Sure! Here's a cheat sheet for all the topics listed in the image based on the structure provided:

Go Basics Cheat Sheet

Packages, Variables, and Functions

Packages

- A Go program is made up of packages.
- The `main` package is the entry point for execution.

```
go


  package main
  import "fmt"

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

Imports

- Import packages using the `import` keyword.
- Can group imports:

```
go


  import (
      "fmt"
      "math"
  )
```

Exported Names

- Identifiers starting with a capital letter are exported.

```
go


  fmt.Println(math.Pi) // Pi is exported from math package
```

Functions

Defining Functions

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

Functions Continued

- You can omit repeating types if they are the same:

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

Multiple Results

- Functions can return multiple values.

```go
func swap(a, b string) (string, string) {
    return b, a
}
```

Named Return Values

- Return values can be named.

```go
func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

Variables

Declaring Variables

```go
var x int
```

```
var y, z string
```

Variables with Initializers

- Variables can be initialized during declaration.

```
go


var a, b int = 1, 2
var c = "hello"
```

Short Variable Declarations

- Use `:=` to declare and initialize variables.

```
go


x := 42
```

Types

Basic Types

- **int**, **float64**, **string**, **bool** are common types.

```
go


var age int = 30
var price float64 = 19.99
var isAvailable bool = true
var name string = "GoLang"
```

Zero Values

- Variables are initialized to their zero value if no value is specified:
  - **0** for numeric types
  - **false** for boolean
  - **""** (empty string) for strings

Type Conversions

- Use explicit conversion:

```
go


var x int = 42
```

```
var y float64 = float64(x)
```

Type Inference

- Compiler infers type from the initializer.

```go
var x = 42  // x is an int
y := 3.14   // y is a float64
```

Constants

Defining Constants

- Use `const` for constants.

```go
const Pi = 3.14
```

Numeric Constants

- Constants can be typed or untyped.

```go
const (
    Big   = 1 << 100
    Small = Big >> 99
)
```

This cheat sheet covers all the listed topics succinctly. Let me know if you want detailed examples or further explanations!

Here's a continuation of the Go cheat sheet based on the new topics in the image:

Control Structures Cheat Sheet

For Loops

Basic `for` Loop

```go
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

For Continued

- Loop initialization, condition, and post statements are optional.

```go
i := 0
for ; i < 10; i++ {
    fmt.Println(i)
}
```

For is Go's `while`

- If the `for` statement has only a condition, it acts like a `while`.

```go
for i < 10 {
    fmt.Println(i)
    i++
}
```

Forever

- Infinite loop using `for` with no condition.

```go
for {
    fmt.Println("Infinite loop")
}
```

If Statements

Basic `if` Statement

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

If with a Short Statement

- A short statement can precede the condition.

```
go


  if v := math.Pow(2, 3); v < 10 {
      fmt.Println("v is less than 10")
  }
```

If and Else

```
go


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

Switch Statements

Basic `switch`

```
go


  switch day {
  case "Monday":
      fmt.Println("Start of the week")
  case "Friday":
      fmt.Println("Almost weekend")
  default:
      fmt.Println("Midweek days")
  }
```

Switch Evaluation Order

- Cases are evaluated top to bottom until a match is found.

```
go


  switch n := 5; {
  case n < 0:
      fmt.Println("Negative")
  case n == 0:
      fmt.Println("Zero")
  default:
      fmt.Println("Positive")
  }
```

Switch with No Condition

- Acts like `if-else`.

```go
switch {
case x < 0:
    fmt.Println("Negative")
case x == 0:
    fmt.Println("Zero")
default:
    fmt.Println("Positive")
}
```

Defer

Basic `defer`

- Used to ensure a function call runs after the surrounding function completes.

```go
defer fmt.Println("world")
fmt.Println("hello")
// Output: hello
//         world
```

Stacking Defers

- Deferred calls are executed in LIFO order (last-in, first-out).

```go
defer fmt.Println("1")
defer fmt.Println("2")
defer fmt.Println("3")
// Output: 3
//         2
//         1
```

This segment complements the earlier cheat sheet. Let me know if you'd like to expand further!

Here's a continuation of the Go cheat sheet based on the new topics:

Pointers, Structs, Arrays, and Slices Cheat Sheet

Pointers

Declaring and Using Pointers

- Pointers store memory addresses.

```go
var p *int
x := 5
p = &x // p points to x
fmt.Println(*p) // Dereference p to get x's value
```

Structs

Defining Structs

- A struct is a collection of fields.

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

Struct Fields

- Access fields with the dot operator.

```go
p := Person{Name: "Alice", Age: 30}
fmt.Println(p.Name)
```

Pointers to Structs

- Struct pointers can access fields directly.

```go
p := &Person{Name: "Bob", Age: 25}
p.Age = 26 // Automatically dereferenced
```

Struct Literals

- Initialize struct with values.

```go
p1 := Person{"Carol", 40}        // Field order
p2 := Person{Name: "Dave"}       // Partial initialization
p3 := Person{}                   // Zero-value initialization
```

Arrays

Declaring Arrays

- Fixed-size sequences.

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

Slices

Slices Are Like References to Arrays

- A slice points to an underlying array.

```go
arr := [5]int{1, 2, 3, 4, 5}
s := arr[1:4] // Slice from index 1 to 3
fmt.Println(s) // [2 3 4]
```

Slice Literals

```go
s := []int{1, 2, 3, 4, 5}
```

Slice Defaults

- Omit start or end to use defaults.

```go
s := []int{1, 2, 3, 4, 5}
fmt.Println(s[:3]) // [1 2 3]
fmt.Println(s[2:]) // [3 4 5]
```

Slice Length and Capacity

- `len` gives the number of elements; `cap` gives the capacity.

```go
s := []int{1, 2, 3}
fmt.Println(len(s)) // 3
fmt.Println(cap(s)) // 3
```

Nil Slices

- A nil slice has no elements or capacity.

```go
var s []int
fmt.Println(s == nil) // true
```

Creating a Slice with `make`

```go
s := make([]int, 5, 10) // len=5, cap=10
```

Slices of Slices

- You can create sub-slices.

```go
s := []int{1, 2, 3, 4, 5}
sub := s[1:4] // [2 3 4]
```

Appending to a Slice

- Add elements with `append`.

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

Range

Basic `range`

- Iterates over elements in a slice, array, or map.

```go
s := []int{1, 2, 3}
for i, v := range s {
    fmt.Println(i, v) // Index and value
}
```

Range Continued

- Use `_` to ignore values.

```go
for _, v := range s {
    fmt.Println(v) // Value only
}
```

This section completes the cheat sheet for pointers, structs, arrays, slices, and **`range`**. Let me know if you'd like further examples!

Here's the continuation of the Go cheat sheet for **Maps** and **Functions**:

Maps and Functions Cheat Sheet

Maps

Declaring and Using Maps

- Maps associate keys to values.

```go
m := make(map[string]int)
m["apple"] = 5
fmt.Println(m["apple"]) // Output: 5
```

Map Literals

- Initialize maps using literals.

```go
m := map[string]int{
    "apple":  5,
    "banana": 10,
}
fmt.Println(m) // Output: map[apple:5 banana:10]
```

Map Literals Continued

- Create an empty map literal.

```
go


  m := map[string]int{}
  m["grape"] = 15
  fmt.Println(m) // Output: map[grape:15]
```

Mutating Maps

- Add or update a key-value pair.

```
go


  m := make(map[string]int)
  m["orange"] = 7
```

- Delete a key.

```
go


  delete(m, "orange")
```

- Check if a key exists.

```
go


  v, ok := m["orange"]
  fmt.Println(v, ok) // Output: 0, false if "orange" does not exist
```

Exercise: Maps

- Iterate through a map.

```
go


  m := map[string]int{"apple": 5, "banana": 10}
  for key, value := range m {
      fmt.Println(key, value)
  }
```

Functions

Function Values

- Functions are first-class values.

```
go


  add := func(a, b int) int {
      return a + b
  }
  fmt.Println(add(2, 3)) // Output: 5
```

Function Closures

- Functions can close over variables in their scope.

```
go


  func adder() func(int) int {
      sum := 0
      return func(x int) int {
          sum += x
          return sum
      }
  }

  pos := adder()
  fmt.Println(pos(1)) // Output: 1
  fmt.Println(pos(2)) // Output: 3
```

This completes the cheat sheet for maps and functions. Let me know if you'd like additional examples or deeper dives into any topic!

Here's the continuation of the Go cheat sheet based on the latest topics:

Methods and Interfaces Cheat Sheet

Methods

Defining Methods

- A method is a function with a receiver.

```
go


  type Rectangle struct {
      Width, Height float64
  }

  func (r Rectangle) Area() float64 {
      return r.Width * r.Height
  }
```

Methods Are Functions

- Methods are just functions with a special receiver argument.

Methods Continued

- You can define multiple methods on a single type.

```
go


  func (r Rectangle) Perimeter() float64 {
      return 2 * (r.Width + r.Height)
  }
```

Pointer Receivers

- Use a pointer receiver to modify the struct.

```
go


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

Pointers and Functions

- A pointer can be passed to a function directly.

```
go


  func ScaleRect(r *Rectangle, factor float64) {
      r.Width *= factor
      r.Height *= factor
  }
```

Methods and Pointer Indirection

- Go handles pointer indirection automatically when calling methods.

```
go


  rect := Rectangle{3, 4}
  rect.Scale(2) // Go interprets as (&rect).Scale(2)
```

Choosing a Value or Pointer Receiver

- Use a value receiver if the method doesn't modify the struct.
- Use a pointer receiver if it modifies the struct or avoids copying large structs.

Interfaces

Defining Interfaces

- An interface specifies a set of methods.

```
go


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

Interfaces Are Implemented Implicitly

- A type implements an interface by defining all its methods.

```
go


  func PrintShape(s Shape) {
      fmt.Println("Area:", s.Area())
      fmt.Println("Perimeter:", s.Perimeter())
  }
```

Interface Values

- An interface value can hold any value that implements the interface.

```
go


  var s Shape
  s = Rectangle{3, 4}
  PrintShape(s)
```

Interface Values with Nil Underlying Values

- An interface value holding a `**nil**` underlying value is not `**nil**`.

```
go


  var s Shape
  fmt.Println(s == nil) // true
  s = nil
  fmt.Println(s == nil) // false if the interface itself is non-nil
```

Nil Interface Values

- An interface value is `nil` only if both the dynamic value and type are `nil`.

The Empty Interface

- The empty interface (`interface{}`) can hold any value.

```
go


  var i interface{}
  i = 42
  i = "hello"
```

Type Assertions

Basic Type Assertion

- Extract the underlying value of an interface.

```
go


  var i interface{} = "hello"
  s := i.(string)
  fmt.Println(s)
```

Safe Type Assertion

```
go


  s, ok := i.(string)
  if ok {
      fmt.Println(s)
  }
```

Type Switches

- A `switch` to handle multiple types.

```
go


  func TypeSwitch(i interface{}) {
      switch v := i.(type) {
      case int:
          fmt.Println("int:", v)
      case string:
          fmt.Println("string:", v)
      default:
```

```
            fmt.Println("unknown type")
        }
    }
```

Stringers

Implementing the `String` Method

- Types can implement the `fmt.Stringer` interface for custom string representations.

```
go


  type Person struct {
      Name string
      Age  int
  }

  func (p Person) String() string {
      return fmt.Sprintf("%s (%d years old)", p.Name, p.Age)
  }
```

This concludes the cheat sheet for methods, interfaces, type assertions, and stringers. Let me know if you'd like further examples or clarifications!

Here's the final part of the Go cheat sheet based on the latest topics:

Advanced Topics Cheat Sheet

Stringers

Custom String Representation

- Implement the `String()` method to define how your type is converted to a string.

```
go


  type Coordinate struct {
      X, Y int
  }

  func (c Coordinate) String() string {
      return fmt.Sprintf("(%d, %d)", c.X, c.Y)
  }
```

Errors

Creating Custom Errors

- Use `errors.New` or `fmt.Errorf` to create errors.

```go
import "errors"

var ErrInvalidInput = errors.New("invalid input")

func validate(input int) error {
    if input <= 0 {
        return ErrInvalidInput
    }
    return nil
}
```

Error with Additional Context

```go
import "fmt"

func process(input int) error {
    if input < 0 {
        return fmt.Errorf("process failed: %d is negative", input)
    }
    return nil
}
```

Readers

Implementing a Custom Reader

- Implement the `Read` method for the `io.Reader` interface.

```go
type MyReader struct{}

func (MyReader) Read(b []byte) (int, error) {
    for i := range b {
        b[i] = 'A'
    }
    return len(b), nil
}
```

Exercise: `rot13Reader`

- A reader that applies the ROT13 substitution cipher.

```go
type rot13Reader struct {
    r io.Reader
}
```

```
func (r13 *rot13Reader) Read(b []byte) (int, error) {
    n, err := r13.r.Read(b)
    for i := 0; i < n; i++ {
        if b[i] >= 'A' && b[i] <= 'Z' {
            b[i] = 'A' + (b[i]-'A'+13)%26
        } else if b[i] >= 'a' && b[i] <= 'z' {
            b[i] = 'a' + (b[i]-'a'+13)%26
        }
    }
    return n, err
}
```

Images

Creating a Custom Image

- Implement the `image.Image` interface.

```
go


import "image"

type MyImage struct{}

func (m MyImage) ColorModel() color.Model {
    return color.RGBAModel
}

func (m MyImage) Bounds() image.Rectangle {
    return image.Rect(0, 0, 100, 100)
}

func (m MyImage) At(x, y int) color.Color {
    return color.RGBA{uint8(x), uint8(y), 255, 255}
}
```

This concludes the cheat sheet. Let me know if you'd like more practical exercises, deeper explanations, or examples on any specific topic!

Here's the continuation of the Go cheat sheet covering **Generics**:

Generics Cheat Sheet

Generics

Type Parameters

- Go supports generics to allow defining functions or types that work with any data type.
- Syntax: Use square brackets `[]` to define type parameters.

Example: A Generic Function

```
go


func PrintSlice[T any](s []T) {
    for _, v := range s {
```

```
        fmt.Println(v)
    }
}
```

- `T` is a type parameter.
- `any` is a predefined type constraint that allows any type.

Usage:

```
go


PrintSlice([]int{1, 2, 3})       // Prints integers
PrintSlice([]string{"a", "b"})  // Prints strings
```

Generic Types

- You can define types that accept type parameters.

Example: A Generic Struct

```
go


type Pair[K, V any] struct {
    Key   K
    Value V
}
```

Usage:

```
go


p := Pair[string, int]{Key: "age", Value: 30}
fmt.Println(p) // Output: {age 30}
```

Example: A Generic Method

- Methods can also use type parameters.

```
go


func (p Pair[K, V]) Display() {
    fmt.Printf("Key: %v, Value: %v\n", p.Key, p.Value)
}
```

Usage:

```
go


p := Pair[string, int]{Key: "age", Value: 30}
p.Display() // Output: Key: age, Value: 30
```

Type Constraints

- Use `interface` to specify constraints on type parameters.

```
go


  type Number interface {
      int | float64
  }

  func Sum[T Number](a, b T) T {
      return a + b
  }
```

Usage:

```
go


  fmt.Println(Sum(3, 4))        // Works with integers
  fmt.Println(Sum(3.5, 4.5))    // Works with floats
```

This concludes the section on **Generics**. Let me know if you'd like more in-depth examples or exercises!

Here's the final part of the Go cheat sheet covering **Concurrency**:

Concurrency Cheat Sheet

Goroutines

Starting a Goroutine

- A goroutine is a lightweight thread managed by the Go runtime.

```
go


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

Wait for Goroutines

- Use channels or synchronization tools (e.g., `sync.WaitGroup`) to wait for goroutines to finish.

Channels

Declaring and Using Channels

- Channels are used to communicate between goroutines.

```go
ch := make(chan int)
go func() {
    ch <- 42 // Send a value into the channel
}()
fmt.Println(<-ch) // Receive a value from the channel
```

Buffered Channels

Defining Buffered Channels

- Buffered channels allow sending data without an immediate receiver.

```go
ch := make(chan int, 2)
ch <- 1
ch <- 2
fmt.Println(<-ch) // Output: 1
fmt.Println(<-ch) // Output: 2
```

Range and Close

Closing Channels

- Use `close` to close a channel.

```go
close(ch)
```

Iterating Over Channels

- Use `range` to receive values until the channel is closed.

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

Select

Basic `select`

- `select` waits on multiple channel operations.

```
go


  ch1 := make(chan int)
  ch2 := make(chan int)

  go func() { ch1 <- 42 }()
  go func() { ch2 <- 43 }()

  select {
  case v := <-ch1:
      fmt.Println("Received from ch1:", v)
  case v := <-ch2:
      fmt.Println("Received from ch2:", v)
  }
```

Default Selection

- Use `default` to avoid blocking.

```
go


  select {
  case v := <-ch:
      fmt.Println("Received:", v)
  default:
      fmt.Println("No data")
  }
```

---

sync.Mutex

Using Mutex for Synchronization

- `sync.Mutex` provides mutual exclusion for critical sections.

```
go


  var mu sync.Mutex
  counter := 0

  func increment() {
      mu.Lock()
      defer mu.Unlock()
      counter++
  }
```

Example: Protecting Shared Data

```go
var mu sync.Mutex
counter := 0

func safeIncrement() {
    mu.Lock()
    counter++
    mu.Unlock()
}

func main() {
    go safeIncrement()
    go safeIncrement()
}
```

This concludes the **Concurrency** section. Let me know if you'd like more practical examples or explanations!

- [ ]