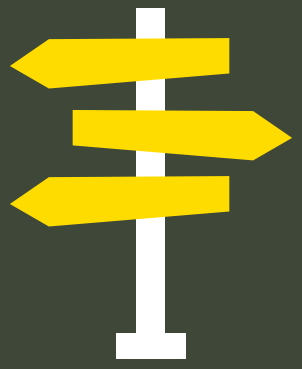
CRISP NOTES

ON GOLANG



Golang Essentials : Quick Reference Guide

AMIT MAHTO
@mahtoamit

Why you should learn Golang ?

✓ Performance

Go is a compiled language, which means it translates directly into machine code, leading to fast execution times

✓ Concurrency

Go has built-in support for concurrent programming with goroutines, making it easier to write programs that can perform multiple tasks simultaneously.

✓ Easy to Learn 📖

Go's syntax is clean and easy to read, making it accessible for beginners while still powerful enough for experienced developers.

✓ Standard Library

Go comes with a robust standard library that includes a wide range of packages for tasks such as HTTP, file I/O, and cryptography.

AMIT MAHTO
@mahtoamit

Uses of GO

✓ Web Development

Go is used to build web applications, either directly through frameworks like Gin or Echo, or indirectly through backend services that serve as the API for frontend applications.

✓ Cloud Services

Many cloud-based applications and services are written in Go due to its efficiency, scalability, and ability to handle concurrent tasks.

✓ Microservices

Go’s simplicity, fast compilation, and concurrency features make it a popular choice for developing microservices architectures.

✓ Distributed Systems

Go's concurrency primitives (goroutines and channels) make it well-suited for developing distributed systems.

AMIT MAHTO
@mahtoamit

Difference in GO & Python

GO	Python
✓ Statically typed	✓ Dynamically typed
✓ Fast run time	✓ Slow run time
✓ Compiled	✓ Interpreted
✓ Supports concurrency through goroutines and channel	✓ No built-in concurrency mechanism
✓ Automatic garbage collection	✓ Automatic garbage collection
✓ Classes & Objects does not support	✓ Classes & Objects supports

AMIT MAHTO
@mahtoamit

Syntx of GO

GO Files consits of following parts

- ✓ Package declaration
- ✓ Import packages
- ✓ Functions
- ✓ Statements and expressions

Package declare

```go
package main
import (
    "fmt"
)

func main() {
  fmt.Println("Hello World!")
}
```

Package import

function

statements/
expressions

AMIT MAHTO
@mahtoamit

Comments in GO

✓ Single Line Comment

These start with // and continue until the end of the line. They are used for short comments or explanations on a single line of code.

```go
fmt.Println("Hello World!") // Tis is Single-Line comment
```

✓ Multi Line Comment

These start with /* and end with */. They can span multiple lines and are typically used for longer explanations or commenting out blocks of code.

```go
func main() {
    /* The below code is
    Multi line comments */
  fmt.Println("Hello World!")
}
```

AMIT MAHTO
@mahtoamit

Variables in GO

- ✓ In Go (Golang), variables are used to store data values.
- ✓ Go uses the var keyword to declare variables. There are several ways to declare variables in Go:

1. Explicit type declaration

```
var age int
var name string
```

Here, age is declared as an integer (int) and name as a string (string).

AMIT MAHTO
@mahtoamit

Variables in GO

2. Declare and initialize

```go
var age int = 30
var name string = "Amit"
```

3. Type inference (var without type)

```go
var age = 26
var name = "Amit"
```

Can be used inside and outside of functions

AMIT MAHTO
@mahtoamit

Variables in GO

4. Short variable declaration (:=)

```
1
2   var age := 26
3   var name := "Amit"
```

Can only be used inside functions

AMIT MAHTO
@mahtoamit

Variable Naming Rule in GO

- ✓ Must start with a letter or underscore (_), not a digit.
- ✓ Can contain only letters, digits, and underscores.
- ✓ Case-sensitive (e.g., age, Age, and AGE are different).
- ✓ No length limit.
- ✓ No spaces allowed.
- ✓ Cannot be a Go keyword like (if, var, func etc

AMIT MAHTO
@mahtoamit

Constants in GO

- If a variable should have a fixed value that cannot be changed, you can use the const keyword.

- The const keyword declares the variable as "constant", which means that it is unchangeable and read-only.

Example

```go
package main
import (
    "fmt"
)
const PI = 3.14
func main() {
  fmt.Println(PI)
}
```

AMIT MAHTO
@mahtoamit

Data Types in GO

- ✓ Data types specify the type of value a variable can hold.
- ✓ Go has three basic data types

1. Boolean

Represents a boolean value (true or false).

```
1  var isVoted bool = true
2
```

2. String

Represents a sequence of characters.

```
1  var city string = "Mumbai"
2
```

AMIT MAHTO
@mahtoamit

Data Types in GO

3. Numeric

Integer Types:

- Signed integers: int, int8, int16, int32, int64

1. Can store both positive and negative values

- Unsigned integers: uint, uint8, uint16, uint32, uint64, uintptr

2. Can only store non-negative values

```go
var age int = 30
var height uint = 180
var smallNumber int8 = -128
```

Floating-point Types:

- float32, float64

```go
var pi float64 = 3.14159

```

AMIT MAHTO
@mahtoamit

Data Types in GO

3. Numeric

Complex Types:

- complex64, complex128

```
var c complex128 = complex(5, 7)
```

AMIT MAHTO
@mahtoamit

Arrays in GO

✓ Arrays are used to store multiple values of the same type in a single variable.

Declaring Arrays

Fixed-size array:

```
var arr [5]int
```

Array with initial values:

```
var arr = [5]int{1, 2, 3, 4, 5}
```

Array with inferred size:

```
array := [...]int{1, 2, 3, 4, 5}
```

AMIT MAHTO
@mahtoamit

Arrays in GO

Accessing Array Elements

Accessing an elements

```go
var arr = [5]int{1, 2, 3, 4, 5}
fmt.Println(arr[0]) // Access the first element
```

Modifying an element:

```go
var arr = [5]int{1, 2, 3, 4, 5}
arr[1] = 10 // Modify the second element
```

AMIT MAHTO
@mahtoamit

Arrays in GO

Array Methods and Operations

Length of an array:

```go
var arr = [5]int{1, 2, 3, 4, 5}
length := len(arr)
fmt.Println(length) // Prints the length of the array

```

Iterating over an array:

```go
var arr = [5]int{1, 2, 3, 4, 5}
for i, v := range arr {
    fmt.Println(i, v)
}
```

AMIT MAHTO
@mahtoamit

Arrays in GO

Array Methods and Operations

Comparing arrays:

Arrays in Go can be compared directly using the == operator, but only if they are of the same type and length.

```go
arr1 := [3]int{1, 2, 3}
arr2 := [3]int{1, 2, 3}
arr3 := [3]int{4, 5, 6}

fmt.Println(arr1 == arr2) // true
fmt.Println(arr1 == arr3) // false
```

Multidimensional arrays:

```go
var matrix [3][3]int
matrix[0][0] = 1 //  sets the first element of the first row to 1.
fmt.Println(matrix) // [[1 0 0] [0 0 0] [0 0 0]]
```

AMIT MAHTO
@mahtoamit

Slices in GO

- ✓ Sliices are a more flexible and powerful data structure compared to arrays.
- ✓ A slice is a lightweight data structure that wraps an underlying array and describes a section of that array.
- ✓ Unlike arrays, slices can dynamically grow and shrink.

Syntax :

Creating a slice

```go
// Creating a slice using a literal
slice := []int{1, 2, 3, 4, 5}

// Creating a slice using make
slice := make([]int, 5) // creates a slice of length 5
```

AMIT MAHTO
@mahtoamit

Slices in GO

Indexing :

You can access individual elements of a slice using square brackets, just like arrays. Indexing starts at 0.

```go
// Creating a slice using a literal
slice := []int{1, 2, 3, 4, 5}

fmt.Println(slice[0]) // prints the first element of the slice
```

Append :

The append() function is used to add elements to a slice.

```go
// Creating a slice using a literal
slice := []int{1, 2, 3, 4, 5}

slice = append(slice, 6) // appends 6 to the end of the slice
```

AMIT MAHTO
@mahtoamit

Slices in GO

Slicing :

You can create a new slice from an existing slice by specifying a range of indices.

```go
// Creating a slice using a literal
slice := []int{1, 2, 3, 4, 5}

subset := slice[1:3] // creates a new slice containing elements from index
```

Length and Capacity:

The len() function returns the number of elements in the slice, and the cap() function returns the capacity of the underlying array.

```go
// Creating a slice using a literal
slice := []int{1, 2, 3, 4, 5}

fmt.Println(len(slice)) // prints the number of elements in the slice
fmt.Println(cap(slice)) // prints the capacity of the underlying array
```

AMIT MAHTO
@mahtoamit

Operators in GO

✓ In Go, operators are symbols that represent computations, comparisons, or logical operations.

1. Arithmetic Operators :
   a. + // Addition
   b. - // Subtraction
   c. * // Multiplication
   d. / // Division
   e. % // Remainder

2. Comparison Operators:
   a. == // Equal to
   b. != // Not equal to
   c. < // Less than
   d. <= // Less than or equal to
   e. > // Greater than
   f. >= // Greater than or equal to

AMIT MAHTO
@mahtoamit

Operators in GO

3. Logical Operators:

a. && // Logical AND
b. || // Logical OR
c. ! // Logical NOT (Unary)

4. Assignment Operators:

a. = // Assignment
b. += // Addition assignment
c. -= // Subtraction assignment
d. *= // Multiplication assignment
e. /= // Division assignment
f. %= // Remainder assignment

AMIT MAHTO
@mahtoamit

Operators in GO

5. Bitwise Operators:
    a. & // Bitwise AND
    b. | // Bitwise OR
    c. ^ // Bitwise XOR (exclusive OR)
    d. << // Left shift
    e. >> // Right shift
6. Unary Operators:
    a. + // Unary plus (no effect)
    b. – // Unary minus
    c. ! // Logical NOT
    d. ^ // Bitwise NOT

AMIT MAHTO
@mahtoamit

Operators in GO

7. Miscellaneous Operators:
    a. & – Address of (creates a pointer)
    b. * – Dereference (access value through a pointer)
    c. <– – Channel receive/send (used in channel operations)

AMIT MAHTO
@mahtoamit

Conditions in GO

✓ In Go, conditional statements are used to execute code blocks based on the evaluation of certain conditions.

Go has the following conditional statements:

a. Use if to specify a block of code to be executed, if a specified condition is true
b. Use else to specify a block of code to be executed, if the same condition is false
c. Use else if to specify a new condition to test, if the first condition is false.
d. Use nested if statements inside if statements, this is called a nested if.
e. Use switch to specify many alternative blocks of code to be executed.

AMIT MAHTO
@mahtoamit

Conditions in GO

if statement :

✓ It executes a block of code if a specified condition is true.

Syntax :

```go
if condition {
  // code block to be executed if condition is true
}
```

Example :

```go
if x > 10 {
  fmt.Println("x is greater than 10")
}
```

AMIT MAHTO
@mahtoamit

Conditions in GO

if–else statement :

- ✓ It executes one block of code if the condition is true, and another block of code if the condition is false.

Syntax :

```
if condition {
  // code block to be executed if condition is true
} else {
  // code block to be executed if condition is false
}
```

Example :

```
if x > 10 {
  fmt.Println("x is greater than 10")
} else {
  fmt.Println("x is not greater than 10")
}
```

AMIT MAHTO
@mahtoamit

Conditions in GO

else if statement:

- ✓ It allows for chaining multiple conditions and executing different blocks of code based on which condition evaluates to true first.

Syntax :

```
if condition1 {
  // code block to be executed if condition1 is true
} else if condition2 {
  // code block to be executed if condition2 is true
} else {
  // code block to be executed if all conditions are false
}
```

Example :

```
if x > 10 {
  fmt.Println("x is greater than 10")
} else if x == 10 {
  fmt.Println("x is equal to 10")
} else {
  fmt.Println("x is less than 10")
}
```

AMIT MAHTO
@mahtoamit

Conditions in GO

Nested if Statement:

- ✓ if statements inside if statements, this is called a nested if.

Syntax :

```
if condition1 {
  // code block for condition1 being true

  if condition2 {
      // code block for condition2 being true
  } else {
      // code block for condition2 being false
  }

} else {
  // code block for condition1 being false
}
```

Example :

```
age := 25
if age > 18 {
    fmt.Println("You are an adult")

    if age < 60 {
        fmt.Println("You are not a senior citizen")
    } else {
        fmt.Println("You are a senior citizen")
    }

} else {
    fmt.Println("You are a minor")
}
```

AMIT MAHTO
@mahtoamit

Conditions in GO

Switch Statement:

- ✓ In Go, switch statements provide a cleaner and more readable way to handle multiple conditional branches

Syntax :

```go
switch expression {
case value1:
    // code to be executed if expression == value1
case value2:
    // code to be executed if expression == value2
default:
    // code to be executed if no cases match
}
```

Example :

```go
day := "Monday"
switch day {
case "Monday":
    fmt.Println("Start of the work week")
case "Friday":
    fmt.Println("End of the work week")
default:
    fmt.Println("Midweek day")
}
```

AMIT MAHTO
@mahtoamit

Loops in GO

For Loop :

- ✓ The for loop loops through a block of code a specified number of times.
- ✓ The for loop is the only loop available in Go.
- ✓ Each execution of a loop is called an iteration.
- ✓ The for loop can take up to three statements.

Syntax :

```
for statement1; statement2; statement3 {
  // code to be executed for each iteration
}
```

statement1 - Initializes the loop counter value.
statement2 - Evaluated for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop ends.
statement3 - Increases the loop counter value.

AMIT MAHTO
@mahtoamit

Loops in GO

Range Based for loop:

✓ This loop iterates over elements of an array, slice, map, or string.

Iterating Over a Slice or Array:

```go
nums := []int{1, 2, 3, 4, 5}
for index, value := range nums {
    fmt.Println(index, value)
}

// If you only need the value:
for _, value := range nums {
    fmt.Println(value)
}
```

AMIT MAHTO
@mahtoamit

Loops in GO

Range Based for loop:

✓ This loop iterates over elements of an array, slice, map, or string.

Iterating Over a a Map:

```go
m := map[string]int{"a": 1, "b": 2, "c": 3}
for key, value := range m {
    fmt.Println(key, value)
}
```

Iterating Over a a String:

```go
s := "hello"
for index, runeValue := range s {
    fmt.Printf("%d: %c\n", index, runeValue)
}
```

AMIT MAHTO
@mahtoamit

Loops in GO

Break & Continue :

- break: Exits the loop immediately.
- continue: Skips the current iteration and moves to the next one.

Example :

```go

for i := 0; i < 10; i++ {
    if i == 5 {
        continue // Skip the rest of the loop when i is 5
    }
    if i == 8 {
        break // Exit the loop when i is 8
    }
    fmt.Println(i)
}
```

AMIT MAHTO
@mahtoamit

Functions in GO

Functions:

- A function in Go is defined using the func keyword.
- A function is a block of statements that can be used repeatedly in a program.
- A function will not execute automatically when a page loads.
- A function will be executed by a call to the function.
- The single most popular Go function is main(), which is used in every independent Go program.

Basic Function:

```go
func functionName() {
  // function body
}
```

AMIT MAHTO
@mahtoamit

Functions in GO

Function with Parameter :

```go
func add(a int, b int) int {
  return a + b
}
```

Function with Multiple Return Values :

```go
func swap(x, y string) (string, string) {
  return y, x
}
```

AMIT MAHTO
@mahtoamit

Functions in GO

Variadic Functions :

- Go supports variadic functions, which can be called with a varying number of arguments.
- In Golang, it is possible to pass a varying number of arguments of the same type as referenced in the function signature.
- To declare a variadic function, the type of the final parameter is preceded by an ellipsis, "...", which shows that the function may be called with any number of arguments of this type.

Example :

```go
func sum(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

AMIT MAHTO
@mahtoamit

Functions in GO

Anonymous Functions :

- ✓ An anonymous function is a function that was declared without any named identifier to refer to it.
- ✓ Anonymous functions can accept inputs and return outputs, just as standard functions do.

Example :

```go
func main() {
    // Define an anonymous function and assign it to a variable
    add := func(a, b int) int {
        return a + b
    }

    // Call the anonymous function through the variable
    result := add(3, 4)
    fmt.Println("The sum is:", result)  // Output: The sum is: 7
}
```

AMIT MAHTO
@mahtoamit

Functions in GO

Closures Functions :

- Closures are a special case of anonymous functions.
- Closures are anonymous functions which access the variables defined outside the body of the function.

Example :

```go
// intSeq returns a function that returns an int.
// The returned function is a closure over the variable i.
func intSeq() func() int {
  i := 0
  return func() int {
    i++
    return i
  }
}

func main() {
  // Create a new instance of the closure
  nextInt := intSeq()

  // Call the closure multiple times
  fmt.Println(nextInt()) // Output: 1
  fmt.Println(nextInt()) // Output: 2
  fmt.Println(nextInt()) // Output: 3

  // Create a new instance of the closure
  newInts := intSeq()
  fmt.Println(newInts()) // Output: 1
}
```

AMIT MAHTO
@mahtoamit

Functions in GO

Deferred Functions :

- ✓ Go has a special statement called defer that schedules a function call to be run after the function completes.

Example :

```go
func first() {
    fmt.Println("First")
}
func second() {
    fmt.Println("Second")
}
func main() {
    defer second()
    first()
}
```

AMIT MAHTO
@mahtoamit

Panic in GO

Panic() :

- In Go, panic is a built-in function that stops the normal execution of a goroutine and begins panicking.
- When a function panics, it immediately stops executing, and the deferred functions are run in the reverse order they were deferred.
- Panic is typically used for unrecoverable errors, such as those that indicate a bug in the program.

Example :

```go
func cleanup() {
  fmt.Println("Cleanup before exiting")
}

func riskyFunction() {
  defer cleanup()
  fmt.Println("About to panic")
  panic("A serious error occurred")
}

func main() {
  fmt.Println("Starting program")
  riskyFunction()
  fmt.Println("This line will not be executed")
}
```

AMIT MAHTO
@mahtoamit

Recover in GO

recover() :

- Go provides a mechanism to recover from panics using the recover function.
- recover can only be called from within a deferred function.
- If recover is called, it stops the panic and returns the value that was passed to panic.

Example :

```go
func cleanup() {
  if r := recover(); r != nil {
      fmt.Println("Recovered from panic:", r)
  }
}

func riskyFunction() {
  defer cleanup()
  fmt.Println("About to panic")
  panic("A serious error occurred")
  fmt.Println("This line will not be executed")
}

func main() {
  fmt.Println("Starting program")
  riskyFunction()
  fmt.Println("Program continues after recovering from panic")
}
```

AMIT MAHTO
@mahtoamit

Maps in GO

map :

- In Go, a map is a built-in data type that associates keys with values.
- It is a collection of key-value pairs, where each key is unique.
- The key type and value type are specified when the map is declared.
- Maps are unordered collections, and there's no way to predict the order in which the key/value pairs will be returned.

Declaring Map:

```go
//Using make Function
m := make(map[string]int){"Alice": 25, "Bob":   30,}

// Using a Map Literal
m := map[string]int{
  "Alice": 25,
  "Bob":   30,
}
```

AMIT MAHTO
@mahtoamit

Maps in GO

Basic Operations:

```go
func main() {
    // Creating an empty map
    m := make(map[string]int)

    // Inserting elements into the map
    m["Alice"] = 25
    m["Bob"] = 30
    fmt.Println("After inserting elements:", m) // Output: map[Alice:25 Bob:30]

    // Updating an element in the map
    m["Alice"] = 26
    fmt.Println("After updating Alice's age:", m) // Output: map[Alice:26 Bob:30]

    // Accessing an element in the map
    aliceAge := m["Alice"]
    fmt.Println("Alice's age:", aliceAge) // Output: Alice's age: 26

    // Deleting an element from the map
    delete(m, "Bob")
    fmt.Println("After deleting Bob:", m) // Output: map[Alice:26]

    // Checking the length of the map
    length := len(m)
    fmt.Println("Length of the map:", length) // Output: Length of the map: 1
}
```

AMIT MAHTO
@mahtoamit

Maps in GO

Iterating Over a Map:

```go
m := make(map[string]int){"Alice": 25, "Bob":   30,}

for key, value := range m {
  fmt.Println(key, value)
}
```

Truncate Map:

```go
func main() {
  var employee = map[string]int{"Mark": 10, "Bob": 20,
    "Alice": 30, "Simon": 40, "Kate": 50}

  // Method - I
  for k := range employee {
    delete(employee, k)
  }

  // Method - II
  employee = make(map[string]int)
}
```

AMIT MAHTO
@mahtoamit

Maps in GO

Merge Maps:

```go
func main() {
  first := map[string]int{"a": 1, "b": 2, "c": 3}
  second := map[string]int{"a": 1, "e": 5, "c": 3, "d": 4}

  for k, v := range second {
    first[k] = v
  }

  fmt.Println(first)
}
```

Nested Map:

```go
func main() {
  nestedMap := map[string]map[string]int{
      "Group1": {
          "Alice": 25,
          "Bob":   30,
      },
      "Group2": {
          "Charlie": 35,
          "Dave":    40,
      },
  }

  // Accessing a nested map value
  fmt.Println(nestedMap["Group1"]["Alice"])
}
```

AMIT MAHTO
@mahtoamit

Struct in GO

Struct :

- In Go, a struct is a composite data type that groups together variables under a single name.
- Each variable in a struct is called a field, and each field has a name and a type.
- The key type and value type are specified when the map is declared.
- structs are used to create more complex data structures by combining multiple pieces of data.

Declaring Struct :

```
type Person struct {
    Name string
    Age  int
}
```

AMIT MAHTO
@mahtoamit

Struct in GO

Basic operation in Struct :

```go
func main() {
  // Creating an instance of the struct
  var person1 Person
  // Assigning values to the struct fields
  person1.Name = "Alice"
  person1.Age = 30

  // Accessing struct fields
  fmt.Println("Name:", person1.Name)
  fmt.Println("Age:", person1.Age)

  // Creating and initializing a struct
  person2 := Person{
      Name: "Bob",
      Age:  25,
  }

  fmt.Println("Name:", person2.Name)
  fmt.Println("Age:", person2.Age)

  // Creating and initializing a struct using field names
  person3 := Person{
      Name: "Charlie",
      Age:  35,
  }

  fmt.Println("Name:", person3.Name)
  fmt.Println("Age:", person3.Age)

  // Creating and initializing a struct without field names (order matters)
  person4 := Person{"Dave", 40}

  fmt.Println("Name:", person4.Name)
  fmt.Println("Age:", person4.Age)
}
```

AMIT MAHTO
@mahtoamit

Struct in GO

Nested Struct :

```go
// Define Address struct
type Address struct {
    City    string
    ZipCode string
}

// Define Person struct with an embedded Address struct
type Person struct {
    Name    string
    Age     int
    Address Address
}

func main() {
    // Creating and initializing a nested struct
    person := Person{
        Name: "Alice",
        Age:  30,
        Address: Address{
            City:    "New York",
            ZipCode: "10001",
        },
    }

    fmt.Println("Name:", person.Name)
    fmt.Println("Age:", person.Age)
    fmt.Println("City:", person.Address.City)
    fmt.Println("ZipCode:", person.Address.ZipCode)
}
```

AMIT MAHTO
@mahtoamit

Struct in GO

Methods on Struct :

```go
// Define Person struct
type Person struct {
    Name string
    Age  int
}

// Define a method on the Person struct
func (p Person) Greet() {
    fmt.Printf("Hello, my name is %s and I am %d years old.\n", p.Name, p.Age)
}

func main() {
    person := Person{Name: "Alice", Age: 30}
    person.Greet() // Output: Hello, my name is Alice and I am 30 years old.
}
```

AMIT MAHTO
@mahtoamit

Interface in GO

Interface :

- Interface describes all the methods of a method set and provides the signatures for each method.
- An Interface is an abstract type.
- To create interface use interface keyword, followed by curly braces containing a list of method names, along with any parameters or return values the methods are expected to have.

Declaring Interface :

```go
// Define an interface
type Speaker interface {
  Speak() string
}
```

AMIT MAHTO
@mahtoamit

Interface in GO

Basic operations in interface :

```go
// Define an interface
type Speaker interface {
    Speak() string
}

// Define a struct
type Person struct {
    Name string
}

// Implement the Speak method for the Person type
func (p Person) Speak() string {
    return "Hello, my name is " + p.Name
}

// Define another struct
type Dog struct {
    Name string
}

// Implement the Speak method for the Dog type
func (d Dog) Speak() string {
    return "Woof! My name is " + d.Name
}

func main() {
    // Create instances of Person and Dog
    person := Person{Name: "Alice"}
    dog := Dog{Name: "Rex"}

    // Call the speak function with different types
    printSpeak(person)
    printSpeak(dog)
}

// Define a function that takes a Speaker interface
func printSpeak(s Speaker) {
    fmt.Println(s.Speak())
}
```

AMIT MAHTO
@mahtoamit

Interface in GO

Empty Interface :

```go
func main() {
    // An empty interface can hold values of any type
    var i interface{}

    i = 42
    fmt.Println(i) // Output: 42

    i = "hello"
    fmt.Println(i) // Output: hello

    i = struct{ Name string }{Name: "Alice"}
    fmt.Println(i) // Output: {Alice}
}
```

AMIT MAHTO
@mahtoamit

Goroutines in GO

Goroutines :

- Goroutines are lightweight threads managed by the Go runtime
- They allow you to perform concurrent tasks efficiently.
- You can create a goroutine by prefixing a function or a method call with the go keyword.
- Goroutines run in the same address space, so access to shared memory must be synchronized.

Example

```go
func sayHello() {
  fmt.Println("Hello from goroutine")
}

func main() {
  go sayHello() // Start a new goroutine

  // Sleep for a while to allow the goroutine to finish
  time.Sleep(1 * time.Second)
  fmt.Println("Hello from main")
}
```

AMIT MAHTO
@mahtoamit

Goroutines in GO

Synchronization with WaitGroup:

- ✓ The sync.WaitGroup is used to wait for a collection of goroutines to finish executing.

Example

```go
func sayHello(wg *sync.WaitGroup) {
    defer wg.Done() // Notify that the goroutine is done
    fmt.Println("Hello from goroutine")
}

func main() {
    var wg sync.WaitGroup

    wg.Add(1) // Increment the counter

    go sayHello(&wg) // Start a new goroutine

    wg.Wait() // Wait for all goroutines to finish
    fmt.Println("Hello from main")
}
```

AMIT MAHTO
@mahtoamit

Channels in GO

Channels:

- Channels provide a way for goroutines to communicate and synchronize their execution.
- Channels are a fundamental feature of the language that enable safe and efficient communication and synchronization between goroutines (concurrently executing functions).
- Channels are created using the built-in make function and can be buffered or unbuffered.

Creating Channels:

```go
c := make(chan int) // Create a channel of type int

```

AMIT MAHTO
@mahtoamit

Channels in GO

Sending and Receiving data from Channels:

```go
func main() {
    c := make(chan int)

    go func() {
        c <- 42 // Send value to channel
    }()

    val := <-c // Receive value from channel
    fmt.Println(val)
}
```

AMIT MAHTO
@mahtoamit

Channels in GO

Buffered Channels:

- ✓ Buffered channels have a specified capacity.
- ✓ When a value is sent to a buffered channel, the sending goroutine is blocked only if the buffer is full.
- ✓ Conversely, the receiving goroutine is blocked only if the buffer is empty.

Example

```go

func main() {
  c := make(chan int, 2) // Create a buffered channel with capacity 2

  c <- 1 // Send value to channel (does not block)
  c <- 2 // Send another value to channel (does not block)

  fmt.Println("Buffered channel contains:", <-c) // Receive value from channel
  fmt.Println("Buffered channel contains:", <-c) // Receive another value from channel
}
```

AMIT MAHTO
@mahtoamit

Channels in GO

UnBuffered Channels:

- Unbuffered channels in Go provide a way for goroutines to communicate with each other synchronously.
- When a value is sent to an unbuffered channel, the sending goroutine is blocked until another goroutine receives the value from the channel.
- Similarly, the receiving goroutine is blocked until a value is sent to the channel.

Example

```go
func main() {
    c := make(chan int) // Create an unbuffered channel

    go func() {
        c <- 42 // Send value to channel (blocks until the value is received)
        fmt.Println("Sent value")
    }()

    val := <-c // Receive value from channel (blocks until a value is sent)
    fmt.Println("Received value:", val)
}
```

AMIT MAHTO
@mahtoamit

Channels in GO

Select Statement:

- ✓ The select statement lets a goroutine wait on multiple communication operations.

Example

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

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

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

    for i := 0; i < 2; i++ {
        select {
        case msg1 := <-c1:
            fmt.Println("Received", msg1)
        case msg2 := <-c2:
            fmt.Println("Received", msg2)
        }
    }
}
```

AMIT MAHTO
@mahtoamit

Files handling in GO

File handling

- The most important package that allows us to manipulate files and directories as entities is the os package.
- The io package has the io.Reader interface to reads and transfers data from a source into a stream of bytes.
- The io.Writer interface reads data from a provided stream of bytes and writes it as output to a target resource.

Create & write a file

```go
func main() {
    // Create a file
    file, err := os.Create("example.txt")
    if err != nil {
        fmt.Println("Error creating file:", err)
        return
    }
    defer file.Close()

    // Write to the file
    _, err = file.WriteString("Hello, World!\n")
    if err != nil {
        fmt.Println("Error writing to file:", err)
        return
    }

    fmt.Println("File written successfully")
}
```

AMIT MAHTO
@mahtoamit

Files handling in GO

Reading from a file :

- ✓ To read from a file, use the os.Open and io/ioutil.ReadAll methods.

```go
func main() {
    // Open a file
    file, err := os.Open("example.txt")
    if err != nil {
        fmt.Println("Error opening file:", err)
        return
    }
    defer file.Close()

    // Read the file content
    content, err := ioutil.ReadAll(file)
    if err != nil {
        fmt.Println("Error reading file:", err)
        return
    }

    fmt.Println("File content:")
    fmt.Println(string(content))
}
```

AMIT MAHTO
@mahtoamit

Files handling in GO

Appending from a file :

- To append to a file, open the file with the os.O_APPEND flag and write to it.

```go
func main() {
    // Open the file in append mode
    file, err := os.OpenFile("example.txt", os.O_APPEND|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println("Error opening file:", err)
        return
    }
    defer file.Close()

    // Append to the file
    _, err = file.WriteString("Appended text.\n")
    if err != nil {
        fmt.Println("Error writing to file:", err)
        return
    }

    fmt.Println("Text appended successfully")
}
```

AMIT MAHTO
@mahtoamit

Files handling in GO

Deleting from a file :

- To delete a file, use the os.Remove method.

```go
func main() {
    // Delete the file
    err := os.Remove("example.txt")
    if err != nil {
        fmt.Println("Error deleting file:", err)
        return
    }

    fmt.Println("File deleted successfully")
}
```

AMIT MAHTO
@mahtoamit

WAS THIS HELPFUL ?

Share with a friend who needs it!

AMIT MAHTO

@mahtoamit