GO PROGRAMMING FOR DATA PROFESSIONAL

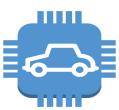
Why Go ?

Performance

Comparable performance to low-level languages like C and C++

Scalability

Ideal for building scalable and distributed systems



Cross Platform

Easily compile and run on different operating systems

Garbage Collection

Garbage collection and memory safety features

Cloud Development

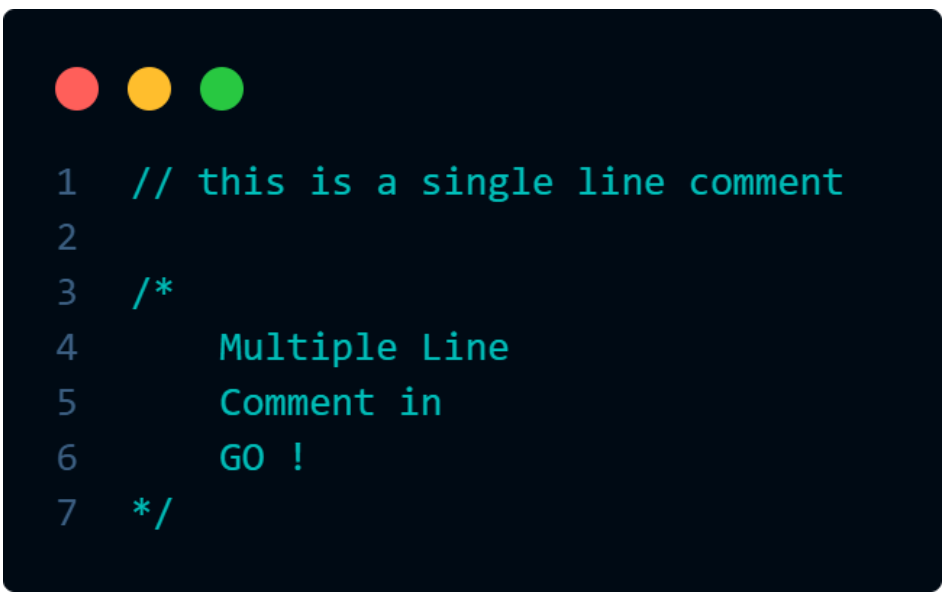
Widely used in cloud infrastructure and microservices architecture

Hello World

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hey Go!")
}
```

Comments

Single / Multiple Line



```
1  // this is a single line comment
2
3  /*
4      Multiple Line
5      Comment in
6      GO !
7  */
```

Basic Data Types

STRING

"Go"

"Is A Programming"

"Language"

BOOLEAN

true

false

NUMBER

INTEGER

"Go"

"Is A Programming"

"Language"

FLOAT

true

false

Why Data Types ?

Its used for classification of data

Help to categorize

Define set of operation that can be performed

Performance Optimization

Memory management

Code clarity

Data Structures

ARRAYS & SLICES

[1, 2, 4, 5,9]

[”red” , “blue”, ”green”]

[1.9, 3.5, 5.6]

MAPS

Collections of key-value pairs

“Go”

“Is A Programming”

“Language”

Integer Types

Data Type	Memory
uint8	bits or 1 byte
uint16	16 bits or 2 bytes
uint32	32 bits or 4 bytes
uint64	64 bits or 8 bytes
int8	8 bits or 1 byte
int16	16 bits or 2 bytes
int32	32 bits or 4 bytes
int64	64 bits or 8 bytes
int	4 bytes for 32-bit machines, 8 bytes for 64-bit machines

Floats

Data Type	Memory
float32	32 bits or 4 bytes
float64	64 bits or 8 bytes

Format Specifiers

Data Type	Format Specifier	Description
General	%v	Default format, suitable for most types
	%+v	Adds field names for structs
	%#v	Go-syntax representation of the value
	%T	Type of the value
	%%	Literal percent sign
Boolean	%t	Boolean: true or false
Integer	%d	Base 10
	%b	Base 2 (binary)
	%o	Base 8 (octal)
	%x	Base 16 (lowercase hexadecimal)
	%X	Base 16 (uppercase hexadecimal)
	%c	Character represented by the corresponding Unicode code point
Floating Point and Complex	%f	Decimal point but no exponent, e.g., 123.456
	%e	Scientific notation (e.g., -1234.456e+78)
	%E	Scientific notation (e.g., -1234.456E+78)
	%g	Exponent as needed, only necessary digits
	%G	Exponent as needed, only necessary digits (uppercase)
String and Byte Slice	%s	Plain string or slice of bytes
	%q	Double-quoted string with Go syntax
	%x	Base 16, with two characters per byte
	%X	Base 16, with two characters per byte (uppercase)
Pointer	%p	Base 16 notation, with leading 0x

Variable

Data Type	Memory
float32	32 bits or 4 bytes
float64	64 bits or 8 bytes

Declare and Initialize

```
1  // declare and initialize
2  var number int = 35
```

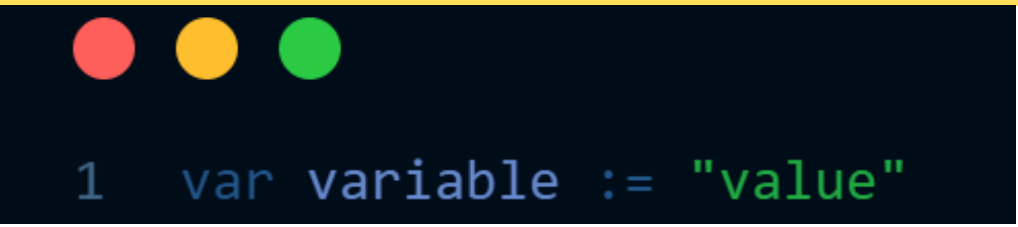
Implicit type declaration

```
1  // declare and initialize
2  var number = 35
```

LinkedIn
@mushtaqalih

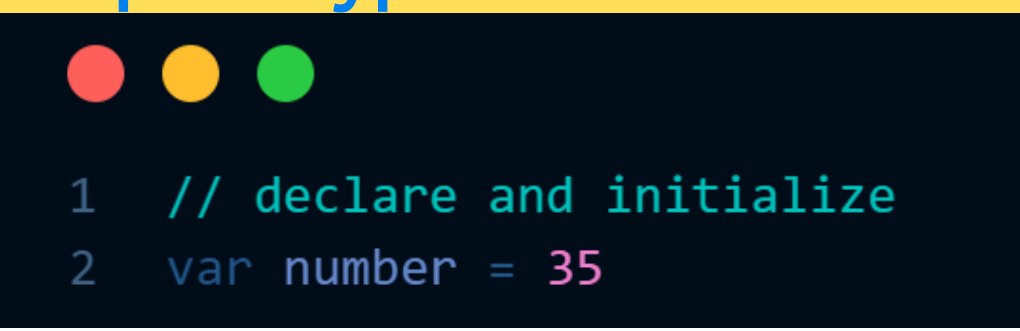
Variable

Short form :=



```
var variable := "value"
```

Implicit type declaration



```
// declare and initialize
var number = 35
```

Static vs Dynamic

Static

- Better performance
- Bugs caught at compile time
- Better Data Integrity

Dynamic

- Faster to type code
- Less learning curve comparibly
- Less rigid

LinkedIn
@mushtaqalih

Go is ?

Static

Supports both explicit declaration and implicit inferred by compiler

Its easy to learn , a very fast learning curve

+

It has features of Dynamically typed languages as well

LinkedIn
@mushtaqalih

Declaring Variable Multiple ways

```go
package main

import "fmt"

func main() {
        var fruit string                //declaring a variable
        fruit = "Apple"                 // assigning a value to the variable
        var vegetable string = "Carrot" // declare and initialize
        drink := "water"                // using short form implicit data type example
        price := 20.5                   // declaring an int variable
        // formatted print statement
        fmt.Printf("Fruit: %v\n"+
                "Vegetable: %v\n"+
                "Drink: %v\n"+
                "Price: %v\n",
                fruit, vegetable, drink, price)
}
```

Not supported

Dynamic type Casting

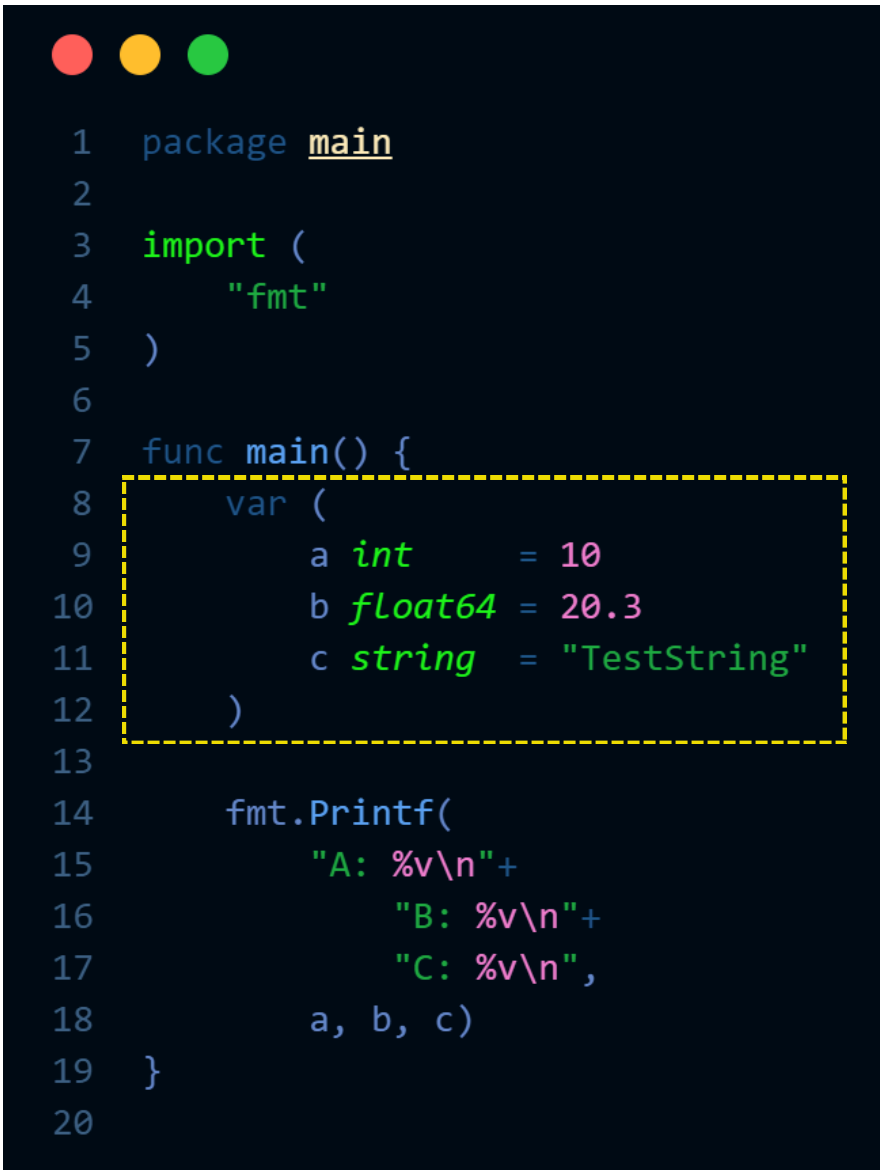
Go does not support dynamic type casting in the same way that some other languages (like Python or JavaScript) do.
Go supports Type assertion

```go
package main

import "fmt"

func main() {
    var a int = 10
    //type assertion
    var f float64 = float64(a)

    fmt.Println(f)
}
```

Multiple Variables Same Data Type

```go
package main

import "fmt"

func main() {
    //note variable type same
    var a, b int = 20, 30
    fmt.Println("A: ", a)
    fmt.Println("B: ", b)
}
```

Multiple Variables Different Data Type



```go
package main

import (
    "fmt"
)

func main() {
    var (
        a int     = 10
        b float64 = 20.3
        c string  = "TestString"
    )

    fmt.Printf(
        "A: %v\n"+
            "B: %v\n"+
            "C: %v\n",
        a, b, c)
}
```

Variable Scope

Different Levels

- Package Level
  Variable declared outside of function but inside a package have a package scope. These variable are visible to any code within the same package.

- Function Level
  Variables declared within a function are local to that function. These variables are only accessible within the function where they are declared.

- Block Level
  Variables declared inside a block (e.g., within if, for, switch, etc.) are local to that block. These variables are only accessible within the block where they are declared.

LinkedIn
@mushtaqalih

Local & Global Variable

Global

- Declared outside of function or a block of code
- Available from any where in a package

Local Variable

- Declared inside a block or function and only accessible from within.

LinkedIn
@mushtaqalih

Variable Scope

Different Levels

- Package Level
  Variable declared outside of function but inside a package have a package scope. These variable are visible to any code within the same package.

- Function Level
  Variables declared within a function are local to that function. These variables are only accessible within the function where they are declared.

- Block Level
  Variables declared inside a block (e.g., within if, for, switch, etc.) are local to that block. These variables are only accessible within the block where they are declared.

LinkedIn
@mushtaqalih

Variable Scope

Package Scope

```go
package main

import "fmt"

var globalVar = "I am a global variable"

func main() {
    fmt.Println("Call GlobalVar from inside Main", globalVar)
    anotherFunction()
}

func anotherFunction() {
    fmt.Println("Call GlobalVar from inside AnotherFunction", globalVar)
}

```

LinkedIn
@mushtaqalih

Variable Scope

Function Scope

```go
package main

import "fmt"

func main() {
    var a string = "I am A from Main"
    anotherFunction()
    fmt.Println(a)
}

func anotherFunction() {
    var a string = "I am A from AnotherFunction"
    fmt.Println(a)
}

```

LinkedIn
@mushtaqalih

Variable Scope

Block Scope

```go
package main

import "fmt"

func main() {
    if true {
        // Block scope
        blockVar := "I am a block variable"
        // Accessible within the block
        fmt.Println(blockVar)
    }
    /**
     fmt.Println(blockVar)
     This would cause an error
     because blockVar is not
    accessible here
    **/
}

```

Default Variable Values

When declared with no value, the default value is assigned to the variable B is a string , A / C are int and float and D is a boolean with default false

```go
package main

import (
    "fmt"
)

func main() {
    var a int
    var b string
    var c float64
    var d bool
    fmt.Printf(
        "A: %v\n"+
            "B: %v\n"+
            "C: %v\n"+
            "D: %v\n",
        a, b, c, d)
}
```

Result

```
[Running]
A: 0
B:
C: 0
D: false
```

LinkedIn
@mushtaqalih

User Input

Single value input

```go
package main

import "fmt"

func main() {
    var name string
    fmt.Println("Please enter your name")
    fmt.Scanf("%s", &name)
    fmt.Printf("Hey %s", name)
}
```

User Input

Multiple Value Input

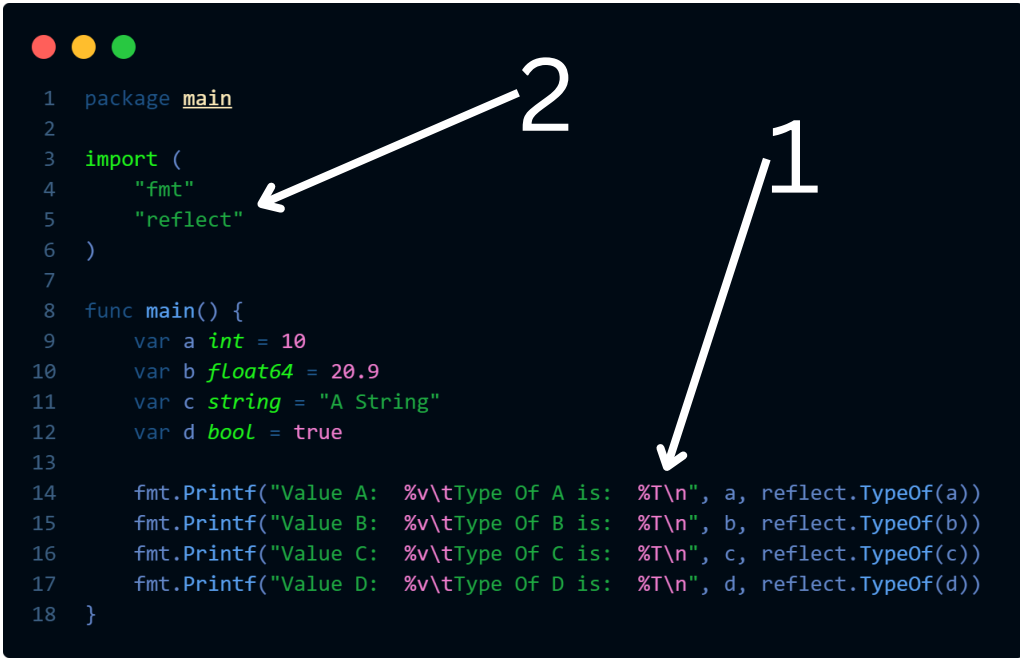
```go
package main

import "fmt"

func main() {
    var name string
    var age int
    fmt.Println("Please enter your name and age")
    fmt.Scanf("%s %d", &name, &age)
    fmt.Printf("Hi My name is %s and Age is %d", name, age)
}

```

```
Please enter your name and age
Ali 50
Hi My name is Ali and Age is 50
```

Find Type of Variable

1. %T specifier is required

2. reflect.TypeOf function is required



Add a subheading

Result

```
Value A:  10     Type Of A is:  int
Value B:  20.9   Type Of B is:  float64
Value C:  A String  Type Of C is:  string
Value D:  true   Type Of D is:  bool
```

you can use the values as well in place of variable names in TypeOf function

LinkedIn
@mushtaqalih

String to Integer Conversion

strconv.Itoa

```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var num int = 100
    var str string = strconv.Itoa(num)
    fmt.Printf("%q", str)
}
```

Required string conversion package

used for quoted string

LinkedIn
@mushtaqalih

Integer to String Conversion

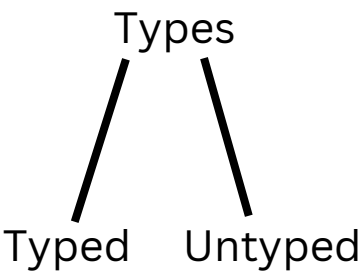
strconv.Atoi


```go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    var str string = "50"
    var num int
    num, _ = strconv.Atoi(str)
    fmt.Printf("%d", num)
}
```

Constants

Constants in Go are immutable values that are fixed at compile-time and cannot be changed during program execution.

Constant Declaration `const Pi = 3.14`



Typed: Declared with a data type

Untyped: Undeclared data type

Constants

Typed

```go
package main

import "fmt"

func main() {
    const firstVar int = 10
    const firstName string = "Ali"

    fmt.Println(firstVar)
    fmt.Println(firstName)
}
```

Constants

Un-Typed

```go
package main

import "fmt"

func main() {
    const firstVar = 10
    const firstName = "Ali"

    fmt.Println(firstVar)
    fmt.Println(firstName)
}
```

Constants

Declaring Multiple

Constants

If you try to declare variable and initialize in a different statement that will give error


```
# command-line-arguments
Constants\main.go:6:8: missing init expr for firstVar

[Done] exited with code=1 in 3.237 seconds
```

Constants

Short hand way to declare a variable will also give an error

Short hand := 

Operators

Arithmetic Operators

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus	a % b

Relational Operators

Relational Operators only return boolean true or false

Operator	Description	Example
==	Equal To	a == b
!=	Not Equal To	a != b
>	Greater Than	a > b
<	Less Than	a < b
>=	Greater than equal to	a >= b
<=	Less than equal to	a <= b

Logical Operators

Operator	Description	Example
&&	AND	a && b
||	OR	a || b
!	NOT	!a

Assignment Operators

Operator	Description	Example
=	Assignment	a = b
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b

Other Operators

Increment / Decrement

Operator	Description	Example
++	Increment	a++
--	Decrement	a--

Pointer

Operator	Description	Example
&	Address of Operator	&a
*	Dereference	*p

Programme Control Flow

Basic if statement

```go
package main

import (
    "fmt"
)

func main() {
    var num int = 10

    if num == 10 {
        fmt.Println(num)
    }
}
```

Programme Control Flow

```go
package main

import (
    "fmt"
)

var num int = 20

func main() {
    // var num int = 10

    if num == 10 {
        fmt.Println(num)
    } else {
        fmt.Println("The value is not 10")
    }
}
```

Enable: To see else block in action

Enable: To see if block in action

LinkedIn
@mushtaqalih

Programme Control Flow

```go
package main

import (
    "fmt"
)

// var fruit float64 = 20
// var fruit float64 = 9
var fruit float64 = 4.25

func main() {

    if fruit >= 10.50 {
        fmt.Println("Expensive")
    } else if fruit >= 5.50 && fruit <= 10.50 {
        fmt.Println("Normal")
    } else {
        fmt.Println("Cheap")
    }
}
```

multiple var , uncomment a particular to run different blocks

Line 15 contains else if and conditional statement

LinkedIn
@mushtaqalih

Programme Control Flow

Switch Case

Basic

```go
package main

import "fmt"

var dayOfWeek string = "Saturday"

func main() {
    switch dayOfWeek {
    case "Monday":
        fmt.Println("Its Monday!")
    case "Tuesday":
        fmt.Println("Its Tuesday!")
    case "Wednesday":
        fmt.Println("Its Wednesday!")
    case "Thursday":
        fmt.Println("Its Thursday!")
    case "Friday":
        fmt.Println("Its Friday Yay !")
    default:
        fmt.Println("Its weekend! Relax & Enjoy")
    }
}

```

LinkedIn
@mushtaqalih

Programme Control Flow

Switch Case

More than one select option

```go
package main

import "fmt"

var dayOfWeek string = "Wednesday"

func main() {
    switch dayOfWeek {
    case "Monday", "Tuesday", "Wednesday", "Thursday":
        fmt.Println("Week is ON work hard!")
    case "Friday":
        fmt.Println("Its Friday Yay tomorrow is rest day!")
    case "Saturday", "Sunday":
        fmt.Println("Its weekend! Relax & Enjoy")
    default:
        fmt.Println("Unknown value")
    }
}

```

- You can apply more than one condition

LinkedIn
@mushtaqalih

Programme Control Flow

Switch Case

fallthrough

```go
package main

import "fmt"

func main() {

    fmt.Println("Ask for a number 2 - 5 to see below numbers")
    var num int = 5

    switch num {
    case 5:
        fmt.Println("5")
        fallthrough
    case 4:
        fmt.Println("4")
        fallthrough
    case 3:
        fmt.Println("3")
        fallthrough
    case 2:
        fmt.Println("2")
        fallthrough
    case 1:
        fmt.Println("1")

    default:
        fmt.Println("0")
    }
}
```

fallthrough keyword is used to move to the next condition and execute without verifying the condition.

LinkedIn
@mushtaqalih

Programme Control Flow

Switch Case

Using conditions.

```go
package main

import "fmt"

func main() {

    var score int = 100

    switch {
    case score >= 90:
        fmt.Println("Excellent!")
    case score >= 60:
        fmt.Println("Good")
    case score >= 40:
        fmt.Println("OK!")
    default:
        fmt.Println("Not ok!")
    }

}

```

LinkedIn
@mushtaqalih

Programme Control Flow

Looping

```go
package main

import "fmt"

func main() {
    for i := 1; i < 6; i++ {
        fmt.Println(i)
    }
}
```

```
main.go"
1
2
3
4
5
```

LinkedIn
@mushtaqalih

Programme Control Flow

Looping (2nd Method)

```go
package main

import "fmt"

func main() {
    i := 10   assignment

    for i <= 100 {  condition
        fmt.Printf("%d\t",i)
        i += 10   increment
    }
    fmt.Println("\n")
}
```

LinkedIn
@mushtaqalih

Programme Control Flow

Infinite for loop

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    var name string
    var verify string

    for {

        fmt.Println("Enter your name: ")
        fmt.Scanln(&name)
        fmt.Println("Your name is ", name)
        fmt.Printf("Do you have any other name Press 'y' else press 'x' ?\n")
        fmt.Scanln(&verify)
        if verify == "x" {
            os.Exit(123)
        } else {
            continue
        }

    }
    fmt.Println("\n")
}
```

Observations

1. Infinite loop , no conditions provided.
2. os package imported and used with Exit function.
3. used if statement inside the loop to provide the exit condition.

LinkedIn
@mushtaqalih

Programme Control Flow

break and continue

```go
package main

import (
    "fmt"
)

func main() {
    fmt.Println("'Continue' For Loop")
    for i := 0; i < 10; i++ {
        if i == 5 {
            continue
        }
        fmt.Printf("%v\t", i)
    }
    fmt.Println()
    fmt.Println("'Break' For Loop")
    for i := 0; i < 10; i++ {
        if i == 5 {
            break
        }
        fmt.Printf("%v\t", i)
    }
}
```

Results

```
'Continue' For Loop
0   1   2   3   4   6   7   8   9
'Break' For Loop
0   1   2   3   4
```

LinkedIn
@mushtaqalih

Complex Data Structures

LinkedIn
@mushtaqalih

Arrays

1. Only Single Type of data can be stored
2. It can store large amount of data
3. The length of an array is fixed
4. Accessing elements of an array is fast

Property of an array

1. len
2. capacity

```
Result
0    0
1    0
2    0
3    0
4    0
```

```go
package main
import "fmt"

func main() {
    var a [5]int
    for i := 0; i < len(a); i++ {
        fmt.Printf("%v\t %v\t", i,
        fmt.Println()
    }
}
```

- **Zeros are default int values**
- **0 - 4 are the index of the array**

Arrays Declaration (3 types)

```go
package main

import "fmt"

func main() {
    //ways to initialize an array
    // full form
    var x [5]int = [5]int{1, 2, 3, 4, 5}
    // implicit declaration
    y := [5]int{1, 2, 3, 4, 5}
    // implicit size
    z := [...]int{1, 2, 3, 4, 5, 6, 7}

    fmt.Println("Array 'x': ")
    for i := 0; i < len(x); i++ {
        fmt.Printf("%v\t %v\t", i, x[i])
        fmt.Println()
    }

    fmt.Println("Array 'y': ")
    for i := 0; i < len(y); i++ {
        fmt.Printf("%v\t %v\t", i, y[i])
        fmt.Println()
    }

    fmt.Println("Array 'z': ")
    for i := 0; i < len(z); i++ {
        fmt.Printf("%v\t %v\t", i, z[i])
        fmt.Println()
    }
}

```

Arrays Indexes

```go
package main

import "fmt"

func main() {

    z := [...]int{1, 2, 3, 4, 5, 6, 7}

    fmt.Println("Array 'z': ")
    for i := 0; i < len(z); i++ {
        fmt.Printf("%v\t %v\t", i, z[i])
        fmt.Println()
    }
}
```

- Array index start at zero
- len is the function to find the array length
- array[index] is used to retrieve the value
- [...] when compiler sees it will find the array length.
- array[index] = new_value , replaces the value of an item in the array

Arrays Indexes

Index out of bounds

```go
package main

import "fmt"

func main() {
    i := 0
    var num [5]int = [5]int{1, 2, 3, 4, 5}
    for i < len(num) {
        fmt.Printf("%v\t", num[i])
        i++
    }
    fmt.Printf("%v", num[5]) //this line will throw error

}
```

```
arrays2\main.go:14:23: invalid argument: index 5 out of bounds [0:5]

[Done] exited with code=1 in 2.917 seconds
```

There are 2 points to remember

1. if you call for an index out of size it will give out of bounds error.
2. if you don't initiate the value i.e leave the value empty it will fill up with default values like 0 or empty string as defined earlier.

Arrays

Using Ranges

```go
package main

import "fmt"

func main() {

	var students [5]string = [5]string{"Ali", "James", "Ravi", "David", "Erik"}

	for index, element := range students {
		fmt.Println(index, " => ", "element", index, element)
	}

}
```

```
0  =>  element 0 Ali
1  =>  element 1 James
2  =>  element 2 Ravi
3  =>  element 3 David
4  =>  element 4 Erik

[Done] exited with code=0 in 1.802 seconds
```

- Range is used to iterate over all the elements of an array.

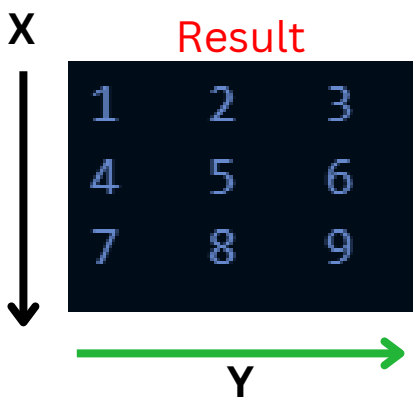
Arrays

Multi-Dimensional Array

```go
package main
import "fmt"

func main() {
    var num [3][3]int = [3][3]int{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

    for i := 0; i < len(num); i++ {
        for j := 0; j < len(num[i]); j++ {
            fmt.Printf("%v\t", num[i][j])
        }
        fmt.Println("")
    }
}
```

iterate over rows

iterate over columns



Slices

A slice in Go is a more flexible and powerful abstraction than an array. It provides a way to work with sequences of elements that are dynamically-sized. Their size can be changed after their creation.Slices are built on top of arrays and offer more functionality and flexibility

- A slice is essentially a view / reference to an underlying array.
- Slices can grow or shrink in size. You can use built-in functions like append() to modify the size of a slice.
- Multiple slices can share the same underlying array.
- Modifications to the slice affect the underlying array and any other slices that reference it.
- It is also possible to create a sub-slice from a slice

Properties

Length	Capacity	Pointer

Slices

```go
package main

import "fmt"

func main() {
    //declare an array
    var arr [5]int = [5]int{1, 2, 3, 4, 5}

    //declare a slice
    slice1_arr_1 := arr[1:3] // this contains elements 2,3

    fmt.Println("Slice 1 array 1", slice1_arr_1)

    //to append
    slice1_arr_1 = append(slice1_arr_1, 7)
    fmt.Println("Slice 1 after append", slice1_arr_1)

    fmt.Println("Original array: ", arr)
}
```

```
Slice 1 array 1 [2 3]
Slice 1 after append [2 3 7]
Original array:  [1 2 3 7 5]

[Done] exited with code=0 in 1.986 seconds
```

- An array's size determines the maximum possible size of any slice created from it.
- **You cannot create a slice larger than the array from which it is derived.**

Dynamic Slices

```go
package main

import "fmt"

func main() {
    //under lying array is taken care by compiler
    initialSlice := []int{1, 2, 3, 4, 5}

    //print initial slice
    fmt.Println(initialSlice)

    for i := 6; i < 15; i++ {
        initialSlice = append(initialSlice, i)

    }

    fmt.Println("Updated slice: ", initialSlice)

}
```

- At times we want to increase the size dynamically for the slice, to overcome that you can create a slice without creating an array, the underlying array will be created by the compiler.
- If you want to add the values that are more then an array size the compiler will create a bigger size array and move the elements to the new arrays

Slicing

array1[start_index:end_index]

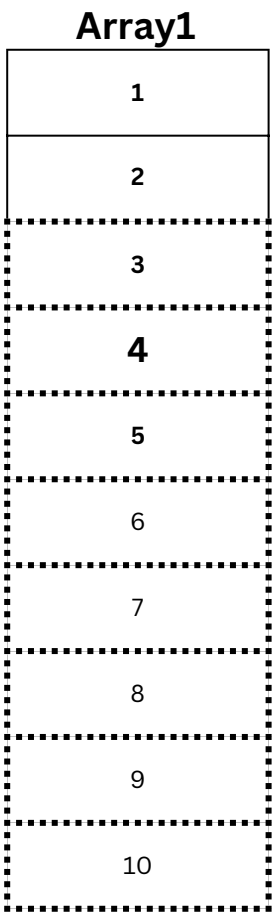
When you create a slice the start index is included but the end index is not included.

Show the whole array
array1[:]

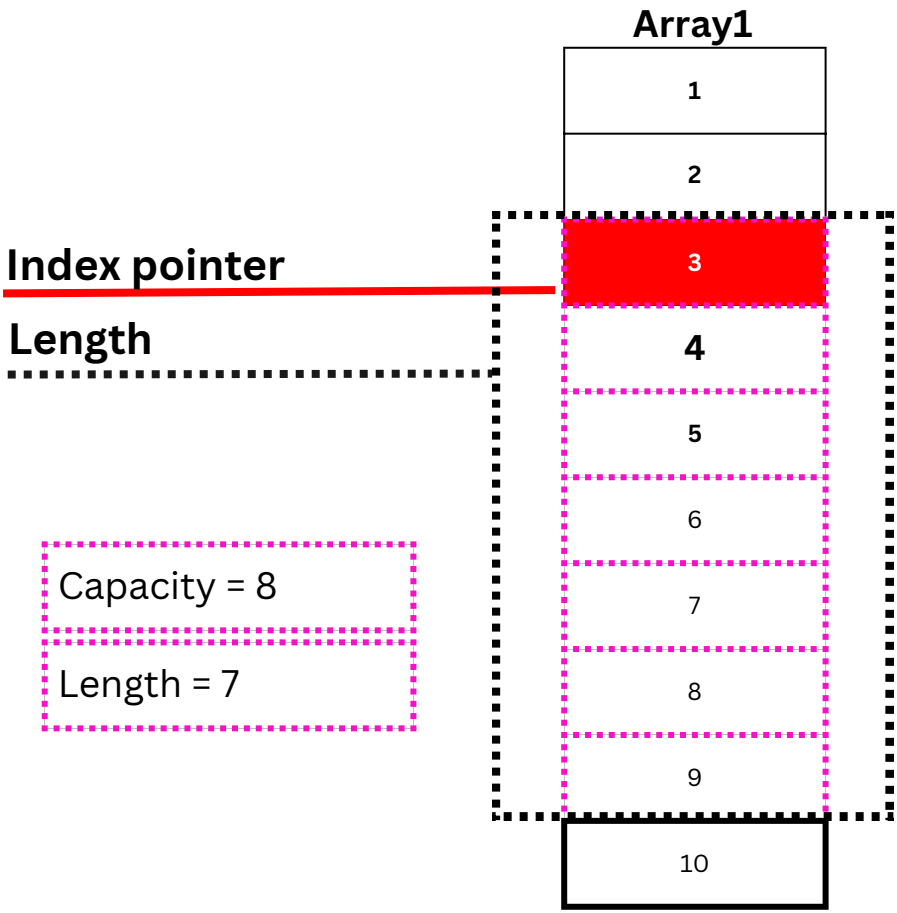
Show value 3-6
array1[2:6]

Show value until 6
array1[:6]

Show value from 4 till end
array1[3:]



Length , Capacity , Pointers



- Capacity is the size from pointer to the end of the array
- Length is the size of the slice
- Pointer is the slice starting address

Subslice

```go
package main

import "fmt"

func main() {
    //under lying array is taken care by compiler
    initialSlice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    sub_slice := initialSlice[1:5]
    //print initial slice
    fmt.Println("Initial Slice: ", initialSlice)
    fmt.Println("Sub Slice: ", sub_slice)

    sub_slice[3] = 900
    fmt.Println("Sub slice after value modification", sub_slice)
    fmt.Println("Initial Slice after value modification in sub_slice\n ", initialSlice)

}
```

1. Both slice refer to the same array
2. changes on one will cause the change to the other as well.

Using make function

```go
package main

import "fmt"

func main() {
    slice1 := make([]int, 5, 10)

    fmt.Printf("Slice with Default Values: %v", slice1)
    fmt.Printf("Slice Length: %v\n", len(slice1))
    fmt.Printf("Slice Capacity: %v\n", cap(slice1))

}
```

```
Slice with Default Values: [0 0 0 0 0]
Slice Length: 5
Slice Capacity: 10
```

- Capacity and length function works on both Slices and Arrays

Array and Slice index difference

```go
package main

import "fmt"

func main() {
    // Create a dynamic slice of 10 elements with values from 1 to 10
    dynamicSlice := make([]int, 10) // Create a slice of length 10
    for i := 0; i < len(dynamicSlice); i++ {
        dynamicSlice[i] = i + 1 // Fill the slice with values 1 to 10
    }

    // Print the original slice with index and values
    fmt.Println("Original Slice:")
    for index, value := range dynamicSlice {
        fmt.Printf("Index %d: Value %d\n", index, value)
    }

    // Create a sub-slice of 5 elements from the dynamic slice
    // Let's take elements from index 2 to index 6
    subSlice := dynamicSlice[2:7]

    // Print the sub-slice with index and values
    fmt.Println("\nSub-Slice:")
    for index, value := range subSlice {
        // Note: Index here is the index in the sub-slice, not in the original slice
        fmt.Printf("Sub-Slice Index %d: Value %d\n", index, value)
    }

    // Print the sub-slice starting index in the original slice
    fmt.Printf("\nSub-Slice starting index in the original slice: %d\n", 2)
}
```

- **Sub-slice has its own indexing which are reference to the array indexes.**

Adding slices

```go
package main

import "fmt"

func main() {

    arr1 := []int{1, 3, 5, 7, 9}
    arr2 := []int{2, 4, 6, 8, 10}

    arr_final := append(arr1, arr2...)
    fmt.Println(arr_final)

}
```

```
[1 3 5 7 9 2 4 6 8 10]

[Done] exited with code=0 in 3.653 seconds
```

Deleting Elements from Slice

To delete an element from a slice in Go, you generally need to create a new slice that excludes the element you want to remove.

There isn’t a built-in method to directly remove elements from slices.

Copying from a slice

Copying data from one slice to another in Go is a common task and is straightforward using the built-in copy function.

func copy(dst, src []T) int

1. Returns number of elements copied
2. Requires destination slice
3. Requires source slices

```go
package main

import "fmt"

func main() {
    // Original slice
    src := []int{1, 2, 3, 4, 5}
    fmt.Println("Source Slice:", src)

    // Create a destination slice with smaller length
    dst := make([]int, 3)

    // Copy data from src to dst
    copied := copy(dst, src)
    fmt.Printf("Number of elements copied: %d\n", copied)

    // Print both slices
    fmt.Println("Destination Slice:", dst)
}

```

Looping a Slice

Looping a slice is similar to looping an Array.

```go
package main

import "fmt"

func main() {
    name_slice := []string{"Ali", "Erik", "James", "Ravi", "David"}

    for i := 0; i < len(name_slice); i++ {
        fmt.Println(name_slice[i])
    }
}
```

```
Ali
Erik
James
Ravi
David

[Done]
```

Conclusion

- Arrays in Go have a fixed size defined at compile time and cannot be resized.
- They are value types, copying an array involves copying its entire contents.
- Slices, can grow or shrink dynamically.
- Slices are built on top of arrays
- Slices are reference types, meaning they point to an underlying array
- Key properties of slices include their length and capacity, and index pointer.
- Copy function can be used to copy data between slices
- Arrays are best suited for fixed-size data collections, while slices offer more flexibility for dynamic data management.

LinkedIn
@mushtaqalih

Maps

Maps are used for storing and retrieving data with unique keys efficiently.

Declare and Initialize

```
name_age := map[string]int{"Ali": 50, "Erik": 25, "Ravi": 33, "Sam": 44}
```

Printing out the map

```
fmt.Println(name_age) // printing out the map
fmt.Println("The length of the map is: ", len(name_age))
```

Creating an empty map

```
score_map := make(map[string]int)
fmt.Println(score_map) //this is empty key map
```

Accessing individual key / value in map

```
//accessing individual items in a map
fmt.Printf("%v\t,%v\t", name_age["Ali"], name_age["Erik"])
```

Maps

Adding new key / value to the map

```
//adding a new key / value to the list
name_age["Rahul"] = 24
fmt.Println(name_age)
```

Updating the value

```
name_age["Ali"] = 40
fmt.Println(name_age)
```

Deleting a particular value

```
delete(name_age, "Rahul")
fmt.Println(name_age)
```

Looping over the list

```
for key, value := range name_age {
    fmt.Println(key, " => ", value)
}
```

Re-initialize / Null the list

```
name_age = make(map[string]int)
fmt.Println("Printing the new length of Map: ", len(name_age))
```

Maps

Iterate and Delete from list

```
fmt.Println("Deleting keys with iteration")
// to loop over the list of key values
for key, value := range name_age {
    fmt.Println(key, "=>", value)
    delete(name_age, key)
}
```

Verify if the value exists

```
package main

import "fmt"

func main() {
        // Declare and initialize a map with user ages
        users := map[string]int{
                "John":    30,
                "Jane":    25,
                "Emily":   35,
                "Michael": 40,
        }

        // Check if the user "Jane" exists and retrieve age
        if age, exists := users["Jane"]; exists {
                // This block executes if "Jane" exists in the map
                fmt.Println("Jane's age is:", age)
        } else {
                // This block executes if "Jane" does not exist in the map
                fmt.Println("Jane is not in the list")
        }

        // Check if the user "Alice" exists and retrieve age
        if age, exists := users["Alice"]; exists {
                // This block executes if "Alice" exists in the map
                fmt.Println("Alice's age is:", age)
        } else {
                // This block executes if "Alice" does not exist in the map
                fmt.Println("Alice is not in the list")
        }
}
```

Functions

LinkedIn
@mushtaqalih

Functions

Functions are fundamental building blocks used to encapsulate code into reusable units.

- Functions help with
1. organize code
2. reduce redundancy
3. improve readability.

- Declaring a function

A function is declared using the **func** keyword, followed by the function name, a list of parameters, a return type, and a block of code.

- Parameters and Return Values

Parameters: Variables passed to the function to provide input.
Return Values: Values that the function returns to the caller after execution.

- Function call

A function is called by using its name followed by parentheses. Parameters are passed within the parentheses if needed.

- Multiple return values

Go functions can return multiple values, which can be useful for functions that need to return more than one result or an error.

- Named Return values

Functions can have named return values, which are variables declared in the function signature. They can be used to simplify code by eliminating the need for explicit return statements.

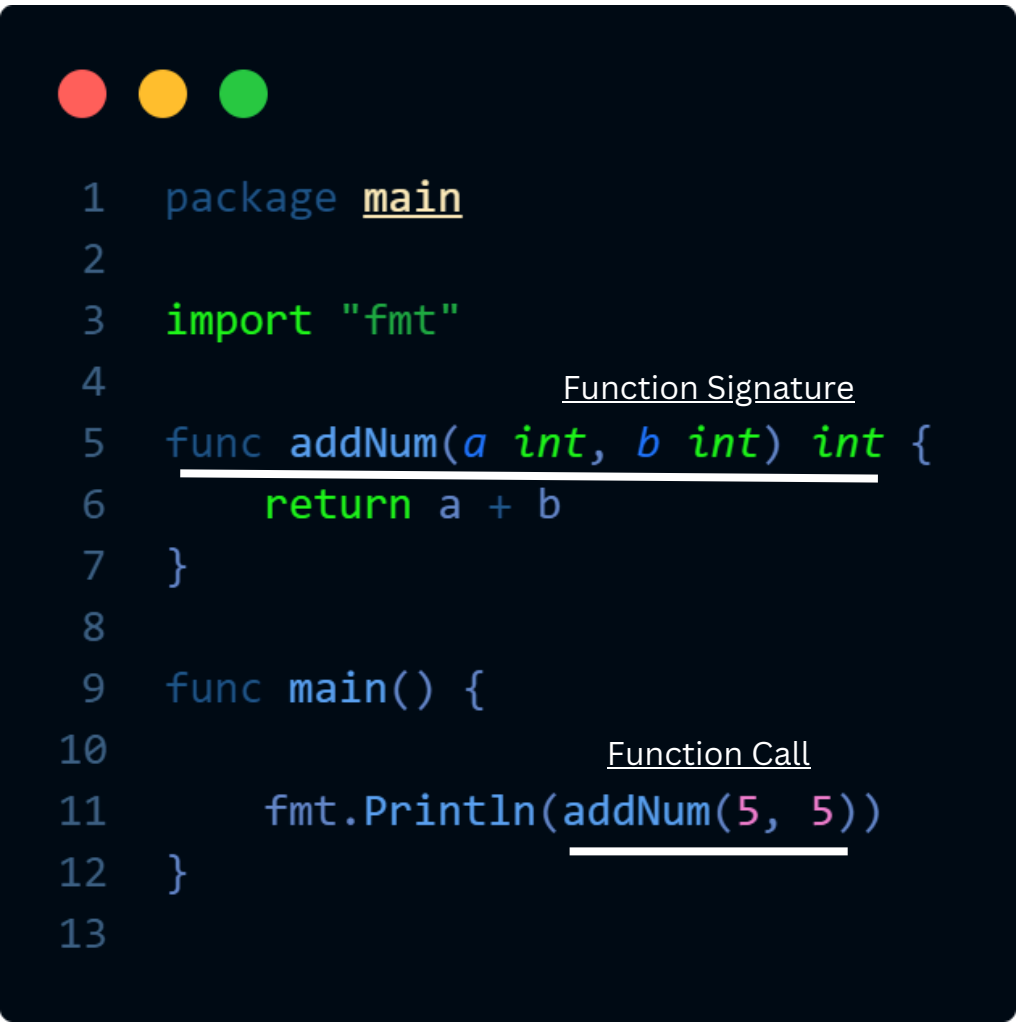
Functions

Functions are fundamental building blocks used to encapsulate code into reusable units.

```go
package main

import "fmt"

// Function to add two integers
func add(a int, b int) int {
    return a + b
}

// Function to multiply two integers and return both the result and the result squared
func multiplyAndSquare(a int, b int) (int, int) {
    product := a * b
    return product, product * product
}

func main() {
    // Call the add function
    sum := add(5, 3)
    fmt.Println("Sum:", sum)

    // Call the multiplyAndSquare function
    product, square := multiplyAndSquare(4, 6)
    fmt.Println("Product:", product)
    fmt.Println("Square of Product:", square)
}
```

Functions

Functions are fundamental building blocks used to encapsulate code into reusable units.




```go
package main

import "fmt"

func addNum(a int, b int) int {
    return a + b
}

func main() {
    fmt.Println(addNum(5, 5))
}
```

Naming Convention

- must begin with letter
- spaces in function name are not allowed
- function names are case sensitive

Functions

Functions are fundamental building blocks used to encapsulate code into reusable units.

```go
package main

import "fmt"

func greeting(str string) {
    fmt.Println("Hey", str)
}
func main() {
    greeting("Ali")
}

```

- Function without any return value

Functions

Functions are fundamental building blocks used to encapsulate code into reusable units.

```go
package main

import "fmt"

func square(num int) int {
    num1 := num * num
    fmt.Println(num1)
    return "String"
}
func main() {
    square(5)
}

```

```
# command-line-arguments
function2\main.go:8:9: cannot use "String" (untyped string constant) as int value in return statement

[Done] exited with code=1 in 1.037 seconds
```

Wrong return type

Functions

Functions are fundamental building blocks used to encapsulate code into reusable units.

```go
package main

import "fmt"

func calc(num1 int, num2 int) (int, int) {
    sum := num1 + num2
    diff := num1 - num2
    return sum, diff
}
func main() {
    sum, diff := calc(5, 5)
    fmt.Println(sum, diff)
}
```

Return multiple values

Functions

Functions are fundamental building blocks used to encapsulate code into reusable units.

```go
package main

import "fmt"

func calc(num1 int, num2 int) (sum int, diff int) {
    sum = num1 + num2
    diff = num1 - num2
    return
}
func main() {
    sum, diff := calc(5, 5)
    fmt.Println(sum, diff)
}
```

- **If you have declared variable in signature you wont need to declare this again in return statement**
- **Return statement is called the terminating statement as it ends the function and returns the value**

Functions

Variadic parameters

- Accepts variable number of arguments
- All the arguments must be of the same data type
- To declare the final parameter is preceded by three dots

```go
package main

import "fmt"

func calc_num(nums ...int) int {
    sum := 0
    for _, number := range nums {
        sum += number
    }
    return sum
}

func main() {
    total := calc_num(10, 15, 30)
    fmt.Println("The total is ", total)
}

```

Functions

Variadic parameters

- Variadic parameter comes last in parameters.

```go
package main

import "fmt"

func calc_num(name string, nums ...int) int {
    sum := 0
    fmt.Println("Name is ", name)
    for _, number := range nums {
        sum += number
    }
    return sum
}

func main() {
    name := "Ali"
    total := calc_num(name, 10, 15, 30)
    fmt.Println("The total is ", total)
}
```

Functions

Blank Identifier _

- underscore is a blank identifier , if let say function returns two values but you need only one for further data processing you can place _ in place of the variable that you dont require.

```go
package main

import "fmt"

func calc(num1 int, num2 int) (int, int) {
    addition := num1 + num2
    substraction := num1 - num2
    return addition, substraction
}
func main() {
    add_total, _ := calc(10, 20)
    _, sub_total := calc(10, 20)

    fmt.Println("Addition result: ", add_total)
    fmt.Println("Substraction result: ", sub_total)
}

```

Functions

Recursive Function

A recursive function is a function that calls itself in order to solve a problem. The idea is to break down a problem into smaller, more manageable instances of the same problem, until reaching a base case that can be solved directly.

- Simplifies Code

  Recursion can simplify the code for problems that have a natural recursive structure, making it easier to write and understand.

- Divide Problems

  Recursion is useful for problems that can be divided into similar sub-problems.

- Elegant Solution

  Some problems are more naturally expressed and solved with recursion, such as tree traversal or mathematical calculations.

Functions

Recursive Function

```go
package main

import "fmt"

func factorial(num int) int {
    if num == 0 {
        return 1
    }

    return num * factorial(num-1)
}

func main() {
    num := 5
    result := factorial(num)
    fmt.Println("The factorial of ", num, "is ", result)
}
```

Functions

Anonymous Function

An anonymous function, also known as a lambda or a function literal, is a function that does not have a name.

It is defined inline, typically where it is used, and can be assigned to a variable, passed as an argument, or returned from another function.

- Concise

This type of function allow you to define and use functions on the fly without needing to create a separate named function.

- Local Scope

They are useful for tasks that are specific to a particular part of the code

- Functional Programming

They support functional programming paradigms where functions are first-class citizens and can be passed around as values.

Functions

Anonymous Function

```go
package main

import "fmt"

func main() {
    num := func(a int, b int) int {
        return a + b
    }(9, 5)

    fmt.Println(num)
}

```

Use Cases

1. Inline callback
2. Use with event handlers
3. best for quick tasks

Functions

Higher Order Function

Higher-order functions are functions that operate on other functions. Specifically, a higher-order function can:

1. Accept one or more functions as arguments.
2. Return a function as its result.

```go
package main

import "fmt"

// Higher-order function that takes a function and applies it to a number
func apply(num int, operation func(int) int) int {
    return operation(num)
}

func main() {
    // Define some simple functions
    addOne := func(n int) int { return n + 1 }
    multiplyByTwo := func(n int) int { return n * 2 }

    // Use the higher-order function with different operations
    result1 := apply(5, addOne)        // Adds 1 to 5
    result2 := apply(5, multiplyByTwo) // Multiplies 5 by 2

    fmt.Println("5 after adding 1:", result1)
    fmt.Println("5 after multiplying by 2:", result2)
}

```

- Line 11 - 12 defines anonymous function
- Line 6 has higher order function that has function in required parameter.
- Line 16-17 call apply function and add addOne as an argument to get the result.

Functions

Deferred Function

a deferred function is a function that is scheduled to run after the surrounding function completes, just before it returns.
Deferred functions are commonly used for cleanup tasks, such as closing files or unlocking resources, and ensure that the cleanup code runs even if an error occurs or the function exits prematurely.

```go
package main

import "fmt"

func main() {
    fmt.Println("Starting main function")

    // Deferred function to demonstrate cleanup
    defer func() {
        fmt.Println("Deferred function executed")
    }()

    // Other operations
    fmt.Println("Performing operations")
}
```

Pointers

Pointers

Pointers are variables that store the memory address of another variable. Instead of holding a data value directly, a pointer holds the address where the data is stored in memory. This allows you to directly manipulate the value at that address.

Pointer uses:

Improves program efficiency:
Passing large structs or arrays to functions by reference (using pointers)

Direct Memory Access:
Allows direct access and modification of memory locations.

Pointers

```go
package main

import "fmt"

func main() {
    var a int = 42
    var p *int = &a // p stores the address of a

    fmt.Println("Value of a:", a) //value 42
    fmt.Println("Address of a:", &a) // Output: memory address of a
    fmt.Println("Pointer p:", p) // Output: memory address of a
    fmt.Println("Value at pointer p:", *p) // Output: 42

    *p = 100   // Modify the value at the address pointed by p
    fmt.Println("New value of a:", a) // Output: 100
}
```

Declaration

- var p *int declares a pointer to an int
- p = &a assigns the address of variable a to pointer p

Dereferencing

- *p accesses the value stored at the memory address held by p.

Modification

- Changing the value of *p will change the value at the address &a as visible on line 14 - 15.

```go
var pointer2_add = &a
fmt.Println(pointer2_add, " => ", *pointer2_add)
```

Pass by Value / Reference

Pass by Value

The function receives a copy of the variable's value. Changes made to the parameter inside the function do not affect the original variable.

```go
package main

import "fmt"

// integer by value
func incrementByValue(num int) {
    num++
}

func main() {
    var x int = 10
    fmt.Println("Original value:", x)

    incrementByValue(x)  // Pass x by value
    fmt.Println("After incrementByValue:", x) // original value unchanged
}
```

Pass by Value / Reference

Pass by Reference

The function receives a reference (or pointer) to the variable. Changes made to the parameter inside the function affect the original variable.

```go
package main

import "fmt"

//  integer pointer (pass by reference)
func incrementByReference(num *int) {
    *num++
}

func main() {
    var x int = 10
    fmt.Println("Original value:", x)

    incrementByReference(&x)
    fmt.Println("After incrementByReference:", x)
}
```

Struct , Method & Interfaces

LinkedIn
@mushtaqalih

Struct vs OOP

Structs and methods are related to Object-Oriented Programming (OOP) concepts, but they do not fully align with traditional OOP as found in languages like Java or C++.

Struct: Go uses structs to define data structures that group together variables (fields) under a single name.

Methods in Go are functions associated with a specific type (often structs).

Encapsulation

Go: Achieved using structs and methods.

OOP: Achieved through classes, which encapsulate data and methods together.

Inheritance

Go: Does not support classical inheritance.

OOP: Supports inheritance, allowing one class to inherit properties and methods from another.

Polymorphism

Go: Achieved through interfaces.

OOP: Achieved through method overriding and interfaces or abstract classes.

Abstraction

Go: Achieved through interfaces.

OOP: Achieved through method overriding and interfaces or abstract classes.

LinkedIn
@mushtaqalih

Struct

```go
package main

import "fmt"

// Define the Person struct
type Person struct {
    Age  int
    Name string
}

// Method for the Person type
func (p Person) Display() {
    fmt.Printf("Name: %s, Age: %d\n", p.Name, p.Age)
}

func main() {
    // Create multiple instances of Person
    person1 := Person{
        Age:  30,
        Name: "Alice",
    }

    person2 := Person{
        Age:  25,
        Name: "Bob",
    }

    person3 := Person{
        Age:  40,
        Name: "Charlie",
    }

    person4 := Person{
        Age:  35,
        Name: "Diana",
    }

    // Call the Display method for each instance
    person1.Display()
    person2.Display()
    person3.Display()
    person4.Display()
}

```

Struct

Accessing struct values

```go
package main

import "fmt"

// Define the Person struct
type Person struct {
    Age  int
    Name string
}

// Function that takes a struct by value
func PrintPerson(p Person) {
    // Modify the copy of the struct
    p.Age = 99
    fmt.Printf("Inside PrintPerson: Name: %s, Age: %d\n", p.Name, p.Age)
}

func main() {
    // Create an instance of Person
    p := Person{
        Age:  30,
        Name: "Alice",
    }

    // Call the function with the struct
    PrintPerson(p)

    // Print the original struct
    fmt.Printf("Outside PrintPerson: Name: %s, Age: %d\n", p.Name, p.Age)
}
```