and write clean code that interacts well with the real world
- It is not parallelism

CONCURRENCY'S USES:

- Physical resource sharing: Multiuser environment since hardware resources are limited.
- Logical resource sharing: Shared file(same piece of information).
- Computation speedup: Parallel execution
- Modularity: Divide system functions into separation processes



CONCURRENCY VS SEQUENTIAL VS PARALLELISM

CONCURRENT CODE

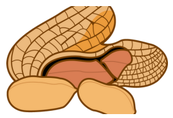
Concurrency is the composition of independently executing computations which may occur at the same time

SEQUENTIAL CODE

Sequential coding refers to the use of a single sequence to access a code in a specific order

PARALLELISM IN CODE

Code which facilitates more than one thing happening at the same which theoretically speeds up processes



IMPLIMENTING CONCURRENCY

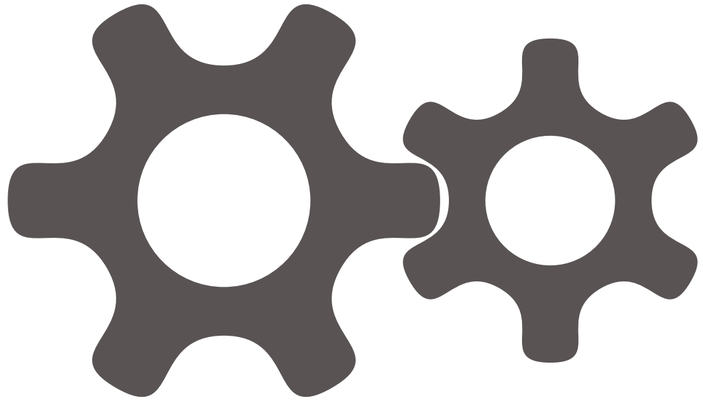
- Goroutines
- Channels
- WaitGroups
- Mutexes

GOROUTINES

Using a boring function

COROUTINES

Computer program components that allow execution to be suspended and resumed, generalizing subroutines for cooperative multitasking



Command to run a function in goroutine : go

```
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

GOROUTINES

Using a boring function

```go
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

```go
func main() {
    boring("hi")
}
```

```
hi 0
hi 1
hi 2
hi 3
hi 4
```

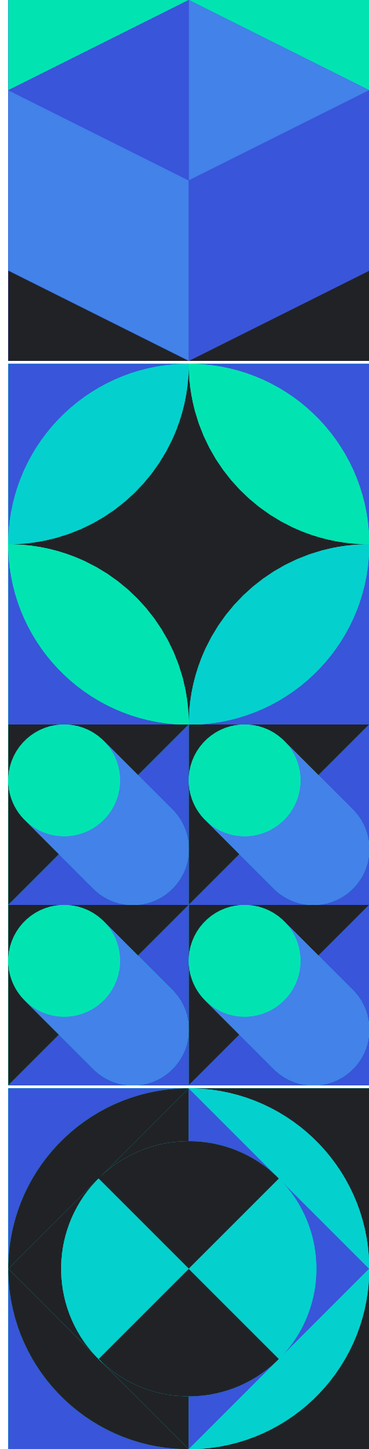
```go
func main() {
    go boring("hi")
}
```

```
[Running] go run "d:\C & C++ Directory\Go\lol.go"

[Done] exited with code=0 in 1.774 seconds
```

GOROUTINES

The Explanation

- It is an independently executing function, launched by the go statement
- It's very cheap possible to have thousands of goroutines
- It is not a thread
- There might be one thread in a program with a thousand goroutines
- Instead, goroutines are multiplexed dynamically onto threads as needed to keep the goroutines running

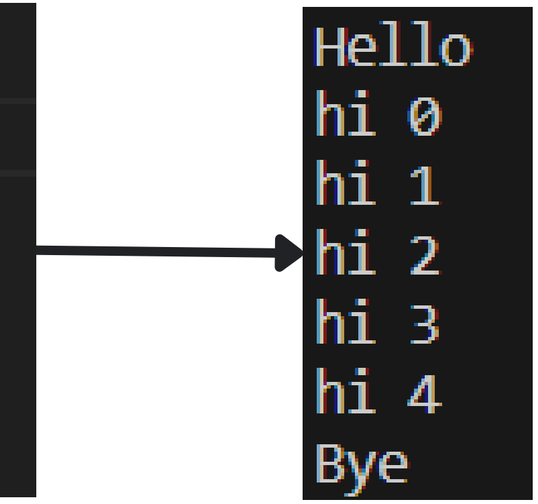


CHANNELS

Communication of goroutines

- In the example below, the goroutine is not communicating with any other function
- It is just mimicking the concept of communication
- Real conversations require communication

```
func main() {
    go boring("hi")
    fmt.Println("Hello")
    time.Sleep(2 * time.Second)
    fmt.Println("Bye")
}
```



```
Hello
hi 0
hi 1
hi 2
hi 3
hi 4
Bye
```

CHANNELS

Communication of goroutines

A channel in Go provides a connection between two goroutines allowing them to communicate

```
//Declaring and initializing
var c chan int
c = make(chan int)
 //or
 c:= make(chan int)
 //sending on a channel
 c<-1
//Recieving from a channel
//The "arrow" indicates direction of data flow
value = <-c
```

In Go channels * are first class values like strings or integers

USING CHANNELS

Channel connects the main and boring goroutines so they can communicate

```go
func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

SENDER

```go
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c)
    }
    fmt.Println("Bye")
}
```

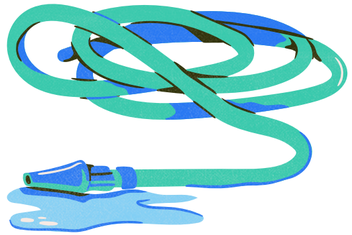
RECIEVER

```
You say: "boring! 0"
You say: "boring! 1"
You say: "boring! 2"
You say: "boring! 3"
You say: "boring! 4"
Bye
```

OUTPUT

IN-DEPTH USE OF CHANNELS

- Generator: function that returns its value as a channel
  - It is like an open pipe I can connect to seal



- Channel as a handle on service:
  - Channels can be used to handle information flow from different services in code

- Multiplexing:
  - Method by which multiple signals are combined into one signal over a shared medium

WAITGROUPS

Loading...

To wait for multiple goroutines to finish, we can use a *waitgroup*.

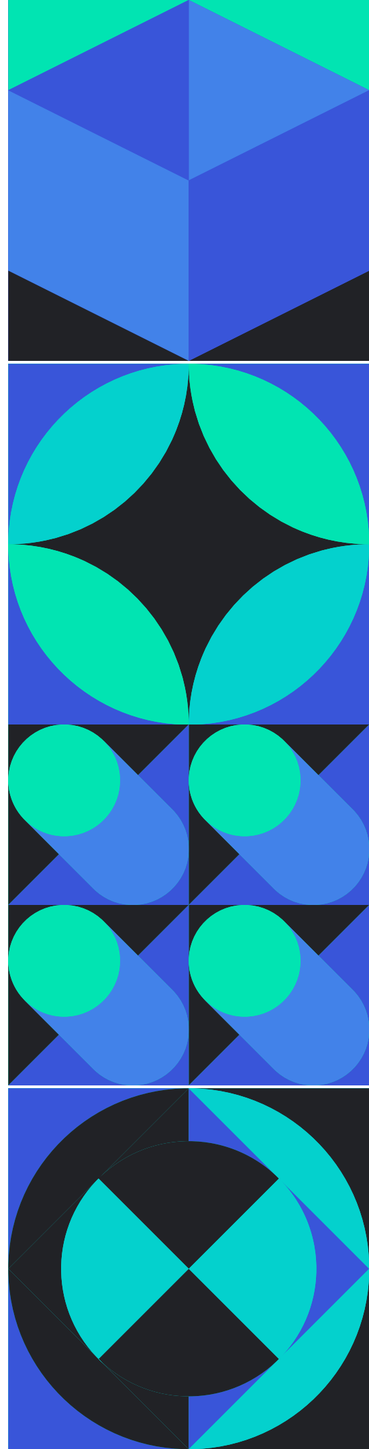
Problem: Program finishes execution before all the goroutines

```go
func runner1() {
    fmt.Println("I am winner 1")
}

func runner2() {
    fmt.Println("I am winner 2")
}

func main() {
    go runner1()
    go runner2()
}
```

```
[Running] go run "d:\C & C++ Directory\Go\lmao.go"

[Done] exited with code=0 in 2.457 seconds
```

Solution: Have a counter count the number of tasks and program only exits after the counter value is zero

= **Waitgroups**



USING WAITGROUPS

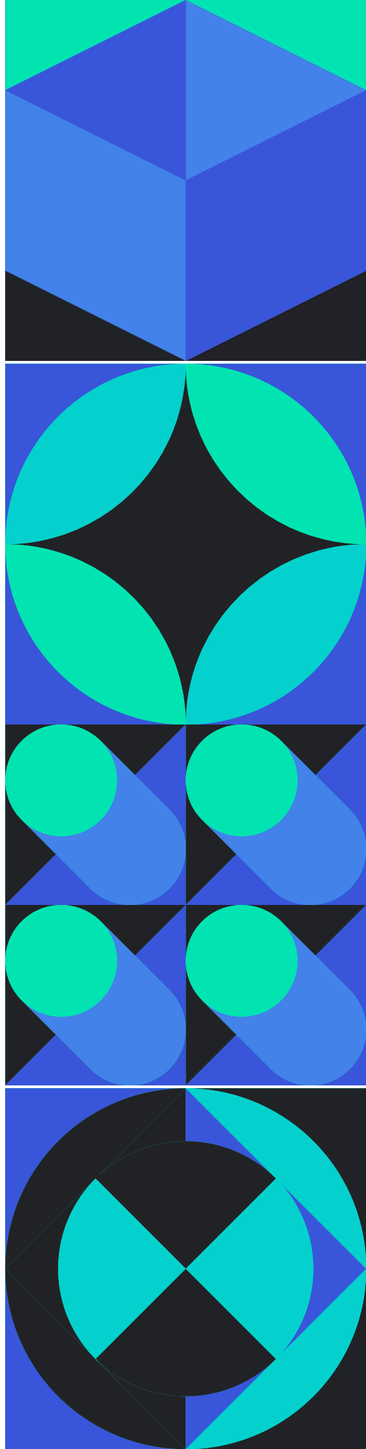
Syntax and application of waitgroups

```go
func runner1(wg *sync.WaitGroup) {
    fmt.Println("I am winner 1")
    defer wg.Done()
}

func runner2(wg *sync.WaitGroup) {
    fmt.Println("I am winner 2")
    defer wg.Done()
}
```

```go
func execute() {
    wg := new(sync.WaitGroup)
    wg.Add(2)

    go runner1(wg)
    go runner2(wg)

    wg.Wait()
}

func main() {
    execute()
}
```

```
PS D:\C & C++ Directory\Go> go run lmao.go
I am winner 2
I am winner 1
```

- If a WaitGroup is explicitly passed into functions, it should be done by a pointer.
- WaitGroup is used instead of stalling program as it does not interfere with the concurrency of the program

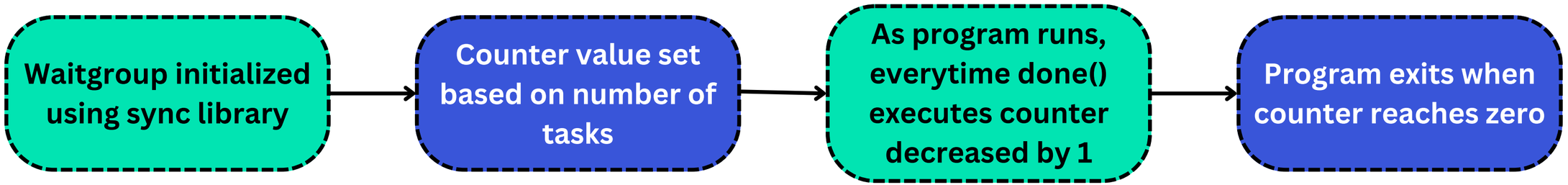


WAITGROUPS: WORKING

Concept and working of a waitgroup

1	**Add(int)**	It increases WaitGroup counter by given integer value.
2	**Done()**	It decreases WaitGroup counter by 1, we will use it to indicate termination of a goroutine.
3	**Wait()**	It Blocks the execution until it's internal counter becomes 0.

Flowchart:



ATOMIC OPERATIONS

Concept of an atomic counter

- It is a method used in Go for managing the state other than communication through channels
- We need to import the sync/atomic library to use it
- It gives us a safe way to access and interact with the data without causing any interference between the waitgroups

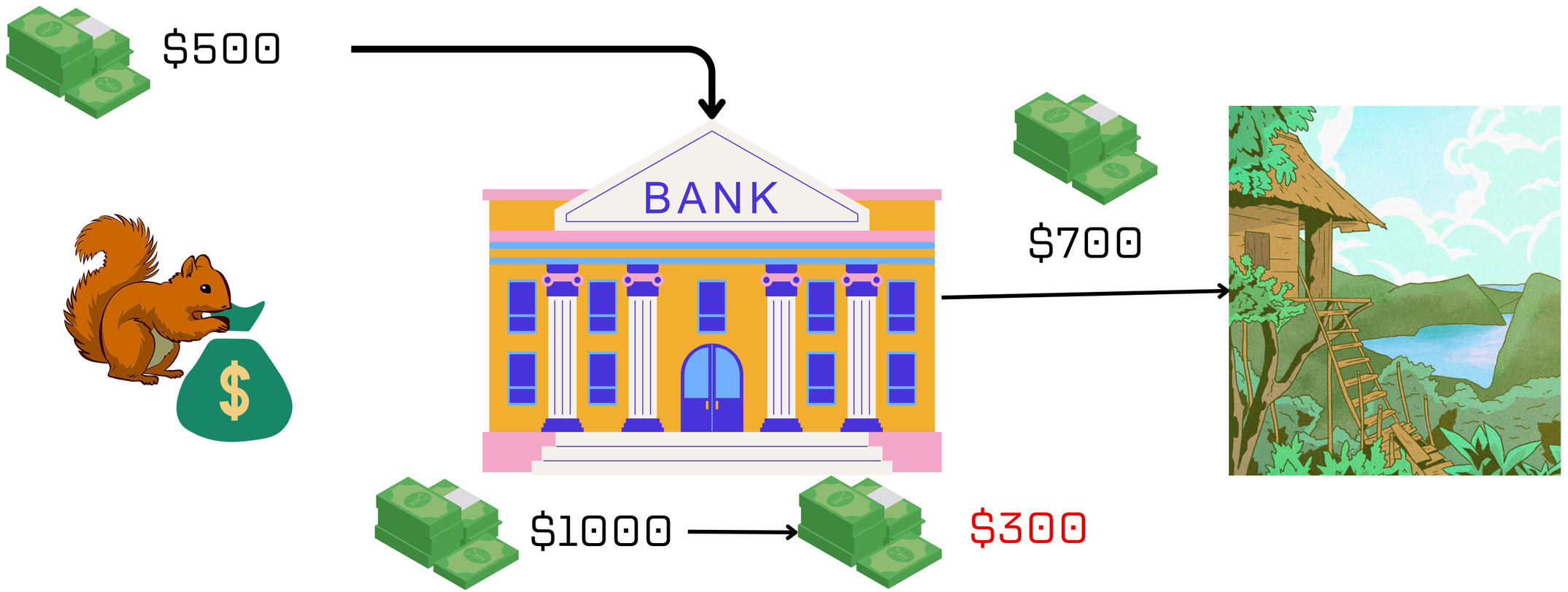
```go
func main() {
    var ops uint64
    var wg sync.WaitGroup
    for i := 0; i < 50; i++ {
        wg.Add(1)

        go func() {
            for c := 0; c < 1000; c++ {
                atomic.AddUint64(&ops, 1)
            }
            wg.Done()
        }()
    }
    wg.Wait()

    fmt.Println("ops:", ops)
}
```

```
PS D:\C & C++ Directory\Go> go run atomic.go
ops: 50000
```

MUTEXES

Concept of an mutexes

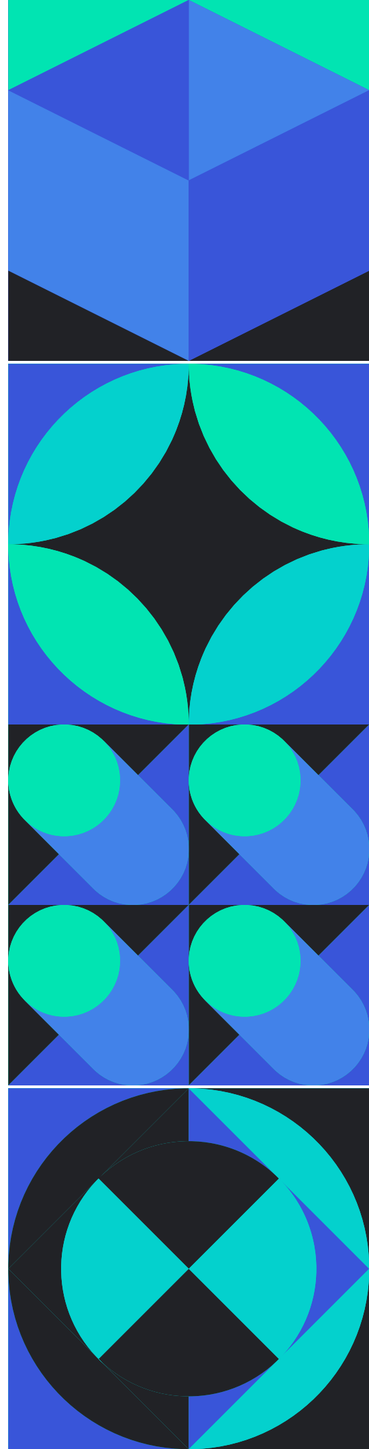
Problem: One goroutines' function overlaps with another goroutines' function and results in an override causing an error



MUTEXES

Concept of an mutexes

- A Mutex is a method used as a locking mechanism to ensure that only one Goroutine is accessing the critical section of code at any point in time.
- This is done to prevent race conditions from happening
- Two methods defined on Mutex:
  - Lock
  - Unlock
- Any code present between a call to Lock and Unlock will be executed by only one Goroutine.
- If one Goroutine already has the lock and if a new Goroutine is trying to get the lock, then the new Goroutine will be stopped until the mutex is unlocked



MUTEXES USAGE

Concept of an mutexes

```go
func deposit(value int, wg *sync.WaitGroup) {
    mutex.Lock()
    fmt.Printf("Depositing %d to account with balance %d\n", value, balance)
    balance += value
    mutex.Unlock()
    wg.Done()
}
```

```go
func withdraw(value int, wg *sync.WaitGroup) {
    mutex.Lock()
    fmt.Printf("Withdrawing %d to account with balance %d\n", value, balance)
    balance -= value
    mutex.Unlock()
    wg.Done()
}
```

```go
func main() {
    fmt.Println("Hello")

    balance = 1000

    var wg sync.WaitGroup
    wg.Add(2)
    go withdraw(700, &wg)
    go deposit(500, &wg)
    wg.Wait()

    fmt.Println(balance)
}
```

The mutex is locked before making any changes so that the other goroutine does not disturb the value while one goroutine is running

```
PS D:\C & C++ Directory\Go> go run mutex.go
Hello
Depositing 500 to account with balance 1000
Withdrawing 700 to account with balance 1500
800
```

THANK YOU FOR LISTENING!