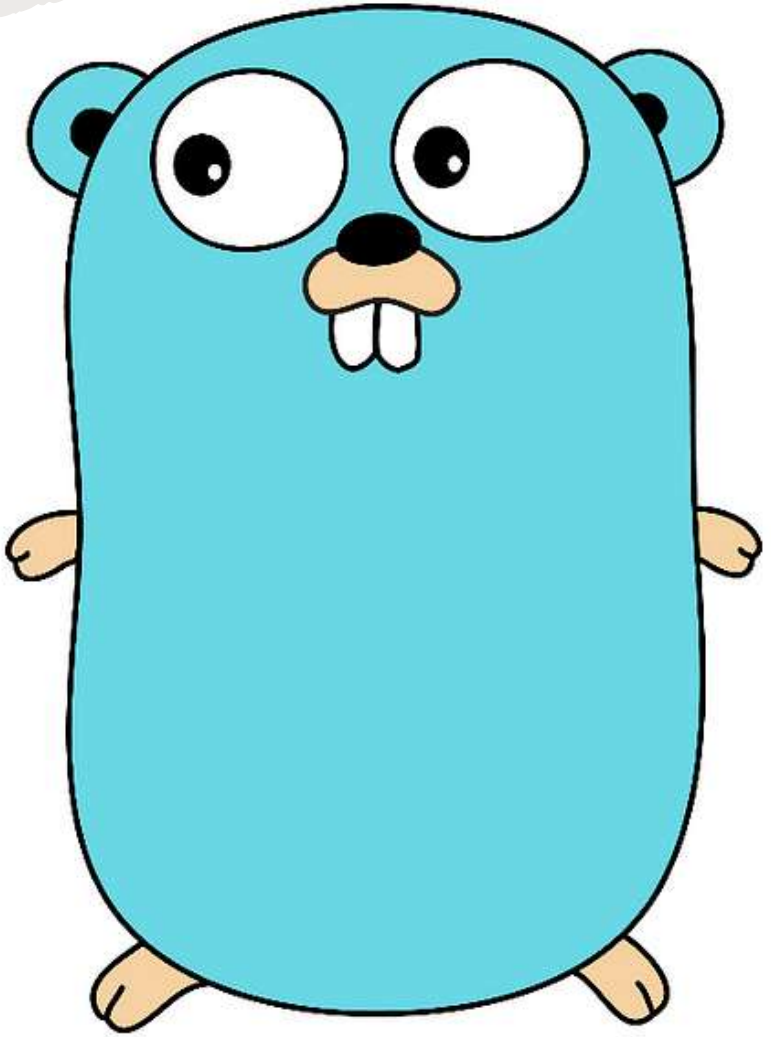
Go Programming Language

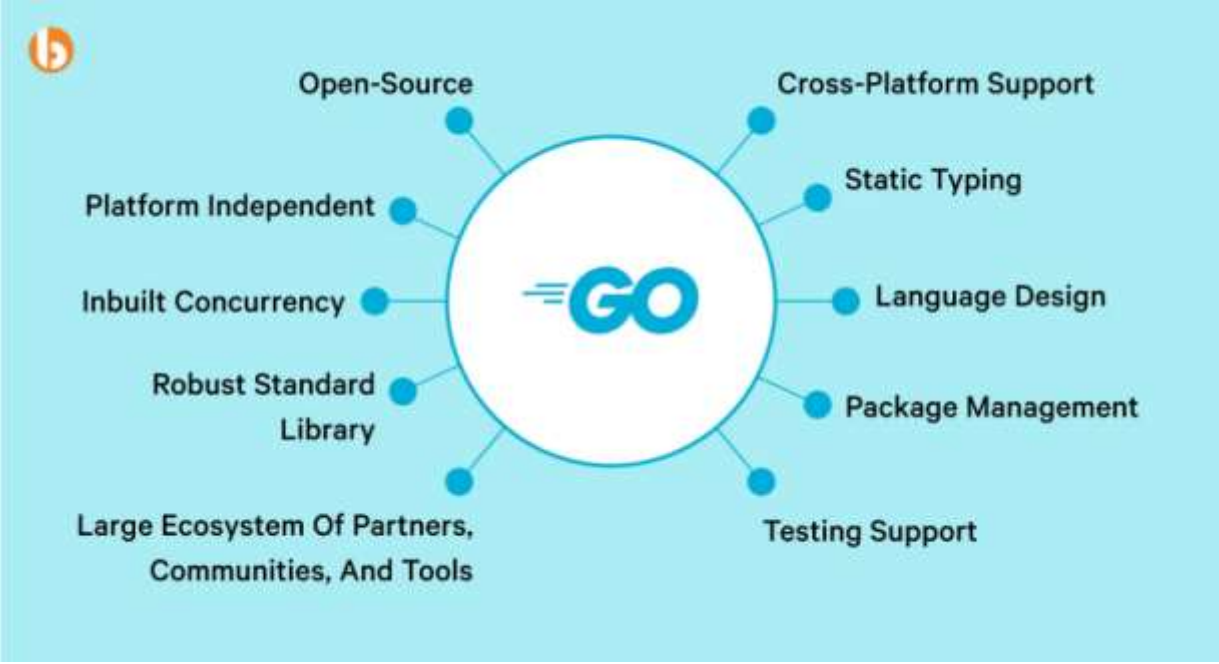


Nội dung trình bày

1. Giới thiệu Golang (slice, syntax cơ bản, pointer, routine)
2. Cơ chế đa luồng Go routine (Phân tích cách hoạt động Go Runtime, Cách Go quản lý các routine, phân quyền ưu tiên giữa các routine)
3. So sánh Golang với Java (Bộ nhớ (thu dọn auto, CPU))
4. OOP in Golang compare with Java , Ví dụ cụ thể
5. Use case (service, backend, api, …)
6. Thực tế Open source, công ty đang sử dụng, 5G, Grap, Map, Docker
7. Software Architecture In Go (Microservice(http, gRPC, Nats.io, Pub/sub…)) , Database (NoSQL, SQL [Redis, Mongdb , mySql , replica database, sync data giữa các sevice])

Go Introduction

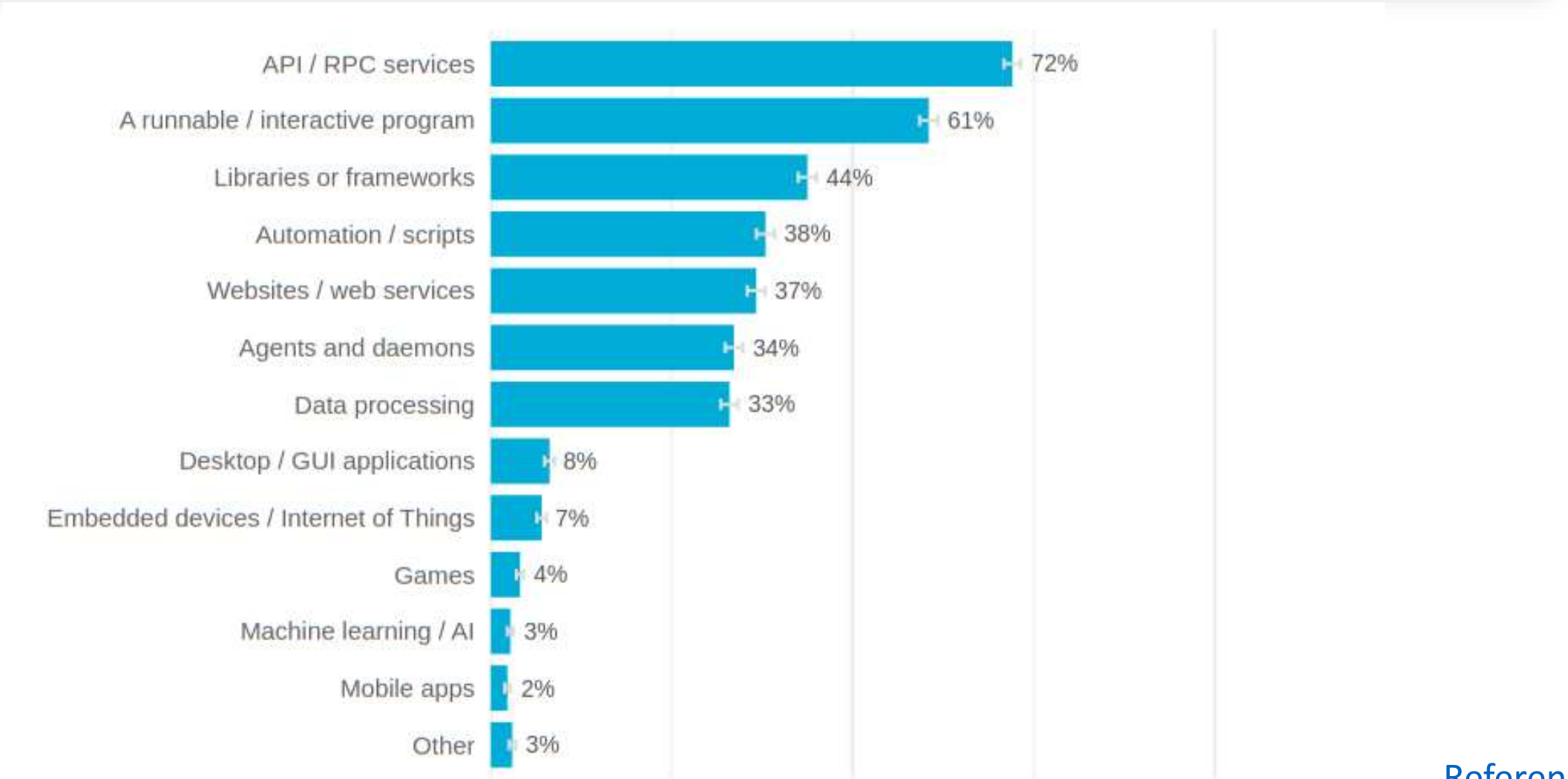
- **What is the Go**
  - Go is a programming language that is precisely statically typed, compiled high-level in nature, and designed at Google by *'Robert Griesemer,' 'Rob Pike,' and 'Ken Thompson' in 2007.*
- **Features of Go**



Who use Go?

Reference

How different experience levels currently use Go



Reference

Go Basic

Go see a bit of code

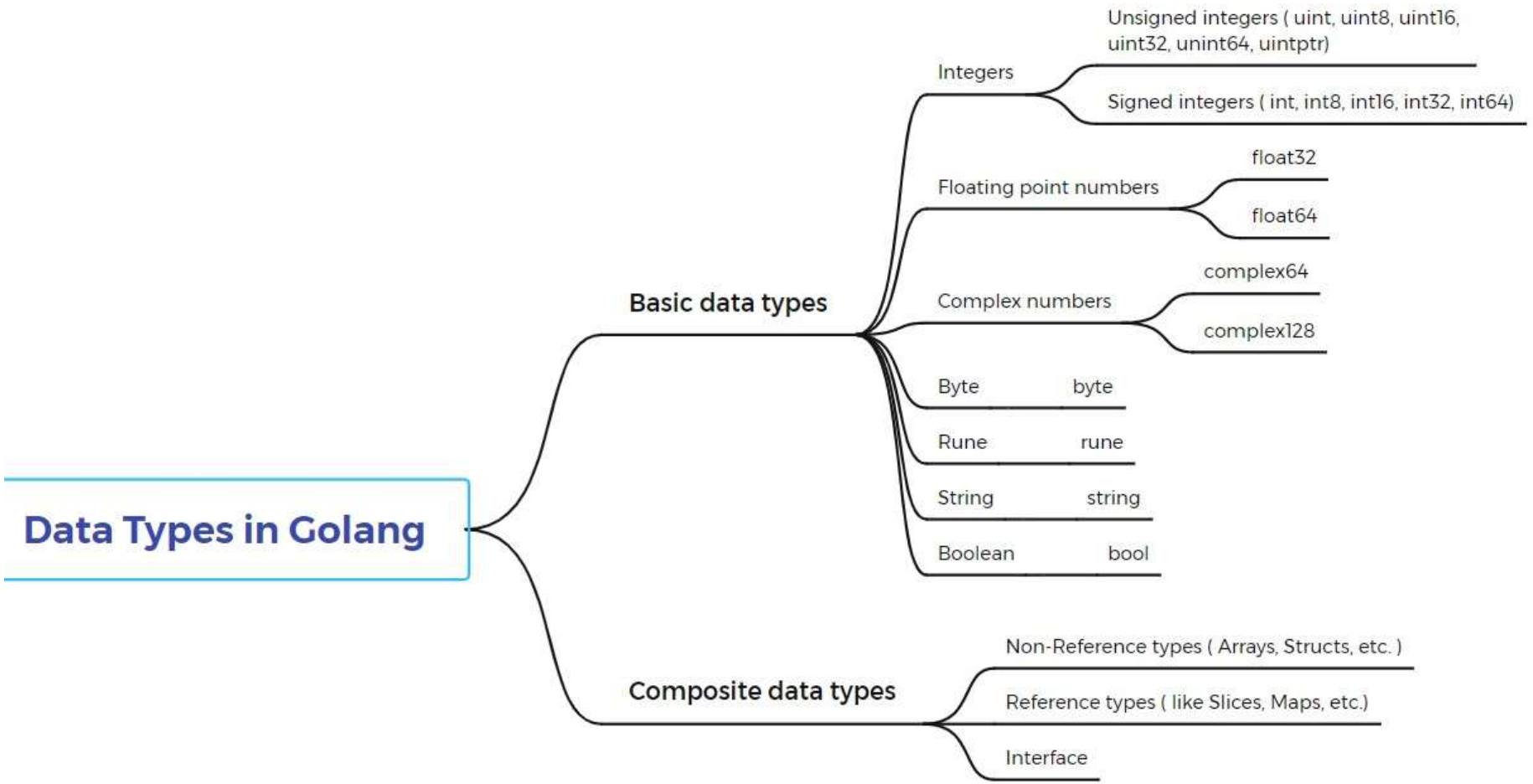
```go
package main

import "fmt"

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

// $ run go main.go
// Return Hello, World
```

Using Package

- Each Go program are compound per packages
- Programs starts from main package
- This example are using the **fmt** and **math** packages

```go
package main

import (
  "fmt"
  "math/rand"
)

func main() {
  fmt.Println("My favorite number is ", rand.Intn(10))
}
```

Data types


Data Types in Golang
Basic data types
Integers
Unsigned integers (uint, uint8, uint16, uint32, unint64, uintptr)
Signed integers (int, int8, int16, int32, int64)
Floating point numbers
float32
float64
Complex numbers
complex64
complex128
Byte byte
Rune rune
String string
Boolean bool
Composite data types
Non-Reference types (Arrays, Structs, etc.)
Reference types (like Slices, Maps, etc.)
Interface

Variable

- Go is statically typed
- Implicitly defined variable - type is inferred by Go compiler
- Explicitly defined variable type - is specified explicitly
- Mutiple variables could be defined together and initialized

```go
package main

func main() {
  var message1 = "Hello Go" // message  would  be of  type string

  var message2 string = "Hello Go"

  var x, y int

  var (
    name = "Golang"
    age = 21
  )
}
```

Type declaration

- Go's type declaration is different from C/C++(or Java/ C#) and is very similar to Pascal – variable / declared name appears before the type

```
C/C++                                                    Pascal:

int ctr = 10                                             var ctr : int = 10
string message = "Hello"                                 var message : string = "Hello"

Go:
var message1 = "Hello from Go"                  // Implicit type declaration- type is inferred
var message2 string = "Hello from Go"           // Explicit type declaration
```

Short variables declarations

- Within a function variables could be defined using a shorthand syntax without using var key word

```
// Option 1 - Explicit type declaration
message string := "Hello World"       // var not is used, := is used instead of =


// Option 2 - Type inferred
message := "Hello World"       // var not is used, := is used, type inferred as string
```

Constants

- Constants are declared like variables but keyword const
- Can not use the syntax :=

```
const PI float32 = 3.14
fmt.Println(PI)

PI := 3.15 // does not compile and shows a compile time error
```

Collection Types

Array, Slice and Map

Array

- Array is a numbered sequence of elements of fixed size
- Arrays could be declared and initialized in a same line
- Go compiler can calculate the length of array if not specified explicitly

```go
var cities[3] string
cities[0] = "Ha Noi"

// Array size is fixed - it could not change after declaring it
fruits :=[3]string{"Apple", "Banana", "Orange"}

// calculate the length of the array if not specified explicitly
fruits :=[...]string{"Apple", "Banana", "Orange"}

// Determine the length of the array
// built-int func len() return length of an array
length := len(fruits)
```

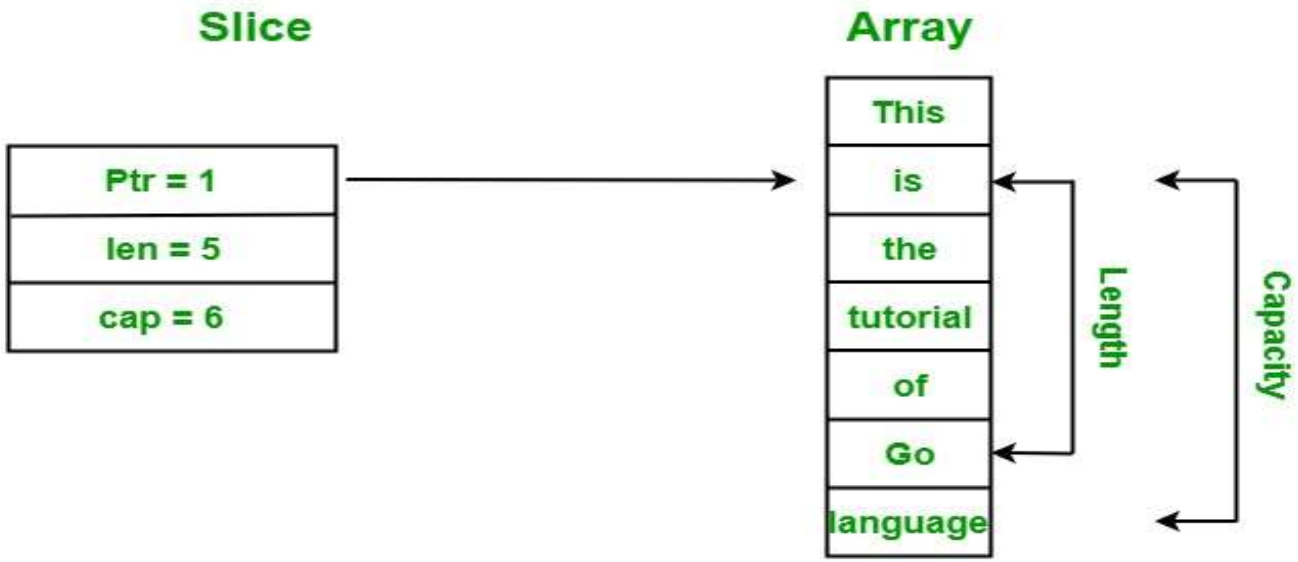
Array

- Iterating through the array

```
fruits :=[...]string{"Apple", "Banana", "Orange"}
// Using for loop
for int i = 0; i < len(fruits); i++{
  fmt.Println(fruits[i])
}
// Using for loop and range keyword
for index, value := range fruits {
   fmt.Println(index, value)
}
```

Slice

- Slice is a dynamically-sized, flexible view into the elements of an array
- A slice has both a *length* and a *capacity*.
- The length of a slice is the number of elements it contains.
- The capacity of a slice is the number of elements in the underlying array, counting from the first element in the slice.



Slice

- Slice is a dynamically-sized, flexible view into the elements of an array.
- A slice is formed by specifying two indices, a low and high bound, separated by a colon:

```
// slices are declared with syntax similar to array. Only length is not specified.
cities := []string {"Ha Noi", "Ninh Binh", "Nam Dinh"}

// declare a slice from an arry
arr := [6]int{1,2,3,4,5,6}
slices := arr[:]            //[1,2,3,4,5,6]
slices := arr[:5]           //[1,2,3,4,5]
slices := arr[1:5]          //[2,3,4,5]

// add elements to a slice
slices := append(slices,7)  // [1.2,3,4,5,6,7]
```

Slice

- Slice could also be defined by built-in make function
- The length and capacity of a slice can be obtained using the len(), cap()

```
cities := make([]string, 3)
cities[0] = "Ha Noi"                        // Allows to set values  like Arrays
cities[1] = "Nam Dinh"
cities[2] = "Ha Nam"
cities = append(cities, "Ninh Binh")           // Use append to add more values
// return ["Ha Noi", "Nam Dinh", "Ha Nam", "Ninh Binh"]

lenght := len(cities) // length = 4
capacity := cap(cities) // capacity = 6
```

Slice

- Slice can also be copied/cloned using copy function

```
cities := make([]string, 3)
cities[0] = "Ha Noi"                        // Allows to set values  like Arrays
cities[1] = "Nam Dinh"
cities[2] = "Ha Nam"
cities = append(cities, "Ninh Binh")           // Use append to add more values
// return ["Ha Noi", "Nam Dinh", "Ha Nam", "Ninh Binh"]

copySlice := make([]string, len(cities))
copy(copySlice, cities)
// return copySlice["Ha Noi", "Nam Dinh", "Ha Nam", "Ninh Binh"]
```

Map

- Map is one of the built in data structure that Go provides. Similar to hashes or dicts in other languages

```
employees := map[int]string{
 1 : "John"
 2 : "Peter"
 3 : "Jame"
}                                   // [1 : "John", 2 : "Peter", 3 : "Jame" ]

// Get a value from a key with name[key]
fmt.Println(employees[1])           // John

// Set a value for a key with name[key]
employees[3] = "Maria"
fmt.Println(employees[3])           // Maria
```

Map

- Map also could be declared by using built in make function

```
// make(map[key-type]val-type)
employees := make(map[int]string)
employees[1] = "John"
employees[2] = "Peter"
employees[3] = "Jame"
fmt.Println(employees)            // [1 : "John", 2 : "Peter", 3 : "Jame" ]

// Get a value from a key with name[key]
fmt.Println(employees[1])           // John

// Set a value for a key with name[key]
employees[3] = "Maria"
fmt.Println(employees[3])           // Maria
```

Function

Function

- Functions are declared with func keyword
- Functions can take one or more arguments and return values

```
func add(a int, b int) int{
    return a +  b
}
func dislplay(message string){
   fmt.Println(message)
}
func main(){
     fmt.Println(add(1,2))  // return 3
     dislplay("Hello")      // return Hello
}
```

Anonymous Functions

- Go supports Anonymous Functions

```go
// add is declared as normal function
func add(a int, b int) int{
    return a +  b
}
// display is declared as anonymous fucntion
var  dislplay = func (message string){
   fmt.Println(message)
}
func main(){
     fmt.Println(add(1,2))  // return 3
     dislplay("Hello")      // return Hello
}
```

Anonymous Functions

- Anonymous Functions could be passed as argument to other functions, and could be returned from other functions. Functions behave like values – could be called function values

```
func main(){
      func(msg string) {
        fmt.Println("Hello" + msg)
      }("John")
      // Display hello John

       func(num1 int, num2 int) {
        fmt.Println(num1 + num2)
      }(10, 20)
      // Display 30

}
```

Anonymous Functions – User-defined Function types

```go
type HigherFunc func(x int, y int) int        // User defined function type
func doWork(anonymous (HigherFunc, num1 int, num2 int) {
      anonymous(num1, num2)
}
  func main(){
     add := func(x int, y int) int{
           return x + y
     }
     sub := func(x int, y int) int{
           return x - y
     }
     result := doWork(add, 10, 20)
     fmt.Println(result) // return 30

       result = doWork(add, 20, 10)
     fmt.Println(result) // return 10

}
```

Variadic Functions

- Variadic functions can be called with any number of trailing arguments.

```
func displayMessage(message string, params ...string){
     fmt.Println(message, params)
}
func main(){
    displayMessage("Hello","Jame") // Display Hello [Jame]
    displayMessage("Hello","Jame","John") // Display Hello [Jame, John ]
}
```

Control Structure

If, For, Switch

Control Structures

- If
- For
- Switch

Go does not support while or do while keywords, though loops similar to while could be written by for

If, else if, else

- The if statement looks as it does in C or Java, except that the () are gone and the {} are required.

```
func main(){
    salary := 1000
  if salary < 50 {
    fmt.Println("you are underpaid")
  } else if salary >= 50 {
    fmt.Println("you are sufficiently paid")
  }
  // Display you are sufficiently paid
}
```

For

- Go has only one looping construct, the for loop.
  - Go does not have while, do while or foreach/for in loops
- The basic for loop look as it does in C and Java, except the () are gone(they are not even optional) and the{} are required

```
func main(){
    for ctr := 0; ctr < 10; ctr++ {
     fmt.Println(ctr)
    }
    // Go does not support while or do while. Same could be achieved using for
    cout := 0
    for cout < 10 {
    fmt.Println(cout)
    cout++
   }
}
```

For

- As in C or Java, you can leave the pre and post statements empty
- Endless or forever loop

```
func main(){
    ctr := 0
    for ; ctr < 10; {
     ctr += 1
     fmt.Println(ctr)
    }

    for {
       fmt.Println("loop")
    }
    // this loop would never end
}
```

Switch

- A switch statement is a shorter way to write a sequence of if – else statements. It runs the first case whose value is equal to the condition expression.
- Go's switch is like the one in C, C++, Java, JavaScript, and PHP, except that Go only runs the selected case, not all the cases that follow
- The break statement is provide in Go automatically
- Another important difference is that Go's switch cases need not be constants, and the values involved need not be integers.

Switch

```go
func main(){
  rating := 2
  switch rating {
  case 4:
    fmt.Println("You are rated Execellent")
  case 3:
    fmt.Println("You are rated Good")
  case 2:
    fmt.Println("You are rated Consistent")
  case 1:
    fmt.Println("You need to improve")
    // Display You are rated Consistent
  }
}
```

Struct, Method and Interface

Type – struct

- Structs are typed collections of named fields. Useful for grouping data together to form records
- The keyword struct to indicate that we are defining a struct type
- Go does not have Class, it support Struct and Interface

```
type struct Employee {
 name string          // field name of type string
 age int              // field age of type int
 salary float32       // field salary of type float32
}
```

Struct – initialization

```
type Employee struct {
 name string          // field name of type string
 age int              // field age of type int
 salary float32       // field salary of type float32
}
func main(){
  // Option 1 - Using new function
  emp := new(Employee)
  emp.name = "James"

   // Option 2 - More like Javascript
  emp := Employee{}
  emp.name = "John"

   // Option 3 - Parameters should be in the same order fields are declared
  emp := Employee{"John",21, 20000000}

  // Option 4 - Parameters should specify field names and assign value to them
  emp := Employee{ name : "John",age : 21, salary : 20000000}
}
```

Struct

Structs can have arrays and other child structs as fields

```
type struct Employee {
 name string          // field name of type string
 age int              // field age of type int
 salary float32       // field salary of type float32
 skills [4]string     // Array field skills of type []string
 Address address      // Nested Child struct as property
}
type Address struct {
  StreetAddress string
  City string
  Country string
}
```

Method

- Go supports methods defined on struct types
- Methods of the struct are actually defined outside of the struct declaration.

```
type  Employee struct {
 name string         // field name of type string
 age int             // field age of type int
 salary float32      // field salary of type float32
}
func(emp Employee) Display() {
  fmt.Println(emp.name, emp.age, emp.salary)
}
func main(){
 employee :=  Employee{"John",21,20000000}
 employee.Display()  // Display {name : John, age: 21, salary: 20000000}
}
```

Method – value receiver type

- Methods can be defined for value receiver types

```
type Employee struct {
 name string          // field name of type string
 age int              // field age of type int
 salary float32       // field salary of type float32
}
func(emp* Employee) increasedAgeByOne() {
    emp.age++
}
func(emp* Employee) increasedAge(increaseBy int) {
    emp.age += increaseBy
}
func(emp* Employee) display() {
    fmt.Println(emp.name, emp.age)
}
func main(){
 employee :=  Employee{"John",21,20000000}
 employee.increasedAgeByOne()
 display()                          // John, 21
 employee.increasedAge(5)
 display()                          // John, 21
}
```

Method – pointer receiver type

```
type Employee struct {
 name string          // field name of type string
 age int              // field age of type int
 salary float32       // field salary of type float32
}
func(emp* Employee) increasedAgeByOne() {
    emp.age++
}
func(emp* Employee) increasedAge(increaseBy int) {
    emp.age += increaseBy
}
func(emp* Employee) display() {
    fmt.Println(emp.name, emp.age)
}
func main(){
 employee :=  Employee{"John",21,20000000}
 employee.increasedAgeByOne()
 display()                          // John, 22
 employee.increasedAge(5)
 display()                          // John, 27
}
```

Interface

- An *interface type* is defined as a set of method signatures.
- A value of interface type can hold any value that implements those methods.

Interface

```go
type  Employee struct {
 name string; age int; salary float32
}
 type  Contractor struct {
 name string; weeklyHour int
}
type Human interface {
  Display()
}
func(emp Employee) Display() {
  fmt.Println(emp.name, emp.age, emp.salary)
}
func(contract Contractor) Display() {
  fmt.Println(contract.name, contract.weeklyHour)
}
func main(){
 employee :=  Employee{"John",21,20000000}
 employee.Display()  // Display {name : John, age: 21, salary: 20000000}
 ctr :=  Contractor{"John",12}
 ctr.Display()       // Display {name : John,weeklyHour : 12 }
}
```

Pointers

- Go has pointers. A pointer holds the memory address of a value.
- The type *T is a pointer to a T value. Its zero value is nil.
- Unlike C, Go has no pointer arithmetic.

```go
func main() {
  i, j := 42, 2701

  p := &i          // point to i
  fmt.Println(*p) // read i through the pointer
  *p = 21          // set i through the pointer
  fmt.Println(i)  // see the new value of i is 21

  p = &j          // point to j
  *p = *p + 6    // add j through the pointer
  fmt.Println(j) // see the new value of j is 2707
}
```

IOTA

- IOTA is a counter which starts with zero
- Increases by 1 after each line
- Is only used with constant

IOTA

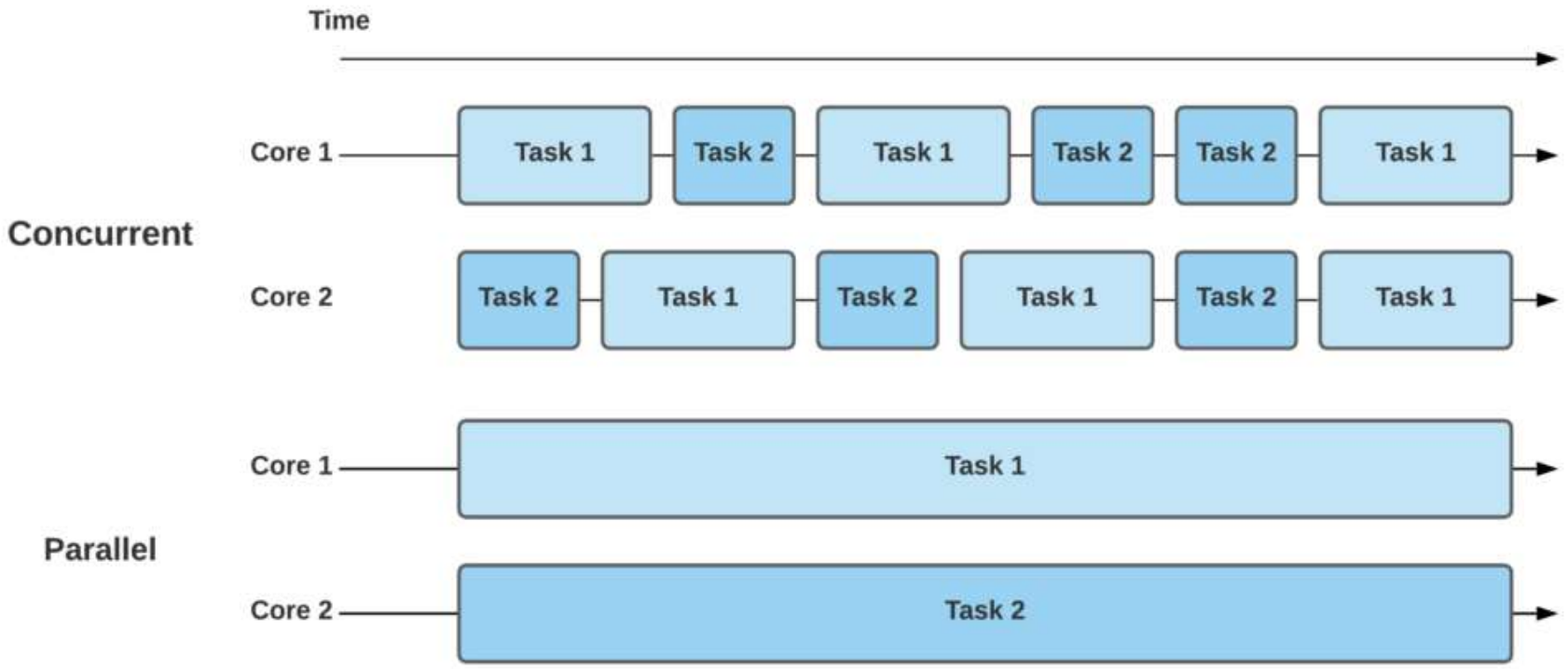
```go
// Auto increment constant without IOTA
const (
    a = 0
    b = 1
    c = 2
)
// Auto increment constant with IOTA
const (
    a = iota
    b
    c
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
}
// Output 0, 1, 2
```

Concurrency

Goroutines, Channel, Select Statement

What is the Concurrency

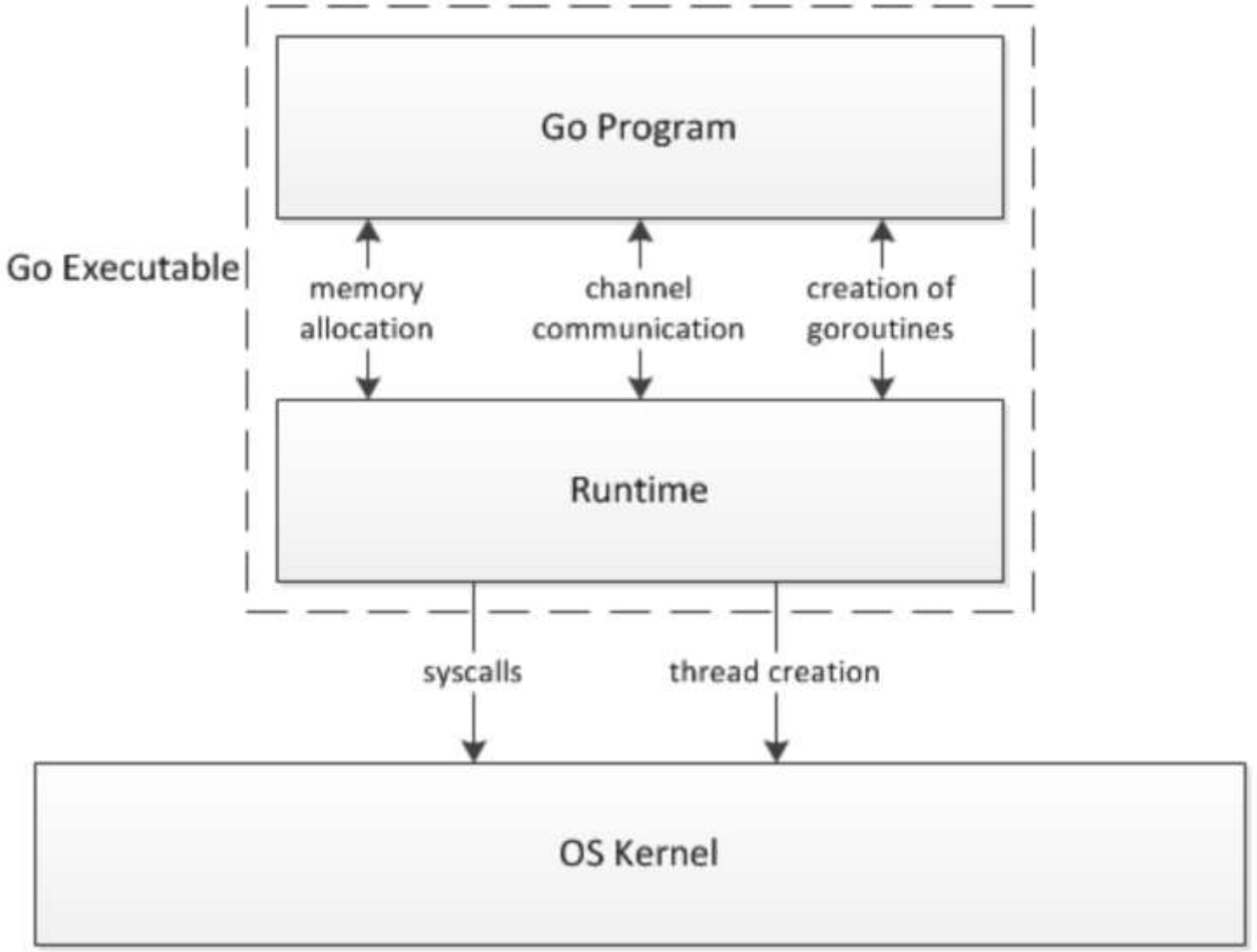
Concurrency is a process that deals with a lot of activities operating simultaneously at once" - Rob Pike



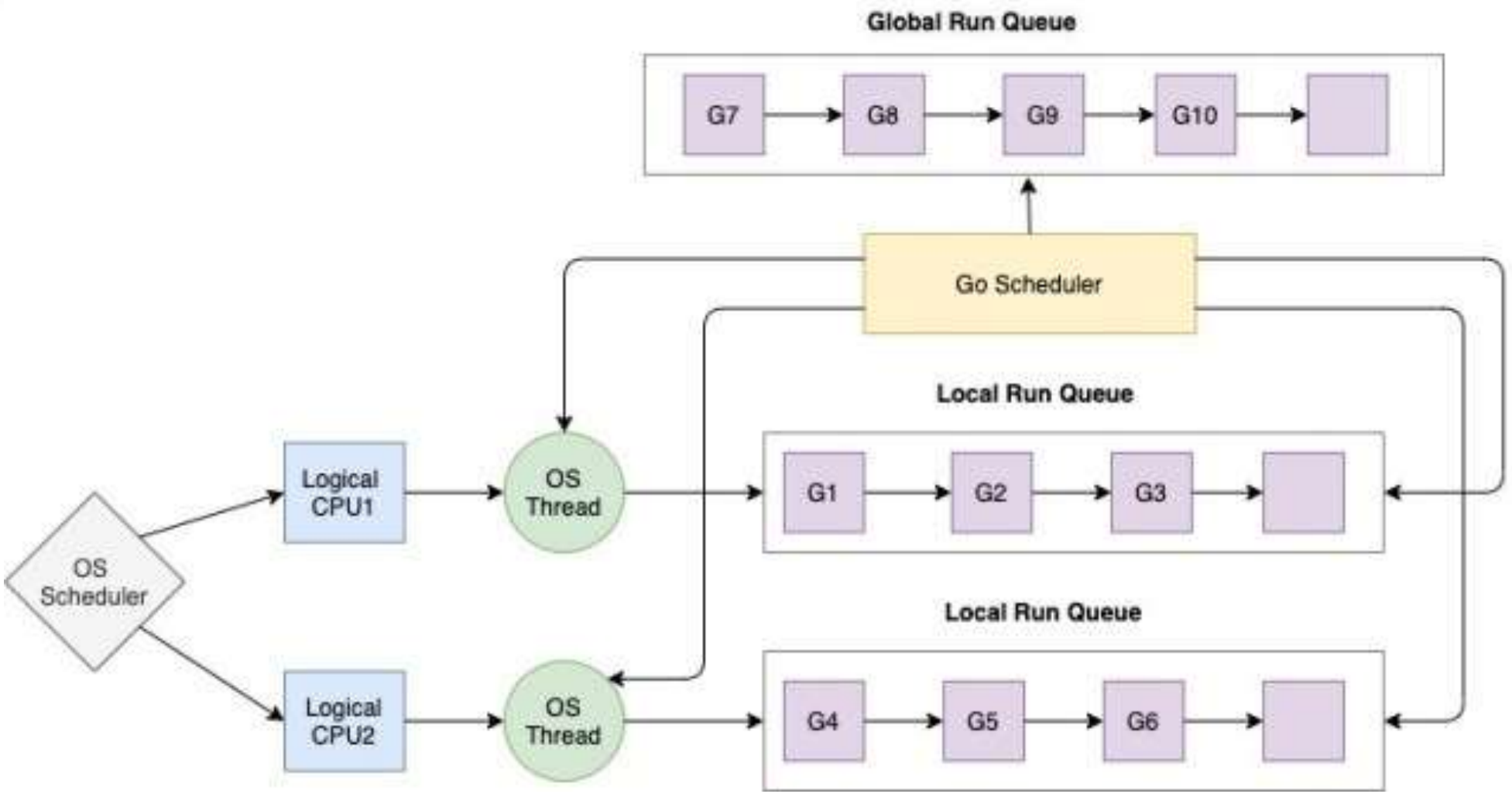
Concurrency in Go

- Go has built-in features Goroutines, Channels to achieve Concurrency
- Go works on the Communicating Sequential Process(CSP) concurrency model

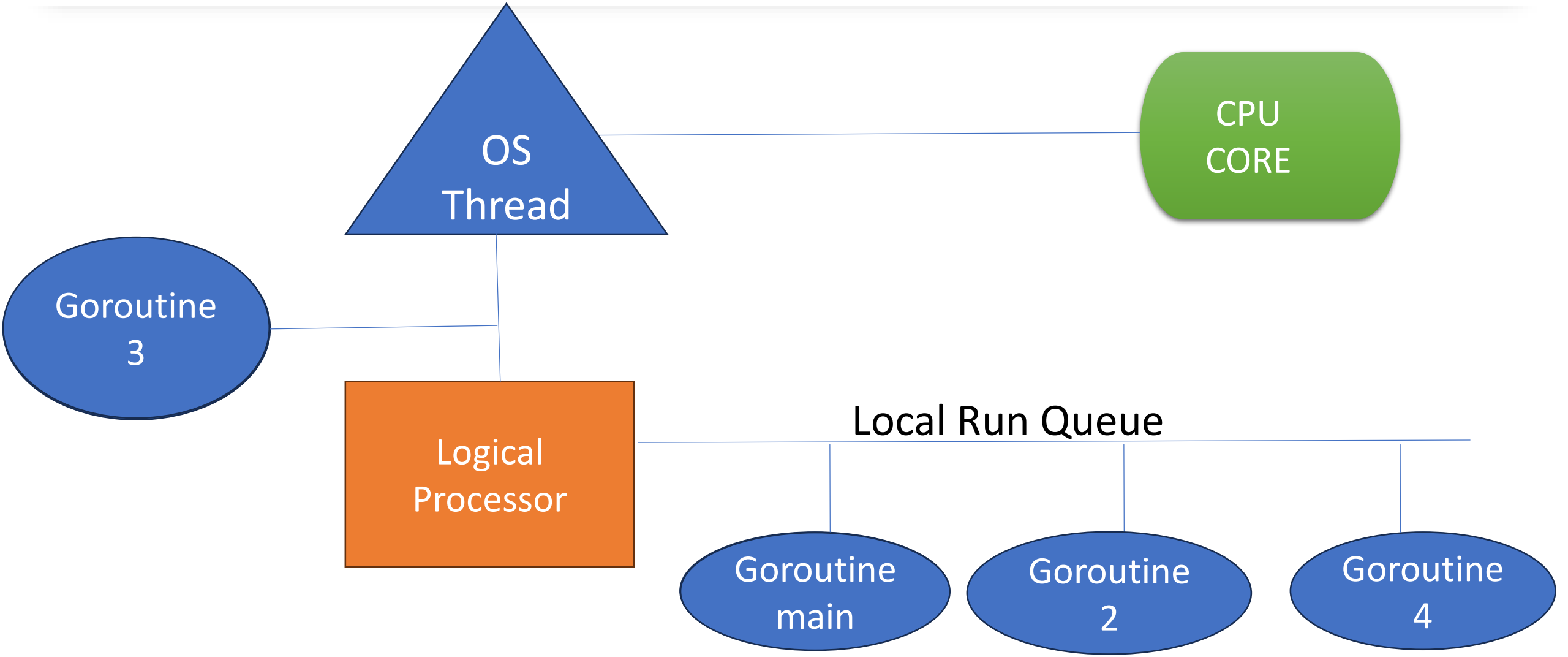
Go runtime, Logical Processor, OS thread, Goroutine



Go runtime, Logical Processor, OS thread, Goroutine



Go runtime, Logical Processor, OS thread, Goroutine

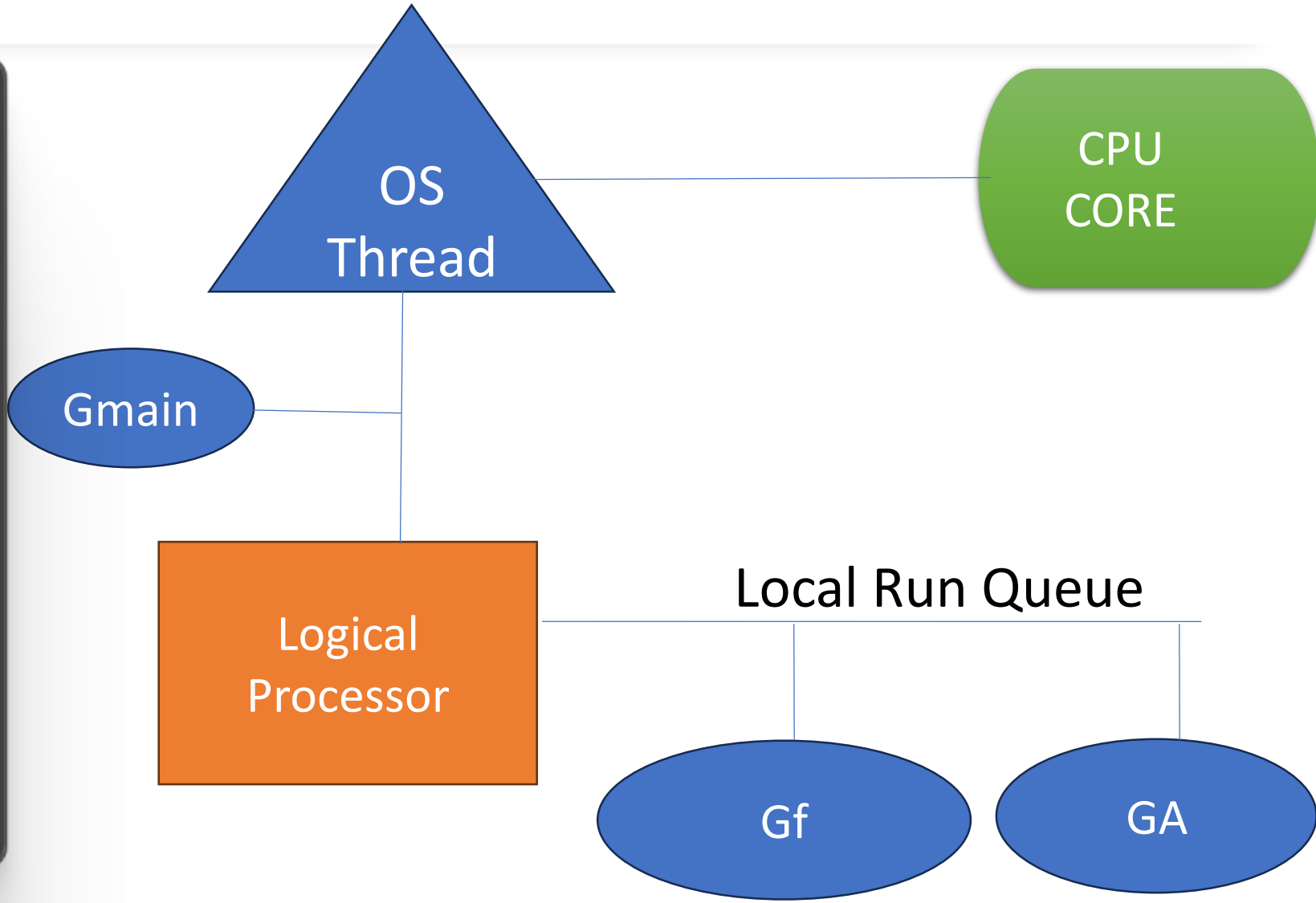


Goroutines

- A goroutine is a lightweight thread in managed by the Go runtime. It can continue its work alongside the main goroutine and thus creating concurrent execution.
- To work with Goroutines, you must prefix the go keyword to the function or method.
- Every program consists of at least one Goroutine and that Goroutine is known as the main Goroutine

Go runtime, Logical Processor, OS thread, Goroutine

```go
func f(from string){
  for i := 0; i < 3; i++{
   fmt.Println(from, ":", i)
  }
}
 func main() {           // Gmain
   f("direct)

   go f("goroutine")     // Gf

   go func(msg string){
     fmt.Println(msg)    // GA
   }("going")

 }
```



Goroutines

```go
func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {

    f("direct")

    go f("goroutine")

    go func(msg string) {
        fmt.Println(msg)
    }("going")

    time.Sleep(time.Second)
    fmt.Println("done")
}
```

```
$ go run goroutines.go
direct : 0
direct : 1
direct : 2
goroutine : 0
going
goroutine : 1
goroutine : 2
done
```

Waitgroups in Go

- Waitgroups allow waiting for goroutines
- when multiple goroutines are being executed it will allow it to wait for each of them to finish
- The sync package contains wait group struct

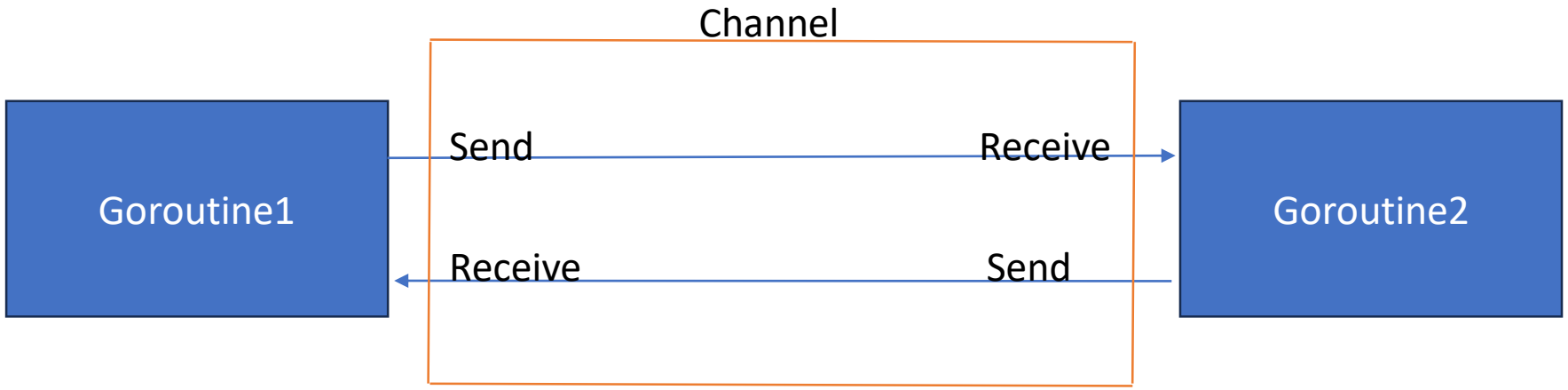
Go Waitgroup Example

```go
// Pass waitgroup as a pointer
func f(from string, wg *sync.WaitGroup) {
  defer wg.Done()                          // Using defer call done
  for i := 0; i < 3; i++ {
    fmt.Println(from, ":", i)
  }
}
func main() {
  var wg sync.WaitGroup
  wg.Add(3)                                // add to the waitgroup couter

  f("direct", &wg)

  go f("goroutine", &wg)

  go func(msg string) {
    defer wg.Done()
    fmt.Println(msg)
  }("going")

  wg.Wait()                                // call wait

  fmt.Println("Done")
}
```

```
direct : 0
direct : 1
direct : 2
going
goroutine : 0
goroutine : 1
goroutine : 2
Done
```

Channel in Go

- Channels are a medium via which goroutines communicate
- It is the primary medium for sending and receiving information.
- Be unidirectional or bidirectional and can send any data through them
- The send or receive operation the channels blocks unless the work is done. Thus allowing them to be synchronized.



Channels syntax

- A channel is dependent on the data type it carries
- **chan** is a keyword which is used to declare the channel using the **make** function.
- To send and receive data using the channel we will use the channel operator which is **<-**

```
// a channel that only carries int
ic := make(chan int)
ic <- 42          // send 42 to the channel
v := <-ic         // get data from the channel
```

Working with Channels(bidirectional)

```go
func SendDataToChannel(ch chan int, value int) {
  ch <- value
}

func main() {

  var v int
  ch := make(chan int) // create a channel

  go SendDataToChannel(ch, 101) // send data via a goroutine

  v = <-ch // receive data from the channel

  fmt.Println(v) // 101
}
```

Working with Channels(unidirectional)

```go
func SendDataToChannel(ch chan<- int, value int) {
  ch <- value
}

func main() {

  ch := make(chan<- int) // create a channel

  go SendDataToChannel(ch, 101) // send data via a goroutine
  go SendDataToChannel(ch, 102) // send data via a goroutine
  go SendDataToChannel(ch, 103) // send data via a goroutine
}
```

Closing a channel

- A channel can be closed after the values are sent through it
- Close function does that and produces a boolean output which can then be used to check whether it is closed or not.

```go
func SendDataToChannel(ch chan string, s string) {
  ch <- s
  close(ch)
}

func main() {

  ch := make(chan string)

  go SendDataToChannel(ch, "Hello World!")

  // receive the second value as ok
  // that determines if the channel is closed or not
  v, ok := <-ch

  if !ok {
    fmt.Println("Channel closed")
  }
  fmt.Println(v) // Hello World!
  f, ok := <-ch
  if !ok {
    fmt.Println("Channel closed")
  }
  fmt.Println(f) // ""
}
```

```
Hello World!
Channel closed
```

Select statement

- Golang select statement is like the switch statement, which is used for multiple channels operation
- This statement blocks until any of the cases provided are ready

```
select {
    case case1:
        // case 1...
    case case2:
        // case 2...
    case case3:
        // case 3...
    case case4:
        // case 4...
    default:
                // default case...
}
```

Using a loop with a channel

- A range loop can be used to iterate over all the values sent through the channel

```
func f(ch chan int, v int) {
    ch <- v
    ch <- v * 2
    ch <- v * 3
    ch <- v * 7
    close(ch)
}

func main() {

    ch := make(chan int)

    go f(ch, 2)

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

```
2
4
6
14
```

Select statement example

```go
func g1(ch chan int) {
  ch <- 12
}

func g2(ch chan int) {
  ch <- 32
}

func main() {

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

  go g1(ch1)
  go g1(ch2)

  select {
  case v1 := <-ch1:
    fmt.Println("Got: ", v1)
  case v2 := <-ch2:
    fmt.Println("Got: ", v2)
  }
  // Display Got 12
}
```

The default case in select channel

- The default case is executed if none of the other cases are ready for execution
- It prevents the select from blocking the main goroutine since the operations are blocking by default

```go
func main() {

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

  go g1(ch1)
  go g1(ch2)

  select {
  case v1 := <-ch1:
    fmt.Println("Got: ", v1)
  case v2 := <-ch2:
    fmt.Println("Got: ", v2)
  default:
    fmt.Println("The default case!")
  }
  //  The default case!
}
```

The race condition

- The race condition appears when multiple goroutines try to access and update the shared data
- They instead fail to update data properly and produce incorrect output
- This condition is called race condition and happens due to repeated thread access

The race condition

```go
func f(v *int, wg *sync.WaitGroup) {
    *v++
    wg.Done()
}

func main() {

    var wg sync.WaitGroup
    var v int = 0

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go f(&v, &wg)
    }

    wg.Wait()
    fmt.Println("Finished", v)
}
```

```
goroutines$ go run .
Finished 865
goroutines$ go run .
Finished 941
goroutines$ go run .
Finished 913
```

Mutex

- A mutex is simply a **mut**ual **ex**clusion in short
- When multiple threads access and modify shared data before they do that they need to acquire a lock
- When the work is done they release the lock and let some other goroutine to acquire the lock.
- Its two methods: Lock and Unlock

Mutex

```go
func f(v *int, wg *sync.WaitGroup, m *sync.Mutex) {
    m.Lock()   // acquire lock
    *v++       // do operation
    m.Unlock() // release lock
    wg.Done()
}
func main() {

    var wg sync.WaitGroup
    // declare mutex
    var m sync.Mutex
    var v int = 0

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go f(&v, &wg, &m)
    }

    wg.Wait()
    fmt.Println("Finished", v)
}
```

```
goroutines$ go run .
Finished 1000
goroutines$ go run .
Finished 1000
goroutines$ go run .
Finished 1000
```

Go vs Java: Memory Management

- Both Go and Java have garbage collectors to help prevent memory leaks
- Generational garbage collectors slow down code execution
- Go's garbage collector is improved with newer versions and is optimized to prevent garbage collection pauses
- Java uses a Generational garbage collector managed by a virtual machine

Reference

Go versus Java fastest performance

The Computer Language 23.03 Benchmarks Game
How programs are measured (Reference)

fannkuch-redux

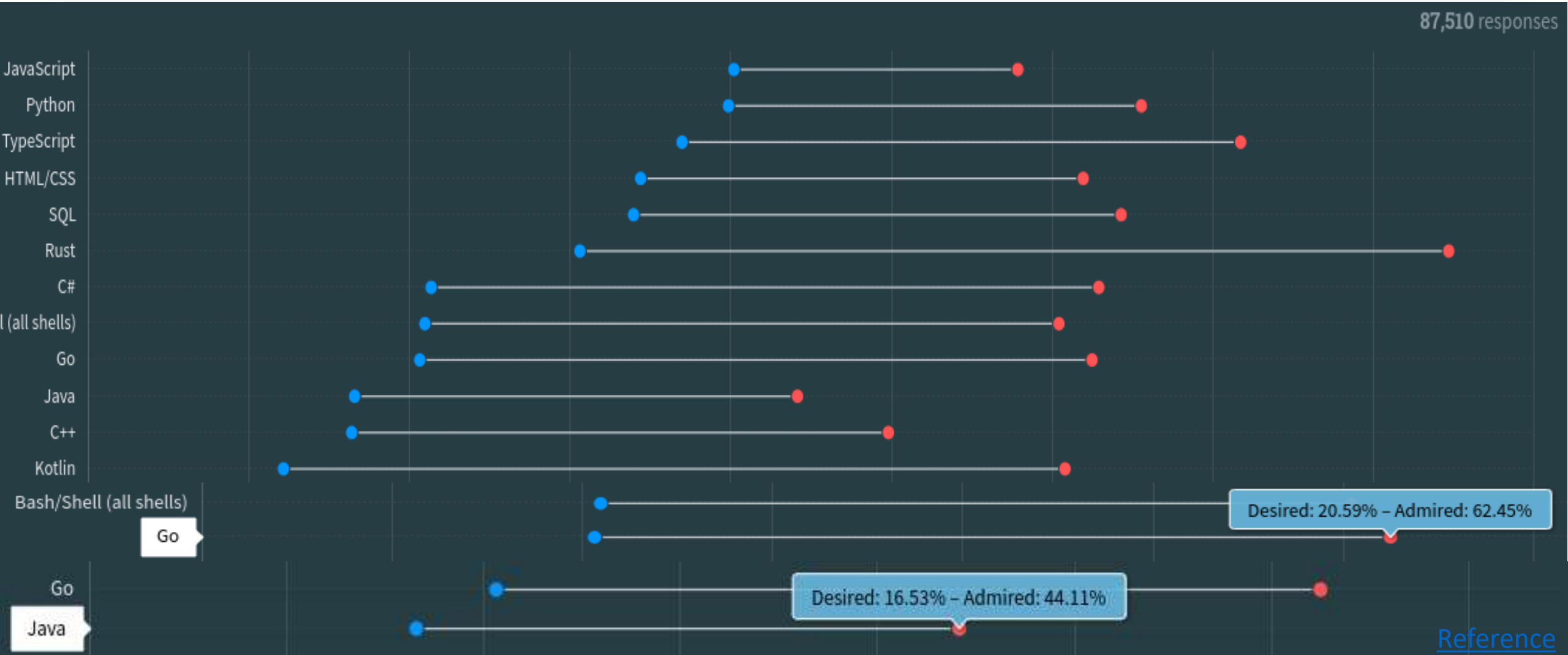
source	secs	mem	gz	cpu secs
Go #3	8.25	10,936	969	32.92
Java #3	40.36	39,932	1257	40.43
Java	10.71	40,324	1282	42.25
Java #2	46.85	38,492	514	46.91
Go	11.83	11,128	900	47.26
Go #2	11.89	12,708	896	47.47

n-body

source	secs	mem	gz	cpu secs
Go #3	6.36	11,244	1200	6.37
Go	6.55	11,244	1310	6.56
Java #5	6.77	40,632	1429	6.82
Java #4	6.86	40,516	1489	6.89
Go #2	6.94	11,244	1215	6.95
Java #2	7.32	40,508	1424	7.37
Java #3	7.38	40,492	1430	7.43
Java	7.76	40,724	1430	7.80

Reference

Compare desired and admired between Java vs Go



Reference

Compare OOP in Go and Java

Class in Java

```java
public class Person {
    // Fields (or instance variables)
    private String name;
    private int age;

    // Constructor method
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method for the "name" field
    public String getName() {
        return name;
    }

    // Getter method for the "age" field
    public int getAge() {
        return age;
    }
       // Method to display information about the person
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

Struct in Go

```go
type Employee struct {
 name string          // field name of type string
 age int              // field age of type int
 salary float32       // field salary of type float32
}
func(emp* Employee) increasedAgeByOne() {
    emp.age++
}
func(emp* Employee) increasedAge(increaseBy int) {
    emp.age += increaseBy
}
func(emp* Employee) display() {
    fmt.Println(emp.name, emp.age)
}
func main(){
 employee :=  Employee{"John",21,20000000}
 employee.increasedAgeByOne()
 display()                          // John, 22
 employee.increasedAge(5)
 display()                          // John, 27
}
```

Inheritance in Java

The **extends keyword** indicates that you are making a new class that derives from an existing class

Inheritance in Java

```java
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}
public class Main {
    public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
  }
}
```

Inheritance in Go

In Go to achieve inheritance structs are used such that the anonymous field property can be used to extend a struct to another struct

```go
type Polygon struct {
    Sides int
}
func (p *Polygon) NSides() int {
    return p.Sides
}
type Triangle struct {
    Polygon // anonymous field
}
func main() {
    t := Triangle{
        Polygon{
            Sides: 3,
        },
    }
    fmt.Println(t.NSides()) // 3
}
```

Encapsulation in Java

- Access modifier keyword: private, publish, protected

```
public class Person {
 private String name;
 private int age;

  public Person(String name, int age){
    this.name = name;
    this.age = age;
}
public void getName() {
  return name;
}
}
```

Encapsulation in Go

- In the Go language, encapsulation is achieved by using packages
- Go provides two different types of identifiers, exported and unexported identifiers
- Encapsulation is achieved by exported elements(variables, functions, methods, fields, structures)
- Exported identifiers are those identifiers which are exported from the package in which they are defined. The first letter of these identifiers is always in capital letter
- Unexported identifiers are those identifiers which are not exported from any package. They are always in lowercase

Encapsulation in Go

- Exported identifiers

```go
package main

import (
    "fmt"
    "strings"
)
func main() {
    // Creating a slice of strings
    slc := []string{"GeeksforGeeks", "geeks", "gfg"}
    // Using ToUpper() function
    for x := 0; x < len(slc); x++ {

        // Exported Method
        res := strings.ToUpper(slc[x])

        // Exported Method
        fmt.Println(res)
    }
}
```

Encapsulation in Go

- Unexported identifiers

```go
package main
import "fmt"

func addition(val ...int) int {     // Unexported function
    s := 0

    for x := range val {
        s += val[x]
    }

    fmt.Println("Total Sum: ", s)
    return s
}

func main() {                       // Main function
    addition(23, 546, 65, 42, 21, 24, 67)
}
```

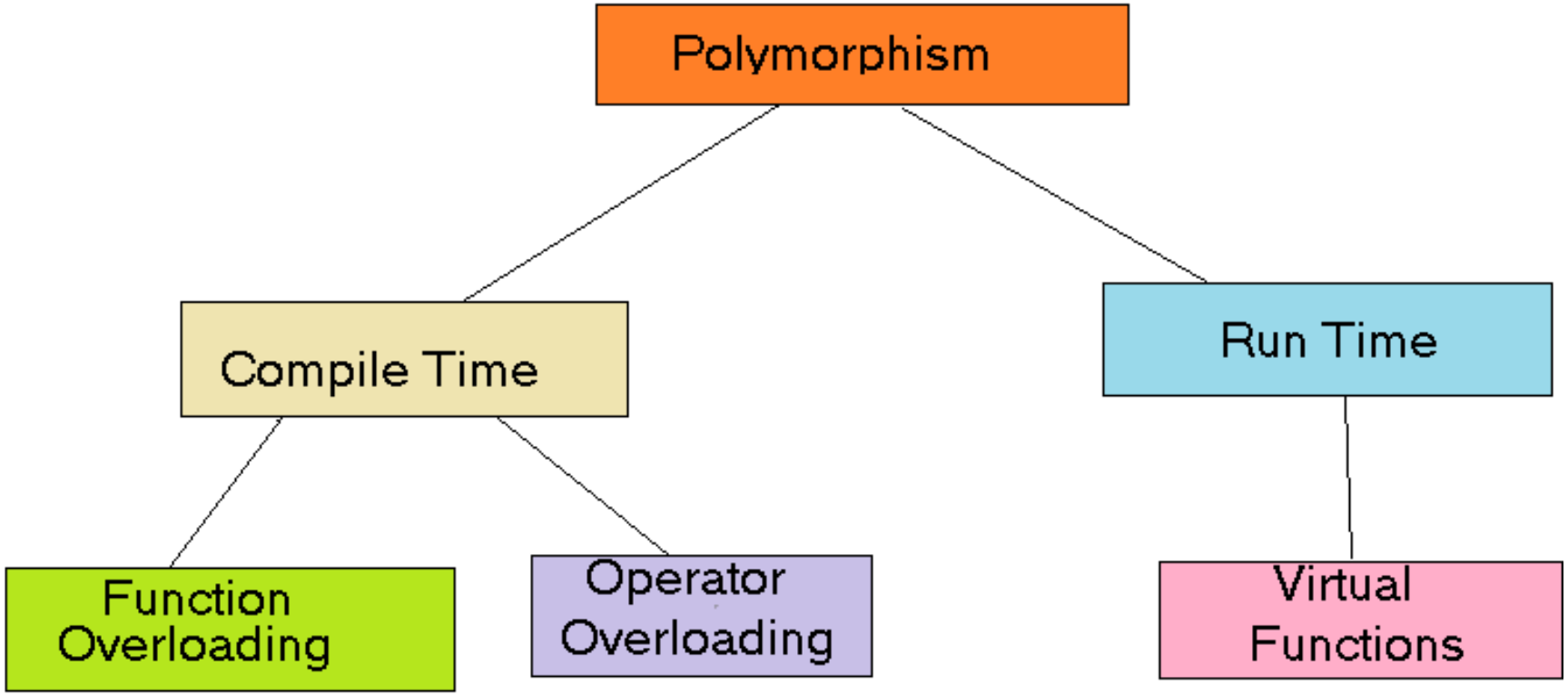
Constructor in Go

- Go language does not support constructors directly like Java, C++

```go
type Person struct {
  age int
}

func PersonDetail(age int) *Person {
  return &Person{age}
}
func (p *Person) Display() {
  fmt.Println("Pham Tuan Duong")
}

func main() {
  person := PersonDetail(10)
  person.Display()

}
```

Polymorphism in Java

- Compile-time Polymorphism
- Runtime Polymorphism



Polymorphism in Go

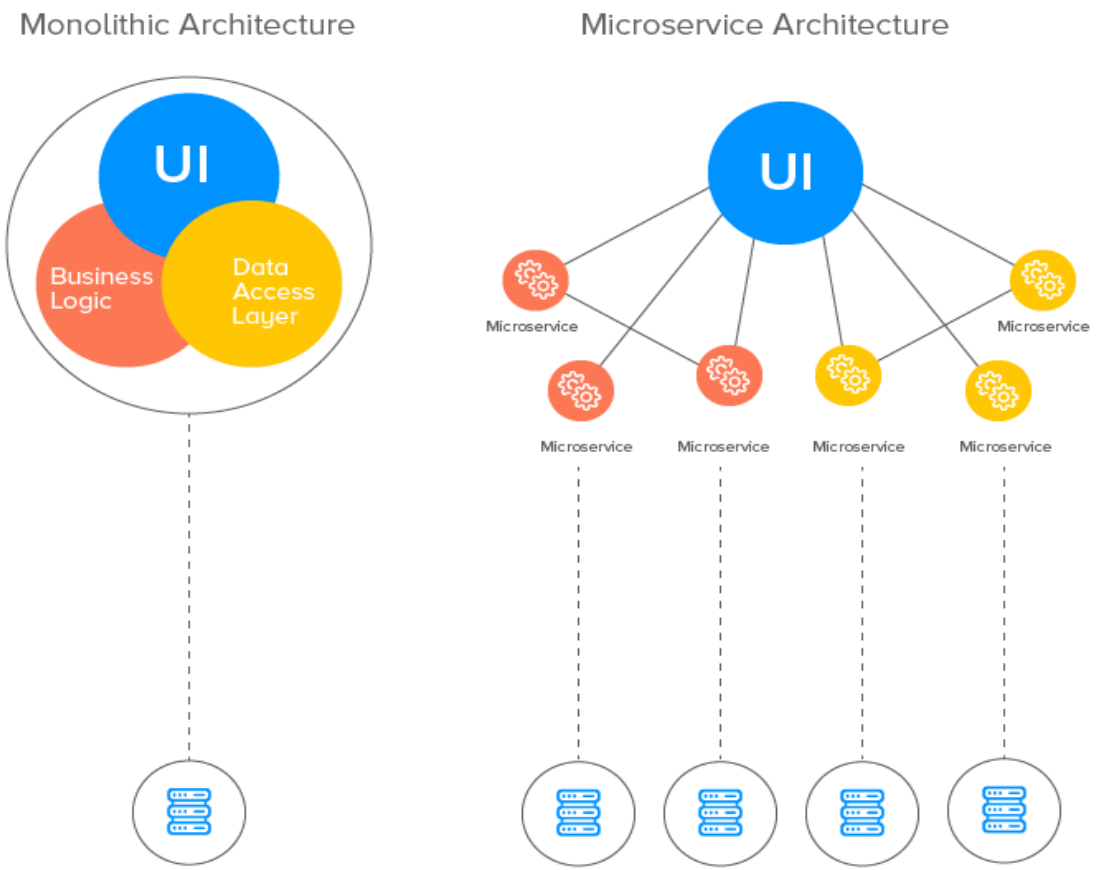
- Go despite not being an OO-language achieves polymorphism through interfaces.
- A type implementing a function defined in interface becomes the type defined as an interface

Polymorphism in Go

```go
// declare interface
type Dog interface {
    Bark()
}

// declare struct
type Dalmatian struct {
    DogType string
}

// implement the interface
func (d Dalmatian) Bark() {
    fmt.Println("Dalmatian barking!!")
}

func main() {
    d := Dalmatian{"Jack"}
    d.Bark()                    // Dalmatian barking!!
}
```

Software Architecture in Go

Monolithic vs Microservices Architecture



Monolithic Architecture

- A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications
- To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface

Monolithic Architecture

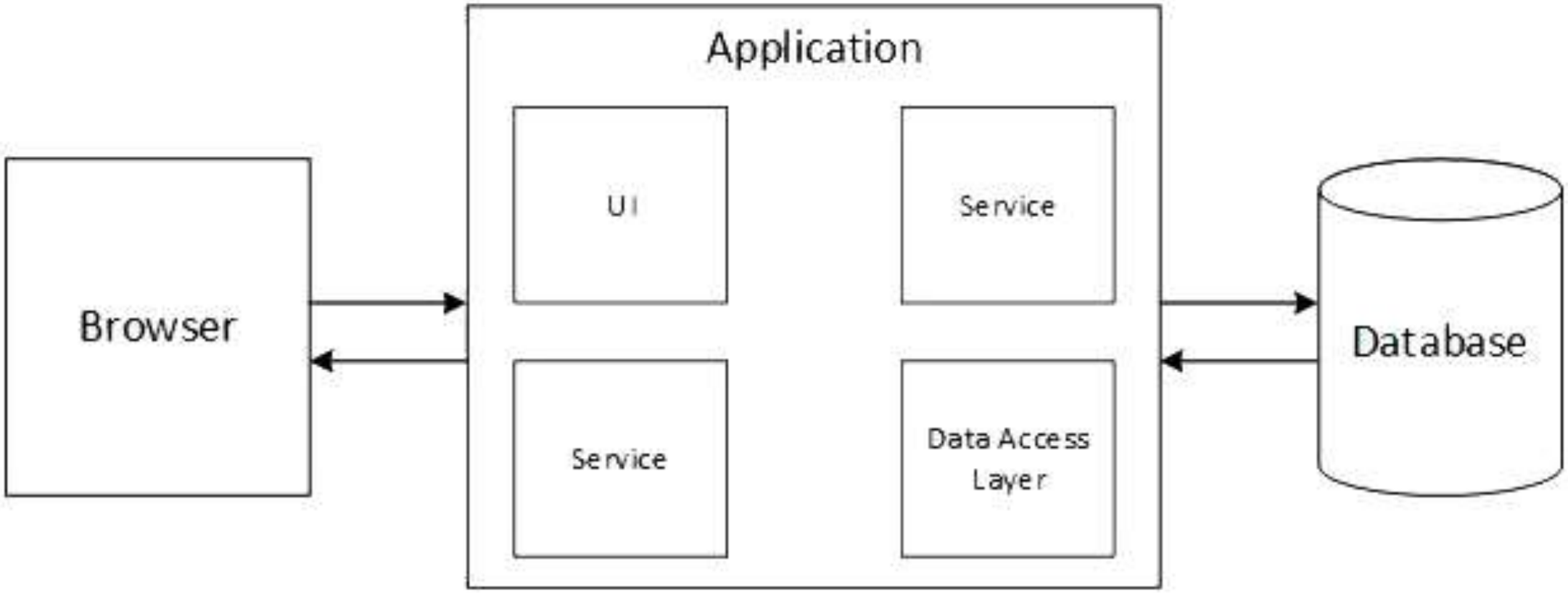
Advantages of a monolithic architecture:

- Easy deployment
- Development
- Performance
- Simplified testing
- Easy debugging

Disadvantages of a monolithic architecture:

- Slower development speed
- Scalability
- Reliability
- Barrier to technology adoption
- Lack of flexibility

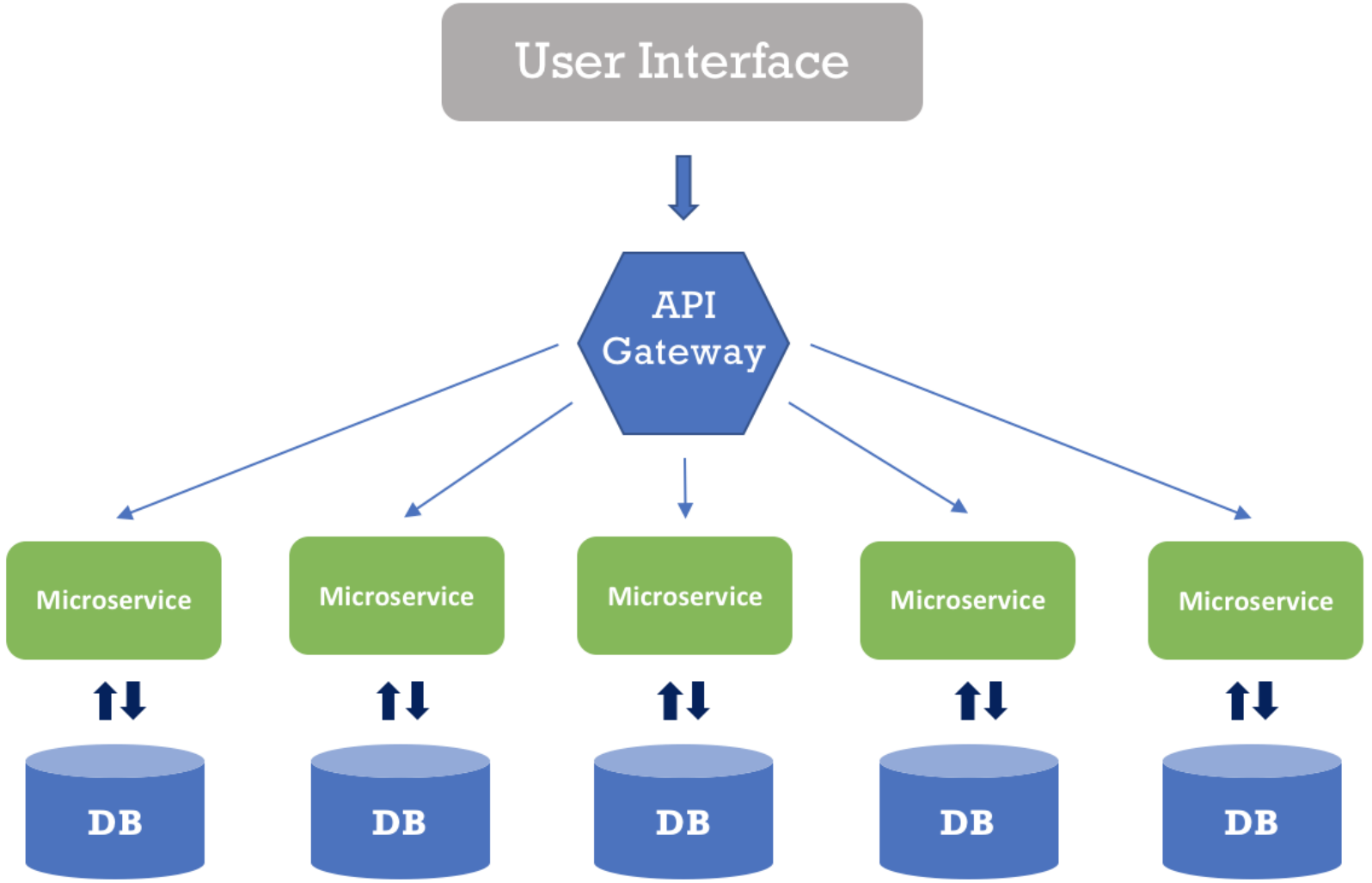
Monolithic Architecture



Microservice Architecture

- Microservice architecture an architectural method that relies on a series of independently deployable services.
- Each service can be developed, updated, deployed, and scaled without affecting the other services

Microservice Architecture



Microservice Architecture

Advantages of a microservice architecture:

- Agility
- Flexible scaling
- Continuous deployment
- Highly maintainable and testable
- Independently deployable
- Technology flexibility
- High reliability

Disadvantages of a microservice architecture:

- Development sprawl
- Exponential infrastructure costs
- Debugging challenges
- Lack of standardization

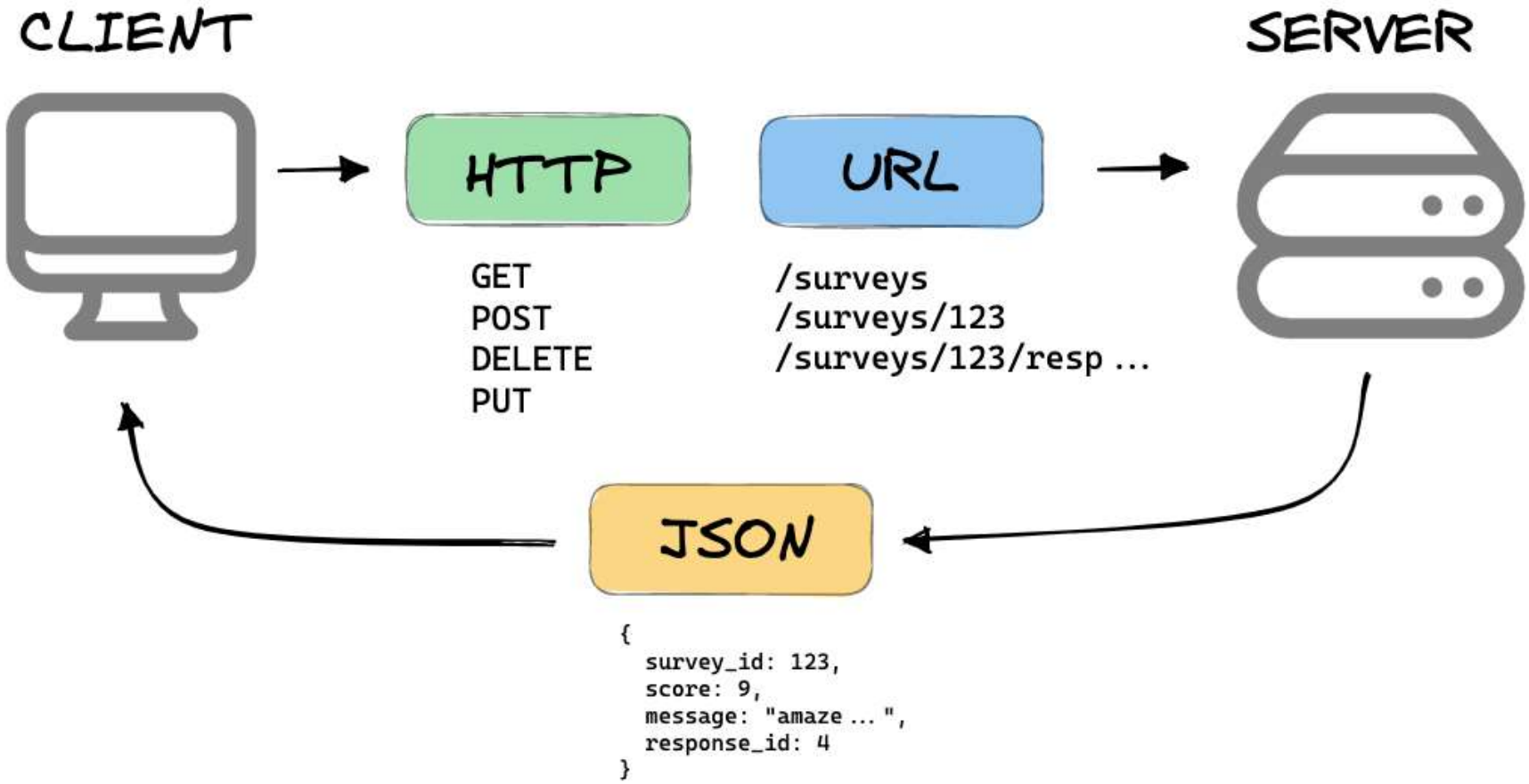
Protocols communicate between Services

Rest API, gRPC

What is Rest and API

- Representational State Transfer (REST) is a software architecture that imposes conditions on how an API should work. It's based on HTTP, the standard communication protocol of the web
- APIs that follow the REST architectural style are called REST APIs
- RESTful APIs manage communications between a client and a server through HTTP verbs, like *POST*, *GET*, *PUT*, and *DELETE*
- An application programming interface (API) defines the rules that you must follow to communicate with other software systems.
- The most common formats of API include: JSON, XML, YAML, Protobuf (Protocol Buffers), YAML

Example a Rest API



What is the RPC? GRPC introduction

- gRPC is an open source remote procedure call framework by Google in 2016. It was a rewrite of their internal RPC infrastructure that they used for years
- GRPC is a popular implementation of RPC

gRPC introduction

- gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment
- gRPC is a popular implementation of RPC