TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Introduction to Go

Corso di Sistemi Distribuiti e Cloud Computing

A.A. 2026/27

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

What is Go?



- "An open source programming language that makes it easy to build simple, secure, and scalable systems" https://go.dev/
- Conceived in 2007 at Google by R. Griesemer, R. Pike and K. Thompson, announced in 2009
- Design goals:
  - Expressive and efficient in both compilation and execution
  - Facilitate writing reliable and robust programs
  - Provide a statically typed, fast compiled language that feels like a dynamically typed, interpreted language
- Ancestors:
  - C: syntax and low-level control
  - CSP (Communicating Sequential Processes) by Tony Hoare: concurrency model https://en.wikipedia.org/wiki/Communicating_sequential_processes

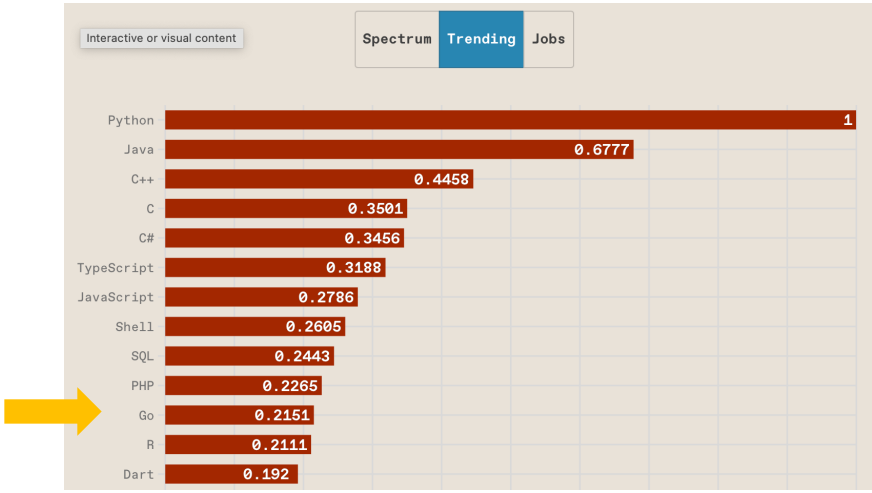
Top programming languages and Go

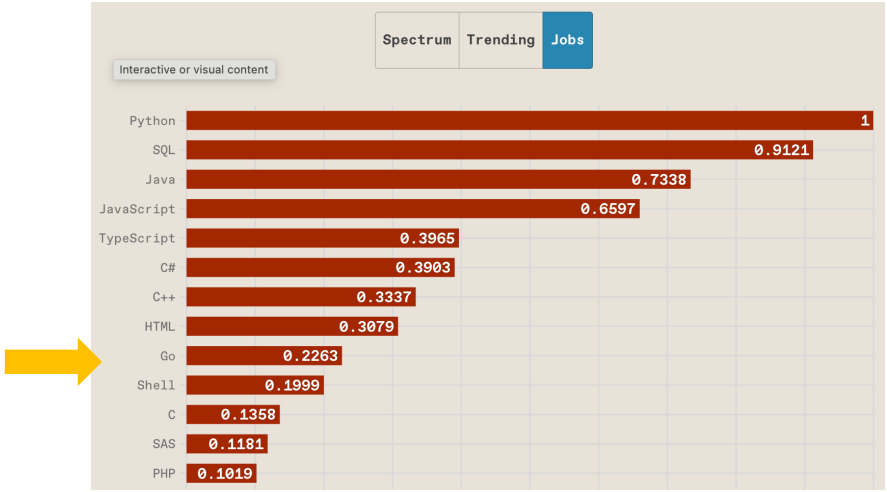
- IEEE Spectrum's ranking of Top Programming Languages 2025

  Does AI mean the end for the Top Programming Languages?

  https://spectrum.ieee.org/top-programming-languages-2025







Go and C

- Go: “C-like” or “C for the 21st century”
- Go inherits from C:
  - Expression syntax
  - Control-flow statements
  - Basic data types
  - Call-by-value parameter passing
  - Pointers
  - Run-time efficiency
  - Static typing

Go for distributed systems

- Why Go for distributed systems?
  - Compiled and statically typed
  - Built for networked services
  - A single static binary easy to deploy
- Concurrency model*:* goroutines, channels, and select vs. threads and locks
- Types and abstraction*:* structs, interfaces, composition over inheritance
- Runtime*:* garbage collection
- Ecosystem for distributed systems*:* net/rpc, gRPC, context, and real-world systems written in Go

Go in the real world

- Containers and orchestration: Docker, containerd, Kubernetes
- Coordination and consensus: etcd, Consul
- Distributed data stores: CockroachDB, Vitess, MinIO
- Messaging: NATS
- Monitoring and tracing: Prometheus, Grafana, Jaeger
- Infrastructure and networking: Terraform, Traefik, Caddy

A taste of Go concurrency model

```
func main() {
  ch := make(chan string, 3)
  for i := 1; i <= 3; i++ {
    go func(id int) { // 3 concurrent goroutines
      time.Sleep(100 * time.Millisecond) // simulate a remote call
      ch <- fmt.Sprintf("reply from server %d", id)
    }(i)
  }

  for i := 0; i < 3; i++ {
    select {
      case r := <-ch:
        fmt.Println(r)
      case <-time.After(time.Second):
        fmt.Println("timeout")
    }
  }
}
```

Editors and plugins for Go

- GoLand https://www.jetbrains.com/go/
- Visual Studio Code
  https://code.visualstudio.com/docs/languages/go
- vim-go: plugin for vim/neovim https://github.com/fatih/vim-go
- Behind the scene: gopls
  - Official Go language server that implements Language Server Protocol (LSP)
    https://github.com/golang/tools/tree/master/gopls
  - LSP: allows editors to provide advanced language features, such as code completion, hover documentation, symbol search, diagnostics and error reporting https://langserver.org

Hello world example

helloworld.go

```
package main

import "fmt"

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

Some notes on the first example

- No semicolon at the end of statements or declarations
- Native Unicode support
- Package-based structure
  - Every Go program is made up of packages (like C libraries or Python modules)
  - Package: one or more `.go` source files in a single directory
- Source file structure
  - Begins with package declaration (which package the file belongs to)
  - Followed by `import` statements for other packages
- Program entry point
  - Execution starts in `main` package and its `main()` function
- `fmt` package
  - Provides functions for formatted output and input

Go tool

- Go is a compiled language
- Go toolchain: fetch, build, and install Go packages and commands
  - Zero configuration: no need for external build scripts or makefiles
- Run program directly: **`go run`**

```
[√ go/src % go run helloworld.go
Hello, 世界
√ go/src %
```

- Build executable binary: **`go build`**

```
[√ go/src % go build helloworld.go
[√ go/src % ls helloworld*
helloworld      helloworld.go
[√ go/src % ./helloworld
Hello, 世界
√ go/src %
```

- Install packages or command: **`go install`**

Packages

- Go program is composed of packages
  - Program starts execution in package `main`
- Packages may contain declarations of types, functions, variables, and constants
- Packages can be very small or very large in scope
- Case determines visibility
  - A name starting with a capital letter is exported (public)
  - A name starting with a lowercase letter is unexported (private)
  - E.g., `Foo` is exported, `foo` unexported
  - E.g., `fmt.Println(math.pi)`
    - Error message: `./prog.go:9:19: undefined: math.pi`

Importing packages

- Go uses **`import`** statement to include other packages
- Imports can be grouped into a parenthesized ("factored") statement for readability

```
package main
import (
    "fmt"
    "math")

func main() {
    fmt.Printf("Now you have %g problems.\n", math.Sqrt(7))
}
```

Functions

- A function can take zero or more arguments
- Type comes *after* the variable name

```
func add(x int, y int) int {
    return x + y
}
```

  - `add` takes as input two arguments of type `int`
- Shorter version for input parameters:

```
func add(x, y int) int {
```

- Function can return any number of results

```
func swap(x, y string) (string, string) {
    return y, x
}
```

  - Commonly used to return result and error values

Functions

```
package main

import "fmt"

func swap(x, y string) (string, string) {
	return y, x
}

func main() {
	a, b := swap("hello", "world")
	fmt.Println(a, b)
}
```

Functions

```
package main

import (
    "fmt"
    "os"
)

func main() {
    data, err := os.ReadFile("example.txt")
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Println("File contents:", string(data))
}
```

Functions

- Go allows naming return values in the function signature
  - Named return variables are initialized to their zero values
  - “Naked” `return` without arguments returns current values

```
package main

import "fmt"

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

func main() {
	fmt.Println(split(17))
}
```

Variables

- `var` statement: declares one or more variables
  - Type comes last
- Can be used at package or function level

```
package main
import "fmt"

var c, python, java bool  // package-level variables

func main() {
  var i int               // function-level variables
  fmt.Println(i, c, python, java)
}
```

- Can include initializers, one per variable
  - If initializer is present, type can be omitted
- Variables declared without an explicit initial value are given their *zero value*
- Short variable declaration: inside functions, use `:=`
  - E.g., `count := 10`

Types

- Usual basic types
  - `bool`, `string`, `int`, `uint`, `float32`, `float64`, …
- Type conversion

```
var i int = 42
var f float64 = float64(i)
```

  - Explicit conversion required between different types (unlike C)
- Type inference
  - Go can infer the variable's type from the value on the right-hand side

```
var i int
j := i // j is an int
```

Flow control statements

- `for`, `if` (and `else`), `switch`
- `defer`

Looping construct

- Go has only one looping construct: **`for`**
- A `for` loop has 3 components
  - *Init* statement
  - *Condition* expression
  - *Post* statement

```
sum := 0
for i := 0; i < 10; i++ {
        sum += i
}
```

- No parentheses around the components
- Braces { } are always required

Looping construct

- Init and post statements are optional
- When omitted, `for` is Go's "while" loop

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

- If you omit the condition, infinite loop

```
for {
}
```

Example: echo

echo.go

```
// Echo prints its command-line arguments.
package main
import (
        "fmt"
        "os"
)
func main() {
        var s, sep string
        for i := 1; i < len(os.Args); i++ {
                s += sep + os.Args[i]
                sep = " "
        }
        fmt.Println(s)
}
```

s and sep implicitly initialized to empty strings

os.Args is a slice of strings (next slides)

```
$ go run echo.go Hello Go World
$ Hello Go World
```

Conditional statements: if

- Go's `if` (and `else`) statements are like `for`:
  - No parentheses `(  )` around the condition
  - Braces `{  }` are always required
  - Can start with a short statement to execute before the condition
- Multiple conditions with `if... else if... else` statement
  - `} else` must be on the same line

```
if optionalStatement1; booleanExpression1 {
    block1
} else if optionalStatement2; booleanExpression2 {
    block2
} else {
    block3
}
```

If: example

```
func pow(x, n, lim float64) float64 {
  if v := math.Pow(x, n); v < lim {
    return v
  } else {
    fmt.Printf("%g >= %g\n", v, lim)
  }
    // can't use v here
    return lim
}
```

%g prints floating points in compact form, choosing %e (exponential) for large value or %f (decimal)

- Notes
  - Optional statement before ; is often used for short variable declarations
  - Variables declared in the if statement (v) are scoped only within that block (including associated `else`)

Conditional statements: switch

- The **`switch`** statement selects one of many cases to execute
  - Cases are evaluated top to bottom, stopping when one case matches
- Differences from C
  - Go only runs the selected case, not all the cases that follow (i.e., C's break is provided automatically)
  - Switch cases need not be constants
  - Values can be of any type (not just integers)

```
switch day {
case "Monday":
    fmt.Println("Start of the week")
case "Friday":
    fmt.Println("Almost weekend!")
case "Saturday", "Sunday":
    fmt.Println("Weekend!")
}
```

Defer statement

- Defers execution of a function *until* the surrounding function returns
  - Deferred call's arguments are evaluated immediately, but the actual call is not executed until the surrounding function has terminated
- Deferred function calls are pushed onto a stack and executed in LIFO order

defer.go

```
package main
import "fmt"

func main() {
   defer fmt.Println("world")
   fmt.Println("hello")
}
```

```
hello
world
```

- Great for cleanup tasks, like closing files or connections, releasing locks

Pointers

- **Pointer**: value that stores the address of a variable
  - Usual operators * and &: & yields the address of a variable, * retrieves the value the pointer points to

pointers.go

```
var p *int
i := 1
p = &i          // point to i
fmt.Println(*p) // output 1
*p = 2          // equivalent to i = 2
fmt.Println(i)  // output 2
```

- Unlike C, Go has no pointer arithmetic
- The zero value for a pointer is `nil`
- Safe for a function to return the address of a local variable
  - Go's runtime ensure a local variable survives beyond the function scope if it escapes, allocating it on the heap instead of the stack

Aggregate data types: structs and arrays

- **Struct**: typed collection of fields
  - Syntax similar to C, fixed size

```
type Vertex struct {
  X int
  Y int
}
v := Vertex{1, 2}
```

  - Fields are accessed using `.` notation, e.g., `fmt.Println(v.X)`
  - Can also be accessed via a struct pointer, e.g., `p := &v`

- **Array**: `[n]T` is an array of `n` elements of type `T`
  - Fixed size (cannot be resized)

```
var a [3]string
a[0] = "Hello"
```

Aggregate data types: slices

- **Slice**: key data type in Go, more flexible than array



- `[ ]T` is a **slice** with elements of type `T`: dynamically-sized, flexible view into an *underlying array*
  - Create a slice by slicing an existing array (or existing slice)
  - Specify two indices: low and high bound separated by a colon, e.g., `s[i : j]`
  - Slice includes the first element, but excludes the last

```
primes := [6]int{2, 3, 5, 7, 11, 13}
var s []int = primes[1:4]
```

```
[3 5 7]
```

- Slice references a section of the underlying array
  - Changing a slice element modifies the corresponding element of underlying array

Slices: operations

- Slices have both length and capacity
- Length of slice `s`: number of elements it contains, use **`len(s)`**
- Capacity of slice `s`: number of elements in the underlying array from the start of the slice, use **`cap(s)`**
- Go performs bounds checking (memory-safe): compile or run-time error if array length is exceeded

- Slices can also be created with **`make`**
  - `make` allows specifying length and optional capacity

```
s := make([]int, 5, 10)
```

  - Underlying array is automatically created, but it is not directly visible

Slices: operations

making-slice.go

- Create an empty slice

```
package main
import "fmt"
func main() {
  a := make([]int, 0, 5)      // len(s)=0, cap(s)=5
  printSlice("a", a)
}

func printSlice(s string, x []int) {
  fmt.Printf("%s len=%d cap=%d %v\n", s, len(x), cap(x), x)
}
```

```
a len=0 cap=5 []
```

Slices: append operation

- New elements can be added to a slice using **append**

```
func append(slice []T, elems ...T) []T
```

– When append a slice, slice may be enlarged if necessary

append.go

```
func main() {
    var s []int
    printSlice(s)

    s = append(s, 0) // works on nil slices
    printSlice(s)

    s = append(s, 1) // slice grows as needed
    printSlice(s)

    s = append(s, 2, 3, 4) // more than one element
    printSlice(s)
}
```

Slices: append operation

- If `len(s) < cap(s)` (slice has extra capacity):
  - New element(s) are added to the underlying array
  - No new array is created
- If `len(s) == cap(s)` (capacity full):
  - Go allocates a new, larger array (usually double the previous capacity)
  - Copies the existing elements to the new array
  - Returns a slice pointing to the new array
  - The old array remains unchanged if other slices point to it

```
s := make([]int, 3, 3)  // len=3, cap=3
s = append(s, 4)        // capacity full
fmt.Println(s)          // [0 0 0 4]
```

Aggregate data types: maps

- A **map** is a collection of key-value pairs
  - Map type `map[K]V` is a reference to a hash table where `K` and `V` are the types of its keys and values
  - Use `make` to create a map

```
m = make(map[string]Vertex)
m["Bell Labs"] = Vertex{
  40.68433, -74.39967,
}
```

- Operations on map: insert or update element, retrieve element, delete element, test if key is present

```
m[key] = element      // insert or update
elem = m[key]         // retrieve
delete(m, key)        // delete
elem, ok = m[key]     // test, ok is true if key exists
```

Range

- `range` iterates over elements in a variety of data structures (arrays, slices, strings, maps and channels); behavior depends on the data type
  - Array/slice/string: `range` provides index and value for each entry
  - Map: `range` provides key and value
- Example: iterating over a slice

range.go

```
package main
import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
```

Range: example

range2.go

```
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println("key:", k)
    }
}
```

_ ignores index (or value) when it is not needed

```
$ go run range2.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
```

You can ignore the value if only the key is needed

https://go.dev/ref/spec#For_statements

Exercise: WordCount

```
package main
import (
        "strings"
        "fmt"
)
func WordCount(s string) map[string]int {
        counts := make(map[string]int)
        words := strings.Fields(s)
        for _, w := range words {
                counts[w] += 1
        }
        return counts
}

func main() {
        // Sample usage
        result := WordCount("this is a test this is only a test")
        fmt.Println(result)
}
```

Anonymous functions and closures

- Go functions can be anonymous, i.e., without a name
  - Defined inline, often used immediately or assigned to a variable

```
f := func(x, y int) int {
    return x + y
}
fmt.Println(f(3, 4)) // 7
```

- Go functions can be closures
  - Closure: nested anonymous function that captures outer variables from its enclosing scope
  - A closure retains its own private state, independent from other instances
  - Other languages support closures (e.g., Python, Rust)

See https://www.calhoun.io/closures-in-go

Closure: example 1

closure.go

```
package main

import "fmt"

func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}

func main() {
    nextInt := intSeq()
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())
    newInts := intSeq()
    fmt.Println(newInts())
}
```

```
$ go run closure.go
1
2
3
1
```

Closure: example 2

closure_fib.go

```
package main

import "fmt"

// fibonacci is a function that returns
// a function that returns an int
func fibonacci() func() int {
    x, y := 0, 1
    return func() int {
        x, y = y, x+y
        return x
    }
}

func main() {
        f := fibonacci()
        for i := 0; i < 10; i++ {
                fmt.Println(f())
        }
}
```

Elegant and efficient!

Methods

- Go does not have classes, but supports **methods** on concrete types, including structs
- A method is a function with a special *receiver* (an extra parameter before the function name) attached to a specific type (e.g., a struct)
  - The receiver appears between `func` and the method name
  - Allows defining behavior on the type

```
type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Interfaces

- **Interface**: a named collection of method signatures
- Implicit interface implementation
  - In Go a type implicitly satisfies an interface by implementing all the methods declared in the interface
  - No explicit declaration needed (unlike Java's `implements`)
  - The type automatically satisfies the interface by implementing the required methods, no need to declare that it implements the interface
- Goal: design abstractions based on behavior (actions), not data
- Interfaces enable polymorphism in Go

Interface: example

- An interface for geometric shapes

interface.go

```
package main

import "fmt"
import "math"

// Define the interface for geometric shapes
type geometry interface {
    area() float64
    perim() float64
}

// Implement the interface on rectangular and circle types
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}
```

Interface: example

```
// To implement an interface in Go, you just need to
// implement all the methods in the interface.

// Implement area and perim methods on rect
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// Implement area and perim methods on circle
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Interface: example

```
// If a variable has an interface type, then you can call
// methods that are in the named interface. Here's a
// generic measure function taking advantage of this
// to work on any geometry
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // The circle and rect struct types both implement the
    // geometry interface so you can use instances of these
    // structs as arguments to measure
    measure(r)
    measure(c)
}
```

```
$ go run interface.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```

Concurrency in Go

- Go provides concurrency as part of the core language
- **Goroutines** and **channels** enable concurrent execution
  - Implements the CSP concurrency model
- Can be used to implement different concurrency patterns

Goroutines

- Goroutine: lightweight thread managed by Go runtime
- Easy to use: prefix a function call with `go` to start a new goroutine

```
go f(x, y, z)   // start a new goroutine running
                // f(x, y, z)
```

- Goroutines share the same address space, so synchronization is needed for accessing shared memory
- Be careful: when `main` function returns, the program exits immediately without waiting for other goroutines to complete
  - See goroutine_termination.go for an example

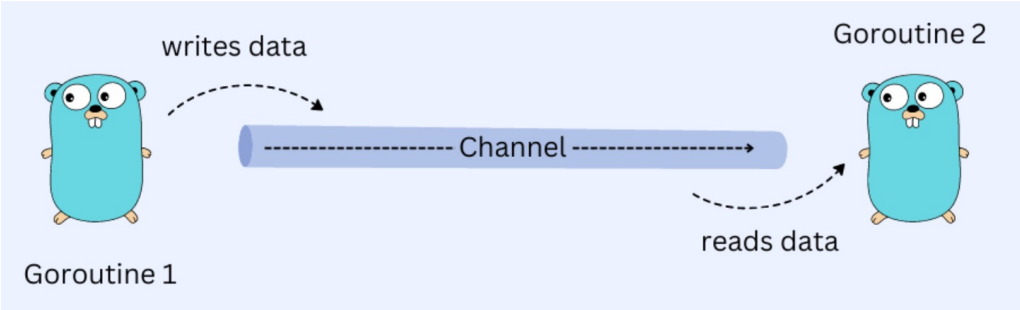
Goroutines

- Goroutines vs threads
  - Goroutines are lightweight user-space threads multiplexed onto OS threads
    - Managed by Go scheduler, which schedules them over OS threads: one OS thread can run many goroutines
  - Goroutines are much cheaper to create and destroy compared to OS threads (~5x) and less memory consuming (~500x)
- Order of execution
  - Goroutines are not guaranteed to run in the declared order
  - They have the same priority and you cannot control the execution order
- Controlling goroutine performance
  - Use `runtime.GOMAXPROCS` to control how many OS threads Go will use simultaneously
    - Typically, number of virtual CPU cores

https://dev.to/gophers/what-are-goroutines-and-how-are-they-scheduled-2nj3

Channels

- Channels enable efficient and safe communication between goroutines by sending values
  - Channel: thread-safe queue managed by Go runtime
- Hides complexity of inter-thread communication
  - Internally, channels use mutexes and semaphores to ensure safe communications



- Multiple goroutines can send values into the same channel
  - Ideal for notifications, multiplexing, etc.
  - Thread-safe communication out of the box!

Channels

- Channel: a typed conduit through which a goroutine can send and receive values using the `<-` channel operator

Data flows in the arrow direction

```
ch <- v    // Send v to channel ch
v := <- ch // Receive from ch and assign to v
```

- Channels can carry values of a specific type (e.g., `int, struct`)
  - By default, channels are bidirectional
- Create channel with `make` before using it

```
ch := make(chan int)
```

- Send and receive block until the other side is ready
  - Goroutines can synchronize without needing explicit locks or condition variables

See https://gobyexample.com/channel-synchronization

Channels: example

channel_sum.go

```
func sum(s []int, c chan int) {
        sum := 0
        for _, v := range s {
                sum += v
        }
        c <- sum // send sum to c
}

func main() {
        s := []int{7, 2, 8, -9, 4, 0}
        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        sum1, sum2 := <-c, <-c // receive from c
        fmt.Println("Total sum:", sum1+sum2)
}
```

- Parallel sum: sum is performed in parallel by 2 goroutines
- **SPMD (Single Program Multiple Data)** pattern for parallelism: each goroutine executes the same program on different parts of data

Channels: example

channel_fib.go

```
package main
import "fmt"
func fib(c chan int) {
        x, y := 1, 1
        for {
                c <- x
                x, y = y, x+y
        }
}
func main() {
        c := make(chan int)
        go fib(c)
        for i := 0; i < 10; i++ {
                fmt.Println(<-c)
        }
}
```

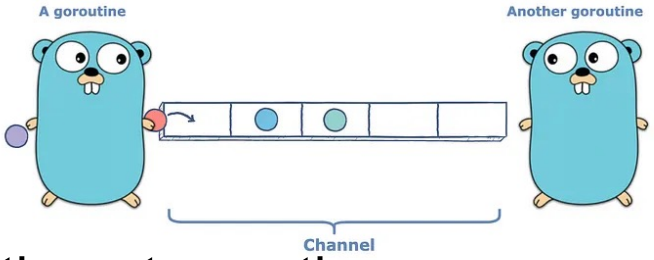
- Fibonacci sequence generator: version using goroutine and channel

Unbuffered vs. buffered channels

- Unbuffered channels*:* by default, channel ops block
  - Go spec.: *If the capacity is zero or absent, the channel is unbuffered and communication succeeds only when both a sender and receiver are ready. If the channel is unbuffered, the sender blocks until the receiver has received the value* https://go.dev/ref/spec#Channel_types
- Buffered channels do not block if they are not full or not empty
  - Specify buffer capacity when creating the channel

    ```
    ch := make(chan int, 100)
    ```

    - If capacity is zero (or not specified), channel is unbuffered
  - Send blocks only when buffer is full
  - Receive blocks only when buffer is empty



  - Useful for decoupling goroutines, allowing them to continue execution without synchronizing at every step

More on channels: close and range

- Close a channel using `close`, e.g., `close(ch)`
  - Sender(s) must stop sending after the channel is closed
  - Receiver(s) can still receive any buffered values
  - Receiver(s) can check if channel is closed using `ok`

    ```
    v, ok := <- ch
    ```

    - `ok` is false if the channel is closed and there are no more values to receive
- Only the sender(s) should `close` a channel
  - Sending on closed channel causes a runtime panic

    ```
    panic: send on closed channel
    ```

  - Receiver(s) can continue to receive values from a closed channel until it is empty (*zero value* is returned)
  - See example https://gobyexample.com/closing-channels

channel_close.go

More on channels: range

- Use `range` to repeatedly receive values from a channel until it is closed

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

More on channels: select

- **`select`** allows a goroutine to wait on multiple channels at once
  - Blocks until one of its cases can proceed, then executes that case
  - If multiple cases are ready, one is chosen randomly
  - If no cases are ready and there is a default one, it is chosen

  https://go.dev/ref/spec#Select_statements

```
select {
    case msg1 := <-ch1:       // receive on ch1
         // Handle msg1
    case msg2 := <-ch2:       // receive on ch2
         // Handle msg2
    }
```

- Useful for multiplexing and handling timeouts

Using select: example

- Fibonacci sequence generator: version with two channels, `ch` for sending values and `quit` for signaling when to stop

channel_fib_select.go

```
package main
import "fmt"

func fibonacci(c, quit chan int) {
        x, y := 1, 1
        for {
                select {
                  case c <- x:      // send Fibonacci number
                       x, y = y, x+y
                  case <- quit:    // receive termination
                       fmt.Println("quit")
                       return
                }
        }
}
```

Using select: example

```
func main() {
        ch := make(chan int)  // unbuffered channel
        quit := make(chan int)
        go func() {           // anonymous goroutine
                for i := 0; i < 10; i++ {
                        fmt.Println(<-ch) // receive Fibonacci val.
                }
                quit <- 0
        }()
        fibonacci(ch, quit) // call to generate the sequence
}
```

More on channels: select

- Non-blocking select with default clause
- Use `default` case to avoid blocking when channels are not ready

```
select {
    case msg1 := <-ch1:        // receive
        // Handle msg1
    case msg2 := <-ch2:        // receive
        // Handle msg2
    case ch3 <- msg3:          // send
        // Handle msg3
    default:
        // ...
    }
```

See example with non-blocking channel operations
https://gobyexample.com/non-blocking-channel-operations

Timers

- Use a timer channel to wait for a specific duration
  - The timer provides a channel (`C`) that notifies you when the duration elapses

```
// to wait 2 seconds
timer := time.NewTimer(time.Second * 2)
    <- timer.C   // blocks until the timer fires
```

  - Timer can be canceled before it fires using `Stop()`
  - See example https://gobyexample.com/timers
  - Useful to implement
    - Timeouts: wait for an event or operation to complete before proceeding
    - Delays: wait for a period of time before performing a task

Exercise: implement mutex using channel

- Go provides mutexes to ensure safe access to shared data across multiple goroutines
  - Go's **`sync.Mutex`** is commonly used for locking and unlocking
  - See example https://gobyexample.com/mutexes
- We can also implement a mutex using channels: how?

```
type Lock struct {
    // ?
}
func NewLock() Lock {
    // ?
}
func (l *Lock) Lock() {
    // ?
}
func (l *Lock) Unlock() {
    // ?
}
```

Exercise: implement mutex using channel

mutex_using_channel.go

```
type Lock struct {
        ch chan bool
}
func NewLock() Lock {
        lock := Lock{make(chan bool, 1)} // buffered channel
        lock.ch <- true  // send
        return lock
}
func (l *Lock) Lock() {
        <-l.ch  // receive
}
func (l *Lock) Unlock() {
        l.ch <- true  // send
}
```

- Note: for most scenarios, using `sync.Mutex` is more idiomatic and efficient

Mutex in Go

- Mutex (`sync.Mutex`)
  - Ensures mutual exclusion, prevents race conditions
  - Only one goroutine can access the critical section at a time
  - Usage:

```
mu.Lock()
counter++
mu.Unlock()
```

- Read/write mutex (`sync.RWMutex`)
  - Allows multiple readers or one writer
  - Improves performance when reads are frequent, writes are rare
  - Readers: `RLock()` / `RUnlock()` → shared access
  - Writer: `Lock()` / `Unlock()` → exclusive access

Synchronization with wait group

- **`sync.WaitGroup`** allows goroutines to collectively wait for an event before proceeding independently again
- Like a concurrent-safe counter: functions `Add`, `Done`, and `Wait`
- When to use
  1. Cleanup: ensure all goroutines (main included) finish before program termination
     - Example https://gobyexample.com/waitgroups
  2. Bulk Synchronous Parallel (BSP) pattern: goroutines work independently for a while, then wait at a barrier before proceeding again; data exchange can occur at barrier
     https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Goroutines and priority scheduling

- Goroutines do not have priority scheduling
  - Go scheduler focuses on fairness and simplicity
- To enforce prioritization, implement it at application-level
  - Adding priority can increase complexity
- Common approaches to simulate priority:
  1. Use multiple channels for different priority classes (e.g., high, low)
  2. Use a priority queue (heap) to manage and order tasks by priority

Additional topics

- Modules
- Variadic functions
- Error handling
- Go tools
- Testing and benchmarking
- RPC in Go
  https://www.ce.uniroma2.it/courses/sdcc2627/slides/DS_Communication-RPC.pdf

- Many more things to explore, this is just an introduction!
  - E.g., HTTP support in `net/http` package

Go modules

- **Module**: collection of related Go packages stored in a directory structure with a `go.mod` file at its root
- The `go.mod` file defines:
  - module path: also used as import path for root directory
  - minimum Go version required by the module
  - dependencies: the other modules required for a successful build, along with their minimum version

- Generate `go.mod` file:

```
$ go mod init <module_name>
```

- Add missing (or remove unused) requirements:

```
$ go mod tidy
```

```
module example.com/mymodule

go 1.17

require github.com/gorilla/mux v1.8.0
```

https://go.dev/doc/tutorial/create-module
https://www.digitalocean.com/community/tutorials/how-to-use-go-modules

Variadic functions

- Variadic functions can accept a varying number of arguments
  - E.g., `fmt.Println` is a variadic function
  - Use `…type` to declare variadic parameters

variadic.go

```
package main

import "fmt"

func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
```

```
func main() {

    sum(1, 2)
    sum(1, 2, 3)

    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

```
$ go run variadic.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

Error handling

- Go uses error values to indicate abnormal states
- Explicit error return: errors are communicated via a separate return value

  "Error handling [in Go] does not obscure the flow of control." (R. Pike)

  – Error is the last return value, by convention

  – A `nil` error position means no error occurred

```
result, err := SomeFunction()
if err != nil {
    // handle error
}
```

- The `error` interface type is a built-in interface

```
type error interface {
    Error() string
}
```

  – `errors.New` constructs a basic error value with a message

Code guidelines and tools

- Go can be somewhat picky
  - Unused variables raise errors (not warnings)
    - Use blank identifier _ for variables you don't care about (e.g., the loop index when you need only the value)
  - In `if-else` statement the `{` must be placed at the end of the same line, e.g.

```
} else {
} else if … {
```

  - Unused `imports` raise an error

- Recommended command-line tools:
  - gofmt: format Go code https://pkg.go.dev/cmd/gofmt

```
$ gofmt -w yourcode.go
```

  - goimports: automatically add/remove imports https://pkg.go.dev/golang.org/x/tools/cmd/goimports
  - godoc: browse package documentation https://pkg.go.dev/golang.org/x/tools/cmd/godoc

Testing and benchmarking in Go

- Go `testing` package: provides tools to write unit tests
  https://pkg.go.dev/testing
- Test files are usually placed in files ending with `_test.go` (e.g., example_test.go)
- Running tests in Go:

  `$ go test`

  – Test function must be named `TestXxx` where `Xxx` is the name of the function being tested; test functions take a `*testing.T` parameter

```
func TestAbs(t *testing.T) {
  got := Abs(-1)
  if got != 1 {
    t.Errorf("Abs(-1) = %d; want 1", got)
  }
}
```

Testing and benchmarking in Go

- Go provides tools to measure code performance by running benchmark tests
- Benchmark tests are placed in `_test.go` files and begin with `Benchmark`

```
func BenchmarkXxx(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Xxx(…)
    }
}
```

  - The benchmark runs a function `b.N` times to gather performance data
  - The test runner adjusts `b.N` to run the functions multiple times for accurate measurement
- To run benchmarks

```
$ go test -bench=.
```

Benchmark/

- Example: benchmarking `make` vs. `append` on slice

References

- Go website https://go.dev
- Go standard library https://pkg.go.dev/std
- Online Go tutorial https://go.dev/tour
- Go playground https://go.dev/play
- Go by examples https://gobyexample.com

- Donovan and Kernighan, The Go programming language, Addison-Wesley, 2016 https://www.gopl.io
- Learn Go programming (7 hours video) https://www.youtube.com/watch?v=YS4e4q9oBaU
- How to code in Go https://www.digitalocean.com/community/tutorial-series/how-to-code-in-go
- More resources https://go.dev/learn