The Go Programming Language

Leonardo Regano

leonardo.regano@unica.it

Industrial Software Development

University Of Cagliari, Italy

Why Go?

- Go was designed at Google (2007) to improve productivity for:
  - large codebases and large teams
  - multicore machines (built-in concurrency support)
  - networked services and infrastructure software
  - fast builds
  - simple dependency management
  - consistent tooling

A very short history of Go

2007

Design starts at Google

Griesemer, Pike, Thompson begin work on a new language aimed at productivity.

2009

Public announcement + open source

Go is publicly introduced and released as an open-source project.

2012

Go 1.0

Compatibility promise begins (Go 1).

2019

Modules & checksum database (Go 1.13)

Proxy + checksum DB used by default for module users.

2022

Go 1.18: generics + fuzzing

Type parameters and native fuzzing arrive.

Hello, Go!

```go
package main

import "fmt"

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

- Every file starts with a package declaration
- main package + main() is the executable entry point
- Imports are explicit; unused imports are compile errors
- gofmt tool standardizes formatting

Core building blocks

Structs+methods

```
type User struct {
    Name string
    Admin bool
}

func (u User) Greeting() string {
    if u.Admin { return "Hi, admin" }
    return "Hi, " + u.Name
}
```

Interfaces

```
type Greeter interface {
    Greeting() string
}

func PrintG(g Greeter) {
    fmt.Println(g.Greeting())
}

// User satisfies Greeter
without "implements"
```

Error handling

```go
f, err := os.Open("config.json")
if err != nil {
    return err
}
defer f.Close()

data, err := io.ReadAll(f)
if err != nil {
    return err
}
// ...
```

- No exceptions: errors are returned (often as the last return value)
- Handling failures close to where they happen is encouraged
- defer statement helps cleanup of system resources (e.g. files) even on early returns
- Works well with tooling (e.g. tests, linters)

Concurrency

```
1 ch := make(chan int)
2
3 go func() {
4     ch <- work() // send
5 }()
6
7 result := <-ch  // receive
```

- goroutines are lightweight concurrent tasks
- channels coordinate and transfer ownership
- select waits on multiple channel operations
- use sync primitives too (mutexes) when appropriate

Go standard tools

- Go is shipped with a lot of useful tools
  - gofmt: automatic code formatter
    - following an officially defined canonical style
    - https://pkg.go.dev/cmd/gofmt
  - go test: standardized unit testing (and fuzzing from Go 1.18)
    - https://pkg.go.dev/testing
  - go vet: flag suspicious constructs in code (that may lead to bugs/vulnerabilities)
    - https://pkg.go.dev/cmd/vet

Pros and tradeoffs

Pros

- Fast build/test cycles; great standard tooling
- Simple, readable syntax for easier onboarding and reviews
- Excellent standard library for networking and services
- Concurrency primitives built-in (goroutines/channels)
- Static typing + GC for built-in safety

Tradeoffs

- Explicit and somewhat verbose error handling
- No classic inheritance/exceptions (by design)
- GC can add latency in some workloads
- Low-level tricks may lead to unsafe behavior

Built-in security

- memory safety:
  - bounds-checked slices and arrays
  - heap managed by the Garbage Collector
- pointers supported…
  - … but no pointer arithmetic by design
- strong static typing + automatic initialization of variables
- out-of-bounds access leads to program panic
  - to avoid memory corruption
  - https://go.dev/doc/effective_go#recover
- crypto module in standard library
  - crypto/tls for modern TLS (1.2/1.3)
  - crypto/* for strong cryptographic primitives (hashes, HMAC, RSA, ECDSA…)
- native fuzzing support from Go 1.18
  - to find edge cases/crashes
- built-in checksum checks for downloaded modules
  - to avoid supply chain attacks

A tour of Go + official resources

- interactive tutorial
  - https://go.dev/tour/
- language specification
  - https://go.dev/ref/spec
- official documentation
  - https://go.dev/doc/
- Go Standard Library documentation
  - https://pkg.go.dev/std
- “Effective Go”: a guide for writing clear Go code
  - https://go.dev/doc/effective_go
- Guides for writing secure code with Go
  - https://go.dev/doc/security/
  - https://go.dev/doc/security/best-practices