Golang Tutorial

Introduction

Go was created by a small team within Google comprising of Robert Griesemar, Rob Pike and Ken Thompson.

1. Strong (type of a variable can't be changed) and statically typed (variables need to be defined at compile time)
2. Excellent community
3. Simplicity, Fast compile times, Built-in concurrency, compiles down to standalone binaries and Garbage collection

Resources

1. golang.org
2. golangbridge.org
3. play.golang.org

Hello, world

```
package main // packages

import (
    "fmt" // to format strings
)

func main() { // main function - entry point
    fmt.Println("Hello, world!") // print function from fmt library
}
```

Setting up a Developer Enviornment

Download Go Binary

Download GoLand from JetBrains

Variables

```
package main

import (
    "fmt"
)

func main() {
    var i int // keyword name type - used when we want declare variable but not use it yet
    i = 42
    var j int = 42 // keyword name type = value - useful when go doesn't have enough enough to automatically pickup type of a variab
    k := 2 // automatic type pickup
    fmt.Println(i)
}
```

Variable Declaration

Outside of the scope of a function, we have to use the complete declaration prototype (`keyword name type`)

```
package main

import (
    "fmt"
)

var i float32 = 50

func main() {
    fmt.Printf("%v, %T", i, i)
}
```

Block of variables declared together;

```
package main

import (
    "fmt"
)

var (
    name string = "foo"
    lname string = "bar"
    numb int = 3
    season int = 11
)

func main() {
    fmt.Printf("%v, %T", i, i)
}
```

Redeclaration and Shadowing

We can't redeclare variables, but can shadow them.

```
package main

import (
    "fmt"
)

var i int = 27

func main() {
    var i int = 42
    i = 13 // can't use i := 13 because we can't define same variable twice within the same scope
    // Shadowing: Inner most scope takes precedence
    fmt.Println(i)
}
```

Visiblity

Uppercase first letter (I) -> Global Scope or Export, i.e., Compiler automatically assigns it to the global scope Lowercase first letter (i) -> Package Scope Lowercase (i) -> Block Scope

```
package main

import (
    "fmt"
)

func main() {
    var i int = 42
    j := 13 // compile time error because j is defined but not used
    fmt.Println(i)
}
```

Naming conventions

PascalCase or camelCase

Capitalize acronyms (e.g., HTTP)

Tip: Length of a variable defines the life-span of a variable.

Type conversions

destinationType(variable)

Integer -> Float32

```
package main

import (
    "fmt"
)

func main() {
    var i int = 42
    var j float32
    j = float32(i) // Explicit conversion is mandatory
    fmt.Println(j)
}
```

Float32 -> Integer (Losing Information)

```
package main

import (
    "fmt"
)

func main() {
    var i float32 = 42.5
    var j int
    j = int(i)
    fmt.Println(j)
}
```

Integer -> String

```
package main

import (
    "fmt"
    "strconv" // String conversion package
)

func main() {
    var i int = 42
    var j string
    j = strconv.Itoa(i) // Method from package
    fmt.Println(j)
}
```

Primitives

Boolean type

```
package main

import (
    "fmt"
)

func main() {
    var n bool = true
    fmt.Printf("%v, %T\n", n, n)
}
```

Booleans can be used for state flagging.

```
package main

import (
    "fmt"
)

func main() {
    n := 1 == 1 // check if 1 equals 1
    fmt.Printf("%v, %T\n", n, n) // generates a boolean
}
```

In Go, everytime we init a variable, it has a zero value and the zero value for booleans is false.

```
package main

import (
    "fmt"
)

func main() {
    var n bool
    fmt.Printf("%v, %T\n", n, n) // generates a boolean
}
```

Numeric types: Integers, floating point and complex numbers

Zero value for all numeric type is zero, or the zero equivalent of that numeric type (e.g., 0.0)

```
package main

import (
    "fmt"
)

func main() {
    n := 42 // Integer (8bit, 16, 32, 64)
    var m uint16 = 42 // Unsigned Integer (8bit, 16, 32)

    // Arithmetic Operations
    fmt.Println(a + b) // Addition
    fmt.Println(a - b) // Subtraction
    fmt.Println(a * b) // Multiply
    fmt.Println(a / b) // Divide
    fmt.Println(a % b) // Modulo Function -> Remainder

    var o int8 = 3
    var p int = 10
    z := 2.1e14 // 10^14
    fmt.Println(p + int(o)) // Type Conversion

    // Bit Operations
    a := 10
    b := 3
    fmt.Println(a & b) // AND
    fmt.Println(a | b) // OR
    fmt.Println(a ^ b) // XOR
    fmt.Println(a &^ b) // ANDNOT

    // Bit Shifting
    fmt.Println(a << 3) // Bitshift left three places
    fmt.Println(a >> 3) // Bitshift right three places

    // Complex Numbers
    var com complex64 = 1 + 2i
    fmt.Printf("%v, %T\n", com, com)
    real(n) // Real part
    imag(n) // Imaginary part

    var com2 complex128 = complex(5, 12) // Another way of declaring complex number
}
```

Text Types

A string in Go is any UTF8 character. This makes them very powerful, but as a result, they can't encode every character.

```
package main

import (
    "fmt"
)

func main() {
    s := "this is a string
    fmt.Printf("%v, %T\n", string(s[2]), s[2]) // slicing the 3rd element in the string - strings in go are actually treated as byte

    s[2] = "u" // compile-time error

    s2 := "add"
    fmt.Printf("%v, %T\n", s+s2, s+s2) // String Concatenation

    by := []byte(s) // Corresponding UTF values of every element in the String as an array - used for functions as a lot of function
}
```

A rune represented UTF32 characters. Any UTF32 character can be upto 32 bits long, but it does NOT have to be that long. Normally, special methods are required to process runes.

```
package main

import (
    "fmt"
)

func main() {
    var r rune = 'a' // Single quotes
    fmt.Printf("%v, %T\n", r, r)

    // Runes are a true-type alias of int32
}
```

Constants

Naming Convention

```
package main

import (
    "fmt"
)

func main() {
    const camelCase int = 2 // Internal Constant
    const CamelCase int = 2 // External Constant
}
```

Typed Constants

```
package main

import (
    "fmt"
)

func main() {
    const camelCase int = 2 // Typed Constant
}
```

Constants can be shadowed, like variables.

Untyped Constants

```
package main

import (
    "fmt"
)

func main() {
    const a = 2 // Untyped Constant
    var b int16 = 3

    fmt.Printf("%v, %T\n", a+b, a+b) // Implicit conversions are possible with constant because the compiler converts every instance
}
```

Enumerated Constants

```
package main

import (
    "fmt"
)

const (
    a = iota
    b // infers pattern of assignment in block and makes this iota automatically
    c // infers pattern of assignment in block and makes this iota automatically
 ) // a counter used for enumerated constants

const (
    ab = iota
)
func main() {
    fmt.Printf("%v", a) // prints 0
    fmt.Printf("%v", b) // prints 1
    fmt.Printf("%v", c) // prints 2
    fmt.Printf("%v", ab) // prints  0 - iota limited to constant block
}
```

Enumeration Expressions

Operations that can be determined at compile time are allowed: Arithmetic, Bitwise and Bitshifting

Arrays and Slices

Arrays

Array members are contiguous in memory. It's very fast in terms of execution. All members must have same data type, and length of an array must be specified explicity at compile time.

Creation of Array

```
package main

import (
    "fmt"
)

func main() {
    grades := [...]int{97, 98, 99} // Create an array just large enough to contain the literals
    var students [3]string // Empty array of length 3
    students[0] = "foo" // Specify value at 0 index
    students[1] = "bar"
    students[2] = "foobar"
    fmt.Printf("Grades: %v", grades)
    fmt.Printf("Student1: %v", students[1]) // Dereferencing
}
```

Built-in Array functions

```
package main

import (
    "fmt"
)

func main() {
    var students [5]string
    students[0] = "foo"
    students[1] = "bar"
    students[2] = "foobar"
    fmt.Printf("No. of Students: %v", len(students)) // Length of Array

    var identityMatrix [3][3]int = [3][3]int{[3]int{1,0,0}, [3]int{0,1,0}, [3]int{0,0,1}} // Identity Matrix
    fmt.Println(identityMatrix)
}
```

Working with Arrays

In Go, Arrays are actually considered values itself. In most other languages, arrays actually point to the values, but in Go, copying a Array creates a literal copy.

```
package main

import (
    "fmt"
)

func main() {
    a := [...]int{1,2,3}
    b := a // Creates a literal copy of a - Slow for large arrays
    b[1] = 5
    fmt.Println(a)
    fmt.Println(b)
}
```

Slices

Everything we can do with an array, we can do with a Slice as well, with a few exceptions.

Creation of Slices

```
package main

import (
    "fmt"
)

func main() {
    a := []int{1,2,3} // Declaration
    fmt.Println(a)
}
```

Built-in Slice functions

```
package main

import (
    "fmt"
)

func main() {
    a := []int{1,2,3} // Declaration
    fmt.Println("%v", len(a)) // Length
    fmt.Println("%v", cap(a)) // Capacity
}
```

Working with Slices

Unlike arrays, slices are reference-types, i.e., they refer to the same underlying data.

```
package main

import (
    "fmt"
)

func main() {
    a := []int{1,2,3}
    b := a // Reference to the same data
    b[1] = 5
    fmt.Println(a)
    fmt.Println(b)

    bigSlice := []int{1,2,3,4,5,6,7,8,9,10}
    b := bigSlice[:] // Slice of all elements
    b := bigSlice[3:] // Slice from 4th element to end
    b := bigSlice[:6] // Slice first 6 elements
    b := bigSlice[3:6] // Slice the 4th, 5th and 6th elements

    //Slicing operations can work with Arrays too
}
```

```
package main

import (
    "fmt"
)

func main() {
    a := make([]int, 3, 100) // Slide Declaration -> int is the data type, 3 is the length of the slice, 100 is the capacity -> for
    a = append(a, 1) // Adds 1 to the end of the zero elements.
    a = append(a, []int{2, 3, 4, 5}...) // Spread Operator to add slide to the end of a slice
    fmt.Println(a)
}
```

```
package main

import (
    "fmt"
)

func main() {
    a := []int{1,2,3,4,5}
    b := append(a[:2], a[3:]...) // slicing from the middle of a slice
}
```

Maps and Structs

Maps

What are maps

Maps take a map key and map them over to some kind of a value.

```
package main

import (
    "fmt"
)

func main() {
    statePop := map[string]int{
        "California": 1111,
        "New York": 2222,
        "Ohio": 3333,
    }
    fmt.Println(statePop)
}
```

Creating Maps

```
package main

import (
    "fmt"
)

func main() {
    statePopMake := make(map[string]int) // Make Method
    statePopMake = map[string]int{
        "California": 1111,
        "New York": 2222,
        "Ohio": 3333,
    }

    statePop := map[string]int{
        "California": 1111,
        "New York": 2222,
        "Ohio": 3333,
    } // Literal Syntax
    fmt.Println(statePop)
}
```

Manipulation

```
package main

import (
    "fmt"
)

func main() {
    statePopMake := make(map[string]int) // Make Method
    statePopMake = map[string]int{
        "California": 1111,
        "New York": 2222,
        "Ohio": 3333,
    }
    statePopMake["Georgia"] = 4444 // adding a key-value pair
    fmt.Println(statePopMake["Ohio"]) // Interrogating one pair
    delete(statePopMake, "Georgia") // deleting a key-value pair -> Expecting the value of this deleted key would now return a value
    pop, ok := statePopulation["Georgia"]
    fmt.Println(pop, ok) // Returns 0, false

    // Return order of a map isn't guaranteed

    fmt.Println(len(statePopulations))

    // Maps are also reference types like slices, i.e., manipulating map in one place is going to affect every other place where the
}
```

Structs

What are structs

A struct type gathers related information together in a very flexible way.

Creating Structs

```
package main

import (
    "fmt"
)

type Doctor struct {
    number int
    actorName string
    companions []string
} // Struct

func main() {
    aDoctor := Doctor {
        number: 3,
        actorName: "aAA",
        companions: []string {
            "aasdfas",
            "asdfasdf",
        },
    } // Field Name Syntax

    fmt.Println(aDoctor.number) // Drilling down on a struct using dot notation
    fmt.Println(aDoctor.companions[1])

    bDoctor := Doctor {
        3,
        "bBB",
        []string {
            "aa",
            "bb",
        },
    } // Positional Syntax -> Not Recommended
}
```

Naming conventions for Structs

```
type Doctor struct { // global struct
    number int
    actorName string
    companions []string // local constituents
} // Struct
```

```
type doctor struct { // local struct
    number int
    actorName string
    companions []string // local constituents
} // Struct
```

```
type Doctor struct { // global struct
    Number int
    ActorName string
    Companions []string // global constituents
} // Struct
```

```
package main

import (
    "fmt"
)

func main() {
    aDoctor := struct{name string}{name: "foo"} // Anonymous struct
    anotherDoctor := aDoctor
    anotherDoctor.name = "bar"
}
```

Embedding

Instead of inheritance model, Go uses a composition model to establish a has-a relationship instead of an is-a relationship.

```
package main

import (
    "fmt"
)

type Animal struct {
    Name string
    Origin string
}

type Bird struct {
    Animal // Embedding
    SpeedKPH float32
    CanFly bool
}

func main() {
    b := Bird{}
    b.Name = "Emu"
    b.Origin = "Australia"
    b.SpeedKPH = 48
    b.CanFly = false
    fmt.Println(b.Name)

    c := Bird{
        Animal: Animal{Name: "Emu", Origin: "Australia"},
        SpeedKPH: 48,
        CanFly: false,
    } // Explicitly have to talk about internal structure in case of literal syntax
    fmt.Println(c.Name)
}
```

Tags

Provide a string of text for description, can also be used externally by custom code for validation etc.

```
package main

import (
    "fmt"
    "reflect" // for tags
)

type Animal struct {
    Name string `required max:"100"` // Tags -> It doesn't make any effect to the struct, it's like a comment, but we can fetch this
    Origin string
}

func main() {
    t := reflect.TypeOf(Animal{})
    field, _ = t.FieldByName("Name")
    fmt.Println("")
}
```

Control Flow: If and Switch statements

If statements

```
package main

import (
    "fmt"
)

func main() {
    if true { // Boolean Test
        fmt.Println("This will be printed")
    }

    if false {
        fmt.Println("This will not be printed")
    }

    statePopulations := map[string]int{
        "California": 111,
        "New York": 222
    }

    if pop, ok := statePopulations["Florida"]; ok { // Execute if Florida exists
        fmt.Println(pop)
    }
}
```

Operators

```
package main

import (
    "fmt"
)

func main() {
    number:=50
    guess:=50

    if guess < 1 || guess > 100 { //OR
        fmt.Println("The guess must be between 1 and 100!")
    }

    if guess >=1 && guess <= 100 { // AND
        if guess < number {
            fmt.Println("Too low")
        }
        if guess > number {
            fmt.Println("Too high")
        }
        if guess == number {
            fmt.Println("Got it")
        }
        fmt.Println(number<=guess, number>=guess, number!=guess)
    }
    fmt.Println(!true) // NOT -> prints false
}
```

If-else and if-else if statements

```
package main

import (
    "fmt"
)

func main() {
    number:=50
    guess:=50

    if guess < 1 {
        fmt.Println("The guess must be more than 1!")
    } else if guess > 100 {
        fmt.Println("The guess must be less than 100!")
    } else {
        if guess < number {
            fmt.Println("Too low")
        }
        if guess > number {
            fmt.Println("Too high")
        }
        if guess == number {
            fmt.Println("Got it")
        }
        fmt.Println(number<=guess, number>=guess, number!=guess)
    }
    fmt.Println(!true) // NOT -> prints false
}
```

Switch statements

Simple cases

```
package main

import (
    "fmt"
)

func main() {
    switch 2 { // 2 is a tag -> comparing everything against this
        case 1: // 1 == 2?
            fmt.Println("one")
        case 2: // 2 == 2?
            fmt.Println("two")
        default: // if none of the cases pass
            fmt.Println("not one or two")
    }
}
```

Cases with multiple tests

```
package main

import (
    "fmt"
)

func main() {
    switch 2 { // 2 is a tag -> comparing everything against this
        case 1, 5, 10:
            fmt.Println("one, five OR ten")
        case 2, 4, 6:
            fmt.Println("two, four OR six")
        default: // if none of the cases pass
            fmt.Println("some other number")

        //NOTE: test cases need to be unique
        //NOTE: break is implicit
    }
}
```

```
package main

import (
    "fmt"
)

func main() {
    switch i:= 2+3; i { // comparing against i
        case 1, 5, 10:
            fmt.Println("one, five OR ten")
        case 2, 4, 6:
            fmt.Println("two, four OR six")
        default: // if none of the cases pass
            fmt.Println("some other number")
    }
}
```

```
package main

import (
    "fmt"
)

func main() {
    i := 10
    switch { // tagless syntax
        case i <= 10:
            fmt.Println("less than equal to 10")
        case i <= 20:
            fmt.Println("less than equal to 20")
        default: // if none of the cases pass
            fmt.Println("greater than 20")
    } // Overlapping allowed
}
```

Falling through

Break is implictic, to add fall through;

```
package main

import (
    "fmt"
)

func main() {
    i := 10
    switch {
        case i <= 10:
            fmt.Println("less than equal to 10")
            fallthrough // next case is also executed instead of logic
        case i <= 20:
            fmt.Println("less than equal to 20")
        default: // if none of the cases pass
            fmt.Println("greater than 20")
    }
}
```

Type switches

```
package main

import (
    "fmt"
)

func main() {
    var i interface{} = 1
    switch i.type() {
        case int:
            fmt.Println("int")
        case float64:
            fmt.Println("float64")
        case string:
            fmt.Println("string")
        default:
            fmt.Println("another type")
    }
}
```

Control Flow: Looping

For Statements

Simple Loops

```
package main

import (
    "fmt"
)

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

```
package main

import (
    "fmt"
)

func main() {
    for i, j:=0, 0; i<5; i, j = i+1, j+1 {
        fmt.Println(i)
    }
}
```

```
package main

import (
    "fmt"
)

func main() {
    i := 0
    for { // Infinite for loop
        fmt.Println(i)
        i++
    }
}
```

Exiting early

```
package main

import (
    "fmt"
)

func main() {
    i := 0
    for { // Infinite for loop
        fmt.Println(i)
        i++
        if i == 5 {
            break // exit loop
        }
    }
}
```

```
package main

import (
    "fmt"
)

func main() {
    i := 0
    for { // Infinite for loop
        if i % 2 == 0 {
            continue // next iteration -> prints odd numbers
        }
        fmt.Println(i)
    }
}
```

Looping through collections

```
package main

import (
    "fmt"
)

func main() {
    s := []int{1,2,3}
    for k, v := range s { // look at collection -> take each value at a time and gives key and value -> Works for arrays, slices, ma
        fmt.Println(k, v)
    }
}
```

Control Flow: Defer, Panic and Recover

Defer

In go, the `defer` keyword actually executes any functions passed into it after the function finishes it final statement but before it actually returns. Defered functions are executed in LIFO (Last in, first out) order. **The arguments are taken at the time of deferral and not at the time of calling.**

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("start")
    defer fmt.Println("middle")
    fmt.Println("end") // end is printed before middle
    // The main looks for any defer functions here
}
```

Panic

```
package main

import (
    "fmt"
)

func main() {
    a, b := 1,0
    ans := a/b // Exception -> Called panic in GO
    panic("something bad happened") // Custom panic
    fmt.Println(ans)
}
```

Deffered calls are handled after panic.

Recover

Recover from panics, only useful in deferred functions because they are handled after panic. Current function will not attempt to continue, but higher functions will continue.

```
package main

import (
    "fmt"
    "log"
)

func main() {
    fmt.Println("start")
    defer func() {
        if err := recover(); err != nil {
            log.Println(err)
        }
    }()
    panic("something bad happened")
    fmt.Println("end")
}
```

Pointers

Creating pointers

```
package main

import (
    "fmt"
)

func main() {
    a := 42
    b := a // copy data from data and assign to b
    fmt.Println(a, b) // Prints 42, 42
    a = 27 // A changes but b remains the same
}
```

```
package main

import (
    "fmt"
)

func main() {
    var a int = 42
    var b *int = &a // Pointer (*)b points to address of (&)a
    fmt.Println(a, b) // Prints a and address of a
    a = 27 // both a and b changed to 27
}
```

Dereferencing pointers

```
package main

import (
    "fmt"
)

func main() {
    var a int = 42
    var b *int = &a
    fmt.Println(a, *b) // Dereferencing pointers -> prints value stores in that memory location
    *b = 14 // both a and b changed
}
```

The `new` function

```
package main

import (
    "fmt"
)

func main() {
    var ms *myStruct
    ms = new(myStruct)
    fmt.Println(ms)
}

type myStruct struct {
    foo int
}
```

Working with `nil`

Used to create pointers to objects

```
package main

import (
    "fmt"
)

func main() {
    var ms *myStruct // Zero value of pointer is nil
    ms = new(myStruct)
    (*ms).foo = 42
    fmt.Println((*ms).foo)
}

type myStruct struct {
    foo int
}
```

Types with internal pointers

```
package main

import (
    "fmt"
)

func main() {
    a := []int{1,2,3} // Internal projection of slice has a pointer to the underlying array
    b := a
    fmt.Println(a, b)
    a[1] = 42
    fmt.Println(a, b)

    // Maps also have internal pointers
}
```

Functions

Basic Syntax

`func` keyword followed by name Uppercase and lowercase define the visibility of the fuunction, like all other things in Go.

```
package main

import (
    "fmt"
)

func main() { // Function
    fmt.Println("Hello, world!")
}
```

Parameters

```
package main

import (
    "fmt"
)

func main() { // Function
    sayMessage("Hello, World!")
}

func sayMessage(msg string){ // Params
    fmt.Println(msg)
}
```

Return values

```
package main

import (
    "fmt"
)

func main() { // Function
    fmt.Println(sayMessage("Hello, World!"))
}

func sayMessage(msg string) string{
    return msg // Return
}
```

Anonymous functions

```
package main

import (
    "fmt"
)

func main() { // Function
    func(){
        fmt.Println("Hello Go!")
    }()
}
```

Functions as types

```
package main

import (
    "fmt"
)

func main() {
    f := func(){
        fmt.Println("Hello Go!")
    }()
    f()
}
```

Methods

```
package main

import (
    "fmt"
)

func main() {
    g := greeter{
        greeting: "Hello",
        name: "Go"
    }
    g.greet() // Calling methods
}

type greeter struct {
    greeting string
    name string
}

func (g greeter) greet() { // Method -> Function executed in known context
    fmt.Println(g.greeting, g.name)
}
```

Interfaces

Basics

```
package main

import (
    "fmt"
    "bytes"
)

func main() {
    var w Writer = ConsoleWriter{}
    w.Write([]byte("Hello Go!"))
}

type Writer interface { // Interface -> Describe behavior
    Write([]byte) (int, error) // Method
}

type ConsoleWriter struct{} // Implicit type -> to add method to a data type

func (cw ConsoleWriter) Write(data []byte) (int, error) { // Write Method oon Console Writer defined here
    n, err := fmt.Println(string(data))
    return n, err
}
```

Composing interfaces

```
package main

import (
    "fmt"
    "bytes"
)

func main() {
    var wc WriterCloser = NewBufferedWriterCloser()
    wc.Write([]byte("Hello"))
    wc.Close()
```

```
    wc.Close()
}

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

type Closer interface {
    Close() error
}

type WriterCloser interface { // Composing -> embedding interfaces
    Writer
    Closer
}

type BufferedWriterCloser struct {
    buffer *bytes.Buffer
}

func (bwc *BufferedWriterCloser) Write(data []byte) (int, error) {
    n, err := bwc.buffer.Write(data)
    if err != nil {
        return 0, err
    }

    v := make([]byte, 8)
    for bwc.buffer.Len() > 8 {
        _, err := bwc.buffer.Read(v)
        if err != nil {
            return 0, err
        }
        _, err = fmt.Println(string(v))
        if err != nil {
            return 0, nil
        }
        return n, nil
    }
    return n, err
}

func (bwc *BufferedWriterCloser) Close() error {
    for bwc.buffer.Len() > 0 {
        data := bwc.buffer.Next(8)
        _, err := fmt.Println(string(data))
        if err != nil {
            return err
        }
    }
    return nil
}

func NewBufferedWriterCloser() *BufferedWriterCloser {
    return &BufferedWriterCloser{
        buffer: bytes.NewBuffer([].byte()),
    }
}
```

Type conversion

```
package main

import (
    "fmt"
    "bytes"
```

```
    "io"
)

func main() {
    var wc WriterCloser = NewBufferedWriterCloser()
    wc.Write([]byte("Hello"))
    wc.Close()

    r, ok := wc.(*BufferedWriterCloser)) // type conversion
    if ok {
        fmt.Println(r)
    } else {
        fmt.println("Conversion failed")
    }
}

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

type Closer interface {
    Close() error
}

type WriterCloser interface {
    Writer
    Closer
}

type BufferedWriterCloser struct {
    buffer *bytes.Buffer
}

func (bwc *BufferedWriterCloser) Write(data []byte) (int, error) {
    n, err := bwc.buffer.Write(data)
    if err != nil {
        return 0, err
    }

    v := make([]byte, 8)
    for bwc.buffer.Len() > 8 {
        _, err := bwc.buffer.Read(v)
        if err != nil {
            return 0, err
        }
        _, err = fmt.Println(string(v))
        if err != nil {
            return 0, nil
        }
        return n, nil
    }
    return n, err
}

func (bwc *BufferedWriterCloser) Close() error {
    for bwc.buffer.Len() > 0 {
        data := bwc.buffer.Next(8)
        _, err := fmt.Println(string(data))
        if err != nil {
            return err
        }
    }
    return nil
}
```

```
func NewBufferedWriterCloser() *BufferedWriterCloser {
    return &BufferedWriterCloser{
        buffer: bytes.NewBuffer([].byte()),
    }
}
```

The empty interface

Interface that doesn't have any method on it. Anything can be cast onto empty interface.

Type switches

```
package main

import (
    "fmt"
)

func main() {
    var i interface{} = 1
    switch i.type() {
        case int:
            fmt.Println("int")
        case float64:
            fmt.Println("float64")
        case string:
            fmt.Println("string")
        default:
            fmt.Println("another type")
    }
}
```

Best Practices

Don't export interfaces for types that will be consumed. Do export interfaces for types that will be used by package. Design functions and methods to receive interfaces whenever possible.

Goroutines

Creating goroutines

```
package main

import (
    "fmt"
    "time"
)

func main() {
    go sayHello() // to spin off a green thread and run the function in that thread
    time.Sleep(100 * time.Millisecond) // to ensure the goroutine invokes a function
}

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

```
package main

import (
    "fmt"
    "time"
)

func main() {
    var msg = "Hello"
    go func(msg string){
        fmt.Println(msg)
    }(msg)
    msg = "Goodbye"
    time.sleep(100 * time.Millisecond)
}

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

Synchronizations

WaitGroups

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg = sync.WaitGroup{}

func main() {
    var msg = "Hello"
    wg.Add(1) // Add one thread
    go func(msg string){
        fmt.Println(msg)
        wg.Done() // tell the group that the thread is done
    }(msg)
    msg = "Goodbye"
    wg.Wait()
}

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

Mutexes

An lock that the application honors.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg = sync.WaitGroup{}
counter = 0
var m = sync.RWMutex{} // infinite readers, only one writer

func main() {
    runtime.GOMAXPROCS(100) // Threads -> Parallelism
    for i:=0; i<10;i++{
        wg.Add(2)
        m.RLock()
        go sayHello()
        m.Lock()
        go increment()
    }
    wg.Wait()
}

func sayHello(){
    fmt.Println("Hello #%v", counter)
    m.RUnlock()
    wg.Done()
}

func increment()
    counter++
    m.Unlock()
    wg.Done()
}
```

Best practices

Don't create goroutines in libraries. When creating a goroutine, know how it will end to avoid subtle memory leaks. Check for race conditions at race time.

Channels

Channel basics

```
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg = sync.WaitGroup{}

func main() {
    ch := make(chan int)
    wg.Add(2)

    go func() { // Recieving goroutine
        i := <- ch
        fmt.Println(i)
        wg.Done()
    }()

    go func() { // Sending goroutine
        ch <- 42
        wg.Done()
    }()

    wg.Wait()
}
```

Restricting data flow

```
package main

import (
    "fmt"
    "sync"
)

var wg = sync.WaitGroup{}

func main() {
    ch := make(chan int)
    wg.Add(2)

    go func(ch <- chan int) { // Recieving only channel
        i := <- ch
        fmt.Println(i)
        wg.Done()
    }(ch)

    go func(go chan<- int) { // Sending only goroutine
        ch <- 42
        wg.Done()
    }(ch)

    wg.Wait()
}
```

Buffered channels

```
package main

import (
    "fmt"
    "sync"
)

var wg = sync.WaitGroup{}

func main() {
    ch := make(chan int, 50) // Internal data store
    wg.Add(2)

    go func(ch <- chan int) { // Recieving only channel
        i := <- ch
        fmt.Println(i)
        wg.Done()
    }(ch)

    go func(go chan<- int) { // Sending only goroutine
        ch <- 42
        ch <- 27 // extra var goes to buffer
        wg.Done()
    }(ch)

    wg.Wait()
}
```

Closing channels

```
package main

import (
    "fmt"
    "sync"
)

var wg = sync.WaitGroup{}

func main() {
    ch := make(chan int, 50) // Internal data store
    wg.Add(2)

    go func(ch <- chan int) { // Recieving only channel
        for i := range <- ch {
            fmt.Println(i)
        }
        wg.Done()
    }(ch)

    go func(go chan<- int) { // Sending only goroutine
        ch <- 42
        ch <- 27
        close(ch) // For loop sees channel is closed, for loop detects it and then wg.Done() is triggered
        wg.Done()
    }(ch)

    wg.Wait()
}
```