Handy Golang Cheatsheet For Quick Reference

With Code Example

Go, also known as Golang, is a statically typed, compiled programming language designed for simplicity and efficiency. Developed by Google, Go emphasizes readability, conciseness, and robustness. With a rich standard library and support for concurrent programming, Go excels in building scalable, high-performance applications for web development, cloud computing, and system programming. Its features include goroutines for lightweight concurrency, channels for communication between goroutines, and interfaces for polymorphism. Go's fast compilation and execution, along with its strong focus on simplicity and efficiency, make it a popular choice for modern software development across various domains.

Hello world in Go

Copy

```go
package main

import "fmt"

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

Output:

Copy

```
Hello, World!
```

Variables in go

In Go, variables are declared using the var keyword followed by the variable name and type (optional). Here's a basic example:

Copy

```go
func main() {
  // Declaring variables
    var age int
    var name string = "John"
    var temperature float64 = 23.5

    // Short variable declaration (Type inference)
    var country string = "USA"
    var height float64 = 175.5

    // Assigning values to variables
    age = 30

    // Multiple variables
    var (
        a int
        b int
        c int
    )

    var (
        x int
        y = 20
        z int = 30
        d int
        e string = "Hello"
        f string
        g string
    )
}
```

Datatypes in Go

In Go, data types are used to define the type of data that a variable can hold. Here are the basic data types in Go along with examples:

Numeric Types:

- `int` : Represents integer numbers.

```
var age int = 30
```

- `int8` , `int16` , `int32` , `int64` : Signed integers with specific bit sizes (8, 16, 32, 64 bits).
- `uint` : Represents unsigned integer numbers.
- `uint8` , `uint16` , `uint32` , `uint64` : Unsigned integers with specific bit sizes (8, 16, 32, 64 bits).
- `float32` , `float64` : Represents floating-point numbers.

```
var temperature float64 = 23.5
```

- `complex64` , `complex128` : Represents complex numbers.

```
var z complex128 = complex(1, 2)
```

Boolean Type:

- `bool` : Represents boolean values (`true` or `false`).

```
var isSunny bool = true
```

String Type:

- `string` : Represents a sequence of characters.

```
var name string = "John"
```

Derived Types:

- `array` : Represents a fixed-size collection of elements of the same type.

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

- `slice` : Represents a dynamic-size collection of elements of the same type.

```
var fruits []string = []string{"apple", "banana", "orange"}
```

- `map` : Represents a collection of key-value pairs.

```
var ages map[string]int = map[string]int{"John": 30, "Alice": 25}
```

- `struct` : Represents a collection of fields.

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

```
var john Person = Person{Name: "John", Age: 30}
```

Pointer Type:

- `pointer` : Represents the memory address of a variable.

```
var ptr *int = &age
```

Function Type:

- `func` : Represents a function.

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

Interface Type:

- `interface` : Represents a set of method signatures.

emphasizes readability, conciseness, and robustness. With a rich standard library and support for concurrent programming, Go excels in building scalable, high-performance applications for web development, cloud computing, and system programming. Its features include goroutines for lightweight concurrency, channels for communication between goroutines, and interfaces for polymorphism. Go's fast compilation and execution, along with its strong focus on simplicity and efficiency, make it a popular choice for modern software development across various domains.

▶ Table of Contents

Hello world in Go

```
package main

import "fmt"

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

Output:

Overview of flow control in Go along with examples for each type:

Conditional Statements:

`if` Statement:

The `if` statement is used for conditional execution of code blocks.

```
func main() {
    x := 10
    if x > 5 {
        fmt.Println("x is greater than 5")
    } else {
        fmt.Println("x is less than or equal to 5")
    }
}
```

`switch` Statement:

The `switch` statement is used to make decisions based on the value of an expression.

```
func main() {
    fruit := "apple"
    switch fruit {
    case "apple":
        fmt.Println("Apple")
    case "banana":
        fmt.Println("Banana")
    default:
        fmt.Println("Unknown fruit")
    }
}
```

`for` Loop:

The `for` loop is used to repeatedly execute a block of code.

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

`for` Loop (while style):

Go doesn't have a `while` loop, but you can achieve the same behavior using a `for` loop with a condition.

```
func main() {
    x := 0
    for x < 5 {
        fmt.Println(x)
        x++
    }
}
```

`for` - `range` Loop:

The `for` - `range` loop is used to iterate over elements in arrays, slices, maps, or strings.

```
func main() {
    numbers := []int{1, 2, 3, 4, 5}
    for index, value := range numbers {
        fmt.Printf("Index: %d, Value: %d\n", index, value)
    }
}
```

Note

- Go provides various flow control mechanisms such as `if`, `switch`, `for`, and `for` - `range`.
- These constructs allow you to make decisions, iterate over collections, and control the flow of execution in your Go programs.

Functions in Go

Overview of functions in Go covering lambdas, multiple return types, and named return values:

Lambdas (Anonymous Functions):

In Go, anonymous functions, also known as lambda functions, can be declared inline without a name. They are often used for short, one-off functions.

```go
func main() {
    // Lambda function to add two numbers
    add := func(a, b int) int {
        return a + b
    }

    result := add(3, 5)
    fmt.Println("Result:", result) // Output: 8
}
```

Multiple Return Types:

Go supports returning multiple values from a function, which is commonly used for error handling or returning multiple results.

```go
// Function to calculate sum and difference of two numbers
func calculate(a, b int) (int, int) {
    sum := a + b
    diff := a - b
    return sum, diff
}

func main() {
    sum, diff := calculate(10, 5)
    fmt.Println("Sum:", sum)   // Output: 15
    fmt.Println("Diff:", diff) // Output: 5
}
```

Named Return Values:

In Go, you can specify names for the return values of a function. Named return values are initialized to their zero values and can be modified in the function body. They are particularly useful for improving code readability.

```go
// Function to divide two numbers and return quotient and remainder
func divide(a, b int) (quotient, remainder int) {
    quotient = a / b
    remainder = a % b
    return // Returns named return values
}

func main() {
    q, r := divide(10, 3)
    fmt.Println("Quotient:", q)   // Output: 3
    fmt.Println("Remainder:", r)  // Output: 1
}
```

> **Note**
>
> - Go supports lambdas, or anonymous functions, which are functions without a name.
> - Functions in Go can return multiple values, which is useful for various scenarios such as error handling.
> - Named return values in Go allow you to specify names for return values, improving code readability and enabling direct assignment of values in the function body.

Packages in Go

Overview of packages in Go covering importing, aliases, packages, and exporting names:

Importing Packages:

In Go, you can import packages using the `import` keyword followed by the package path. You can import multiple packages in a single `import` statement, and you can also use blank identifiers `_` to import packages for their side effects only.

```go
package main

import (
```

```
    "fmt"
    "math/rand"
    "time"
)
```

Aliases:

You can provide an alias for an imported package using the `import` statement to avoid name conflicts or to make the code more readable.

Copy

```
package main

import (
    "fmt"
    myrand "math/rand"
)
```

Packages:

In Go, a package is a collection of Go source files that reside in the same directory. You can think of a package as a namespace that organizes a set of related functions, types, and variables.

Copy

```
package mypackage

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

Exporting Names:

In Go, names that start with a capital letter are exported from the package and can be accessed by other packages. Names that start with a lowercase letter are not exported and are only accessible within the package.

Copy

```
package mypackage

// Exported function
func Add(a, b int) int {
    return a + b
}

// Not exported (only accessible within the package)
func multiply(a, b int) int {
    return a * b
}
```

Note

- Go allows you to import packages using the `import` keyword followed by the package path.
- You can provide an alias for an imported package to avoid name conflicts or improve readability.
- Packages in Go organize related functions, types, and variables into namespaces.
- Exported names in Go start with a capital letter and are accessible by other packages, while names starting with a lowercase letter are not exported and are only accessible within the package.

Concurrency in Go

Concurrency in Go is a powerful feature that allows you to write efficient and scalable programs. Here's an overview of key concepts in concurrency in Go, including goroutines, buffered channels, closing channels, and wait groups:

Goroutines:

Goroutines are lightweight threads of execution that allow functions to run concurrently with other functions. You can create a new goroutine using the `go` keyword followed by a function call.

Copy

```
package main

import (
    "fmt"
    "time"
)

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

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

    // Wait for a moment to allow the goroutine to execute
    time.Sleep(100 * time.Millisecond)
}
```

Buffered Channels:

Channels are a built-in synchronization primitive in Go used for communication and synchronization between goroutines. Buffered channels have a capacity specified during creation, allowing them to hold a limited number of values without a receiver.

Copy

```
package main

import (
    "fmt"
)

func main() {
    // Create a buffered channel with capacity 3
    ch := make(chan int, 3)

    // Send values to the channel
    ch <- 1
    ch <- 2
    ch <- 3

    // Receive values from the channel
    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Closing Channels:

Closing a channel indicates that no more values will be sent on it. You can close a channel using the `close` function. A closed channel can still be read from, but it will always return the zero value immediately.

Copy

```
package main

import (
    "fmt"
)

func main() {
    ch := make(chan int)

    go func() {
        for i := 0; i < 5; i++ {
            ch <- i
        }
        close(ch) // Close the channel after sending all values
    }()

    for num := range ch {
        fmt.Println(num)
    }
}
```

WaitGroup:

The `sync` package provides the `WaitGroup` type, which allows you to wait for a collection of goroutines to finish. You can add goroutines to the `WaitGroup` using the `Add` method, and then wait for them to finish using the `Wait` method.

```go
package main

import (
    "fmt"
    "sync"
)

func worker(id int, wg *sync.WaitGroup) {
    defer wg.Done() // Decrease the WaitGroup counter when the function finishes

    fmt.Printf("Worker %d starting\n", id)
    // Simulate some work
    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 1; i <= 3; i++ {
        wg.Add(1) // Increment the WaitGroup counter before starting a goroutine
        go worker(i, &wg)
    }

    wg.Wait() // Wait for all workers to finish
    fmt.Println("All workers done")
}
```

Note

- Goroutines are lightweight threads of execution in Go.
- Channels facilitate communication and synchronization between goroutines.
- Buffered channels have a capacity and can hold a limited number of values.
- Closing a channel indicates that no more values will be sent on it.
- WaitGroups allow you to wait for a collection of goroutines to finish before proceeding.

Error control in Go

Error control in Go is crucial for writing robust and reliable code. Here's an overview of how to handle errors and use the `defer` statement and deferring functions in Go:

Defer:

The `defer` statement is used to schedule a function call to be executed after the surrounding function returns. This can be useful for cleanup tasks or ensuring that resources are released, regardless of whether the function exits normally or with an error.

```go
package main

import "fmt"

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

In this example, the `Cleanup` function will be called after the `main` function returns, ensuring that it is executed regardless of whether an error occurs or not.

Deferring Functions:

You can also defer the execution of a specific function call. This is particularly useful for closing resources such as files or database connections.

```go
package main

import "fmt"

func cleanup() {
    fmt.Println("Cleanup")
}

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

In this example, the `cleanup` function will be called after the `main` function returns, ensuring that resources are properly cleaned up.

Error Handling:

Error handling in Go typically involves returning an error value from a function and checking for errors at the caller's side using the `if err != nil` pattern.

Copy

```go
package main

import (
    "errors"
    "fmt"
)

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

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

In this example, the `divide` function returns an error if the divisor is zero. The caller checks for errors using `if err != nil` and handles them accordingly.

> **Note**
>
> - Error control in Go is essential for writing robust and reliable code.
> - The `defer` statement is used to schedule a function call to be executed after the surrounding function returns.
> - Deferring functions can be used to ensure cleanup tasks or resource releases.
> - Error handling in Go typically involves returning an error value from a function and checking for errors at the caller's side using the `if err != nil` pattern.

Structs in Go

Structs in Go are composite data types that allow you to group together different types of data under a single name. Here's an overview of defining structs, using struct literals, and pointers to structs in Go:

Defining Structs:

You define a struct using the `type` keyword followed by the name of the struct and a set of field declarations within curly braces `{}`.

Copy

```go
// Define a struct named Person with fields Name and Age
type Person struct {
    Name string
    Age  int
}
```

```
func main() {
    // Create a variable of type Person and initialize its fields
    var p1 Person
    p1.Name = "Alice"
    p1.Age = 30

    // Accessing struct fields
    fmt.Println("Name:", p1.Name)
    fmt.Println("Age:", p1.Age)
}
```

Struct Literals:

You can initialize a struct using a struct literal, which specifies the values of its fields.

Copy

```
type Point struct {
    X, Y int
}

func main() {
    // Create a Point struct using a struct literal
    p := Point{X: 10, Y: 20}

    fmt.Println("X coordinate:", p.X)
    fmt.Println("Y coordinate:", p.Y)
}
```

Pointers to Structs:

You can also use pointers to structs, which allow you to modify the struct's fields directly.

Copy

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

func main() {
    // Create a pointer to a Person struct using the address-of operator (&)
    p := &Person{Name: "Bob", Age: 25}

    // Accessing and modifying struct fields using pointers
    fmt.Println("Name:", p.Name)
    fmt.Println("Age:", p.Age)

    p.Age = 30
    fmt.Println("Updated Age:", p.Age)
}
```

Note

- Structs in Go allow you to group together different types of data under a single name.
- You define structs using the `type` keyword followed by the name of the struct and a set of field declarations within curly braces `{}`.
- Struct literals are used to initialize structs with specific values for their fields.
- Pointers to structs allow you to modify the struct's fields directly and are created using the address-of operator `&`.

Methods in Go

Methods are functions associated with a particular type. Here's an overview of methods in Go, including receivers and mutation:

Receivers:

A receiver is a parameter in a method declaration that associates the method with a specific type. Methods with receivers are called on instances of that type.

Copy

```go
// Define a struct type
type Rectangle struct {
    Width, Height float64
}

// Method with a receiver of type Rectangle
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

func main() {
    // Create an instance of Rectangle
    rect := Rectangle{Width: 10, Height: 5}

    // Call the Area method on the Rectangle instance
    fmt.Println("Area:", rect.Area()) // Output: 50
}
```

In this example, the `Area` method is associated with the `Rectangle` type through the receiver `(r Rectangle)`. When you call `rect.Area()`, Go automatically passes `rect` as the receiver to the `Area` method.

Mutation:

Methods in Go can mutate the receiver if the receiver is a pointer type. This allows methods to modify the state of the receiver.

Copy

```go
type Counter struct {
    Count int
}

// Method with a pointer receiver
func (c *Counter) Increment() {
    c.Count++
}

func main() {
    // Create an instance of Counter
    counter := Counter{Count: 0}

    // Call the Increment method, which mutates the receiver
    counter.Increment()

    // Print the updated count
    fmt.Println("Count:", counter.Count) // Output: 1
}
```

In this example, the `Increment` method has a pointer receiver `(c *Counter)`, allowing it to modify the `Count` field of the `Counter` instance directly.

Note

- Methods in Go are functions associated with a particular type.
- A receiver is a parameter in a method declaration that associates the method with a specific type.
- Methods can be called on instances of the associated type.
- Methods with pointer receivers can mutate the state of the receiver, while methods with value receivers cannot.

Interfaces in Go

Interfaces in Go provide a way to specify behavior, enabling polymorphism and decoupling of components. Here's an overview of how to define a basic interface, implement it with a struct and methods, and an example of using interfaces:

Basic Interface:

An interface in Go is defined as a set of method signatures. Any type that implements all the methods of an interface implicitly implements that interface.

Copy

```go
package main
```

```
import "fmt"

// Define a basic interface named Shape
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

Struct and Methods:

You can define a struct type and implement the interface methods for that struct.

Copy

```
package main

import (
    "fmt"
    "math"
)

// Define a struct named Circle
type Circle struct {
    Radius float64
}

// Implement the Area method for Circle
func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

// Implement the Perimeter method for Circle
func (c Circle) Perimeter() float64 {
    return 2 * math.Pi * c.Radius
}
```

Interface Example:

You can now use the interface and the struct together, treating the struct instances as instances of the interface.

Copy

```
package main

import "fmt"

// Define a basic interface named Shape
type Shape interface {
    Area() float64
    Perimeter() float64
}

// Define a struct named Circle
type Circle struct {
    Radius float64
}

// Implement the Area method for Circle
func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}

// Implement the Perimeter method for Circle
func (c Circle) Perimeter() float64 {
    return 2 * math.Pi * c.Radius
}

func main() {
    // Create a Circle instance
    c := Circle{Radius: 5}

    // Use the interface to call the methods
    var s Shape = c
    fmt.Println("Area:", s.Area())
    fmt.Println("Perimeter:", s.Perimeter())
}
```

Note

In this example, we define a `Shape` interface with `Area()` and `Perimeter()` methods. We then define a `Circle` struct with `Radius` field and implement the `Area()` and `Perimeter()` methods for the `Circle` struct. Finally, we create a `Circle` instance and use it as an instance of the `Shape` interface. This demonstrates the power of interfaces in Go, allowing different types to share common behaviors through method signatures.