Here’s a concise guide to help you get started with Golang

Auther : SHUBHAM BASU | GOLANG TRAINER | +91-7020660328

Introduction to Go

- **Creator:** Developed by Google (Robert Griesemer, Rob Pike, and Ken Thompson).
- **Purpose:** Simplicity, efficiency, and concurrency.
- **Main Features:**
  - Statically typed.
  - Compiled language with fast execution.
  - Built-in support for concurrent programming.

Basic Syntax

1. **Structure of a Go Program:**

```go
package main // Every Go program starts with the 'main' package.

import "fmt" // Importing standard library packages.

func main() { // The entry point function.
    fmt.Println("Hello, World!") // Print to console.
}
```

2. **Variable Declaration:**

```go
// Explicit type
var x int = 10

// Type inference
y := 20

// Multiple variables
var a, b, c = 1, true, "GoLang"
```

3. **Constants:**

```go
const pi = 3.14
```

4. **Functions:**

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

func main() {
    result := add(2, 3)
    fmt.Println(result)
}
```

Control Structures

1. **If-Else:**

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

2. **Switch:**

```
switch day := 5; day {
case 1:
    fmt.Println("Monday")
case 5:
    fmt.Println("Friday")
default:
    fmt.Println("Unknown day")
}
```

3. **For Loop:**

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
```

4. **While Equivalent:**

```
i := 0
for i < 5 {
    fmt.Println(i)
    i++
}
```

Data Structures

1. **Arrays:**

```go
var arr [5]int
arr[0] = 10
fmt.Println(arr)
```

2. **Slices:**

```go
nums := []int{1, 2, 3}
nums = append(nums, 4) // Append new elements
fmt.Println(nums)
```

3. **Maps (Dictionaries):**

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

Pointers

```go
var x = 10
var p *int = &x // Pointer to x
fmt.Println(*p) // Dereferencing the pointer
```

Structs

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

func main() {
    p := Person{Name: "Alice", Age: 25}
    fmt.Println(p.Name)
}
```

Concurrency

1. **Goroutines:**

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

func main() {
    go sayHello() // Runs in a separate thread
    fmt.Println("Main Function")
}
```

2. **Channels:**

```
ch := make(chan int)

go func() {
    ch <- 42 // Send data
}()

fmt.Println(<-ch) // Receive data
```

Error Handling

```
func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return a / b, nil
}

func main() {
    result, err := divide(10, 0)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println(result)
    }
}
```

File Handling

```go
import (
    "os"
    "log"
)

func main() {
    file, err := os.Create("test.txt")
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    file.WriteString("Hello, File!")
}
```

Useful Go Commands

- `go run file.go` – Run the program.
- `go build` – Compile the program.
- `go fmt` – Format your code.
- `go test` – Run tests.
- `go mod init` – Initialize a module.
- `go get` – Fetch dependencies.

Here's an extended version of the Go notes including **methods** and **interfaces**:

Methods in Go

- Methods are functions with a receiver argument.
- Receivers can be either value receivers or pointer receivers.

1. **Defining Methods:**

```go
package main

import "fmt"

type Rectangle struct {
    Length, Width float64
}

// Method with value receiver
func (r Rectangle) Area() float64 {
    return r.Length * r.Width
}

// Method with pointer receiver
```

```go
func (r *Rectangle) Scale(factor float64) {
    r.Length *= factor
    r.Width *= factor
}

func main() {
    rect := Rectangle{Length: 10, Width: 5}

    fmt.Println("Area:", rect.Area())

    rect.Scale(2) // Modifies the original rect
    fmt.Println("Scaled Area:", rect.Area())
}
```

Interfaces in Go

- Interfaces define a set of methods that a type must implement.
- They allow for polymorphism and dynamic behavior.

1. **Defining an Interface:**

```go
package main

import "fmt"

type Shape interface {
    Area() float64
    Perimeter() float64
}

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}

func (c Circle) Perimeter() float64 {
    return 2 * 3.14 * c.Radius
}

func main() {
    var s Shape
    s = Circle{Radius: 5}

    fmt.Println("Area:", s.Area())
    fmt.Println("Perimeter:", s.Perimeter())
}
```

2. **Empty Interface (`interface{}`):**
    - Represents any type.
    - Useful for creating generic structures.

```
func printValue(v interface{}) {
    fmt.Println("Value:", v)
}

func main() {
    printValue(42)
    printValue("Hello")
    printValue(3.14)
}
```

3. **Type Assertion:**

```
func describe(i interface{}) {
    switch v := i.(type) {
    case int:
        fmt.Println("Integer:", v)
    case string:
        fmt.Println("String:", v)
    default:
        fmt.Println("Unknown Type")
    }
}

func main() {
    describe(42)
    describe("GoLang")
    describe(3.14)
}
```

Practical Use of Interfaces

1. **Custom Error Handling:**

```
package main

import "fmt"

type MyError struct {
    Message string
}

func (e MyError) Error() string {
```

```go
    return e.Message
}

func doSomething() error {
    return MyError{Message: "Something went wrong"}
}

func main() {
    err := doSomething()
    if err != nil {
        fmt.Println(err)
    }
}
```

2. **Using Interfaces with Structs:**

```go
package main

import "fmt"

type Animal interface {
    Speak() string
}

type Dog struct{}
type Cat struct{}

func (d Dog) Speak() string {
    return "Woof!"
}

func (c Cat) Speak() string {
    return "Meow!"
}

func main() {
    animals := []Animal{Dog{}, Cat{}}
    for _, animal := range animals {
        fmt.Println(animal.Speak())
    }
}
```

Embedding Interfaces

- Go allows interfaces to be embedded within each other.

```go
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

Key Points

- Methods with **value receivers** work on copies of the original value.
- Methods with **pointer receivers** allow modifying the original value.
- An **interface** is satisfied implicitly when a type implements all its methods.
- Use the **empty interface (`interface{}`)** for generic types but avoid overusing it, as it can lead to less type safety.

Struct Embedding in Go

Struct embedding is a feature in Go that allows you to include one struct inside another. This provides a way to achieve composition and reuse functionality from embedded structs.

How Struct Embedding Works

- When you embed a struct into another struct, all the fields and methods of the embedded struct are promoted to the outer struct.
- The outer struct can directly access the fields and methods of the embedded struct.

Basic Example

```go
package main

import "fmt"

// Embedded struct
type Address struct {
    City, State string
}

// Outer struct
type Person struct {
    Name    string
    Age     int
    Address // Embedding Address struct
}

func main() {
    p := Person{
        Name: "Alice",
        Age:  30,
        Address: Address{
            City:  "Pune",
            State: "Maharashtra",
        },
    }

    // Accessing fields of the embedded struct directly
    fmt.Println("Name:", p.Name)
    fmt.Println("Age:", p.Age)
    fmt.Println("City:", p.City)   // Promoted field
    fmt.Println("State:", p.State) // Promoted field
}
```

Method Promotion with Struct Embedding

Methods of the embedded struct are also promoted to the outer struct.

```go
package main

import "fmt"

// Embedded struct
type Address struct {
    City, State string
}

func (a Address) FullAddress() string {
    return a.City + ", " + a.State
}

// Outer struct
type Person struct {
    Name    string
    Age     int
    Address
}

func main() {
    p := Person{
        Name: "Bob",
        Age:  40,
        Address: Address{
            City:  "Mumbai",
            State: "Maharashtra",
        },
    }

    // Calling method of embedded struct
    fmt.Println("Full Address:", p.FullAddress()) // Promoted method
}
```

Overriding Embedded Fields or Methods

The outer struct can override fields or methods of the embedded struct.

1. **Overriding Fields:**

```go
package main

import "fmt"

type Address struct {
    City string
}

type Person struct {
    Name    string
    Address
    City string // Overrides the City field of Address
}

func main() {
    p := Person{
        Name: "Charlie",
        Address: Address{
            City: "Delhi",
        },
        City: "Bangalore", // Overriding City field
    }

    fmt.Println("Outer City:", p.City)       // Outer City field
    fmt.Println("Embedded City:", p.Address.City) // Embedded struct field
}
```

2. **Overriding Methods:**

```go
package main

import "fmt"

type Address struct {
    City, State string
}

func (a Address) FullAddress() string {
    return a.City + ", " + a.State
}

type Person struct {
    Name    string
    Age     int
    Address
}

// Overriding method
func (p Person) FullAddress() string {
    return p.Name + " lives in " + p.Address.FullAddress()
}

func main() {
    p := Person{
        Name: "Diana",
        Age:  35,
        Address: Address{
            City:  "Chennai",
            State: "Tamil Nadu",
        },
    }

    // Call the overridden method
    fmt.Println(p.FullAddress())
}
```

Anonymous Embedding

You can embed structs anonymously (without giving them a field name). This is the usual way of struct embedding in Go.

- If you give the embedded struct a name, you will need to access fields and methods using that name.

```go
package main

import "fmt"

type Address struct {
    City string
}

type Person struct {
    Name string
    Age  int
    Addr Address // Named embedding
}

func main() {
    p := Person{
        Name: "Eve",
        Age:  28,
        Addr: Address{
            City: "Kolkata",
        },
    }

    // Accessing fields of the named embedded struct
    fmt.Println("City:", p.Addr.City) // Must use Addr to access City
}
```

Key Points

1. Struct embedding promotes fields and methods of the embedded struct to the outer struct.
2. Fields and methods in the outer struct can override those in the embedded struct.
3. Anonymous embedding (no field name) is the idiomatic way of struct embedding in Go.
4. Struct embedding is **composition**, which Go prefers over inheritance.

Anonymous Functions in Go

An **anonymous function** in Go is a function without a name. It is commonly used as a **closure**, where it captures and uses variables from its surrounding scope.

Basic Syntax

```go
package main

import "fmt"

func main() {
    // Defining and calling an anonymous function immediately
    func() {
        fmt.Println("Hello from an anonymous function!")
    }()

    // Defining and assigning an anonymous function to a variable
    greet := func(name string) {
        fmt.Printf("Hello, %s!\n", name)
    }

    greet("Alice") // Calling the anonymous function
    greet("Bob")
}
```

Returning a Value

Anonymous functions can return values like any other function.

```go
package main

import "fmt"

func main() {
    add := func(a, b int) int {
        return a + b
    }

    result := add(5, 3)
    fmt.Println("Sum:", result)
}
```

Closures

An anonymous function can capture variables from its surrounding scope.

```go
package main

import "fmt"

func main() {
    // Closure capturing `count` variable
    count := 0
    increment := func() int {
        count++
        return count
    }

    fmt.Println(increment()) // 1
    fmt.Println(increment()) // 2
    fmt.Println(increment()) // 3
}
```

Passing Anonymous Functions as Arguments

Anonymous functions can be passed as arguments to other functions.

```go
package main
```

```
import "fmt"

func operate(a, b int, op func(int, int) int) int {
    return op(a, b)
}

func main() {
    // Passing an anonymous function as an argument
    result := operate(10, 5, func(x, y int) int {
        return x + y
    })
    fmt.Println("Addition:", result)

    result = operate(10, 5, func(x, y int) int {
        return x * y
    })
    fmt.Println("Multiplication:", result)
}
```

Returning Anonymous Functions

A function can return an anonymous function, which is often used to create closures.

```
package main

import "fmt"

func multiplier(factor int) func(int) int {
    return func(x int) int {
        return x * factor
    }
}

func main() {
    double := multiplier(2)
    triple := multiplier(3)

    fmt.Println("Double of 4:", double(4)) // 8
    fmt.Println("Triple of 4:", triple(4)) // 12
}
```

Anonymous Goroutines

Anonymous functions are often used as goroutines for concurrent execution.

```
package main
```

```go
import (
    "fmt"
    "time"
)

func main() {
    go func() {
        fmt.Println("Hello from a goroutine!")
    }()

    time.Sleep(time.Second) // Wait for the goroutine to finish
}
```

Practical Use Cases

1. **Event Handlers:**

```go
package main

import "fmt"

func onClick(handler func()) {
    fmt.Println("Button clicked!")
    handler()
}

func main() {
    onClick(func() {
        fmt.Println("Performing an action.")
    })
}
```

2. **Filtering Data:**

```go
package main

import "fmt"

func filter(nums []int, predicate func(int) bool) []int {
    result := []int{}
    for _, num := range nums {
        if predicate(num) {
            result = append(result, num)
        }
    }
    return result
}
```

```go
func main() {
    numbers := []int{1, 2, 3, 4, 5, 6}
    even := filter(numbers, func(n int) bool {
        return n%2 == 0
    })

    fmt.Println("Even numbers:", even)
}
```

Key Points

1. **Scope**: Anonymous functions can access variables from their outer scope (closures).
2. **Flexibility**: Useful for short-lived operations like callbacks, filtering, and custom logic.
3. **Readability**: Overuse can hurt code readability; use them where it makes sense.