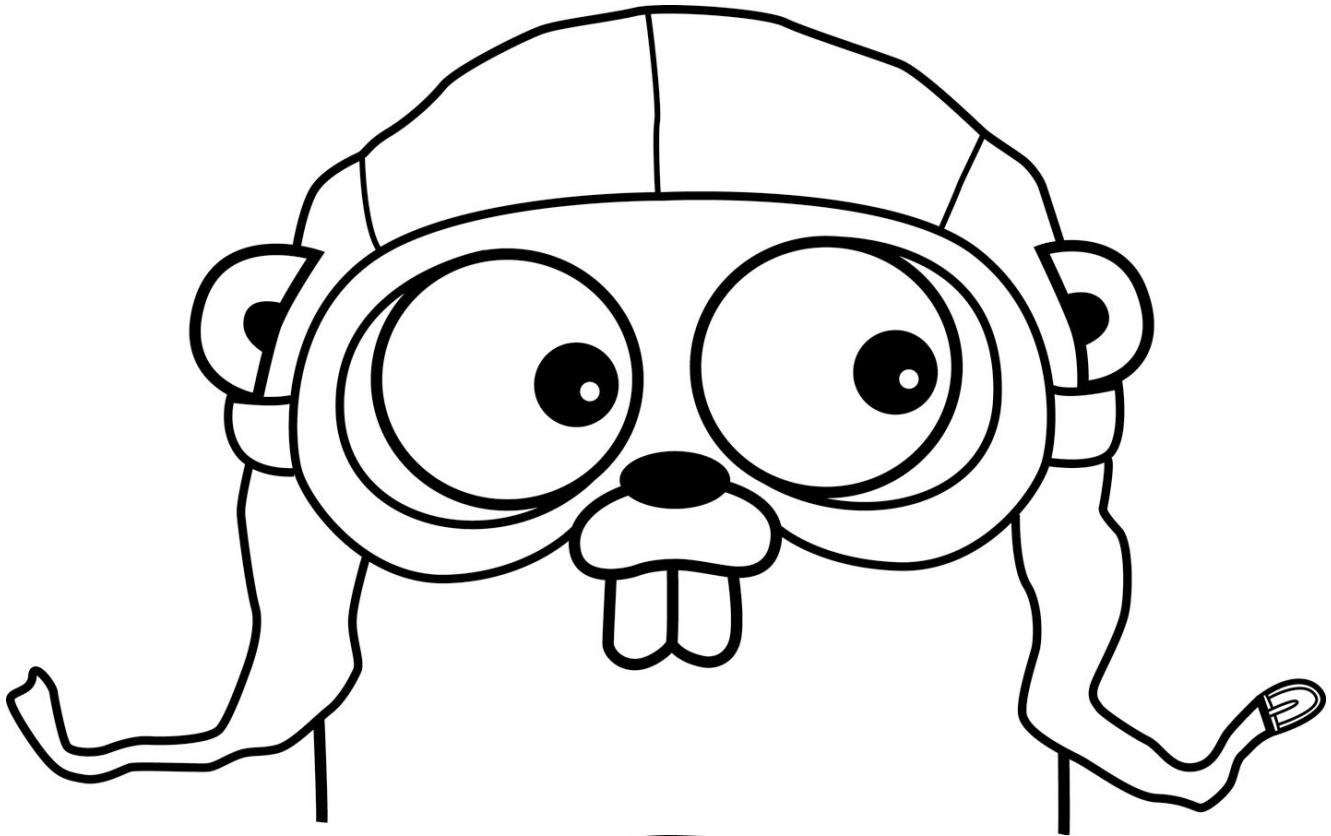
OCTOBER 29, 2021

Developing Concurrent Pentesting Tools in Go



Agenda

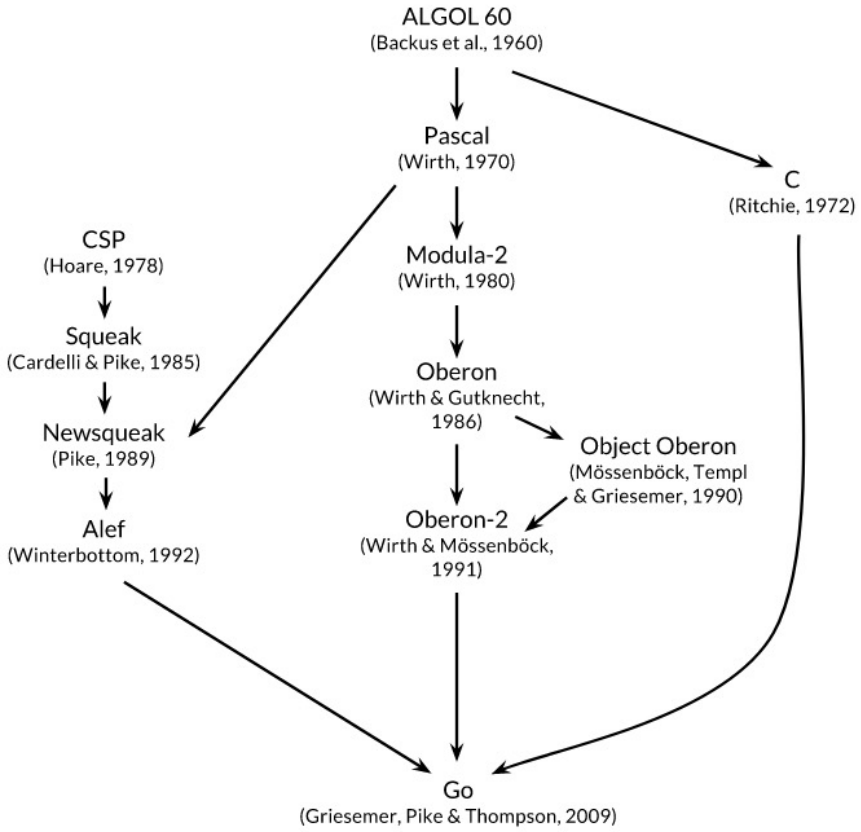
1 Language Features

2 Collaboration

3 Concurrency

4 DevOps

Origins of Go



ALGOL 60, Pascal, Modula-2, Oberon

The syntax for packages, imports, declarations and method declarations is greatly influenced by Modula-2 and Oberon.

CSP, Squeak, Newsqueak, Alef

The implementation of concurrency and communication via channels is inspired by CSP and subsequent languages that were derived from it.

C

From C, Go inherited its expression syntax, control-flow statements, basic data types, call-by-value parameter passing, pointers, and C's emphasis on programs that compile to efficient machine code and cooperate naturally with the abstractions of current operating systems.

Early team

Robert Griesemer (left)

Worked on Google V8 JavaScript engine, Sawzali language and the Java HotSpot Virtual machine.

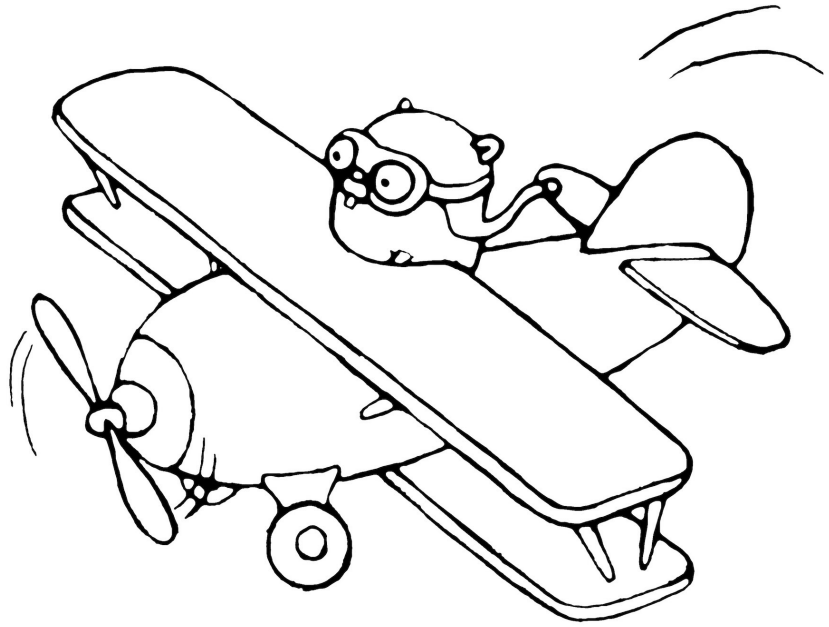
Rob Pike (middle)

Worked on Unix, Plan 9 from Bell Labs, Inferno Operating System, several books (The Practice of Programming, The Unix Programming Envrionment, first implementation of UTF-8

Ken Thompson (right)

Worked on the B Programming Language, founder of Unix, work on regular expressions, first implementation of UTF-8

Go is simple



Feature Highlights

Statically Typed

The type of a variable is known at compile time which allows checks at compile time, hereby preventing many trivial bugs. Declaring and initializing variables can be made easy using **short declarations**.

```
Basic Data types
  Integers
  Floating point numbers
  Complex
  Boolean
  Strings
  Constants

Composite types
  Arrays
  Slices
  Maps
  Structs

var i, j int = 1, 2
var number := 39
```

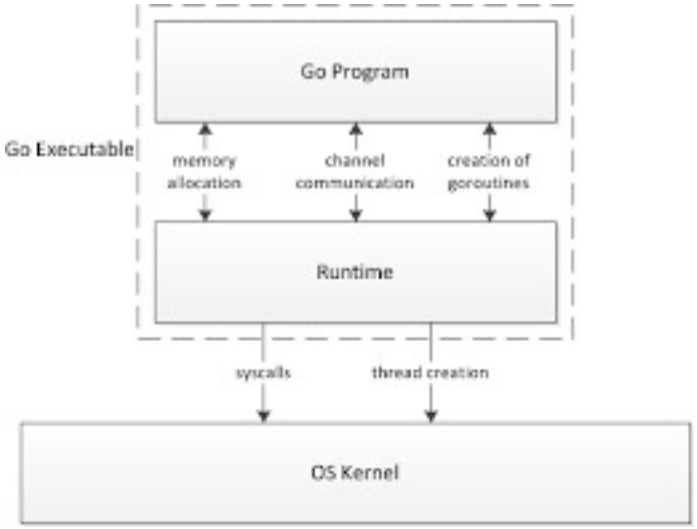
Error Handling

The existence of the **error** type and the explicit return of errors by the standard library functions, gently influence the process of writing code by encouraging error handling.

```
package main

import (
  "fmt"
)

func Sqrt(x float64) (float64, error) {
  return 0, nil
}

func main() {
  fmt.Println(Sqrt(2))
  fmt.Println(Sqrt(-2))
}
```

Garbage Collector

In go there is no need to explicitly allocate memory to different data structures ...however, that doesn't mean that memory is never to be accounted for.



Feature Highlights

If statements

Go has regular **if-statements** for handling control flow. Notice the lack of parentheses, but braces are required.

```
package main

import (
  "fmt"
  "math"
)

func pow(x, n, lim float64) float64 {
  if v := math.Pow(x, n); v < lim {
    return v
  }
  return lim
}

func main() {
  fmt.Println(
    pow(3, 2, 10), pow(3, 3, 20),
  )
}
```

For loops

Go has also regular **for loops** as you would expect from C-like languages. It also has variations similar to while and loop forever. Here again, parenthesis are not required.

```
package main

import (
  "fmt"
)

func main() {
  sum := 0

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

Function declarations

In go **objects** are simply values or variables that have methods, and **methods** are defined as functions associated with a particular type.

```
package main

import "fmt"

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

func main() {
  fmt.Println(add(42, 13))
}
```

Feature Highlights

Defer

Something about **defer**

```
package main

import "fmt"

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

Pointers

Something about **function**

```
package main

import "fmt"

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

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

  p = &j         // point to j
  *p = *p / 37   // divide j through ptr
  fmt.Println(j) // see the new value of j
}
```

Objects & Methods

In go **objects** are simply values or variables that have methods, and **methods** are defined as functions associated with a particular type.

```
package main

import (
 "fmt"
 "math"
)

type Vertex struct {
  X, Y float64
}

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

func main() {
  v := Vertex{3, 4}
  fmt.Println(v.Abs())
}
```

Feature Highlights

Structs

Something about **structs**

```
package main

import "fmt"

type Vertex struct {
  X int
  Y int
}

func main() {
  v := Vertex{1, 2}
  v.X = 4
  fmt.Println(v.X)
}
```

Objects & Methods

Something about **structs and their methods**

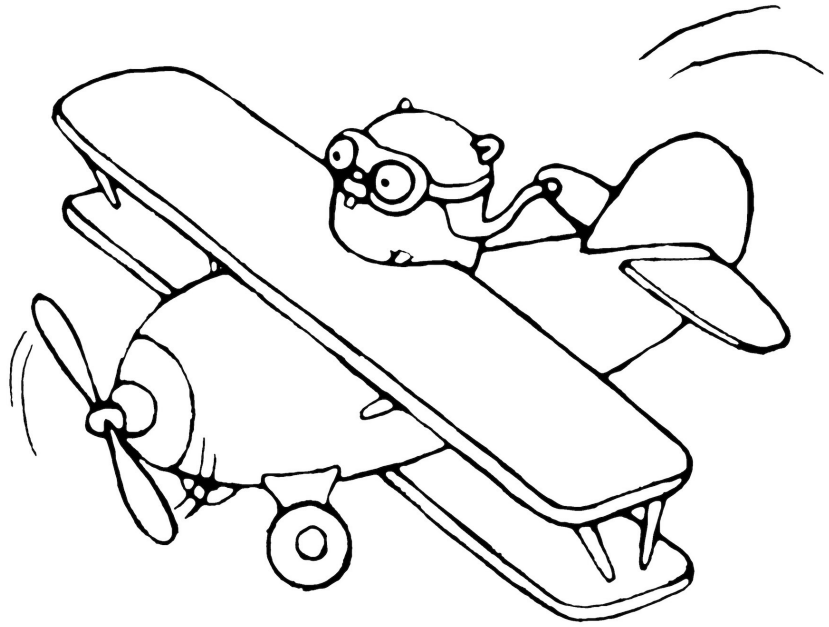
```
package main

import (
  "fmt"
  "math"
)

type Vertex struct {
  X, Y float64
}

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

func main() {
  v := Vertex{3, 4}
  fmt.Println(v.Abs())
}
```

Interfaces

Interface types express generalizations about the behaviors of other types. They let us write flexible functions without being tied to the details of one particular implementation.

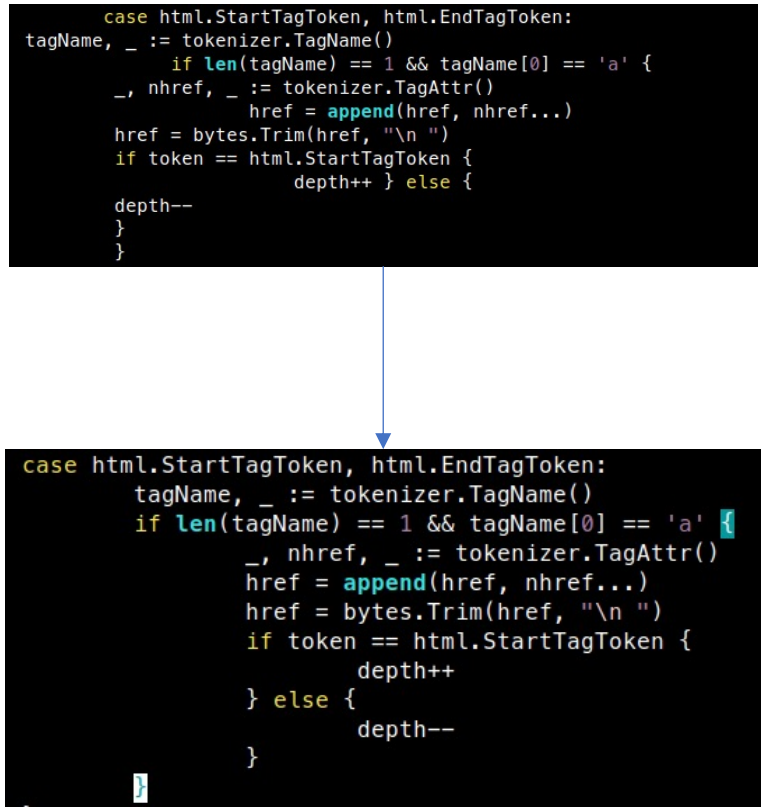
```
package main

import "fmt"

type I interface {
  MyMethod()
}

type T struct {
  S string
}

func (t T) MyMethod() {
  fmt.Println(t.S)
}

func main() {
  var i I = T{"hello"}
  i.MyMethod()
}
```

Collaboration & Tools



go fmt

gofmt is a tool that automatically formats Go source code.



Easier to write

Never worry about minor formatting concerns while coding.

Easier to read

When all code looks the same you need not mentally convert other's formatting style into something you can understand.

Easier to maintain

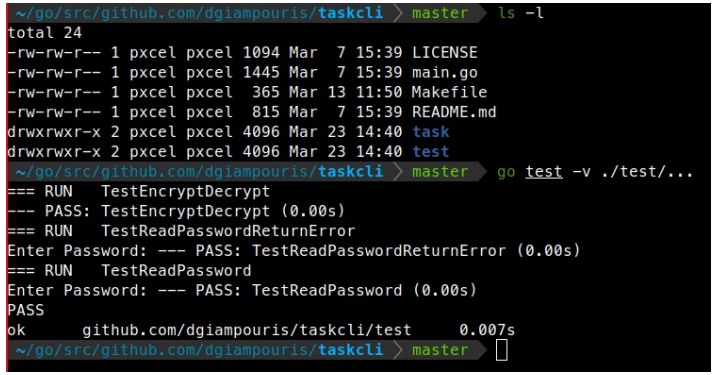
Mechanical changes to the source don't cause unrelated changes to the file's formatting; diffs show only the real changes.

Uncontroversial

Never have a debate about spacing or brace position ever again!

go test

The **go test** subcommand is a test driver for Go packages that are organized according to certain conventions. In a package directory, files whose names end with _test.go are not part of the package ordinarily built by **go build** but are part of it when built by **go test**.



Test functions

Functions whose name begins with **Test**, that exercise some program logic for correct behavior. **go test** calls the test function and reports the result, which is either **PASS** or **FAIL**.

Benchmark functions

Functions whose name begins with **Benchmark**, that measure the performance of some operation. **go test** reports the mean execution of the operation.

Example functions

Functions whose name begins with **Example**, that provide machine-checked documentation.

Packages & Imports

The purpose of any package system is to make the design and maintenance of large programs practical by grouping related features together into units.

```
package task

import (
        "encoding/binary"
        "fmt"
        "log"
        "math"
        "os"
        "strconv"

        bolt "go.etcd.io/bbolt"
)
```

```
go get go.etcd.io/bbolt/...
```

Modularity

Allows packages to be shared and reused by different projects, distributed within an organization, or made available to the wider world.

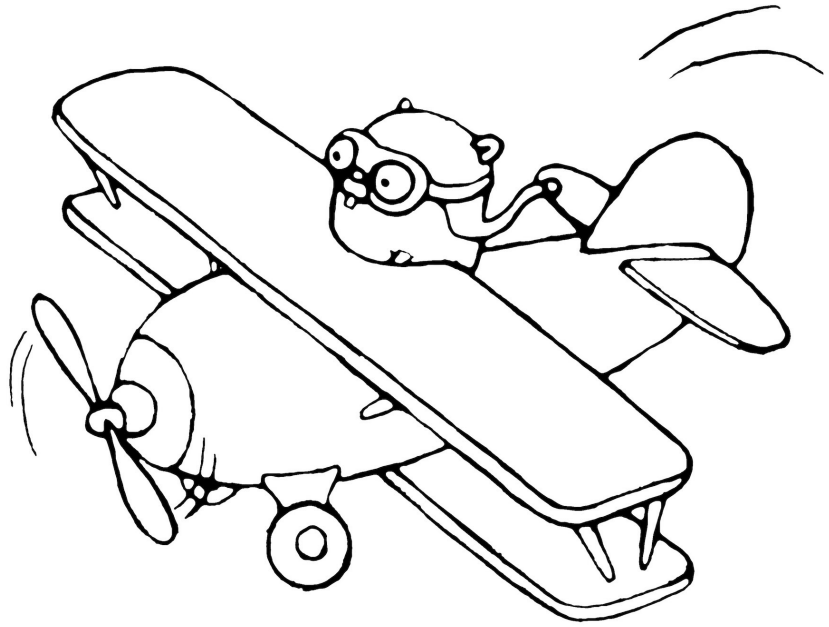
Encapsulation

Control which names are visible or exported outside the package.

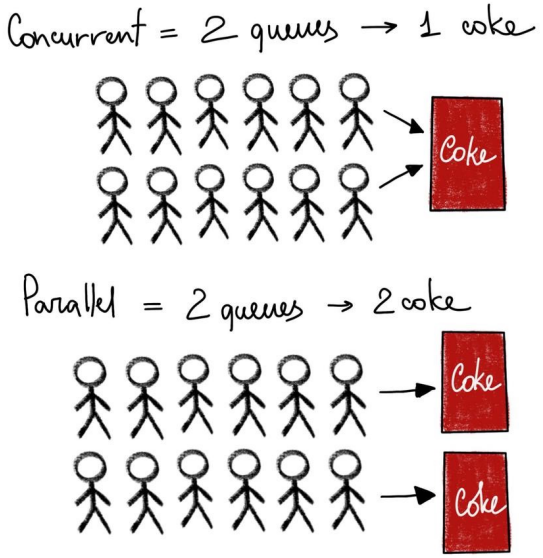
Separation

Each package defines a distinct name space that encloses its identifiers.

Concurrency patterns in Go



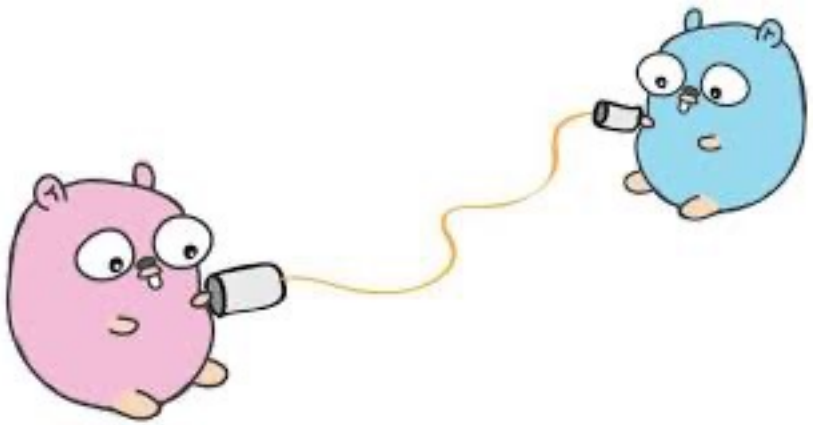
Concurrency is not parallelism



- Parallelism is about parallel processing
- Concurrency is about structuring a program, about *design*
  - Design your program as a collection of independent processes
  - Design these processes to eventually run in parallel
  - Design your code so that the outcome is always the same
    - Such that there are no race conditions
    - Such that there are no deadlocks
    - Such that more workers implies faster execution
- In the Go Programming Language, ideas originated from the paper Communicating Sequential Processes (Tony Hoare, 1978)
  - Each process is built for sequential execution
  - Data is communicated between processes via channels: no shared state!
  - Scale by adding more of the same

Channels & Go routines

A concurrently executing activity is called a **goroutine**. A function call prefixed by the keyword **go** causes the function to be called in a newly created goroutine. Complimentary, a **channel** is a communication mechanism that lets one goroutine send values to another



Channels

```
package main

import "fmt"

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)
  x, y := <-c, <-c // receive from c

  fmt.Println(x, y, x+y)
}
```

Go routines

```
package main

import (
  "fmt"
  "time"
)

func say(s string) {
  for i := 0; i < 5; i++ {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(s)
  }
}

func main() {
  go say("world")
  say("hello")
}
```

Example for a TCP scanner

```go
package main

import (
  "fmt"
  "net"
  "sort"
)
```

First, we do some formalities

```go
func worker(ports, results chan int) {
  for p := range ports {
    address := fmt.Sprintf("scanme.nmap.org:%d", p)
    conn, err := net.Dial("tcp", address)
    if err != nil {
      results <- 0
      continue
    }
    conn.Close()
    results <- p
  }
}
```

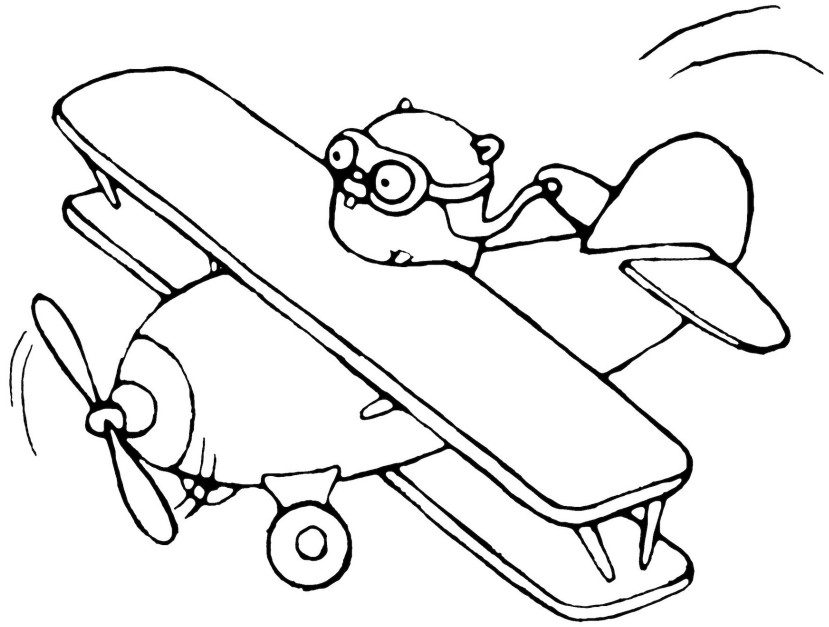
Second, the worker function will attempt to establish a TCP session on a port

Then, from the main function

- We create two chanels
- Call the worker function as a go routine
- Initialize the ports channel
- Wait for results to come in on the results channel
- Close the program when all is done with open ports listed

```go
func main() {
  ports := make(chan int, 100)
  results := make(chan int)
  var openports []int

  for i := 0; i < cap(ports); i++ {
    go worker(ports, results)
  }

  go func() {
    for i := 1; i <= 1024; i++ {
      ports <- i
    }
  }()

  for i := 0; i < 1024; i++ {
    port := <-results

    if port != 0 {
      openports = append(openports, port)
    }
  }

  close(ports)
  close(results)
  sort.Ints(openports)
  for _, port := range openports {
    fmt.Printf("%d open \n", port)
  }
}
```

What are it's applications

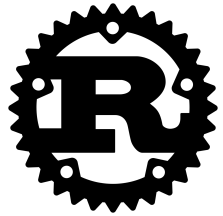


Where is Go used?

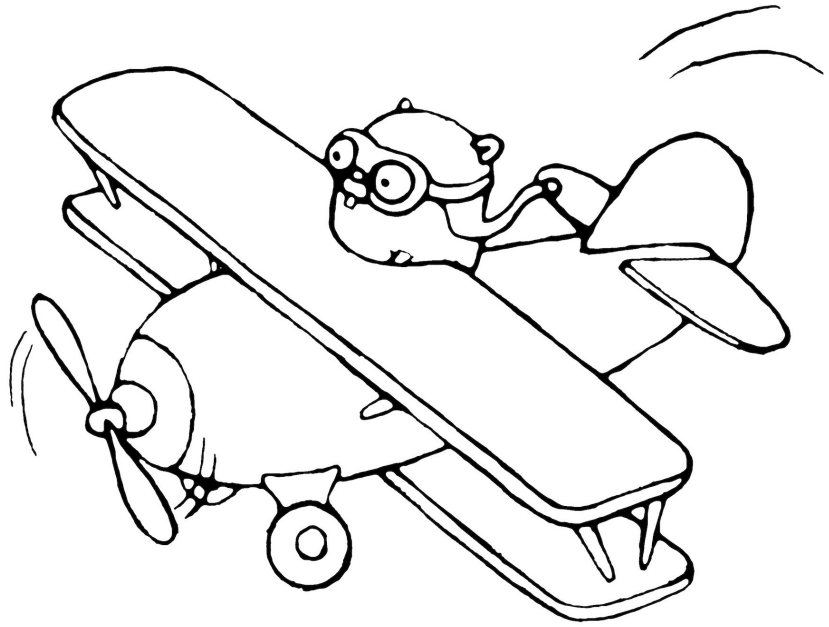
- Go is build for:
  - Running and compiling fast
  - Concurrent 'server/system application' (C++/Java)
  - Easy to write tests, easy to write code as compared to C++/Java
  - Nice balance between performance and ease of development
- Go froze the API since v1 release (2012) guaranteeing stability
- Go has currently a nice ecosystem with many packages and libraries
- Go is already regarded as ”the DevOps language”

Go versus ...

- Python:
  - Go is statically typed, Python is dynamically typed
  - Go is more verbose, no classic OOO, harder learn but much much faster
  - Go compiles to machine code, Python is interpreted (written in C)
- Java:
  - Both are statically typed
  - Go is easier to read, no classic OOO, faster
  - Go compiles to machine code, Java to JVM
- Rust:
  - Both are statically typed, memory safe languages compiling to machine code
  - Go focusses a bit more on speed of development rather than execution (Rust is faster)
  - Go was deliberately simple (small syntax, few features)



How to Go from here?



How to go from here?

- The official site with the base packages is golden:
  - https://golang.org/
  - https://pkg.go.dev
- Official Golang tour:
  - https://tour.golang.org/welcome/1
- GoByExamples:
  - https://gobyexample.com/
- Book ‘The Go Programming Language’ (Donovan, Kernighan):
  - https://www.gopl.io/
- Gophercises:
  - https://gophercises.com/