



Golang

Parameswari Ettiappan

A black and white photograph of a woman with curly hair, wearing a light-colored sweater, sitting at a desk and working on a laptop. She is looking down at the screen. A large orange diagonal bar starts from the top right and extends towards the center of the image.

High performance. Delivered.

consulting | technology | outsourcing



Goals

- Understand Go architecture and memory model
- Write idiomatic, concurrent Go programs
- Build modular and testable packages
- Use Go for systems, networking, and microservice development
- Implement secure, resilient microservices using external packages and patterns
- Use **GitHub Copilot** effectively for code suggestions, testing, refactoring, and documentation

Software Requirements



- **1. Operating System Requirements**
- Go is **cross-platform** and runs on all major OSes:
- **Linux** (x86_64, ARM64, PPC64, etc.)
- **Windows** (x86_64, ARM64)
- **macOS** (Intel & Apple Silicon)
- Other supported OS/architectures: FreeBSD, OpenBSD, NetBSD, Solaris
- Officially supported platforms: <https://go.dev/dl/>

Software Requirements



⚙️ 2. Go Compiler / SDK

- Download from go.dev/dl
- Current stable (as of 2025): **Go 1.23**
- Installation includes:
 - Go compiler (go)
 - Go tools (gofmt, godoc, etc.)
 - Standard library

System requirements (approx.):

- Disk space: **200–300 MB**
- Memory: minimum **2 GB RAM** (for builds)



📁 3. Environment Variables

- **GOROOT** → Points to Go installation path (usually auto-set).
- **GOPATH** → Workspace directory for Go projects (default = `$HOME/go`).
- **PATH** → Must include `$GOROOT/bin` and `$GOPATH/bin`.

Software Requirements



Example (Linux/macOS – `~/.bashrc`):

bash

```
export GOROOT=/usr/local/go
export GOPATH=$HOME/go
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

On Windows (PowerShell profile):

powershell

```
$env:GOROOT="C:\Go"
$env:GOPATH="$env:USERPROFILE\go"
$env:PATH+=";$env:GOROOT\bin;$env:GOPATH\bin"
```



📦 4. Build Tools & Dependency Management

- **Go Modules (go mod)** → official dependency management.
 - Initialize: `go mod init myapp`
 - Add deps: `go get github.com/gin-gonic/gin`
- No need for external build tools (like Maven/Gradle); Go handles builds with `go build`.



🧪 5. Testing & Quality Tools

- go test → built-in unit testing.go
- lint → style checker (optional).go
- vet → static analysis.go
- fmt → auto-formatting (included in Go).

Software Requirements



-
-  **6. IDE / Editors**
 - **VS Code** (with Go plugin by Google)
 - **GoLand (JetBrains)** – commercial, powerful
 - Vim/Neovim, Sublime, Atom (plugins available)

Software Requirements



-
- 7. Optional Tools for Advanced Development
 - Docker → containerize Go apps.
 - Git → version control for Go modules.
 - Task runners like make or Taskfile (cross-platform).
 - Profiling tools: pprof (comes with Go).



-
- **8. Security & Networking**
 - TLS libraries are in the Go standard library (`crypto/tls`).
 - For REST APIs → frameworks like Gin, Echo, Fiber.
 - For gRPC → google.golang.org/grpc.



Golang

- **Go (Golang) is an open-source, statically typed, compiled language created at Google in 2007 (publicly released in 2009).**
- Designed by **Robert Griesemer, Rob Pike, and Ken Thompson** (Unix/C pioneers).
- Purpose: a language that combines the **simplicity of scripting languages with the performance and safety of compiled languages like C/C++.**



Key Goals of Go

- **Simplicity** → minimal keywords, easy to learn.
- **Performance** → compiled, close to C speed.
- **Concurrency** → first-class support with goroutines & channels.
- **Portability** → cross-platform, built-in toolchain.
- **Productivity** → batteries-included standard library, fast builds.



Introduction

- Go is a programming language that was born out of frustration at Google.
- Developers continually had to pick a language that executed efficiently but took a long time to compile, or to pick a language that was easy to program but ran inefficiently in production.
- Go was designed to have all three available at the same time: fast compilation, ease of programming, and efficient execution in production.



Main Features

- **Compiled & Fast** → produces standalone binaries, no external runtime.
- **Statically Typed** → type-checked at compile time.
- **Garbage Collected** → automatic memory management.
- **Concurrency** → lightweight threads (goroutines) + communication (channels).
- **Cross-Platform** → Linux, Windows, macOS, FreeBSD, ARM, etc.
- **Rich Standard Library** → networking, web, crypto, JSON, testing.
- **Built-in Tools** → go build, go test, go fmt, go mod.



Where is Go used?

- **Cloud & Infrastructure** → Docker, Kubernetes, Terraform, Istio are written in Go.
- **Web APIs & Microservices** → Gin, Echo, Fiber frameworks.
- **DevOps Tools** → CLI tools (kubectl, Hugo, etc.).
- **Networking & Distributed Systems** → gRPC, NATS, etc.

Does Golang have a future?



- **🌟 Why Go Has a Bright Future**
- **1. Backed by Google + Big Ecosystem**
- Created by Google engineers (Rob Pike, Ken Thompson, Robert Griesemer).
- Used internally at Google for massive-scale systems.
- Supported by major companies like **Uber, Netflix, Dropbox, Cloudflare, Twitch, PayPal**

Does Golang have a future?



- **2. Cloud-Native Standard**
- **Docker, Kubernetes, Terraform, Prometheus, Grafana** → all written in Go.
- Became the **de-facto language for DevOps, cloud-native, and container tooling**.
- Cloud providers (AWS, GCP, Azure) provide Go SDKs first-class.

Does Golang have a future?



-
- **3. Performance + Concurrency**
 - Almost as fast as C/C++ but much simpler.
 - **Goroutines + channels** make concurrency **easy & scalable**.
 - Perfect for **microservices, streaming, and networking** workloads.

Does Golang have a future?



-
- **4. Developer Productivity**
 - Small language, easy to learn.
 - Built-in tooling (go fmt, go mod, go test).
 - No heavy runtime (like JVM or .NET).
 - Cross-compilation: build binaries for Linux, Windows, macOS, ARM in one command.

Does Golang have a future?



- **5. Growing Community & Libraries**
- Rich frameworks: **Gin, Echo, Fiber, Buffalo** for web APIs.
- Strong ecosystem: **gRPC, GraphQL, Kafka clients, ML packages, etc.**
- Active open-source contributions, Go conferences, and meetups worldwide.

Does Golang have a future?



- **6. Job Market & Demand**
- High demand for Go developers in **startups and large enterprises.**
- Particularly strong in roles like:
 - Cloud Engineer
 - Backend Developer
 - DevOps/SRE
 - Blockchain / Web3 engineer
- Salaries for Go developers are **competitive and often higher than average backend roles.**

Does Golang have a future?



-
-  **Challenges / Limitations**
 - Simpler than Java/C# → fewer built-in abstractions (deliberately minimal).
 - Lacks **generics until Go 1.18** (now added, but still evolving).
 - Not as mature in **GUI / Data Science / AI** compared to Python/Java.

Does Golang have a future?



-  **Future Outlook (2025 and beyond)**
-  **Stable & growing:** Go 1.23 released, continuous evolution.
-  **Native concurrency model** will remain unmatched for cloud workloads.
-  **DevOps & cloud-native ecosystem** guarantees long-term adoption.
-  **Generics (added in Go 1.18)** makes Go more flexible → more adoption for libraries & enterprise systems.
-  **Enterprise adoption rising** → banks, fintechs, and SaaS companies increasingly adopting Go for performance + scalability.



Is Golang better than Python and C++?

Aspect	Golang	C++
Performance	Very fast, but not as low-level as C++.	<input checked="" type="checkbox"/> Extreme performance, direct memory control.
Complexity	<input checked="" type="checkbox"/> Simple syntax, minimal keywords.	<input checked="" type="checkbox"/> Complex (templates, pointers, manual memory).
Memory Management	Garbage collector (automatic).	Manual (more control, but harder).
Concurrency	<input checked="" type="checkbox"/> Built-in (goroutines, channels).	<input checked="" type="checkbox"/> Threads, mutexes → harder to manage.
Use Cases	Cloud services, networking, APIs, DevOps tools.	Game engines, OS kernels, embedded systems, high-performance computing.
Portability	Cross-platform builds with ease.	Cross-platform but build systems (CMake, etc.) add complexity.



Golang vs C and C++

Linux custom development

Requirement	C	C++	Go
Size requirements in devices	Lowest	Low (1.8MB more)	Low (2.1 MB more, however will increase with more binaries)
Setup requirements in Yocto	None	None	None
Buffer under/overflow protection	None	Little	Yes
Code reuse/sharing from CFEngine	Good	Easy (full backwards compatibility)	Can import C API
Automatic memory management	No	Available, but not enforced	Yes
Standard data containers	No	Yes	Yes
JSON	json-c	jsoncpp	Built-in
HTTP library	curl	curl	Built-in
SSL/TLS	OpenSSL	OpenSSL	Built-in (TLS is a part of crypto/tls package)*



Golang vs C and C++

Aspect	Golang	C
Level	High-level (modern, simplified).	Low-level (close to hardware).
Memory Management	✓ Garbage collector (automatic).	✗ Manual (malloc/free).
Performance	Very fast (close to C, but not equal).	✓ Extremely fast (baseline for systems programming).
Concurrency	✓ Built-in goroutines + channels.	✗ No native concurrency (threads via libraries like pthreads).
Complexity	✓ Simple, readable, small spec.	✗ More boilerplate, error-prone.
Use Cases	Cloud-native apps, servers, APIs, DevOps tools.	OS kernels, embedded systems, drivers, hardware-level programming.



Golang vs C Speed Comparison

- ⚡ Go vs C — Speed Comparison
- 1. Execution Speed
- C:
 - Compiled directly to machine code, minimal runtime.
 - Extremely fast (baseline for systems programming).
 - No garbage collector → predictable performance.
- Go:
 - Also compiled to machine code.
 - Slightly slower than C due to **garbage collection (GC)** and runtime checks (e.g., array bounds, type safety).
 - Typically **10–30% slower** than optimized C in raw compute benchmarks.



Golang vs C Speed Comparison

- **3. Concurrency**
- **C:** Uses **threads + mutexes** (via pthreads). High performance, but complex to implement safely.
- **Go:** Built-in **goroutines + channels** → lightweight concurrency model.
 - Goroutines are much lighter than OS threads.
 - For concurrent workloads, Go can **outperform C in developer productivity** (but not raw speed).



Golang vs C Speed Comparison

- **4. Compilation Speed**
- **C:** Slow for large projects (complex toolchains, headers, linking).
- **Go:** Designed for **very fast compilation** (great for microservices and CI/CD).



Golang vs C Speed Comparison

Task	C	Go
Raw math (loops, arrays)	<input checked="" type="checkbox"/> Fastest	~10–20% slower
String handling	Fast (manual)	Slightly slower, but safer
Concurrency (100k tasks)	Heavy (threads, more RAM)	<input checked="" type="checkbox"/> Goroutines scale easily
Memory usage	Very efficient	Higher (due to GC + runtime)
Compilation time	Slower on big projects	<input checked="" type="checkbox"/> Much faster



Golang vs C Speed Comparison

-  **When to Choose**
- **C** → When absolute maximum performance and full hardware control is required (OS kernels, embedded, drivers, real-time systems).
- **Go** → When you need **near-C speed** but with **simplicity, concurrency, safety, and developer productivity** (APIs, cloud-native apps, microservices, servers).



Golang vs C and C++

GO OS / GO ARCH	amd64	386	arm	arm64	ppc64le	ppc64	mips64le	mips64	mipsle
aix						X			
android	X	X	X	X					
darwin	X				X				
dragonfly	X								
freebsd	X	X	X						
illumos					X				
ios					X				
js									
linux	X	X	X	X	X	X	X	X	X
netbsd	X	X	X						
openbsd	X	X	X	X					
plan9	X	X	X						
solaris	X								
windows	X	X	X	X					



Golang

- Golang has many advantages over other programming languages.
 - Golang has an easy learning curve
 - Go codes are easily readable. Over and above, an easy-to-use tool
 - Golang has good documentation, which will support developers significantly.
 - It compiles and executes codes at speed
 - Golang is portable. It means that it will work on all platforms
 - It has automatic garbage collection
 - It performs independent error handling
 - Golang reduces runtime errors and dependencies.
 - It has good memory safety and management
 - Golang supports scalability
 - It has a large user community.



Golang vs Python

Aspect	Golang (Go)	Python
Type System	<input checked="" type="checkbox"/> Statically typed (type-checked at compile time).	<input checked="" type="checkbox"/> Dynamically typed (checked at runtime).
Speed	<input checked="" type="checkbox"/> Compiled → much faster, close to C.	<input checked="" type="checkbox"/> Interpreted → slower (5–30x slower than Go).
Concurrency	<input checked="" type="checkbox"/> First-class support (goroutines, channels).	<input checked="" type="checkbox"/> Limited (Global Interpreter Lock / GIL restricts true parallelism).
Ease of Learning	Simple, small spec, easy to learn.	Even simpler, beginner-friendly.
Ecosystem	Strong for cloud, APIs, microservices, DevOps tools .	<input checked="" type="checkbox"/> Huge ecosystem → AI, ML, Data Science, Web frameworks, scripting .
Use Cases	Cloud-native apps, APIs, microservices, networking, CLI tools.	Data science, machine learning, scripting, automation, web apps (Django/Flask).



Golang vs Python

Memory Management	Garbage collected, efficient.	Garbage collected, but slower.
Syntax	C-like, minimalistic.	Very readable, English-like.
Community	Large, growing fast (DevOps, cloud).	Massive (data science, academia, startups, enterprises).
Deployment	Produces a single binary , easy deployment (no runtime needed).	Needs Python runtime + dependencies (can be tricky in production).



Golang vs Python

- **Key Strengths**
- **Go**
- Fast, compiled, efficient concurrency → great for **microservices & cloud infrastructure**.
- Produces static binaries → easy to ship in Docker/K8s.
- Used by **Docker, Kubernetes, Terraform, Prometheus**.
- **Python**
- Huge ecosystem for **AI/ML, data science, scientific computing**.
- Super beginner-friendly.
- Widely used in **automation, scripting, web apps, prototyping**.



Where To Use

Domain	Best Choice
Microservices, APIs, Cloud, DevOps tools	Go
Machine Learning, AI, Data Science, Scripting	Python
General-purpose scripting & automation	Python
High-performance concurrent servers	Go



Golang vs Java

Java

- Object-oriented language
- Virtual machine
- Bigger community
- Slower

Go

- Supports concurrency
- No error handling
- Easier to read
- Faster

Java & Go

- Server-side programs
- C language family
- Uses garbage collector



Golang vs Java

Aspect	Golang (Go)	Java
Type System	<input checked="" type="checkbox"/> Statically typed, simple	<input checked="" type="checkbox"/> Statically typed, object-oriented
Performance	<input checked="" type="checkbox"/> Compiled to native machine code → very fast, low overhead	Runs on JVM , slightly slower but still high-performance with JIT optimizations
Concurrency	<input checked="" type="checkbox"/> Built-in with goroutines and channels (lightweight, scalable)	Threads, executors, and CompletableFuture; powerful but heavier
Memory Management	Garbage collected, lightweight runtime	Garbage collected (mature GC, multiple strategies)
Ease of Learning	Small language spec, easy for beginners	Rich and verbose, larger learning curve
Ecosystem	Strong for cloud-native, DevOps, microservices	<input checked="" type="checkbox"/> Massive ecosystem (enterprise apps, Android, Spring, Hadoop, banking/finance)



Golang vs Java

Tooling	Built-in: go build, go test, go fmt, go mod	<input checked="" type="checkbox"/> Mature ecosystem: Maven, Gradle, JUnit, IntelliJ, Spring Boot
Deployment	<input checked="" type="checkbox"/> Single binary, easy to deploy in containers	Requires JVM; container-friendly but heavier
Community	Growing (cloud-native, startups, infra tools)	<input checked="" type="checkbox"/> Huge, decades-old, enterprise-backed
Use Cases	Cloud apps, APIs, microservices, DevOps tools (Docker, K8s, Terraform)	Enterprise apps, Android apps, banking systems, large-scale backends



Golang vs Java

- **Key Strengths**
- **Go**
- Designed for **simplicity and concurrency**.
- Tiny runtime, fast startup, static binary → perfect for **containers & microservices**.
- Used in **Kubernetes, Docker, Prometheus, Terraform**.
- Great fit for **startups and modern cloud infrastructure**.
- **Java**
- Rich **enterprise ecosystem** (Spring Boot, Hibernate, Kafka, Spark).
- JVM portability → runs everywhere.
- Highly optimized for long-running enterprise apps.
- Used in **banks, telecom, big data, Android, large-scale enterprise software**.



Golang vs Java - When to Use

Domain	Best Choice
Cloud-native microservices, DevOps tools, APIs	Go
Enterprise software, banking, legacy systems	Java
Android apps	Java/Kotlin
Infrastructure (K8s, Docker, Terraform)	Go
Big Data (Spark, Hadoop, Kafka ecosystem)	Java

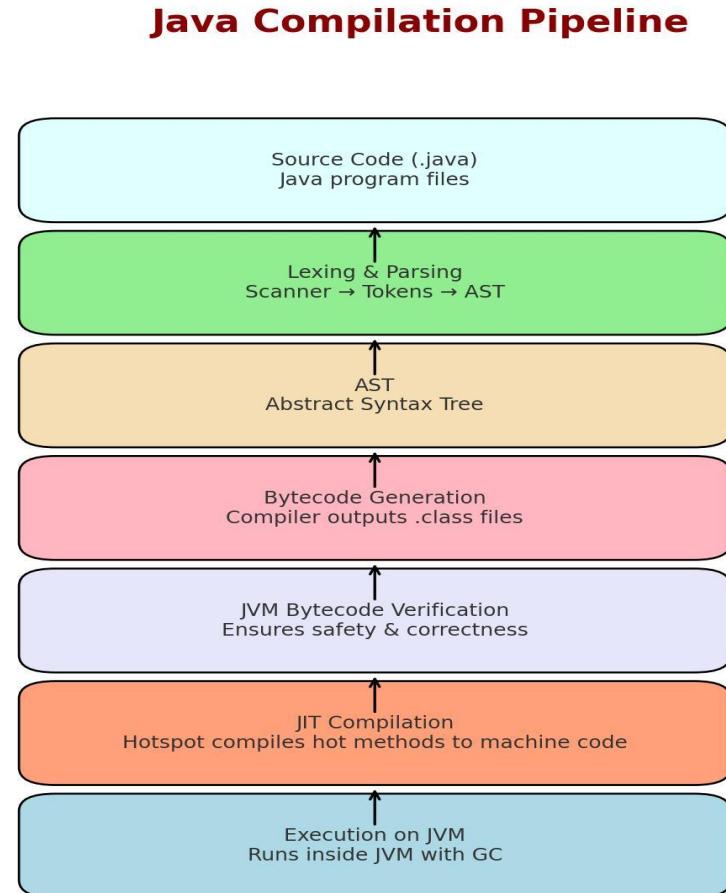
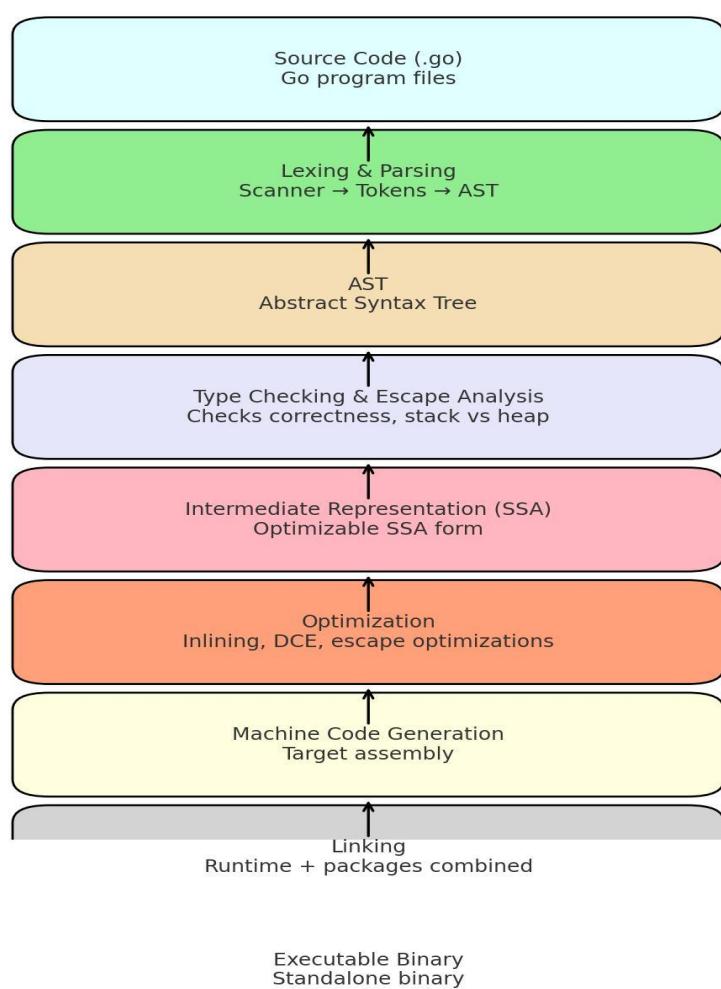


Golang vs Java

- **Verdict**
- **Go** → Simplicity, concurrency, cloud-native tooling.
Best for **modern microservices** and **DevOps platforms**.
- **Java** → Stability, ecosystem, enterprise adoption.
Best for **large-scale enterprise applications, finance, Android, and big data**.
- Many companies actually **use both**:
- **Go** for cloud infrastructure and high-performance APIs.
- **Java** for business logic-heavy enterprise systems.



Golang Compilation Model





Golang Compilation Process

⚡ Go Compilation Steps

1. Source Code (.go files)

- You write Go code in .go files.
- Organized into **packages** (each folder = a package).



Golang Compilation Process

- **2. Lexing (Tokenization)**
- The **scanner (lexer)** reads your code and breaks it into tokens: keywords, identifiers, literals, operators.
- Example:

```
go  
  
x := 42
```

→ Tokens: IDENT(x) , := , NUMBER(42)



Golang Compilation Process

- **3. Parsing**
- Tokens are structured into an **AST (Abstract Syntax Tree)**.
- AST represents the program in tree form (functions, variables, control flow).



Golang Compilation Process

- **4. Type Checking & Escape Analysis**
- Compiler checks types (static typing).
- **Escape Analysis** decides:
 - If variable stays on **stack** (fast, no GC).
 - Or must be placed on **heap** (managed by GC).



Golang Compilation Process

- **5. Intermediate Representation (SSA)**
- Go compiler converts code into **Static Single Assignment (SSA)** form.
- Easier to optimize: constant folding, inlining, dead-code elimination, etc.



Golang Compilation Process

- **6. Optimization**
- Performs code optimizations:
 - Inline functions
 - Remove dead code
 - Optimize memory allocations (stack vs heap)



Golang Compilation Process

- **7. Machine Code Generation**
- Compiler backend generates **architecture-specific assembly** (x86, ARM, etc.).
- Uses **Plan 9 assembler syntax** internally.



Golang Compilation Process

- **8. Linking**
- Links:
 - Your program's machine code
 - Go runtime (scheduler, GC, etc.)
 - Standard library packages
- Produces a **single executable binary** (static by default).



Golang Compilation Process

- **9. Executable Binary**
- Final output = **self-contained binary** (no JVM or interpreter).
- Easy to deploy (great for Docker, Kubernetes).



Golang Compilation Process

- Example

```
bash
```

```
go build main.go
```

Produces `main` (Linux/macOS) or `main.exe` (Windows).

```
bash
```

```
./main
```

Runs directly — no external runtime needed.



Golang Compilation Process

- Example

```
bash
```

```
go build main.go
```

Produces `main` (Linux/macOS) or `main.exe` (Windows).

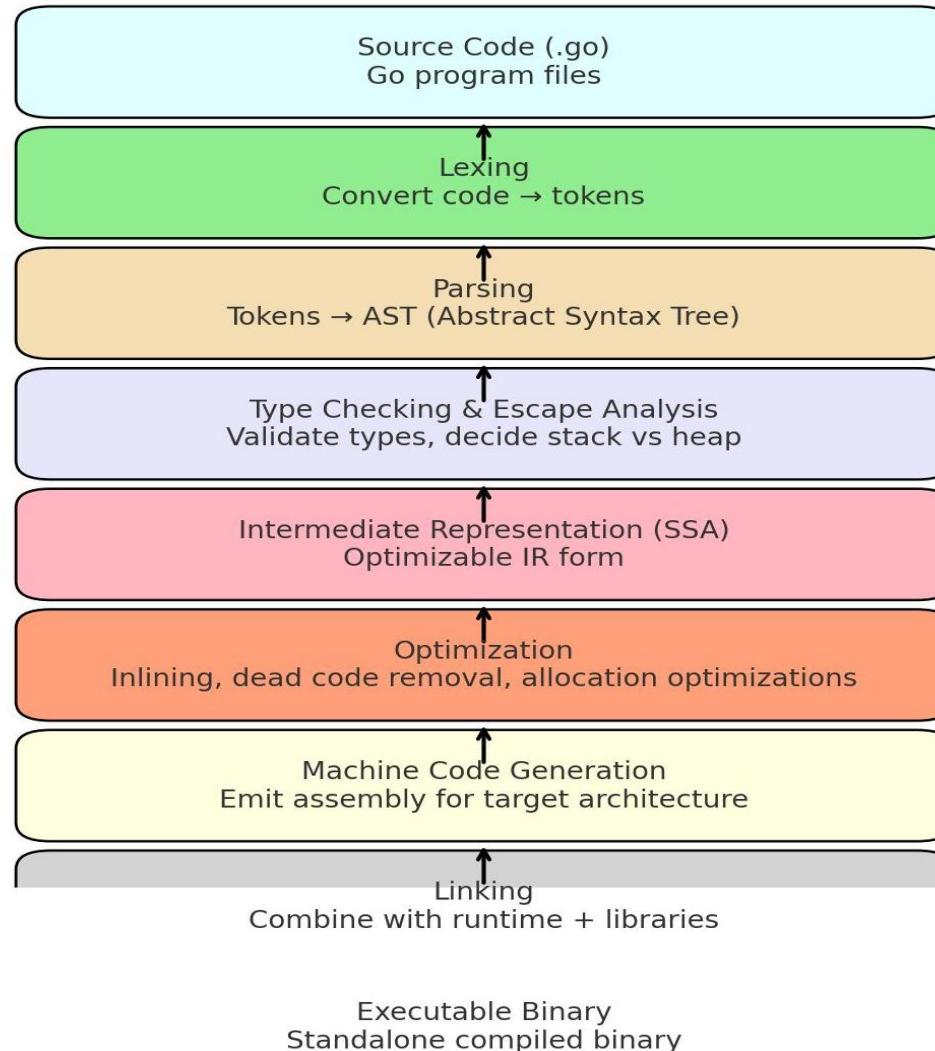
```
bash
```

```
./main
```

Runs directly — no external runtime needed.

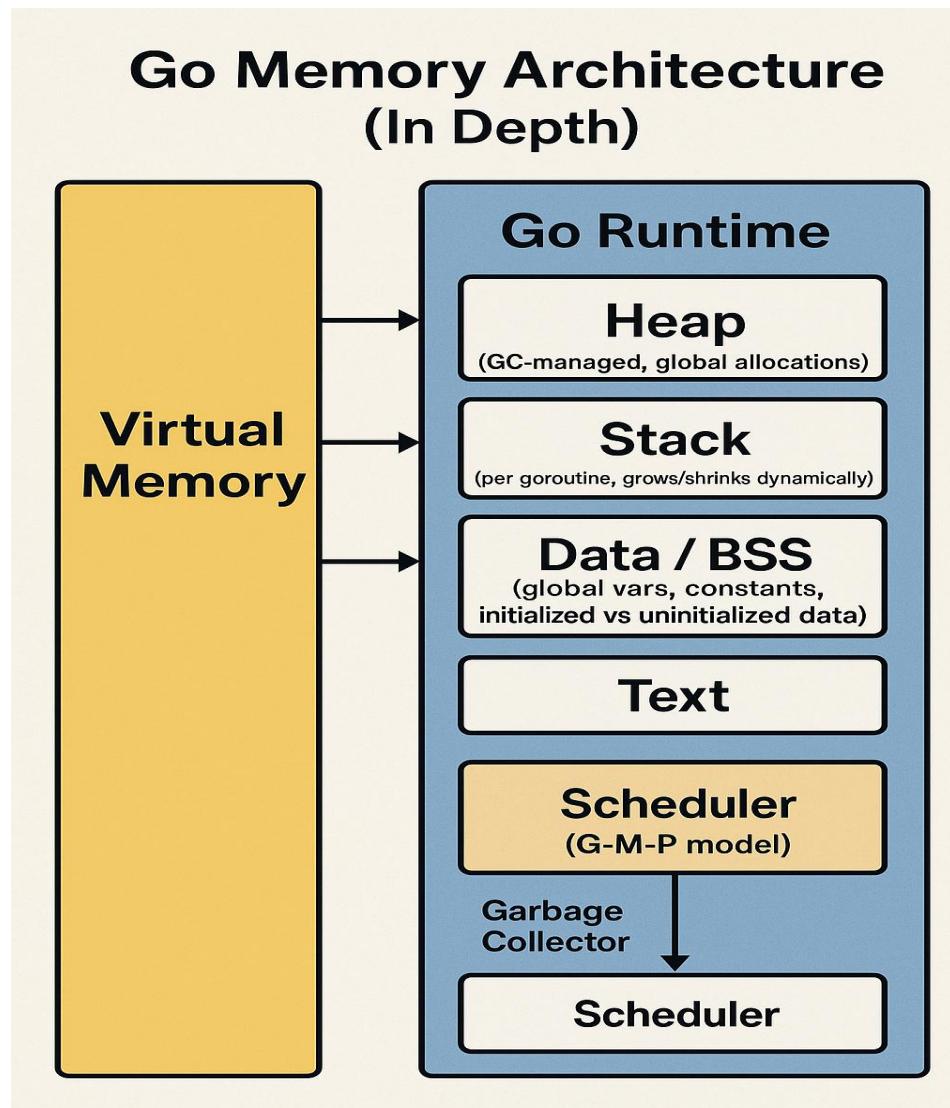


Golang Compilation Process





Go Memory Architecture





Go Memory Architecture

- 🧠 **Go Memory Architecture (Internal)**
- At a high level, Go's memory system is built around:
- **Heap** → for dynamically allocated objects.
- **Stack** → for function calls and local variables.
- **Garbage Collector (GC)** → manages heap, reclaims unused memory.
- **Allocator (mcache, mcentral, mheap)** → highly optimized for concurrency.

1 Stack



- Each **goroutine** has its **own stack**.
- Starts small (~2 KB) and **grows/shrinks dynamically** (unlike C, which is fixed ~1 MB per thread).
- Stores:
 - Local variables
 - Function call frames
 - Return addresses
- No GC overhead for stack memory (reclaimed automatically when a goroutine ends).



2. Heap

- Shared memory pool for long-lived objects (like slices, maps, interfaces, closures).
- Allocations happen when data **escape's** function scope → determined by **escape analysis**.

- Example:

```
go

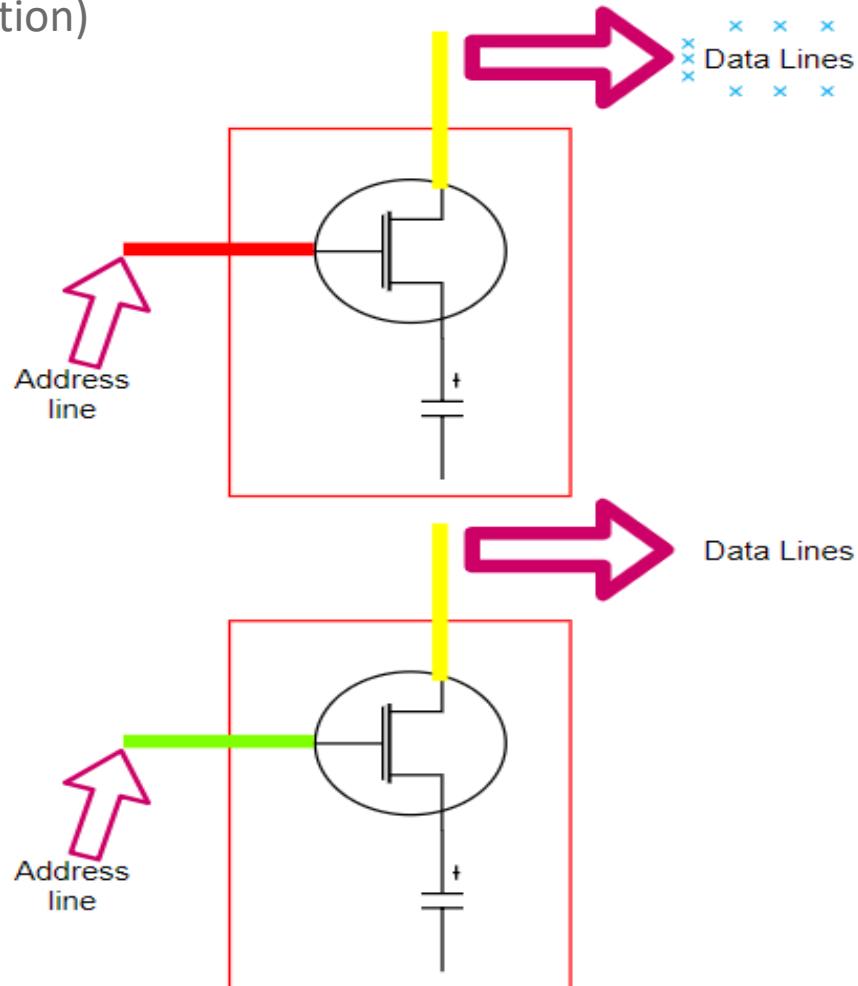
func foo() *int {
    x := 42
    return &x // escapes → heap
}
```

- Heap is divided into **spans (pages)** → optimized for small allocations.

Physical and Virtual Memory



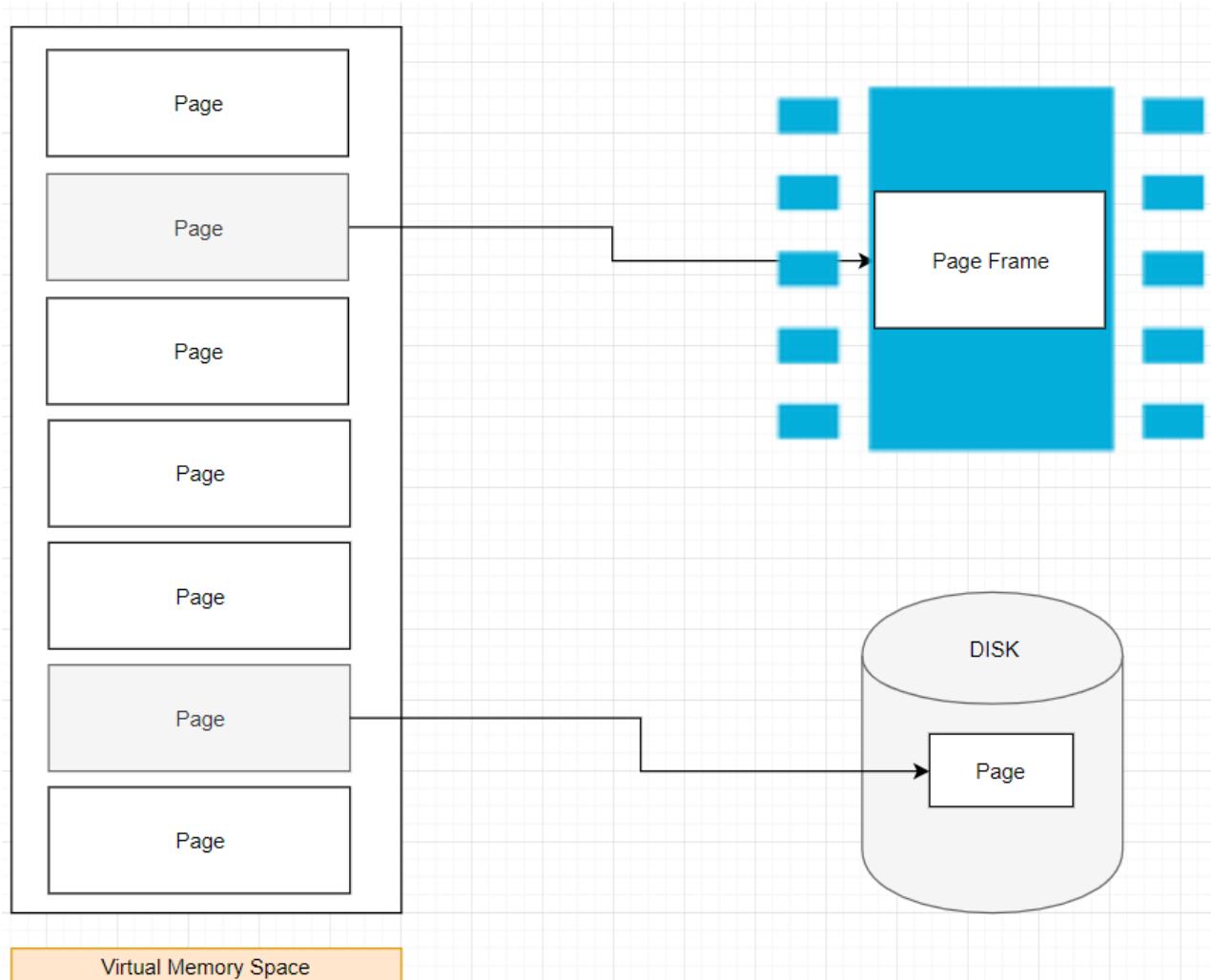
A simple illustration of a Physical Memory Cell (Not an exact representation)



Physical and Virtual Memory



As the size of physical RAM is limited, so Each Process runs in its own memory sandbox — “virtual address space,” known as **Virtual Memory**.





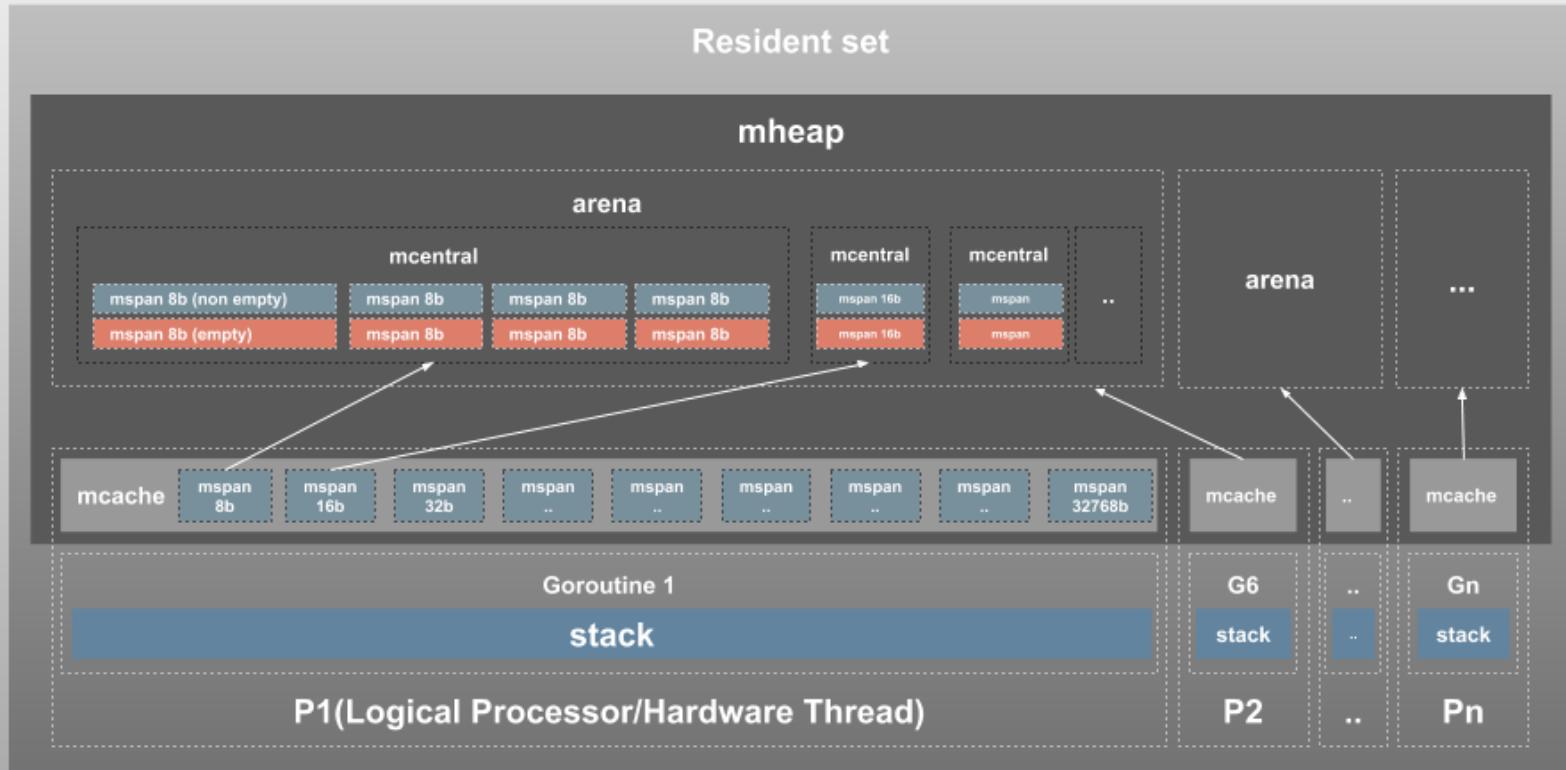
Resident Set

- Each Go program process is allocated some virtual memory by the Operating System(OS), this is the total memory that the process has access to.
- The actual memory that is used within the virtual memory is called **Resident Set**.

Process Virtual Memory



Process Virtual Memory



Process Virtual Memory



Page Heap (global pool of pages)



mheap (global heap manager)



mcentral (per size class)



mspan (metadata describing span)



mcache (per-P local cache of spans)

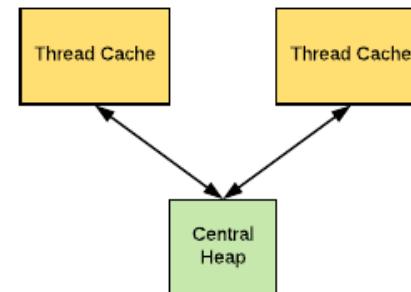


Goroutines



TMalloc

- **💡 What is tcmalloc?**
- **tcmalloc = Thread-Caching Malloc**, originally from Google.
- It's a **high-performance memory allocator** designed to reduce contention between threads when allocating and freeing memory.
- Go's runtime **memory allocator (mcache → mcentral → mheap)** is heavily inspired by tcmalloc's ideas.





Core Principles of tcmalloc

1. Per-thread caches (thread-local freelist)

- Each thread has a small cache of objects of various sizes.
- Allocation from this cache = **lock-free, O(1)**.
- Frees go back into the thread's cache first.

2. Size classes

- Objects are grouped into **size classes** (8B, 16B, 32B, ... up to a cutoff).
- Each cache entry corresponds to a size class.

3. Central free lists

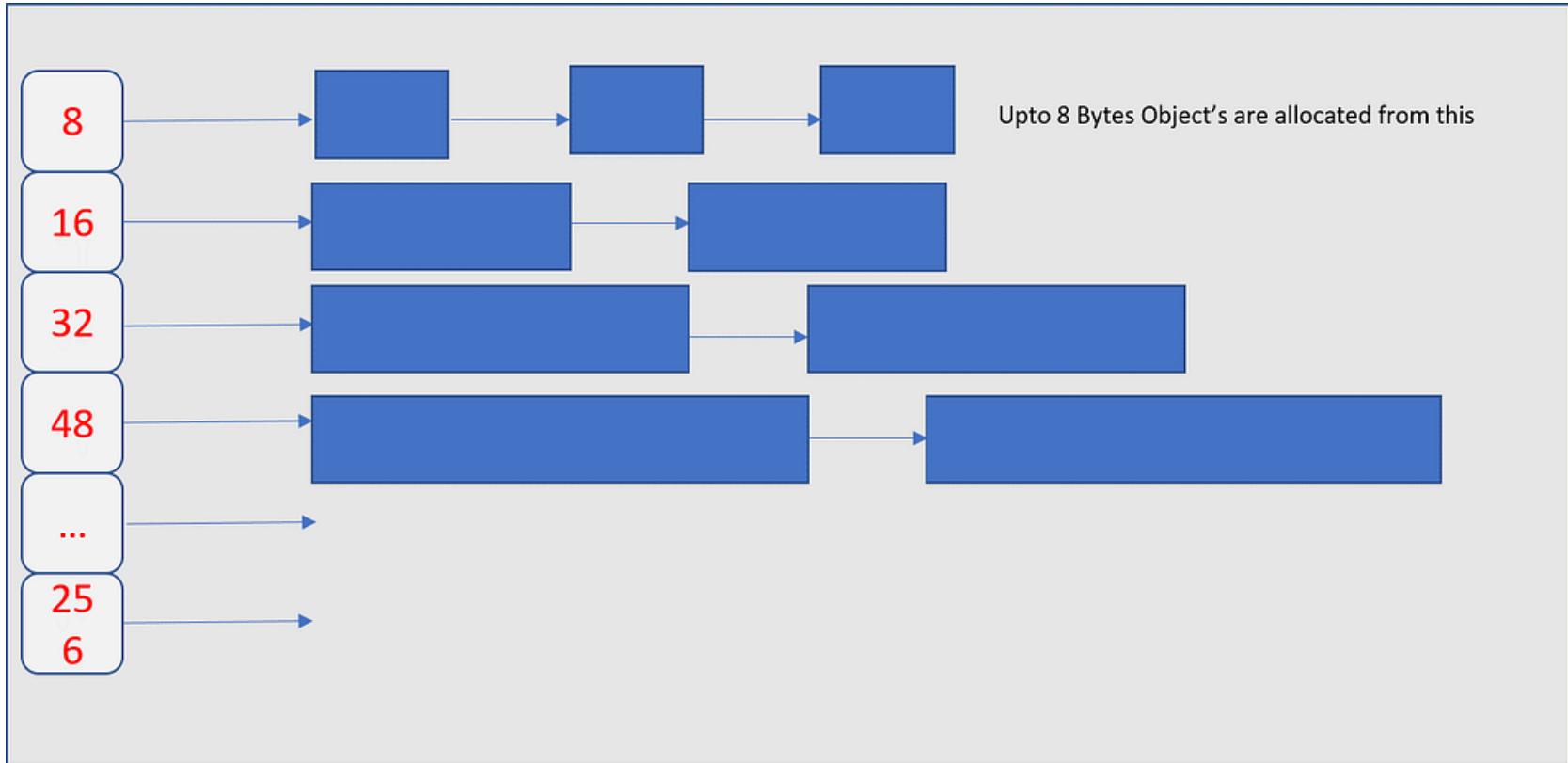
- When a thread's cache runs empty/full, it fetches/releases a **batch** of objects from/to a central free list.
- This reduces contention because lock acquisition happens **per batch**, not per allocation.

4. Page heap (global heap)

- Large allocations (e.g., >32KB) bypass caches and go directly to the global heap.
- The central free list itself gets memory from the **page heap**, which is managed in terms of pages (usually 8KB or multiples).



Core Principles of tcmalloc



Thread Cache (Each Thread gets this Thread Local Thread Cache)



Page Heap

- The heap managed by TCMalloc consists of a collection of pages, **where a set of consecutive pages can be represented by span**.
- When allocated Object is larger than 32K, Pages Heap is used for allocation.



Page Heap How it Works

1. Page as a unit

- Memory is managed in pages (e.g., 8 KB).
- Large objects take multiple contiguous pages.

2. Span allocation

- The page heap carves memory into **spans** (contiguous groups of pages).
- Each span is represented by an `mspan` (metadata struct).
- Spans are then subdivided into objects of the same **size class**.

3. Free lists

- Page heap maintains lists of **free spans** of various sizes.
- When an `mcentral` needs more memory, it asks the page heap for a span.

4. Large allocations

- If an allocation >32 KB, Go bypasses size classes and directly requests a **large span** from the page heap.

5. Garbage Collection interaction

- The page heap interacts with the GC:
 - GC marks/free memory → freed spans go back to the page heap.
 - Reused later by other size classes.



mheap

- **What is mheap?**
- **mheap = global heap manager** in Go's runtime.
- It manages the **entire heap space**, which is requested from the **Page Heap (OS memory)**.
- Acts as the **central authority** that hands out spans to **mcentral**.
- Works closely with the **garbage collector** to recycle freed memory



mheap

-  **Role of mheap**
- Allocates **spans** (contiguous groups of pages).
- Maintains lists of **free spans** by size.
- Handles **large object allocations** directly (bypassing size classes).
- Interacts with **GC** to reclaim unused spans.
- Supplies spans to **mcentral** when its free list runs empty.



mheap

mheap Lifecycle

1. Initialization

- At runtime start, mheap is initialized.
- Gets memory from the OS (via mmap/sbrk).

2. Span allocation

- When mcentral needs more spans, it requests from mheap.
- mheap finds a suitable free span (or requests more pages from OS).

3. Large objects (>32 KB)

- Allocated directly by mheap, bypassing mcentral/mcache.
- Uses multiple contiguous pages.

4. Garbage Collection

- During GC, spans are marked/swept.
- Freed spans are returned to mheap's free lists.



💡 What is mcentral?

- **mcentral** is a **per-size-class central cache of spans**.
- Each **size class** (e.g., 8B, 16B, 32B, ... up to 32KB) has its own mcentral.
- It sits between **mheap** (global heap) and **mcaches** (per-P local caches).
- Purpose: to hand out spans to mcache when they run empty.

-  **Role of mcentral**
- Manages **free spans** for a particular size class.
- Maintains two linked lists of spans:
 - **Empty list** → spans currently being used (some objects still allocated).
 - **Non-empty list** → spans with free objects available.
- Allocates spans to **mcache** in batches.
- Returns spans back to **mheap** when completely free.

⌚ How mcentral Works

1. mcache request

- When a goroutine needs memory, it asks its **mcache**.
- If mcache is empty → it requests a new span from mcentral.

2. Serving spans

- mcentral finds a span in its **non-empty list**.
- Hands the span over to the requesting mcache.

3. Replenishment from mheap

- If mcentral has no non-empty spans, it requests a fresh span from **mheap**.

4. Returning spans

- When a span is completely freed (all objects unused), mcentral returns it to mheap.



mspan

- **What is mspan?**
- An **mspan** is the **metadata structure** that describes a **span** (a contiguous chunk of memory from the heap).
- Spans are the unit of allocation inside Go's runtime.
- Every span is made up of one or more **pages** (default page = 8 KB).



mspan



Lifecycle of an mspan

- 1. Created by mheap** → when mcentral requests memory.
- 2. Assigned to size class** → span is subdivided into objects of fixed size.
- 3. Used by mcache** → goroutines allocate objects from this span.
- 4. GC interaction** → gcmarkBits used to mark live objects.
- 5. Freed** → when all objects are free, span is returned to mcentral (and possibly to mheap).



mspan

mspan





Arena

- **What is an Arena in Go?**
- An **Arena** is a **large contiguous region of virtual memory** reserved by the Go runtime to serve as the heap.
- The heap is divided into **arenas**, and each arena is further divided into **pages** (default: 8 KB).
- Arenas are managed by the **mheap**, which carves them into spans for allocation.
- You can think of arenas as **big chunks of memory** grabbed from the OS that Go slices into smaller pieces.



Arena

Arena Characteristics

- Size: typically, **64 MB** per arena on 64-bit systems.
- Alignment: arenas are aligned in memory for efficient indexing.
- Indexed by a global **heapArena array**.
- Each arena is subdivided into:
 - **Pages** (8 KB each).
 - **Spans** (group of one or more pages, tracked by an `mspan`).



Arena

-  **Arena Lifecycle**
- **Heap Growth**
 - If mheap cannot satisfy a request, it asks the OS for more memory.
 - A new **arena** is mapped into the process address space.
- **Span Carving**
 - mheap divides arenas into spans and assigns them to size classes.
- **Allocation**
 - Goroutines allocate from mcache → spans → pages inside an arena.
- **Garbage Collection**
 - GC reclaims unused objects.
 - Freed spans can be reused within the same arena.



Arena in the Allocation Hierarchy

OS (mmap/sbrk)



Arena (64 MB)



Pages (8 KB each)



Spans (1+ pages, tracked by mspan)



mcentral / mcache



Goroutines



mcache

- **What is mcache?**
- **mcache = per-P (per-processor) memory cache** in Go's allocator.
- Each **P** (from the Go scheduler's **G-M-P model**) owns one **mcache**.
- It's where **goroutines actually allocate memory from** most of the time.
- Designed to be **lock-free, fast, and local**, avoiding global contention.



mcache

Role of mcache

- Stores **spans** (from **mcentral**), one span per size class.
- Each span provides many objects of the same size.
- When a goroutine needs memory:
 1. If a span for that size class exists in **mcache** and has free slots → allocate immediately (fast path).
 2. If span is exhausted → refill from **mcentral** (slower path).



mcache

-  **How mcache Works**
- **Fast Allocation**
 - Goroutine allocates memory.
 - mcache picks a slot from its current span → O(1), no locks.
- **Refill (Slow Path)**
 - If current span is exhausted:
 - mcache requests a new span from **mcentral**.
 - If mcentral is empty → mcentral requests from **mheap**.
- **Return Spans**
 - When a span is fully freed (all objects unused), it's returned to **mcentral**.



3. Hierarchy

- **3 Allocator Hierarchy**
- Go uses a **hierarchical memory allocator** inspired by **tcmalloc**:
- **mheap** → Global heap, managed by GC.
- **mcentral** → Manages spans (groups of objects of same size class).
- **mcache (per-P cache)** → Thread-local cache for each processor (P in Go scheduler).
 - Reduces lock contention by letting goroutines allocate from local caches.



Go Internal Memory Allocator Concepts

1 mheap

- The **global heap** managed by the Go runtime.
 - Represents the **entire memory region** used for dynamic allocations.
 - Divided into **spans** (continuous memory blocks).
 - Managed by the **garbage collector (GC)**.
 - Central allocator of memory, but expensive to access directly (locks required).
- 👉 Rarely allocated from directly. Instead, goroutines use **mcache** (per P-thread cache) which pulls memory from mcentral → mheap.



Go Internal Memory Allocator Concepts

2 mspan

- An mspan is a **descriptor (metadata)** for a block of memory in the heap.
- Each span represents a **contiguous region of memory** divided into objects of the same size class.
- Keeps track of:
 - Which objects are allocated vs free (bitmap).
 - The **size class** (e.g., 8B, 16B, 32B, ...).
 - Links to free lists for fast allocation.
- Helps the allocator quickly find free space for an object of a given size.



Go Internal Memory Allocator Concepts

3 span

- A **unit of allocation** on the heap.
- Backed by an `mspan` (the metadata).
- Groups together multiple objects of the same size.
- Allocations smaller than 32KB come from spans, sized by **size classes**.
- For large objects (>32KB), Go allocates a dedicated span directly from `mheap`.



Go Internal Memory Allocator Concepts

Goroutine → mcache (local, per-P) → mcentral (shared per size class) → mheap (global)

- **mcache:** per-processor cache, lock-free, fastest allocation.
- **mcentral:** manages a pool of mspans for a given size class, shared across processors.
- **mheap:** global allocator, fallback when mcentral is empty.



Go memory usage (Stack vs Heap)

- Main function is kept in a “main frame” on the Stack
- Every function call is added to the stack memory as a frame-block
- All static variables including arguments and the return value is saved within the function frame-block on the Stack
- All static values regardless of type are stored directly on the Stack. This applies to global scope as well
- All dynamic types are created on the Heap and is referenced from the Stack using Stack pointers. Objects of size less than 32Kb go to the mcache of the P. This applies to global scope as well

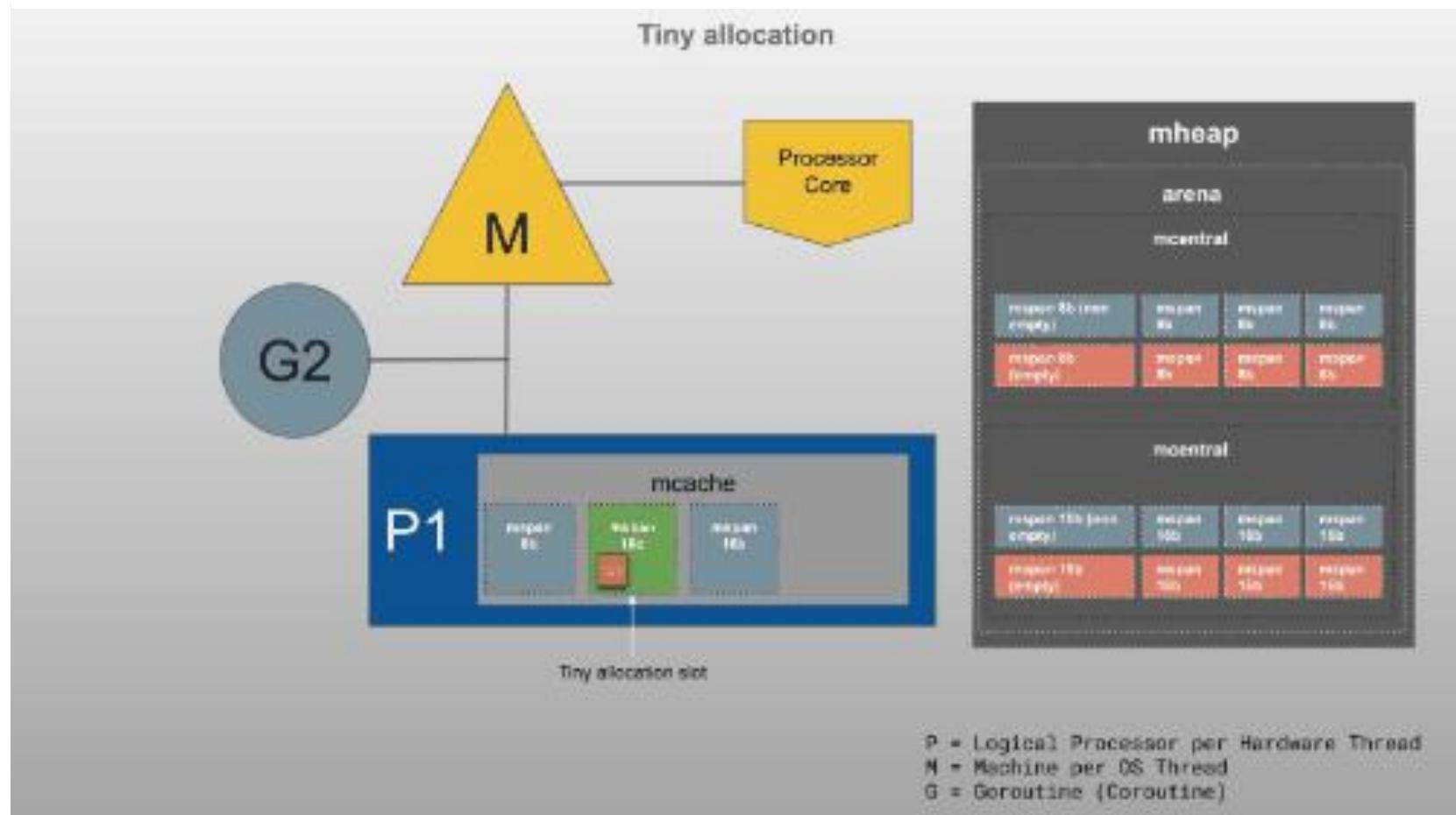


Go memory usage (Stack vs Heap)

- The struct with static data is kept on the stack until any dynamic value is added at that point the struct is moved to the heap
- Functions called from the current function is pushed on top of the Stack
- When a function returns its frame is removed from the Stack
- Once the main process is complete, the objects on the Heap do not have any more pointers from Stack and becomes orphan

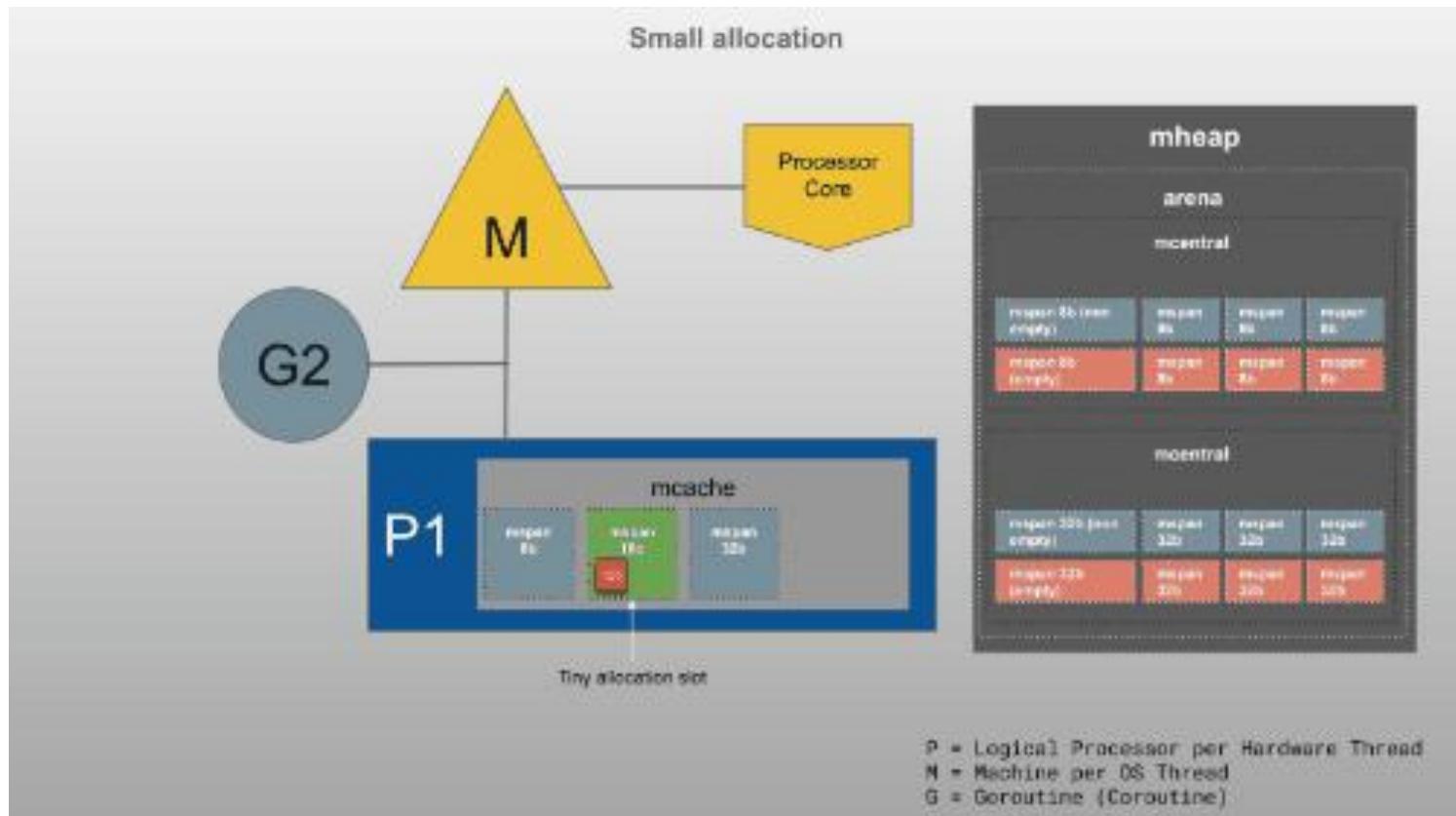
Memory Allocation

Tiny(size < 16B): Objects of size less than 16 bytes are allocated using the `mcache`'s tiny allocator. This is efficient and multiple tiny allocations are done on a single 16-byte block.



Memory Allocation

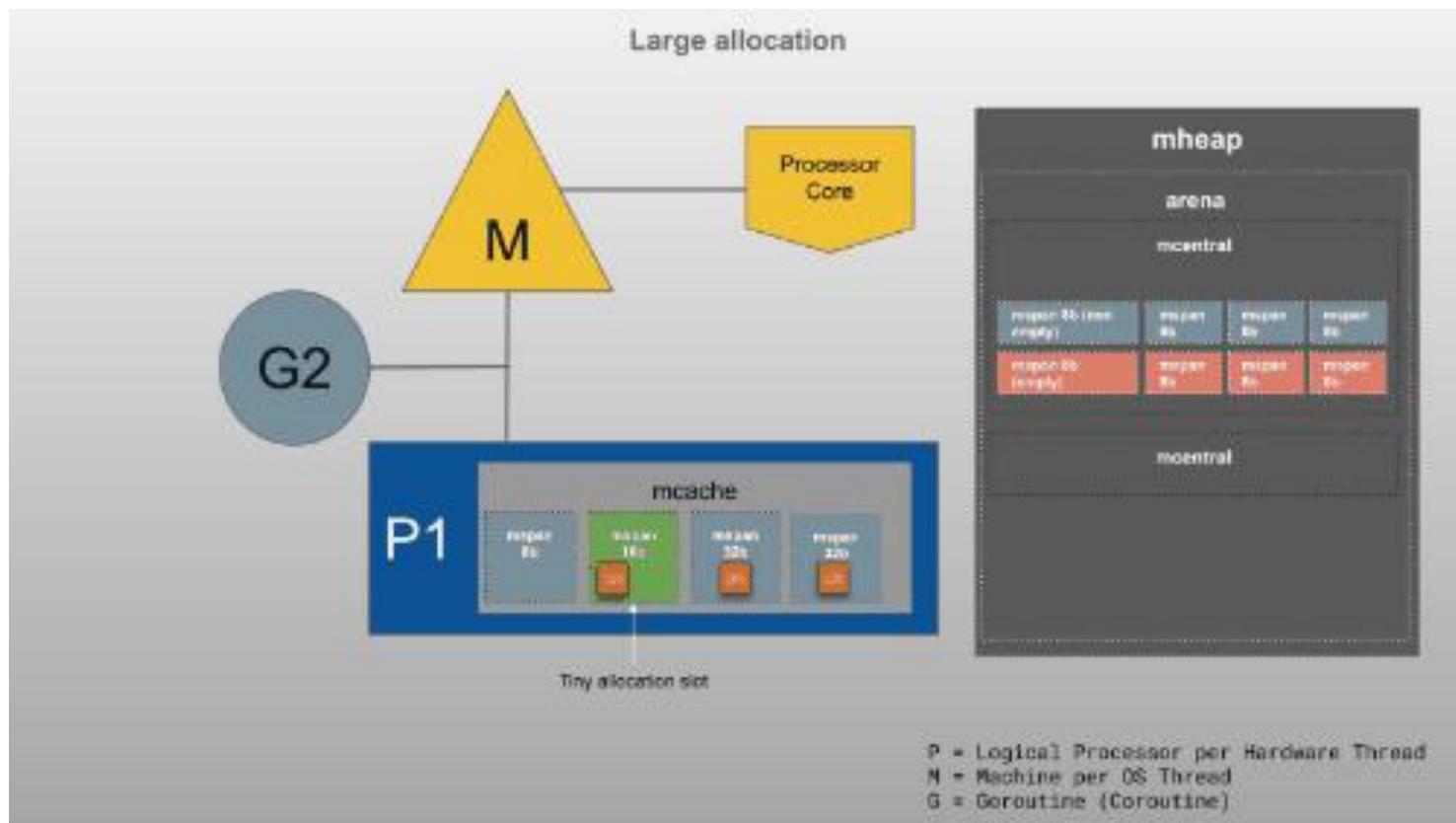
Small(size 16B ~ 32KB): Objects of size between 16 bytes and 32 Kilobytes are allocated on the corresponding size class(**mspan**) on **mcache** of the **P** where the **G** is running.





Memory Allocation

Large(size > 32KB): Objects of size greater than 32 kilobytes are allocated directly on the corresponding size class of `mheap`. If the `mheap` is empty or has no page runs large enough then it allocates a new group of pages (at least 1MB) from the OS.





4. Garbage Collector (GC)

-  **Garbage Collector (GC)**
- Go uses a **non-generational, concurrent, tri-color, mark-and-sweep GC**.
- Improvements in Go 1.5+ made GC **low-latency ($\sim<1\text{ms}$ pauses)**.
- Process:
 - **Mark phase** → traverse object graph, mark reachable objects.
 - **Sweep phase** → free unmarked objects.
 - Runs concurrently with goroutines (not fully stop-the-world).
- Modern Go GC is tuned for **low latency ($<100\ \mu\text{s}$ per MB allocated)** rather than max throughput.



GC

- **Mark Setup (Stop the world):** When GC starts, the collector turns on write barriers so that data integrity can be maintained during the next concurrent phase.
- This step needs a very small pause as every running Goroutine is paused to enable this and then continues.



5. Analysis

- **5 Escape Analysis**
- Compiler decides if a variable lives on **stack** (fast, no GC) or **heap** (needs GC).
- Example:

```
go

func main() {
    x := 42
    fmt.Println(x) // stays on stack
}
```

- If a variable **escapes** function scope → allocated on heap.
- Run with:

```
bash

go build -gcflags=-m main.go
```



6. Layout

- **6 Memory Layout in Process**
- **A Go program's memory layout typically looks like this:**





7. Concurrency & Memory

- **7 Concurrency & Memory**
- Each goroutine's stack is independent → no data races unless explicitly shared.
- Shared data (maps, slices, channels) lives on heap → needs synchronization (mutexes, channels).
- Scheduler (M:N model) + allocator design makes **millions of goroutines possible**.



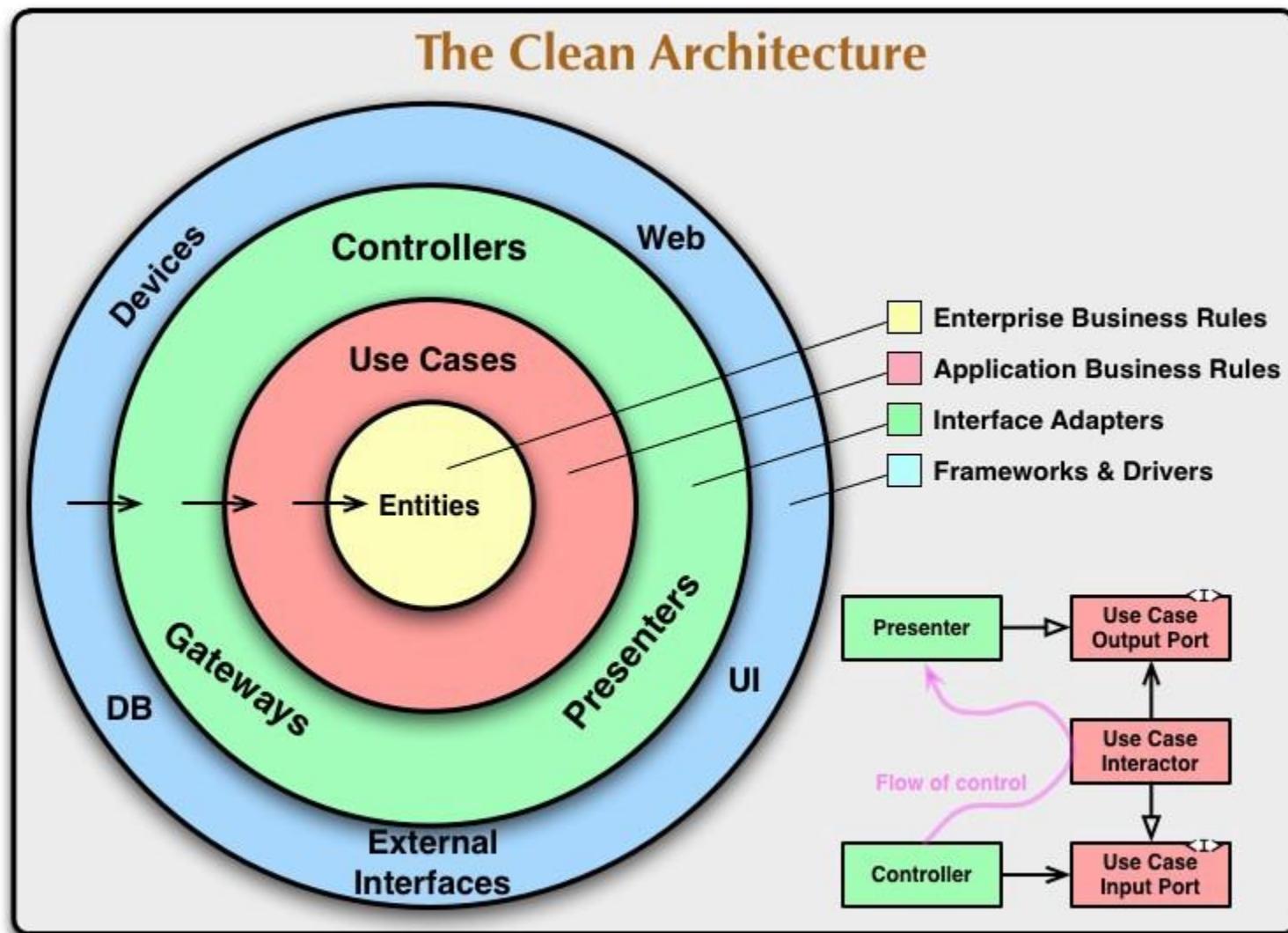
Key Takeaways

- **Stack = fast, auto-managed per goroutine.**
- **Heap = global, GC-managed for long-lived/shared data.**
- **Allocator = mcache/mcentral/mheap** for low-latency allocations.
- **GC = concurrent, tri-color, low-pause** → keeps latency low.
- **Escape Analysis** decides heap vs stack allocation.



What is Clean Architecture?

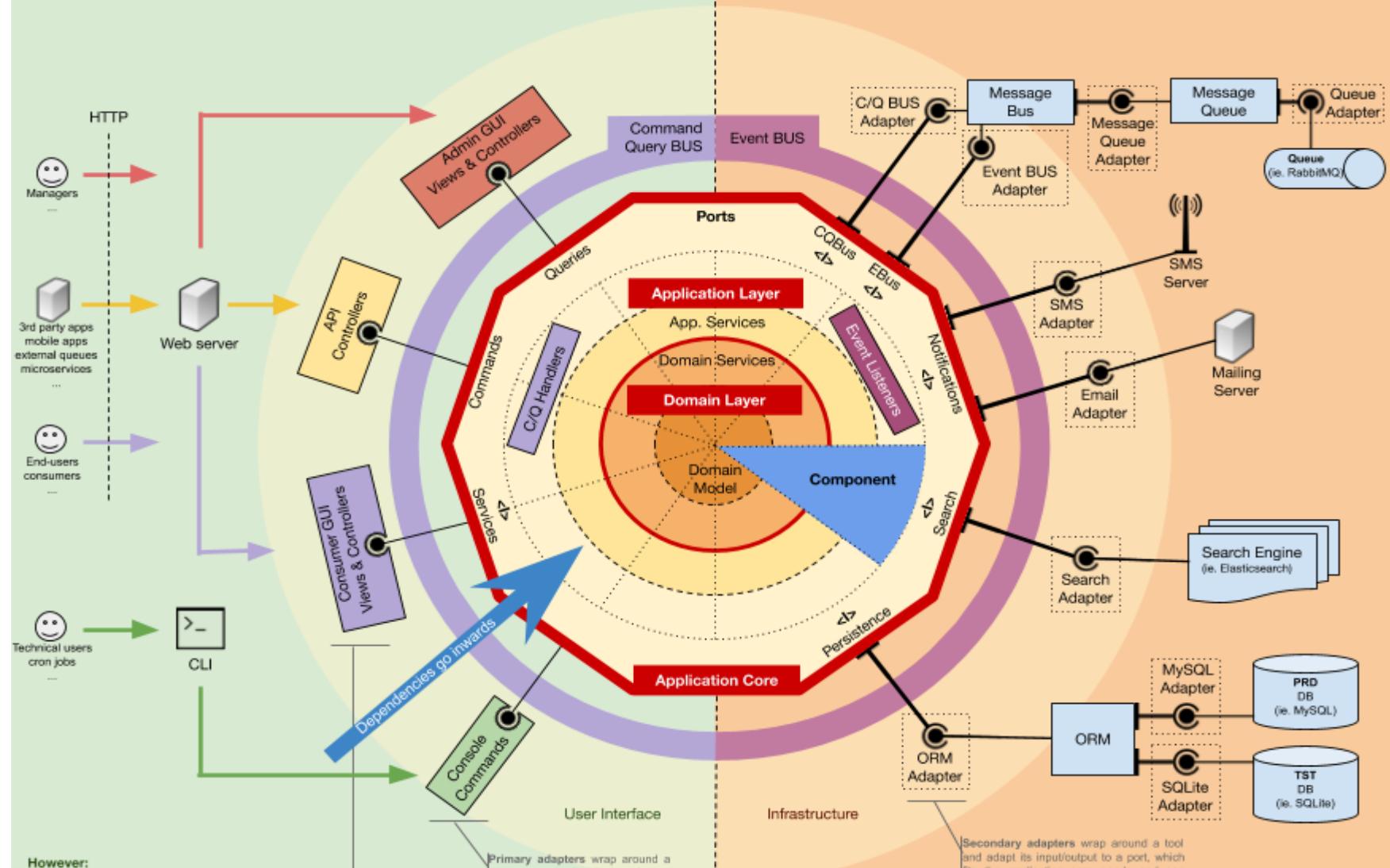
- Popularized by Robert C. Martin (Uncle Bob).
- Organizes code into **layers with clear boundaries**.
- Rule: **Inner layers shouldn't depend on outer layers.**
- Goal: keep **business rules (core logic)** independent from details (DB, HTTP, frameworks).



Explicit Architecture

Primary/Driving Adapters

Secondary/Driven Adapters





Go Questions

How do we run the code in our project?

What does '*package main*' mean?

What does '*import "fmt"*' mean?

What's that '*func*' thing?

How is the `main.go` file organized?



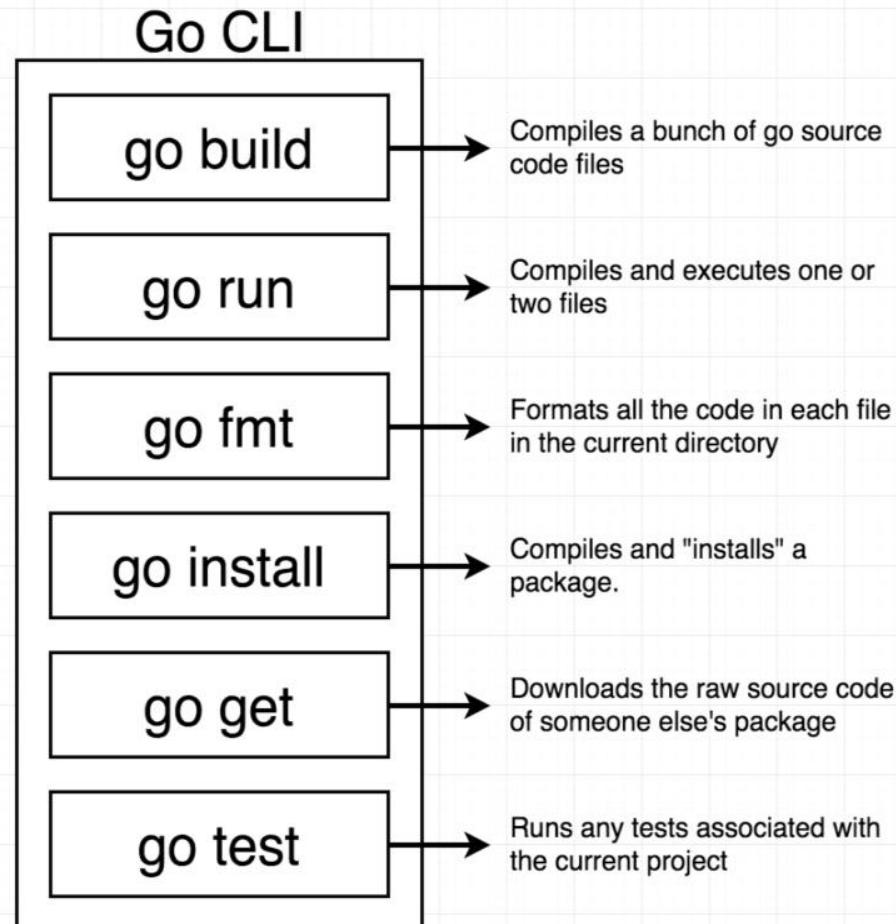
Go Order Of Invocation





Go Compile and Run

How do we run the code in our project?





What is a Go Module?

📦 What is a Go Module?

- A **module** is a collection of Go packages stored in a file tree with a go.mod file at the root.
- The go.mod file defines:
 - The module path (usually a repo URL or project name).
 - The Go version.
 - The dependencies and their versions.

Since Go 1.16, modules are the **default** (replacing GOPATH-based workflow).



🛠 Step 1: Create a new module

- 🛠 Step 1: Create a new module

Inside your project folder:

```
bash

mkdir myapp
cd myapp
go mod init myapp
```

This creates a `go.mod` file:

```
go

module myapp

go 1.22
```



🛠 Step 2: Add code and local packages

- In Go, a package is a collection of source files in the same directory that are compiled together.
- Every Go program is made up of packages.
- The entry point is always the main package (with a `main()` function).

📁 Package structure example

```
go

myapp/
    └── go.mod
    └── main.go          // package main
        └── greetings/
            └── greetings.go // package greetings
```



Go Packages

The screenshot shows a file explorer window with the following structure:

- Root folder: GOLANGTESTWS
 - day0
 - welcome.go
 - util
 - getotp.go
 - go.mod

The screenshot shows a code editor with the welcome.go file open. The code is as follows:

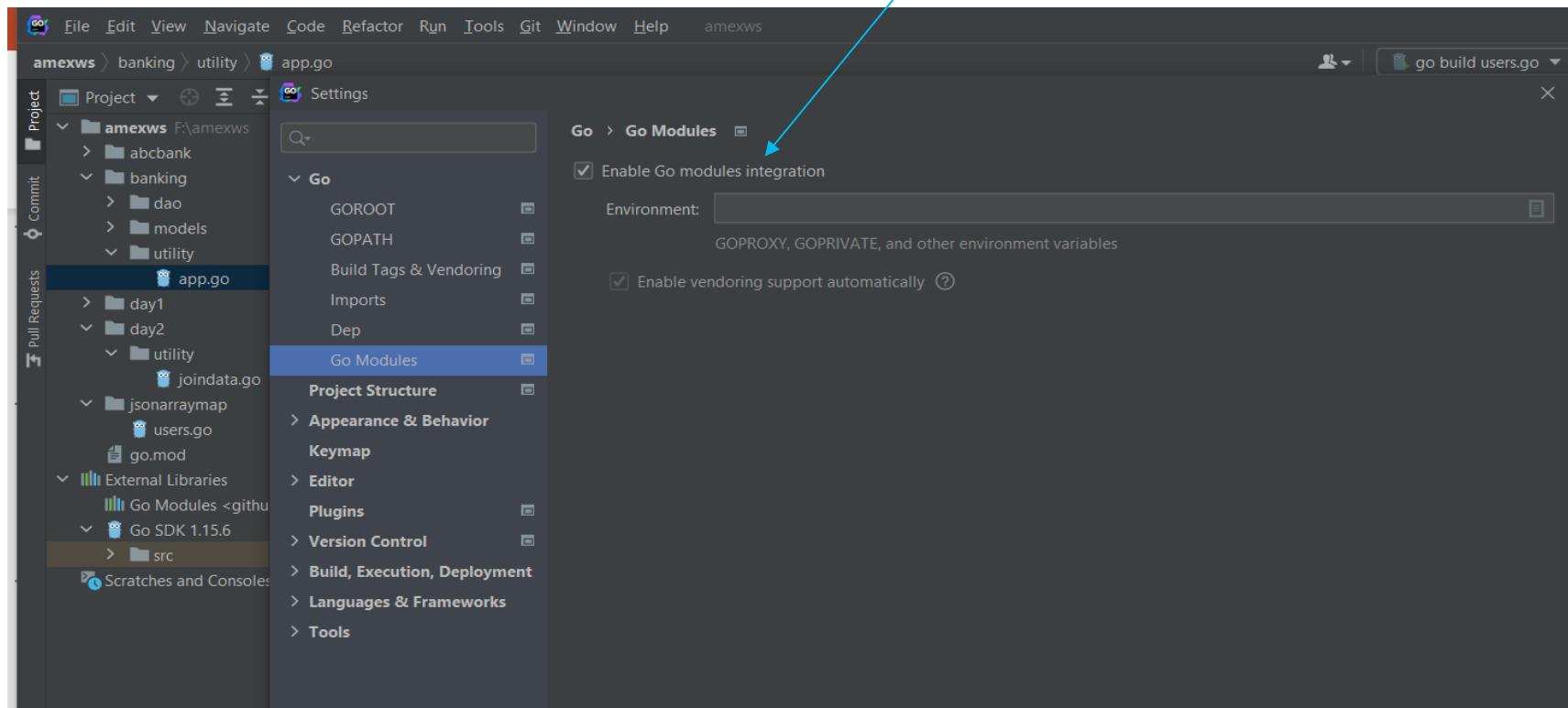
```
day0 > -GO welcome.go > ...
1 package main
2
3 import (
4     "fmt"
5     "github.com/eswaribala/golangtestws/util"
6 )
7
8 func main() {
9     otp := util.GetOTP()
10    fmt.Println("Your OTP is:", otp)
11 }
12
```

Go mod init github.com/eswaribala/golangtestws
#from root go to day0 folder
Go run .



Go mod in Goland

- Go mod init github.com/amexws/abcbank
- Set File settings Go mod preferences enable integration





Key rules about packages

🔑 Key rules about packages

1. Package name = directory name (usually).

- All files in a directory must have the same package statement.

2. Exported vs unexported:

- Identifiers starting with an **uppercase letter** are exported (public).
- Lowercase = private to the package.

3. Main package:

- package main with a func main() is the entry point.

4. Imports:

- Standard library: import "fmt"
- External: import "github.com/google/uuid"
- Local: relative to module name in go.mod.



What does package main mean

What does
'package main'
mean?

Package Main

main.go

```
package main

import"fmt"

funcmain() {
    fmt.Println("Hi
there!")
}
```

support.go

```
package main

func support() {
    fmt.Println("I help!")
}
```

helper.go

```
package main

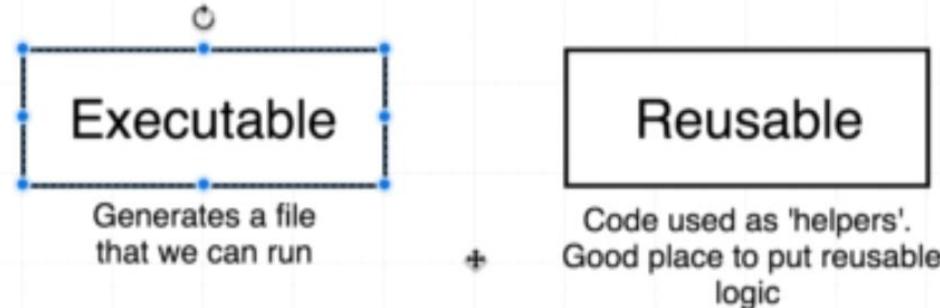
func help() {
    fmt.Println("I help too")
}
```



Types of packages

What does
'package main'
mean?

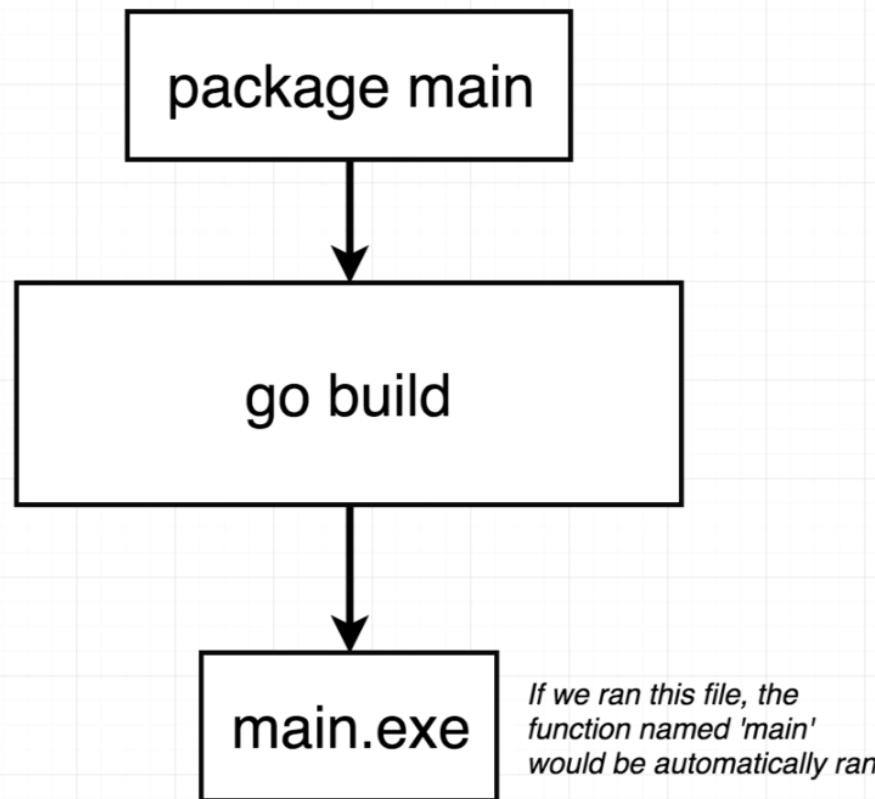
Types of Packages





Go Package Main

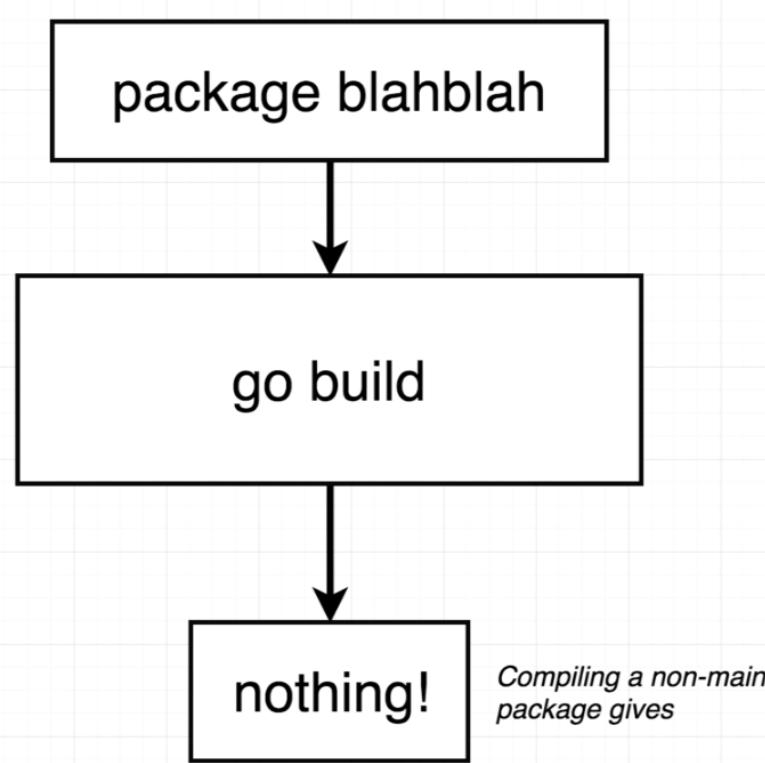
What does
'package main'
mean?





Go Packages

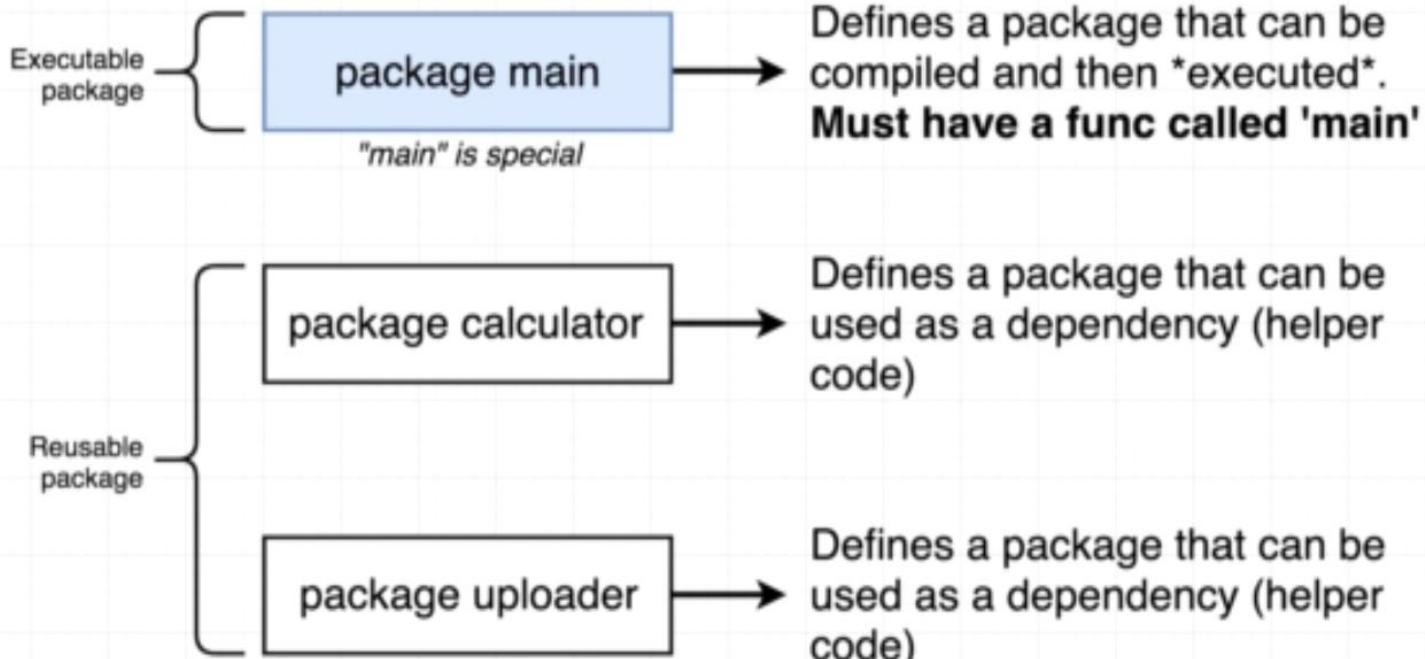
What does
'package main'
mean?





Multi Module GO

What does 'package main' mean?





External Packages

- go get github.com/bxcodec/faker/v3

A screenshot of a terminal window from a code editor. The window has tabs for PROBLEMS (1), OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined in blue), and PORTS. The status bar shows "powershell" and a dropdown menu icon. The terminal content shows the command "go get github.com/bxcodec/faker/v3" being run, followed by several lines of output indicating the download of multiple modules and a deprecation warning.

```
● PS E:\golangtestws> go get github.com/bxcodec/faker/v3
go: downloading github.com/bxcodec/faker/v3 v3.8.1
go: downloading github.com/bxcodec/faker v2.0.1+incompatible
go: module github.com/bxcodec/faker/v3 is deprecated: use github.com/go-faker/faker/v4 instead.
go: added github.com/bxcodec/faker/v3 v3.8.1
❖ PS E:\golangtestws>
```



External Packages

```
package main

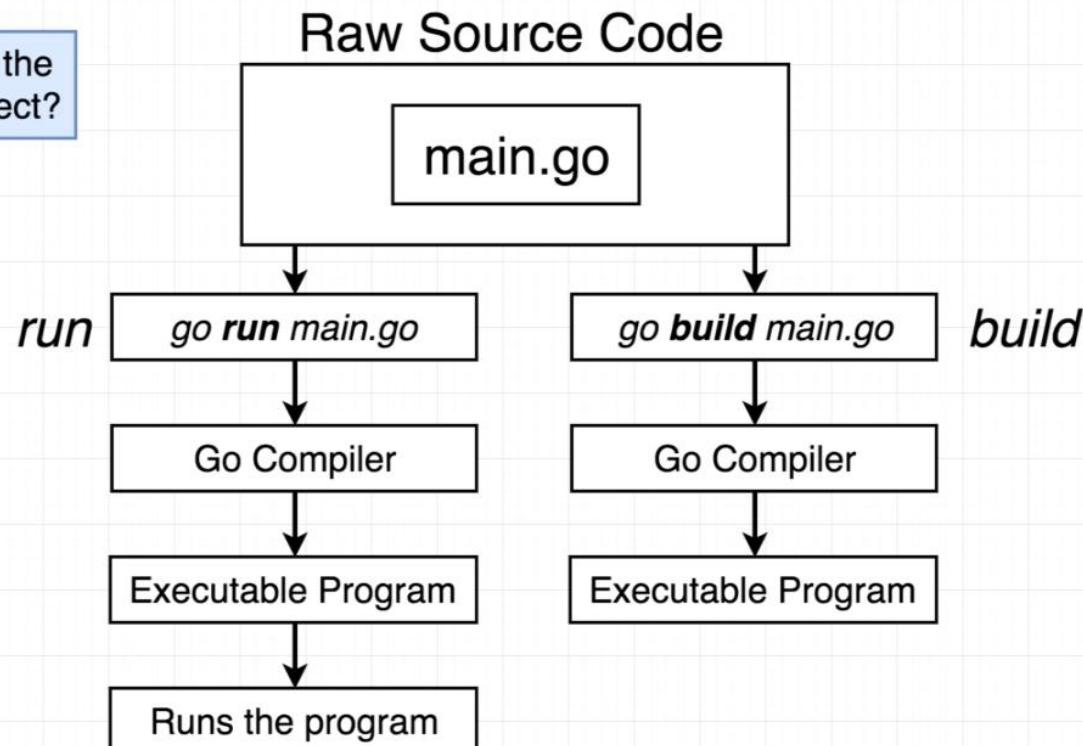
import (
    "fmt"
    "github.com/bxcodec/faker/v3"
)

func GenerateNamesArray() {
    var names [5]string
    for i.. := 0; i < len(names); i++ {
        names[i] = faker.Name()
    }
    fmt.Println(names)
}
```



Go Compilation Process

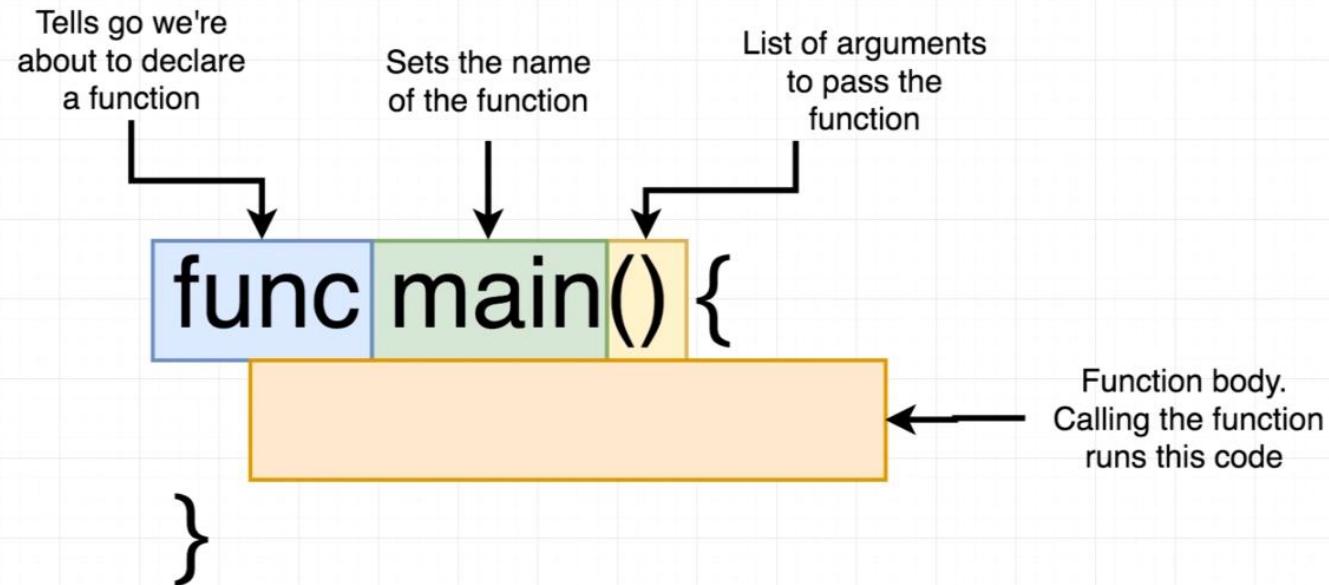
How do we run the code in our project?





Go Main Structure

What's that 'func' thing?



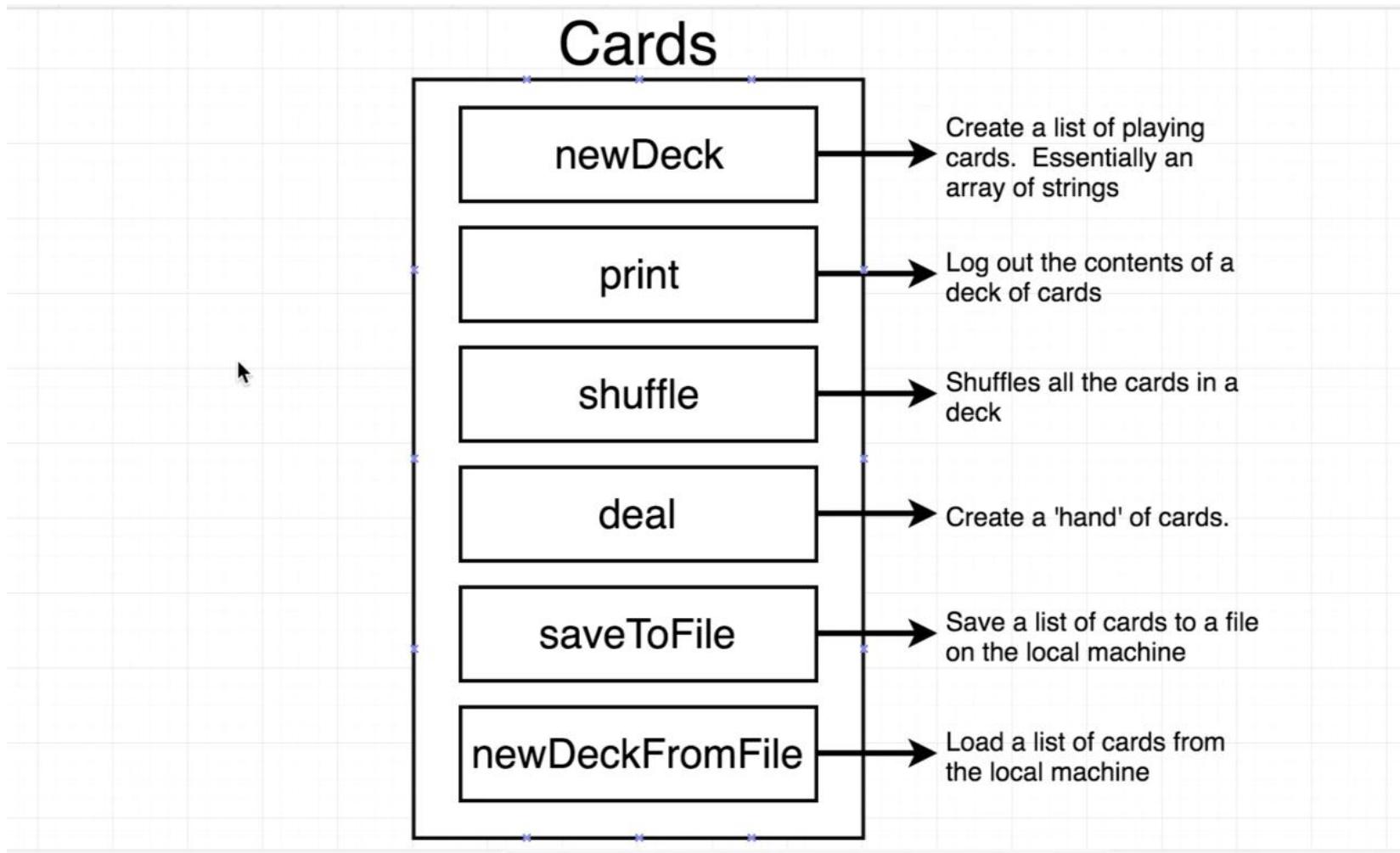


Go Main Structure

How is the main.go file organized?

package main	<i>Package declaration</i>
import "fmt"	<i>Import other packages that we need</i>
func main() { fmt.Println("hi there") }	<i>Declare functions, tell Go to do things</i>

Go Use case





Golang Structure

- Every Go Program Contains the following Parts
 - Declaration of Packages
 - Package Importing
 - Functions
 - Variables
 - Expression and Statements
 - Comments

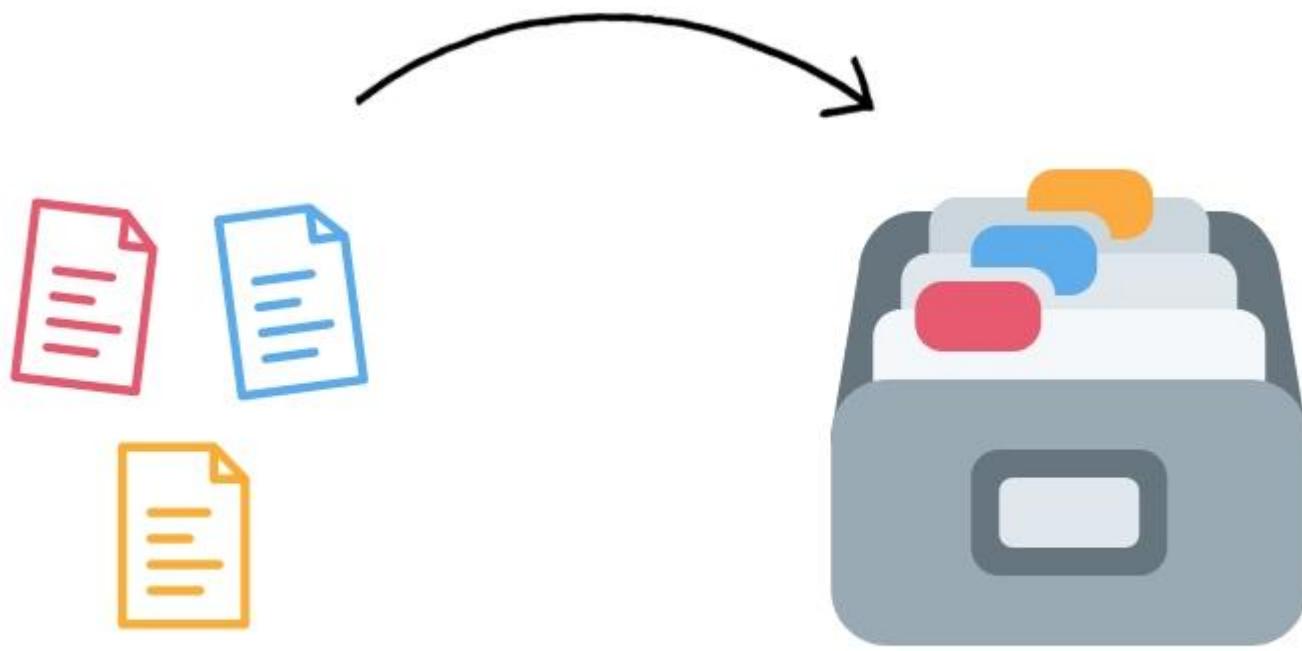


Golang Execution Steps

- Step1) Write the Go Program in a Text Editor.
- Step2) Save the program with “.go” as the program file extension.
- Step3) Go to command Prompt.
- Step4) In the command prompt, we have to open the directory where we have saved our Program.
- Step5) After opening the directory, we have to open our file and click enter to compile our code.
- Step6) If no errors are present in our code, then our program is executed, and the following output is displayed:



Packages



Go source files

Go Package



Packages – Standard Libraries

1. Core / Language Support

- `fmt` → formatted I/O (`fmt.Println`, `fmt.Sprintf`)
- `errors` → error handling (`errors.New`, `errors.Is`)
- `strconv` → string ↔ numeric conversions
- `strings` → string manipulation
- `unicode`, `unicode/utf8` → Unicode runes, UTF-8 encoding
- `reflect` → runtime reflection
- `unsafe` → low-level pointer operations



Packages – Standard Libraries

2. I/O and Files

- `os` → operating system interface (files, env vars, process)
- `io` → basic I/O primitives (`Reader`, `Writer`)
- `io/ioutil` → old helpers (deprecated in favor of `os` and `io`)
- `bufio` → buffered I/O
- `path`, `path/filepath` → path handling (Unix-style and OS-native)
- `embed` → embedding static files into binaries



Packages – Standard Libraries

3. Time and Dates

- `time` → time, duration, timers, tickers
 - `context` → request-scoped cancellation, deadlines
-

4. Math

- `math` → floating-point math
 - `math/rand` → pseudo-random numbers
 - `crypto/rand` → secure random numbers
 - `math/big` → arbitrary-precision integers and floats
 - `math/cmplx` → complex number math
-



Packages – Standard Libraries

5. Networking & Internet

- `net` → TCP/UDP sockets, DNS
- `net/http` → HTTP client/server
- `net/url` → URL parsing
- `net/mail` → parsing email headers
- `net/smtp` → sending email
- `html`, `html/template` → escaping, templating



Packages – Standard Libraries

6. Data Encoding / Decoding

- `encoding/json` → JSON encode/decode
 - `encoding/xml` → XML encode/decode
 - `encoding/csv` → CSV reader/writer
 - `encoding/base64` , `hex` , `binary` → encodings
 - `compress/gzip` , `zlib` , `flate` → compression
 - `archive/zip` , `tar` → archive formats
-



Packages – Standard Libraries

7. Cryptography & Security

- `crypto` → common interfaces
- `crypto/sha256`, `sha512`, `md5` → hashing
- `crypto/hmac` → HMAC signatures
- `crypto/aes`, `des`, `rsa`, `ecdsa`, `ed25519` → encryption/signatures
- `crypto/tls` → TLS (SSL)
- `crypto/x509` → certificates

8. Concurrency / Sync

- `sync` → mutexes, waitgroups
- `sync/atomic` → low-level atomic ops
- `runtime` → goroutine and GC info
- `runtime/debug` → debugging
- `runtime/pprof` → profiling



Packages – Standard Libraries

7. Cryptography & Security

- `crypto` → common interfaces
- `crypto/sha256`, `sha512`, `md5` → hashing
- `crypto/hmac` → HMAC signatures
- `crypto/aes`, `des`, `rsa`, `ecdsa`, `ed25519` → encryption/signatures
- `crypto/tls` → TLS (SSL)
- `crypto/x509` → certificates

8. Concurrency / Sync

- `sync` → mutexes, waitgroups
- `sync/atomic` → low-level atomic ops
- `runtime` → goroutine and GC info
- `runtime/debug` → debugging
- `runtime/pprof` → profiling



Packages – Standard Libraries

7. Cryptography & Security

- `crypto` → common interfaces
- `crypto/sha256`, `sha512`, `md5` → hashing
- `crypto/hmac` → HMAC signatures
- `crypto/aes`, `des`, `rsa`, `ecdsa`, `ed25519` → encryption/signatures
- `crypto/tls` → TLS (SSL)
- `crypto/x509` → certificates

8. Concurrency / Sync

- `sync` → mutexes, waitgroups
- `sync/atomic` → low-level atomic ops
- `runtime` → goroutine and GC info
- `runtime/debug` → debugging
- `runtime/pprof` → profiling

9. Testing

- `testing` → unit tests (`go test`)
- `testing/quick` → randomized property testing
- `net/http/httptest` → HTTP test servers



Your First Program

```
package main

import "fmt"

// this is a comment

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



package main

- This is known as a “package declaration”.
- Every Go program must start with a package declaration.
- Packages are Go's way of organizing and reusing code.
- There are two types of Go programs: executables and libraries.
- **Executable** applications are the kinds of programs that we can run directly from the terminal. (in Windows they end with .exe)
- **Libraries** are collections of code that we package together so that we can use them in other programs.



Packages and imports Every Go

```
package main

func main() {
    print("Hello, World!\n")
}
```

```
import "fmt"
import "math/rand"
```

```
import (
    "fmt"
    "math/rand"
)
```



Code Location

```
import "github.com/mattetti/goRailsYourself/crypto"
```

```
$ go get github.com/mattetti/goRailsYourself/crypto
```



Import fmt

- The import keyword is how we include code from other packages to use with our program.
- The fmt package (shorthand for format) implements formatting for input and output.



Fmt package



fmt Package Overview

Common functions

- **Printing**
 - `fmt.Print()` → print values
 - `fmt.Println()` → print values + newline
 - `fmt.Sprintf(format, args...)` → C-style formatted output
- **Formatting (return as string)**
 - `fmt.Sprint()`
 - `fmt.Sprintln()`
 - `fmt.Sprintf(format, args...)`
- **Input**
 - `fmt.Scan()`, `fmt.Scanf()`, `fmt.Scanln()` → read from `os.Stdin`



Fmt package

Verb	Meaning (for Printf / Sprintf)
%v	default format (any value)
%+v	include field names for structs
%#v	Go syntax representation
%T	type of the value
%d	decimal integer
%b	binary integer
%x / %X	hex integer (lower/upper)
%f	float decimal
%e / %E	scientific notation
%t	boolean
%s	string
%q	quoted string
%p	pointer (address)



Fmt package

```
// Print / Println
fmt.Println("Hello")           // Hello
fmt.Println("Go", 2025)         // Go 2025

// Printf with format verbs
n := 42
fmt.Printf("Decimal: %d\n", n)    // Decimal: 42
fmt.Printf("Binary: %b\n", n)     // Binary: 101010
fmt.Printf("Hex: %x\n", n)        // Hex: 2a

// Floats
f := 3.14159
fmt.Printf("Float: %.2f\n", f)    // Float: 3.14

// Strings
s := "golang"
fmt.Printf("String: %s\n", s)      // String: golang
fmt.Printf("Quoted: %q\n", s)       // Quoted: "golang"

// Structs
u := User{"Alice", 30}
fmt.Printf("Default: %v\n", u)      // {Alice 30}
fmt.Printf("With fields: %+v\n", u) // {Name:Alice Age:30}
fmt.Printf("Go syntax: %#v\n", u)   // main.User{Name:"Alice", Age:30}
fmt.Printf("Type: %T\n", u)         // main.User

// Sprintf -> returns string
msg := fmt.Sprintf("Hi %s, age %d", u.Name, u.Age)
fmt.Println(msg)                  // Hi Alice, age 30
```



How To Write Comments in Go

```
// This is a comment
```

```
/*
```

```
Everything here  
will be considered  
a block comment  
*/
```



Naming Variables: Rules and Style

- The naming of variables is quite flexible, but there are some rules to keep in mind:
- Variable names must only be one word (as in no spaces).
- Variable names must be made up of only letters, numbers, and underscores (_).
- Variable names cannot begin with a number.

Valid	Invalid	Why Invalid
userName	user-name	Hyphens are not permitted
name4	4name	Cannot begin with a number
user	\$user	Cannot use symbols
userName	user name	Cannot be more than one word



Global and Local Variables

```
package main
```

```
import "fmt"
```

```
var g = "global"
```

```
func printLocal() {
```

```
    l := "local"
```

```
    fmt.Println(l)
```

```
}
```

```
func main() {
```

```
    printLocal()
```

```
    fmt.Println(g)
```

```
}
```

A large, light blue starburst shape with the word "global" written in black inside it, connected by a blue arrow from the "g" in "global" in the code.

A large, light blue starburst shape with the word "local" written in black inside it, connected by a blue arrow from the "l" in "local" in the code.



Data Types

Category	Type(s)	Example Declaration	Example Value	Zero Value
Boolean	bool	<code>var b bool = true</code>	true, false	false
Integer (signed)	int, int8, int16, int32 (rune), int64	<code>var x int32 = -42</code>	-42	0
Integer (unsigned)	uint, uint8 (byte), uint16, uint32, uint64, uintptr	<code>var y uint16 = 500</code>	500	0
Float	float32, float64	<code>var pi float64 = 3.1415</code>	3.1415	0.0
Complex	complex64, complex128	<code>z := complex(2, 3)</code>	(2+3i)	0+0i



Data Types

String	string	name := "GoLang"	"GoLang"	"" (empty string)
Pointer	*T	var p *int	pointer to int	nil
Array	[N]T	var arr [3]int = [3]int{1,2,3}	[1 2 3]	[0 0 0]
Slice	[]T	s := []int{1,2,3}	[1 2 3]	nil
Map	map[K]V	m := map[string]int{" Alice":25}	map[Alice:25]	nil
Struct	struct {}	type P struct {Name string}	P{Name:"Bob"}	field zero-values
Interface	interface{} or custom	var i interface{} = 42	42	nil
Function	func	var f func(int) int	function reference	nil
Channel	chan T	c := make(chan int)	channel reference	nil



Data Types

Data Type	Range
uint8	0 to 255
uint16	0 to 65535
uint32	0 to 4294967295
uint64	0 to 18446744073709551615

Data Type	Range
int8	-128 to 127
int16	-32768 to 32767
int32	-2147483648 to 2147483647
int64	-9223372036854775808 to 9223372036854775808



Data Types

Go has the following architecture-independent integer types:

```
uint8      unsigned 8-bit integers (0 to 255)
uint16     unsigned 16-bit integers (0 to 65535)
uint32     unsigned 32-bit integers (0 to 4294967295)
uint64     unsigned 64-bit integers (0 to 18446744073709551615)
int8       signed 8-bit integers (-128 to 127)
int16      signed 16-bit integers (-32768 to 32767)
int32      signed 32-bit integers (-2147483648 to 2147483647)
int64      signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
```

Floats and complex numbers also come in varying sizes:

```
float32    IEEE-754 32-bit floating-point numbers
float64    IEEE-754 64-bit floating-point numbers
complex64  complex numbers with float32 real and imaginary parts
complex128 complex numbers with float64 real and imaginary parts
```

There are also a couple of alias number types, which assign useful names to specific data types:

```
byte       alias for uint8
rune      alias for int32
```



Data Types

`uint` unsigned, either 32 or 64 bits

`int` signed, either 32 or 64 bits

`uintptr` unsigned integer large enough to store the uninterpreted bits of a pointer value



Keywords

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	If	range	type
continue	for	import	return	var



Variables & inferred typing

The var statement declares a list of variables with the type declared last.

```
var (
    name      string
    age       int
    location string
)
```

Or even

```
var (
    name, location string
    age           int
)
```



Variables & inferred typing

Variables can also be declared one by one:

```
var name      string
var age       int
var location  string
```

A var declaration can include initializers, one per variable.

```
var (
    name      string = "Prince Oberyn"
    age       int    = 32
    location  string = "Dorne"
)
```

If an initializer is present, the type can be omitted, the variable will take the type of the initializer (inferred typing).

```
var (
    name      = "Prince Oberyn"
    age       = 32
    location = "Dorne"
)
```



Variables & inferred typing

You can also initialize variables on the same line:

```
var (
    name, location, age = "Prince Oberyn", "Dorne", 32
)
```

Inside a function, the `:=` short assignment statement can be used in place of a var declaration with implicit type.

```
func main() {
    name, location := "Prince Oberyn", "Dorne"
    age := 32
    fmt.Printf("%s (%d) of %s", name, age, location)
}
```

A variable can contain any type, including functions:

```
func main() {
    action := func() {
        //doing something
    }
    action()
}
```



Constants

```
const Pi = 3.14
const (
    StatusOK          = 200
    StatusCreated     = 201
    StatusAccepted    = 202
    StatusNonAuthoritativeInfo = 203
    StatusNoContent   = 204
    StatusResetContent = 205
    StatusPartialContent = 206
)
```

```
const (
    Pi      = 3.14
    Truth   = false
    Big     = 1 << 62
    Small   = Big >> 61
)

func main() {
    const Greeting = ""
    fmt.Println(Greeting)
    fmt.Println(Pi)
    fmt.Println(Truth)
    fmt.Println(Big)
}
```



Basic Types

```
package main

import (
    "fmt"
    "math/cmplx"
)

var (
    goIsFun bool        = true
    maxInt  uint64      = 1<<64 - 1
    complex complex128 = cmplx.Sqrt(-5 + 12i)
)

func main() {
    const f = "%T(%v)\n"
    fmt.Printf(f, goIsFun, goIsFun)
    fmt.Printf(f, maxInt, maxInt)
    fmt.Printf(f, complex, complex)
}
```

```
bool(true)
uint64(18446744073709551615)
complex128((2+3i))
```



Type conversion

The expression **T(v)** converts the value **v** to the type **T**. Some numeric conversions:

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

```
package main
b := 125.0
c := 390.8
fmt.Println(int(b))
fmt.Println(int(c))

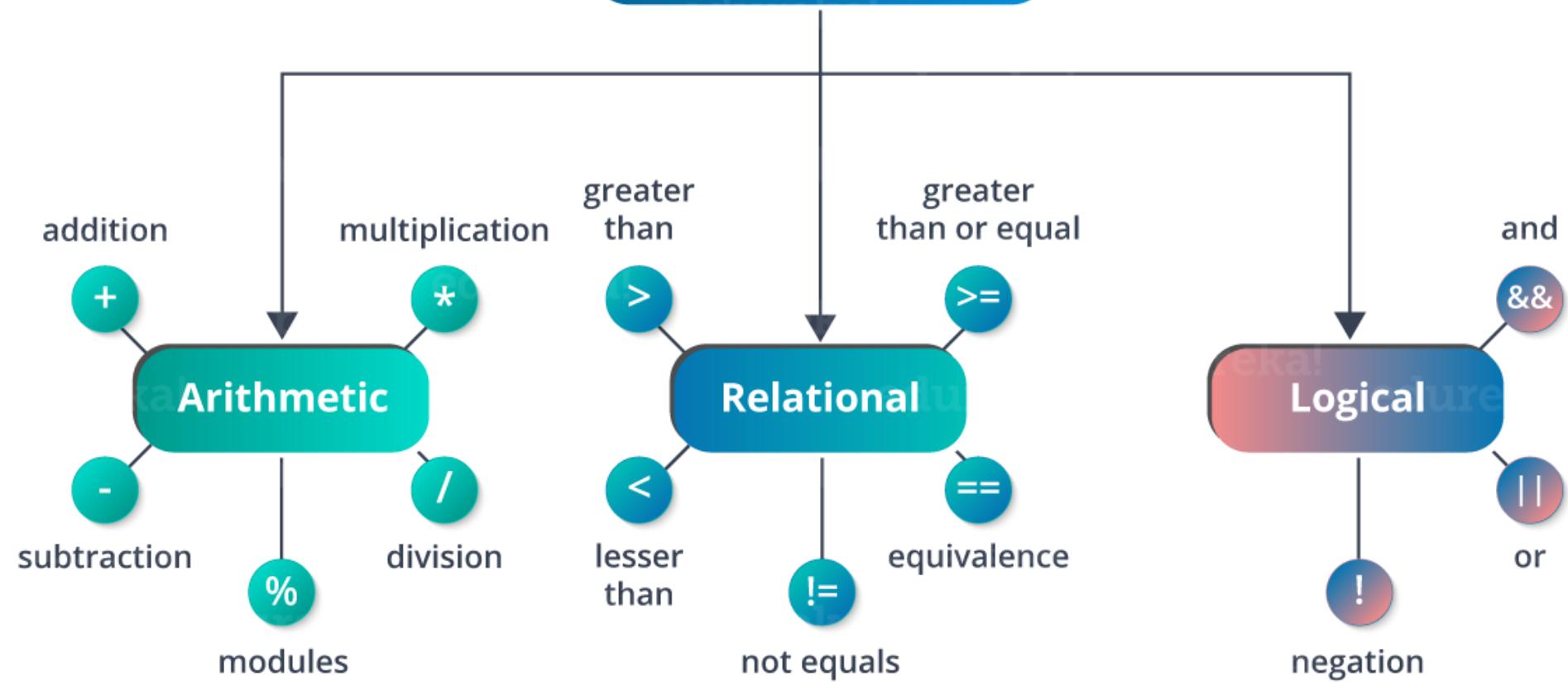
import (
    "fmt"
    "strconv"
)

func main() {
    a := strconv.Itoa(12)
    fmt.Printf("%q\n", a)
}
```



Operators

Types of Operators





Array

📦 Arrays in Go

◆ Definition

- An **array** is a **fixed-length** sequence of elements of the same type.
- Length is part of the type ([5]int is not the same as [10]int).
- `var arr [3]int // array of 3 ints, initialized to [0 0 0]`



Array

Slices

"Five of Spades" "Three of Diamonds" "Five of Diamonds"

*Every element in a slice must
be of same type*

"Five of Spades" 55525235 "Five of Diamonds"



Array

```
import "fmt"

func main() {
    cards := []string{"Ace of Diamonds", newCard()}
    cards = append(cards, "Six of Spades")

    for i, card := range cards {
        fmt.Println(i, card)
    }
}

func newCard() string {
    return "Five of Diamonds"
}
```



Array

```
index of this  
element in the  
array  
for index, card := range cards {  
    fmt.Println(card)  
}  
Current card  
we're iterating  
over  
Take the slice  
of 'cards' and  
loop over it  
Run this one time  
for each card in the  
slice
```





Array

Question 3:

How do we iterate through each element in a slice and print out its value?

```
colors := []string{"Red", "Yellow", "Blue"}  
○ for value in colors {  
    fmt.Println(value)  
}
```

```
colors := []string{"Red", "Yellow", "Blue"}  
○ colors.forEach(func(value, index) {  
    fmt.Println(value)  
})
```

```
colors := []string{"Red", "Yellow", "Blue"}  
○ for index, color := colors {  
    fmt.Println(index, color)  
}
```

```
colors := []string{"Red", "Yellow", "Blue"}  
○ for index, color := range colors {  
    fmt.Println(index, color)  
}
```



Arrays

- Define Array
 - *[capacity]data_type{element_values}*
 - *[4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral"}*
- `coral := [4]string{"blue coral", "staghorn coral", "pillar coral", "elkhorn coral"}`
- `fmt.Println(coral)`



Arrays

- `var myArray1 = [3]int` // will be filled with so called zero values, for integers: 0
- `var myArray2 = [5]int{1,2,3,4,5}` // number of values between {} can not be larger than size (ofc)
- `var myArray3 = [...]int{1,2,3,4}` // the compiler will count the array elements for you



Arrays

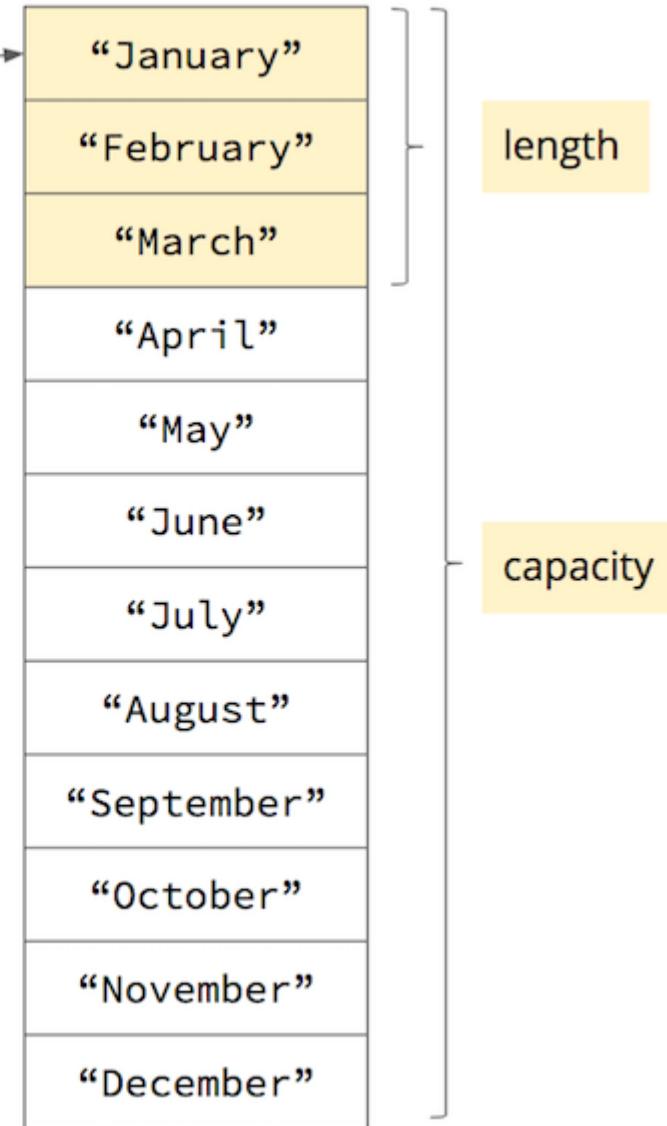
- ar variable_name [SIZE1][SIZE2]...[SIZEN]
variable_type
- a := [3][4]int{
 - {0, 1, 2, 3} , // initializers for row indexed by 0
 - {4, 5, 6, 7} , // initializers for row indexed by 1 */
 - {8, 9, 10, 11} // initializers for row indexed by 2
- }



Slices

```
//slice of array  
Q1 := months[0:3]
```

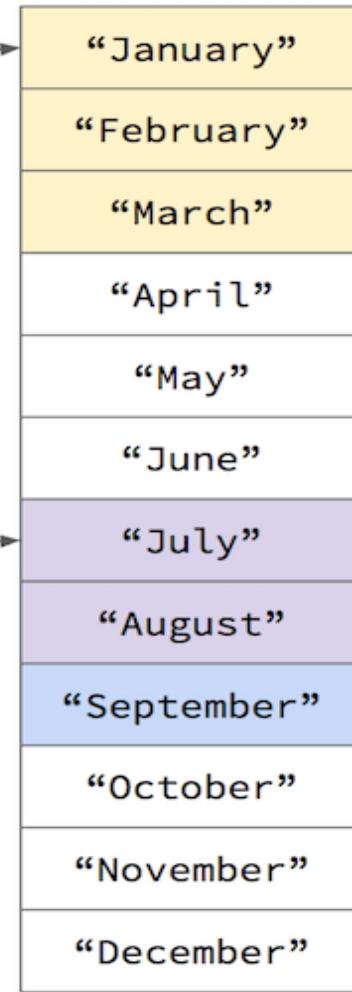
```
{  
    data: *,  
    length: 3,  
    capacity: 12,  
}
```



Slices

```
Q1 := months[0:3]
```

```
{  
    data: *,  
    length: 3,  
    capacity: 12,  
}
```



```
Q3 := months[6:9]
```

```
{  
    data: *,  
    length: 3,  
    capacity: 6,  
}
```

```
SEASummer := months[6:8]
```

```
{  
    data: *,  
    length: 2,  
    capacity: 6,  
}
```



Creating slices

- From an existing array
- We already stated slices are an abstraction over the array, so let's grab a slice from an array.
- `myArray := [10]int{0,1,2,3,4,5,6,7,8,9} // firstly we need an array to make the slice out of`
- `mySlice1 := myArray[1:5] // slice from 1st up to 5th element, so [1 2 3 4]`
- `mySlice2 := mySlice1[0:2] // slicing a slice // [1 2]`



Creating slices

- var mySlice3 := make([]int, 4, 4) // mySlice has length and capacity 5, and it's filled with zero values for int, so it's [0 0 0 0]
- mySlice5 := make([]int, 2, 4) // now it's [0 0 nil nil]
 - // but be aware that nil
 - // values don't print, so
 - // when printed it will be
 - // just [0 0]



Arrays

```
regNos := make([]int32, 100)
```

```
//Assign values
```

```
for r:=range regNos{
```

```
    regNos[r] = rand.Int31n(1000)
```

```
}
```

```
//print values
```

```
for index, val := range regNos{
```

```
    fmt.Printf("%d = %d\n", index, val)
```

```
}
```

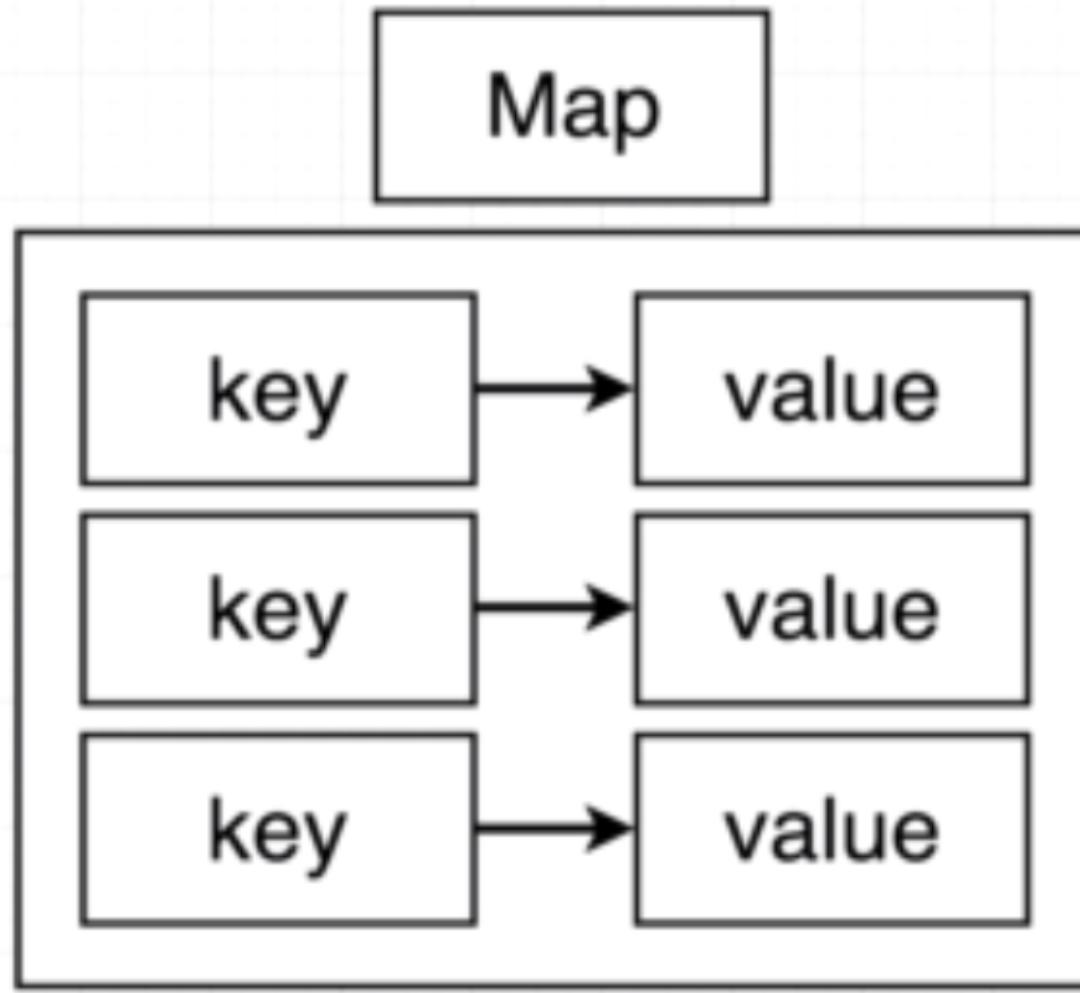


Map

- A map maps keys to values.
- The zero value of a map is nil. A nil map has no keys, nor can keys be added.
- The make function returns a map of the given type, initialized and ready for use.

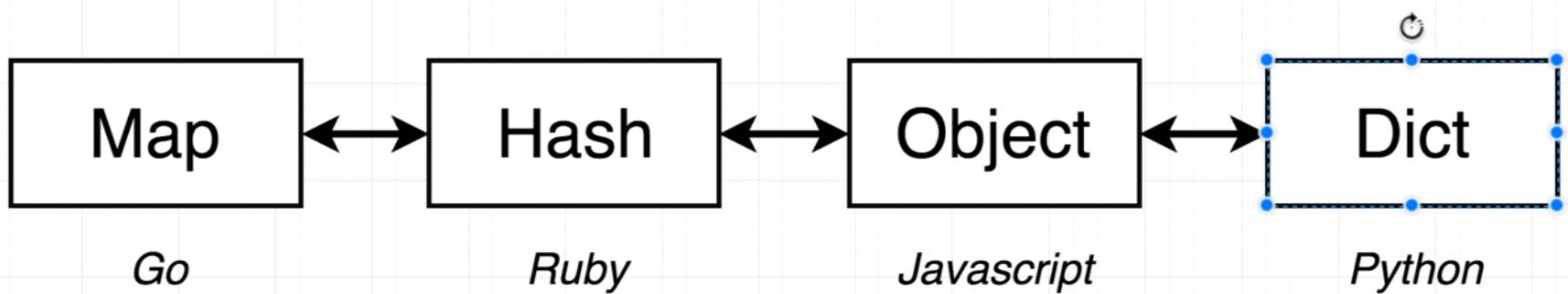


Map



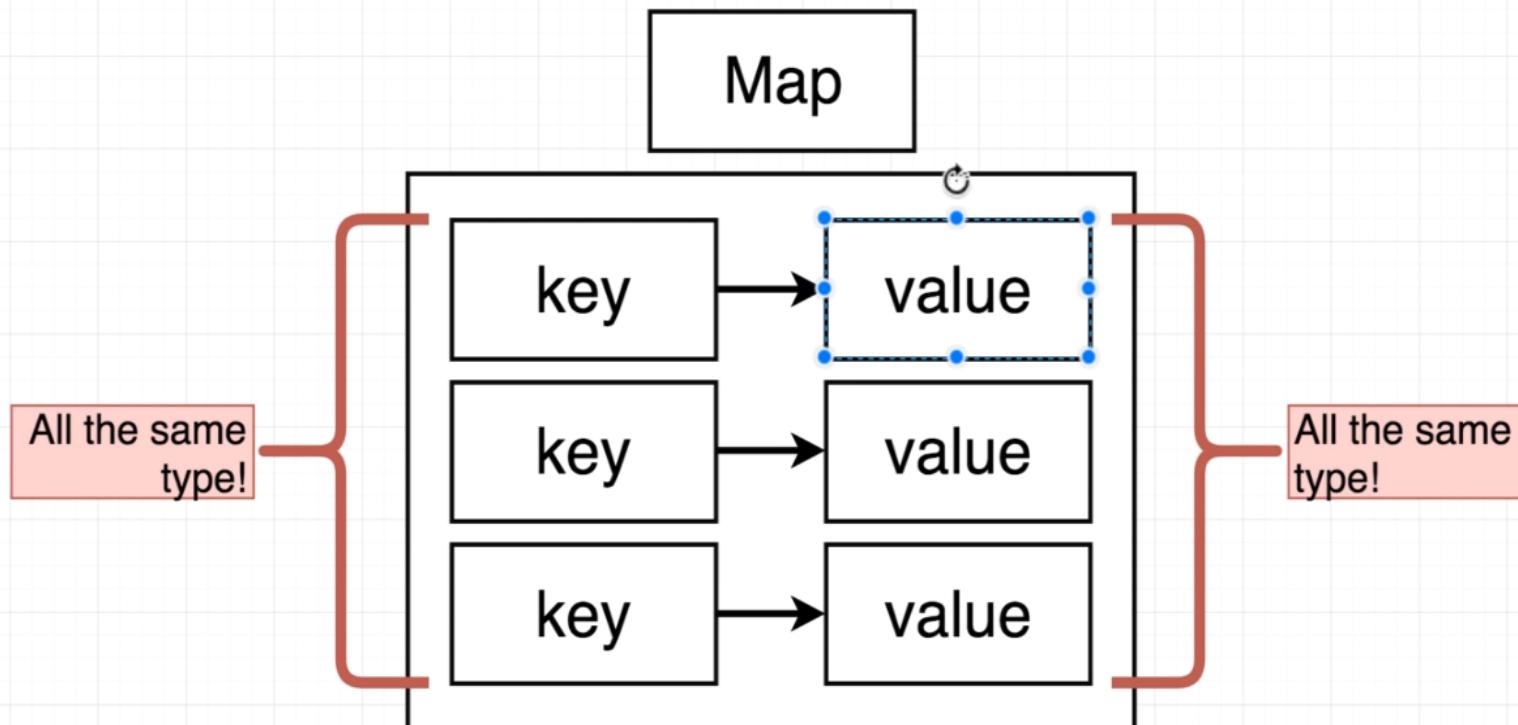


Map





Map





Map

Map

All keys must be the same type	Use to represent a collection of related properties
--------------------------------	---

All values must be the same type	Don't need to know all the keys at compile time
----------------------------------	---

Keys are indexed - we can iterate over them	Reference Type!
---	-----------------

Struct

Values can be of different type	You need to know all the different fields at compile time
---------------------------------	---

Keys don't support indexing	Use to represent a "thing" with a lot of different properties
-----------------------------	---

Value Type!



Map

Dynamic map (int, customer)

The screenshot shows a Go code editor interface with the following details:

- Project Tree:** The project structure is visible on the left, showing packages like `amexws`, `banking`, `dao`, and `models`.
- Code Editor:** The main window displays `customerdao.go` with the following code:

```
func CreateCustomerList() {
    customerList := make(map[int]models.Customer)
    //create customers
    fmt.Println("Enter no of customers to be created")
    var count int
    fmt.Scanln(&count)
    for i := 0; i < count; i++ {
        customerList[i] = models.Customer{ Account_Number: rand.Int63n( n: 1000000),
                                         Name: "customer" + (strconv.Itoa(rand.Int())),
                                         AddressRef: models.Address{ Door_No: "428998", Street_Name: "x", City: "Chennai", State: "TN"},
                                         Contact_Number: 9952056789, Email: "sample@gmail.com", Password: "test@123" }
    }
    for key, value := range customerList {
        fmt.Printf("\nThe Customer Id is #{} and value is #{}")
    }
}

func FindAllCustomers() models.Customer {
    //create customer instances
    var customers models.Customer
    //for i:=0;i<len(customers);i++ {
    customers = models.Customer{ Account_Number: rand.Int63n( n: 1000000),
                               Name: "customer" + (strconv.Itoa(rand.Int())), AddressRef: models.Address{ Door_No: "428998", Street_Name: "x", City: "Chennai", State: "TN"},
```
- Annotations:** A blue arrow points from the text "Dynamic map (int, customer)" at the top right towards the `make(map[int]models.Customer)` line in the code.
- Run Bar:** At the bottom, there is a "Run" bar with the command `go build github.com/amexws/Map`.



Map

- In Go, a **map** is a built-in data type that associates **keys** with **values**.

Declaring a Map

```
go

// Method 1: Using make
m := make(map[string]int)

// Method 2: Using map literal
n := map[string]string{
    "foo": "bar",
    "baz": "qux",
}
```



Map

```
// Create an empty map
numbers := make(map[int]string)

// Assign values in a loop
for i := 1; i <= 5; i++ {
    numbers[i] = fmt.Sprintf("Value-%d", i)
}

// Print the map
for k, v := range numbers {
    fmt.Println(k, "=>", v)
}
```



Map

```
-- mapdemo.go -- GenCustomer  
func GenCustomer() {  
    for i := 0; i < 100; i++ {  
        customers[i] = faker.FirstName()  
    }  
    fmt.Println("Length of map:", len(customers))  
    for k, v := range customers {  
        println(k, v)  
    }  
    //delete(customers, 10)  
    // Remove customers with ID < 50  
    for k := range customers {  
        if k < 50 {  
            delete(customers, k)  
        }  
    }  
  
    fmt.Println("Length of map:", len(customers))  
}
```

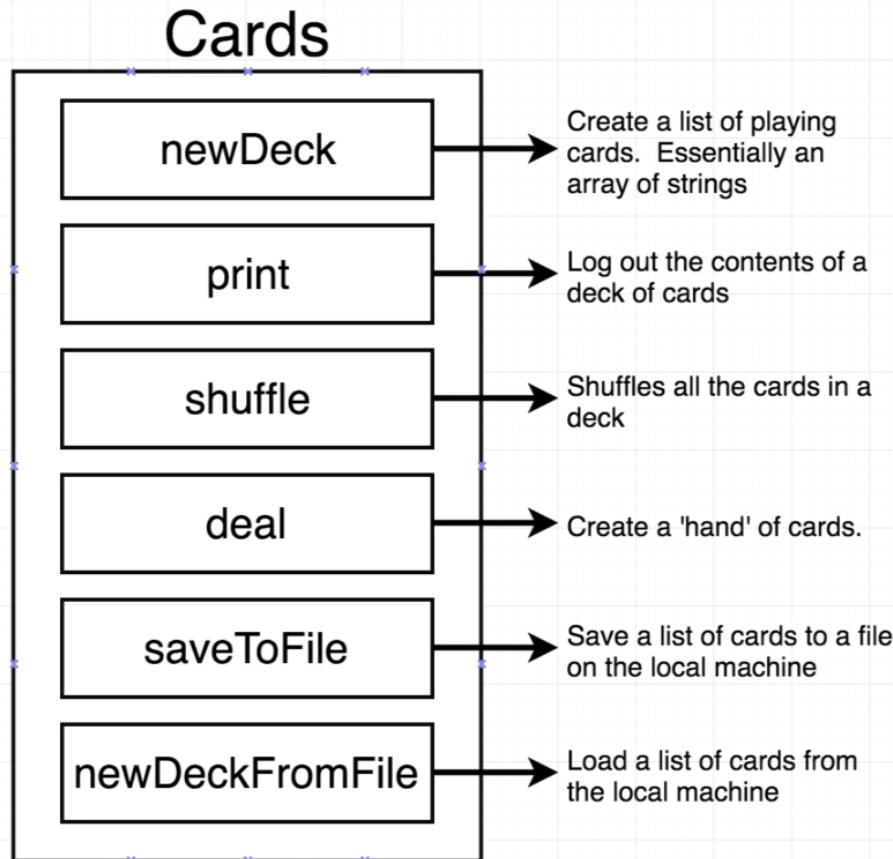


Variadic Function

```
package main

func Accumulate(nums ...int) int {
    total := 0
    for _, num := range nums {
        total += num
    }
    return total
}
```

Use case





Use case

```
var card string = "Ace of Spades"
```

We're about to
create a new
variable

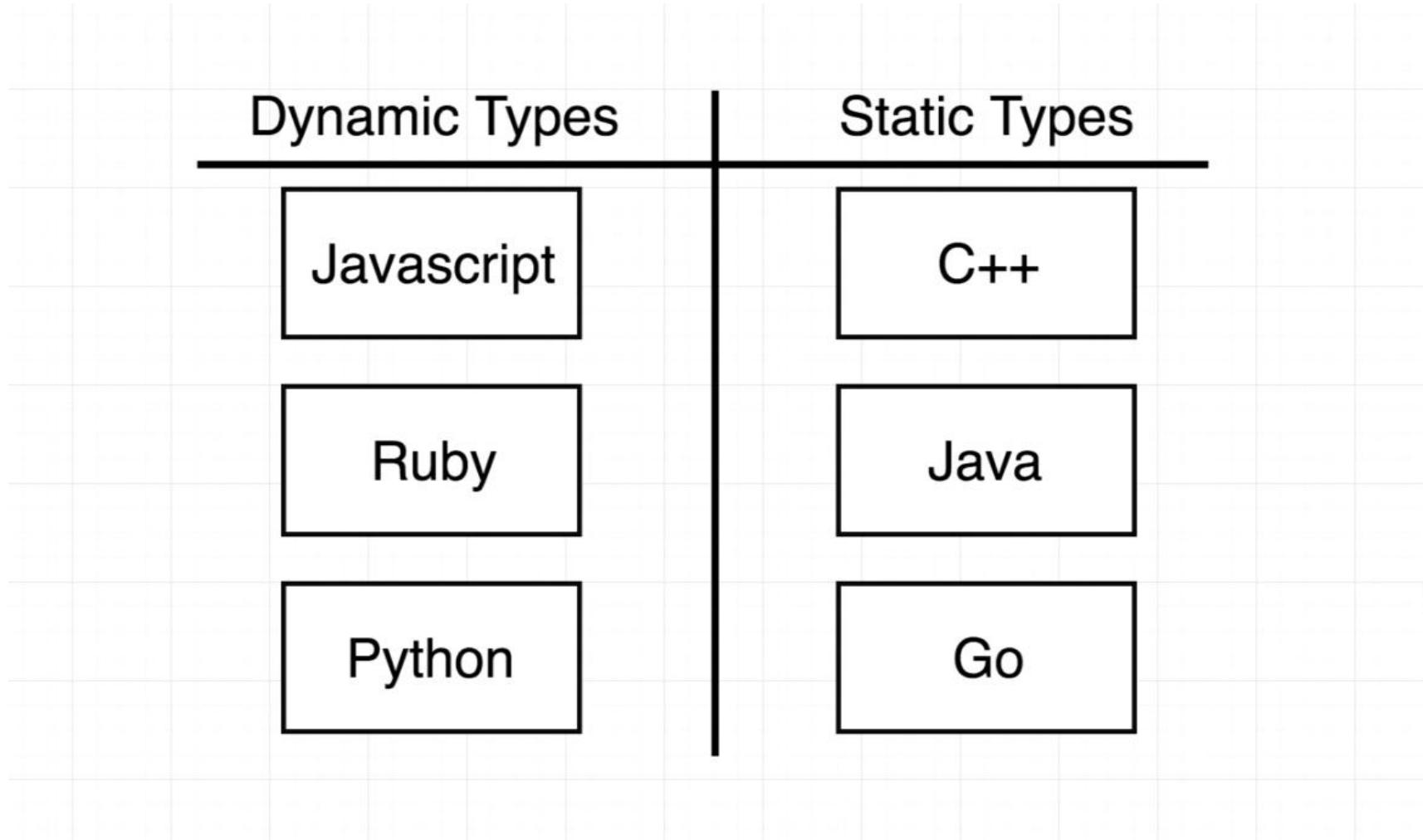
The name of the
variable will be
'greeting'

Only a "string" will
ever be assigned
to this variable

Assign the value
"Ace of Spades"
to this variable



Use case





Use case

Question 4:

Will the following code compile? Why or why not?

```
1 | paperColor := "Green"  
2 | paperColor := "Blue"
```

Yes, because we are creating and assigning a value to the variable 'paperColor' twice.

No, because a variable can only be initialized one time. In this case, the ':=' operator is being used to initialize 'paperColor' two times

No, because ':=' is not a valid operator



Use case

Question 6:

This might require a bit of experimentation on your end :). Remember that you can use the Go Playground at <https://play.golang.org/> to quickly run a snippet of code.

Is the following code valid?

```
1 | package main
2 |
3 | import "fmt"
4 |
5 | deckSize := 20
6 |
7 | func main() {
8 |     fmt.Println(deckSize)
9 | }
```

Yes

No



Functions

Define a function
called 'newCard'

When executed, this
function will return a
value of type 'string'

```
func newCard() string {
```

```
}
```



Understanding init in Go

- In Go, the predefined `init()` function sets off a piece of code to run before any other part of your package.
- This code will execute as soon as the package is imported.
- It can be used when you need your application to initialize in a specific state, such as when you have a specific configuration or set of resources with which your application needs to start.
- It is also used when importing a side effect, a technique used to set the state of a program by importing a specific package.



Understanding init in Go

- ♦ **What is init in Go?**
- init is a **special function** in Go that runs **automatically** before main().
- It is mainly used for **initialization logic** (setting defaults, connecting to configs, seeding random, etc.).
- You **cannot call init() explicitly**.



Understanding init in Go

1 Basic Example

```
go

package main

import "fmt"

func init() {
    fmt.Println("Init function runs before main")
}

func main() {
    fmt.Println("Main function runs")
}
```

Output:

```
pgsql

Init function runs before main
Main function runs
```



Understanding init in Go

2 Multiple init Functions

You can define **multiple init functions** in the same package (even in different files). They all execute before `main`, in the order they appear **within the package**.

```
go

package main

import "fmt"

func init() {
    fmt.Println("First init")
}

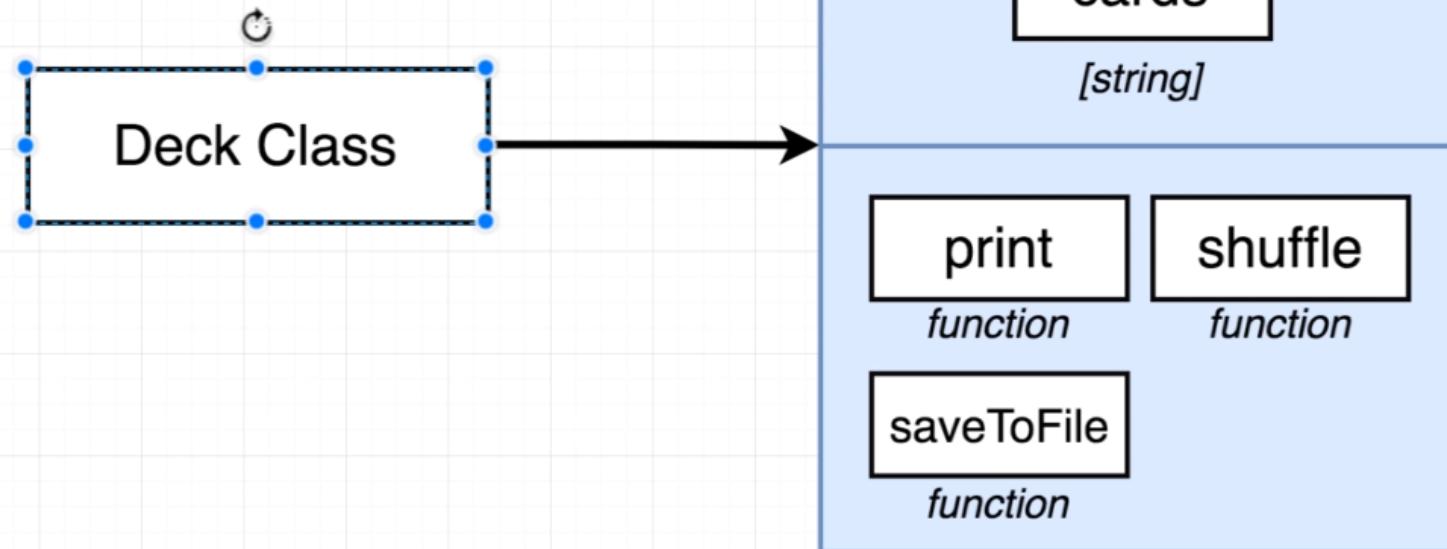
func init() {
    fmt.Println("Second init")
}

func main() {
    fmt.Println("Main")
}
```

OO vs Go Structure

Cards - OO Approach

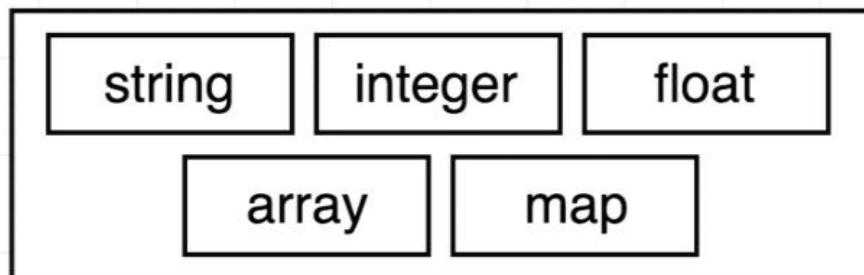
Deck Instance





OO vs Go Structure

Base Go Types



We want to "extend" a base type and add some extra functionality to it

`type deck []string`

Tell Go we want to create an array of strings and add a bunch of functions specifically made to work with it

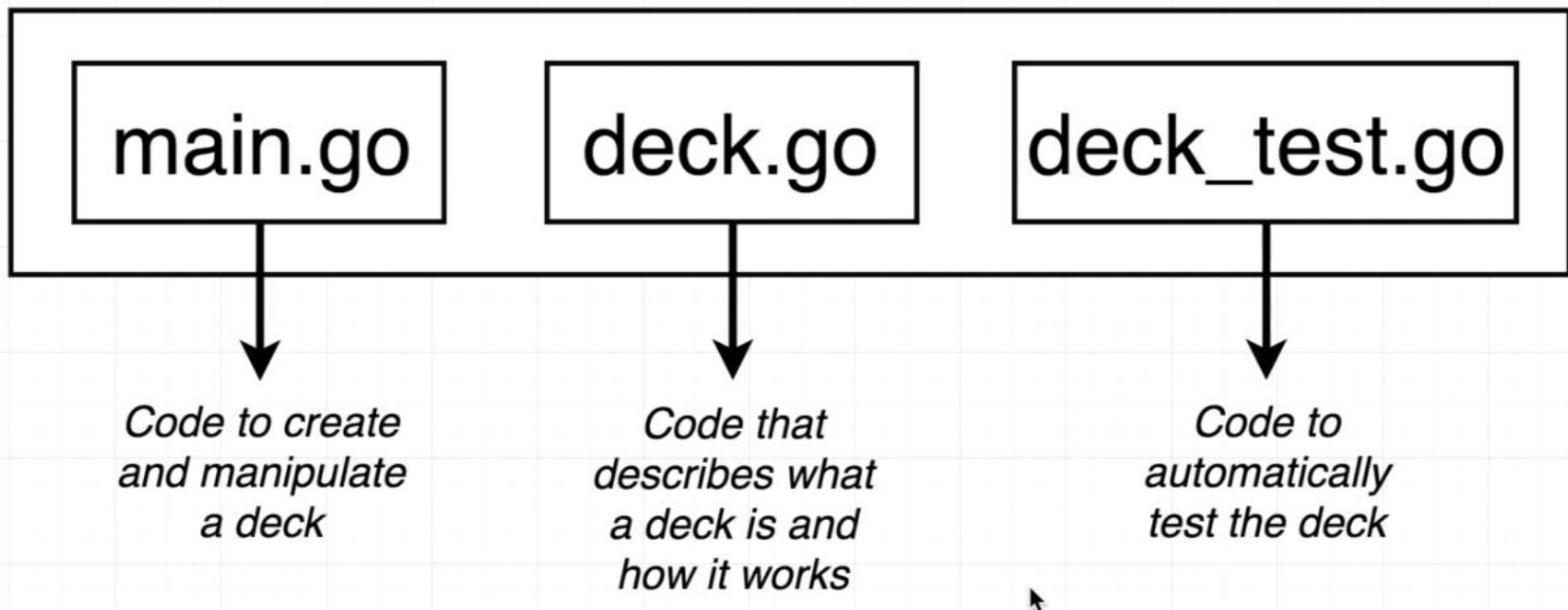
`Functions with 'deck'
as a 'receiver'`

A function with a receiver is like a "method" - a function that belongs to an "instance"



OO vs Go Structure

'cards' folder





Custom Type

```
go

package main

import "fmt"

// Define a custom type
type Age int

func main() {
    var a Age = 25
    fmt.Println("Age is:", a)
}
```



Custom Type

```
}

type min int
type max int

func main() {
    otp := util.GetOTP()
    fmt.Println("Your OTP is:", otp)
    // Print / Println
    fmt.Print("Hello")      // Hello
    fmt.Println("Go", 2025)  // Go 2025
```

```
func GenId(minId min, maxId max) int {
    return rand.Intn(int(maxId)-int(minId)+1) + int(minId)
}
```



Case Study

```
cards = deck{}
```

```
cardSuits := []string{"Spades", "Hearts", "Diamonds"}
```

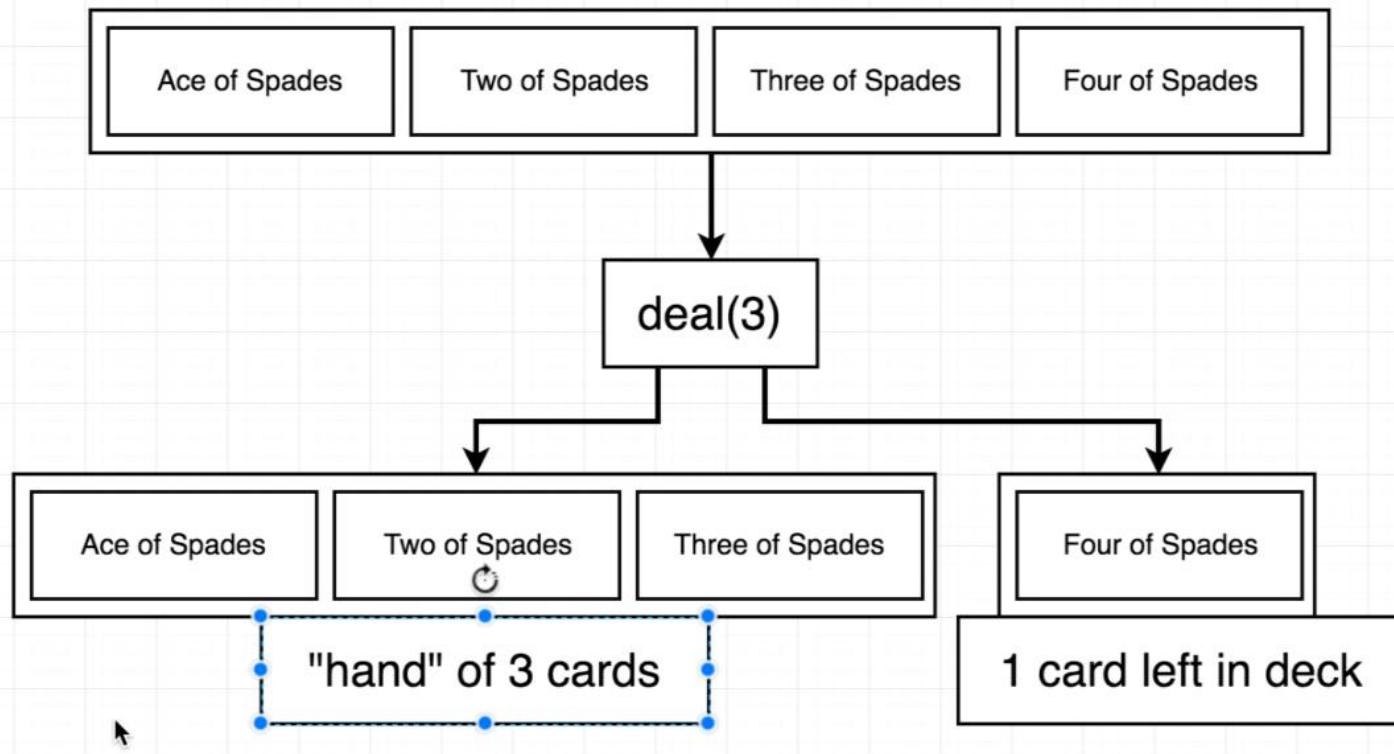
```
cardValues := []string{"Ace", "Two", "Three"}
```

```
for each suit in cardSuits
```

```
    for each value in cardValues
```

```
        Add a new card of 'value of suit' to the 'cards' deck
```

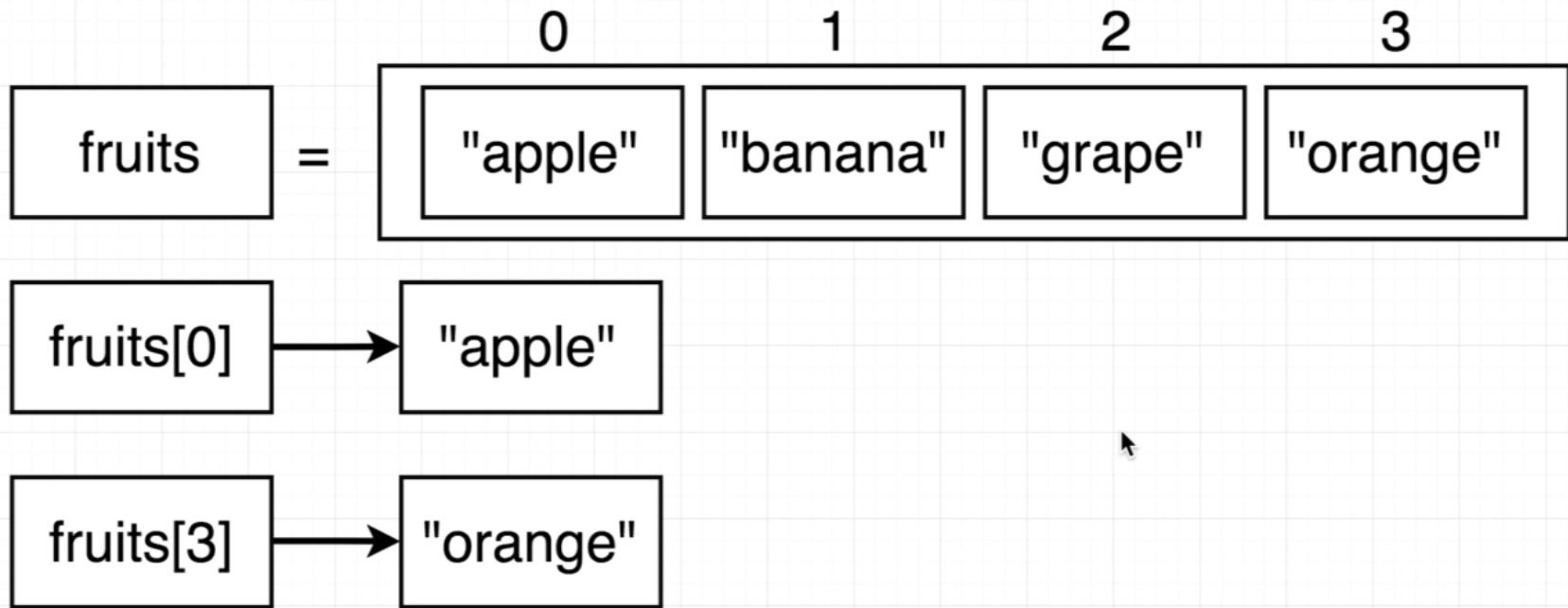
Case Study





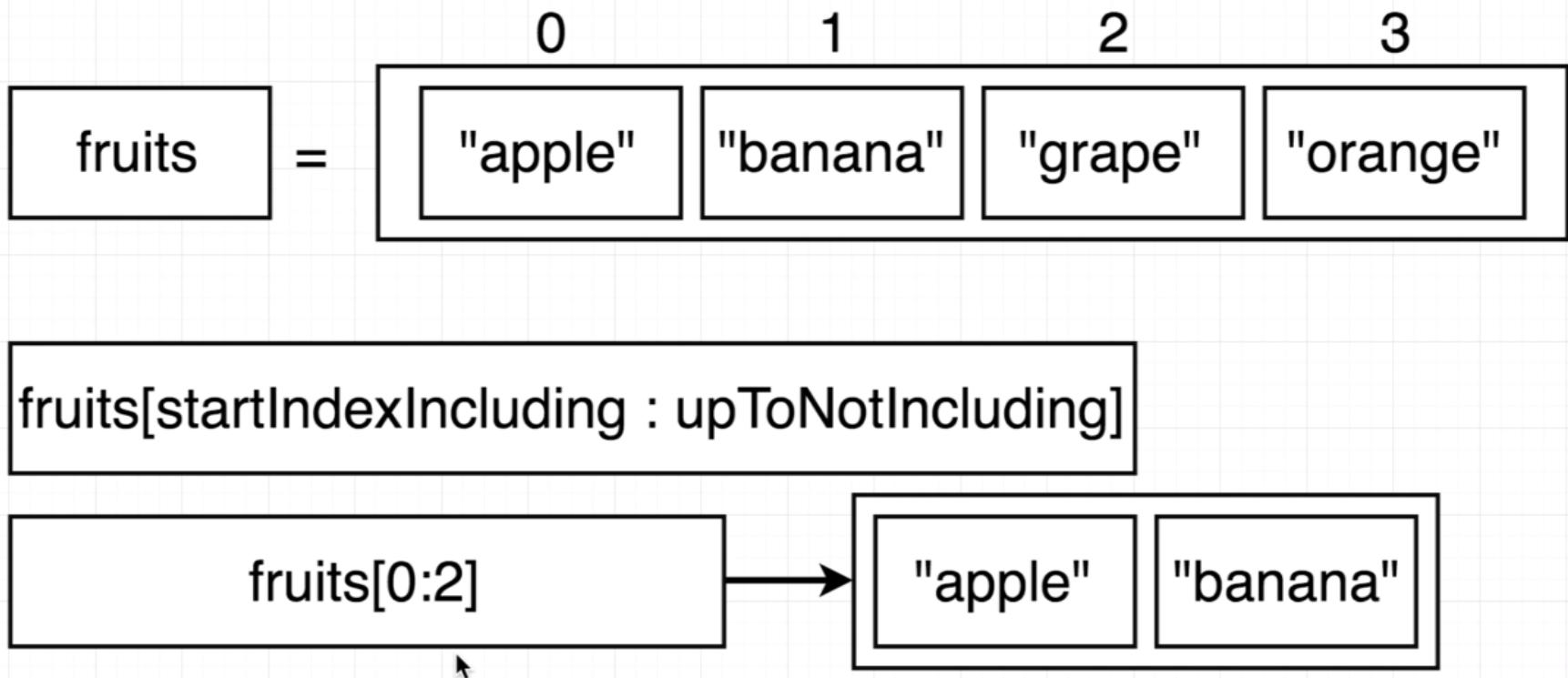
Case Study

Slices are zero-indexed





Case Study





Byte Slice

- In Go, `[]byte` is a slice of bytes (`uint8` values).
- Often used for strings, files, network data, etc.

1 Declaring a byte slice

```
go

var b []byte          // nil slice
b = []byte{65, 66, 67} // slice with values
fmt.Println(b)         // [65 66 67]
fmt.Println(string(b)) // "ABC"
```

2 Converting string → byte slice

```
go

s := "Hello"
b := []byte(s)          // string → []byte
fmt.Println(b)           // [72 101 108 108 111]

s2 := string(b)         // []byte → string
fmt.Println(s2)          // Hello
```



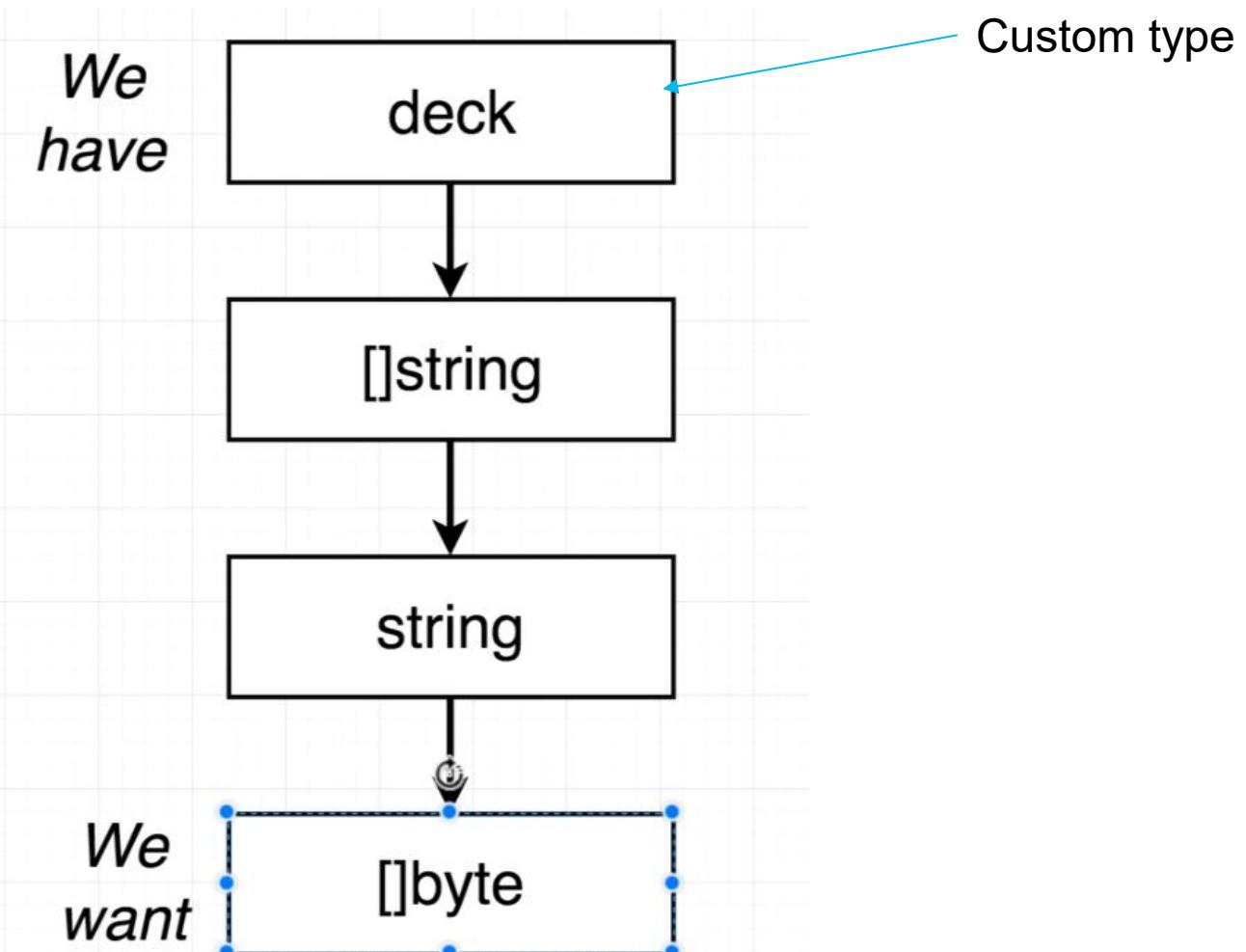
Byte Slice

```
import (
    "crypto/rand"
    "fmt"
)

func main() {
    b := make([]byte, 8)    // slice of 8 bytes
    _, err := rand.Read(b) // fills slice with random bytes
    if err != nil {
        panic(err)
    }
    fmt.Println("Random bytes:", b)
}
```



Byte Slice





Pointers

- **◆ What is a pointer?**
- A pointer holds the **memory address** of a value.
- In Go, the type of a pointer is written with `*T` (pointer to type `T`).
- The `&` operator gives you the address of a variable.
- The `*` operator dereferences a pointer (gets the value at that address).



What is the need of the pointers?

- **1. Avoid Copying Large Data**
- By default, Go passes values **by copy**.
If you pass a large struct or array to a function, it makes a full copy (expensive in memory & CPU).
- **2. Modify Values Inside Functions**
- Functions in Go normally work on copies.
If you want a function to actually **modify the original**, you need a pointer.
- **3. Work with Structs Efficiently**
- Structs can be large; passing by pointer makes function calls cheaper and allows updates.



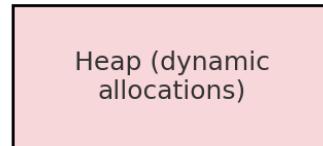
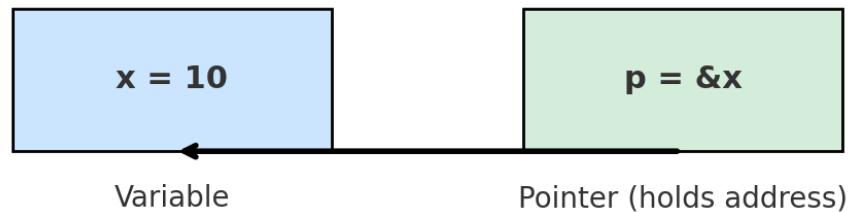
What is the need of the pointers?

- **4. Optional / Nil Values**
- Pointers let you represent “no value” with nil.
- **5. Share Data Between Functions**
- Pointers let multiple functions access & update the **same underlying data**, instead of working on copies.
- **6. Under the Hood of Slices, Maps, Channels**
- Slices, maps, and channels in Go already **use pointers internally**.
- That’s why when you pass a slice/map/channel to a function, modifications are visible outside the function, **without explicit pointers**.



What is the need of the pointers?

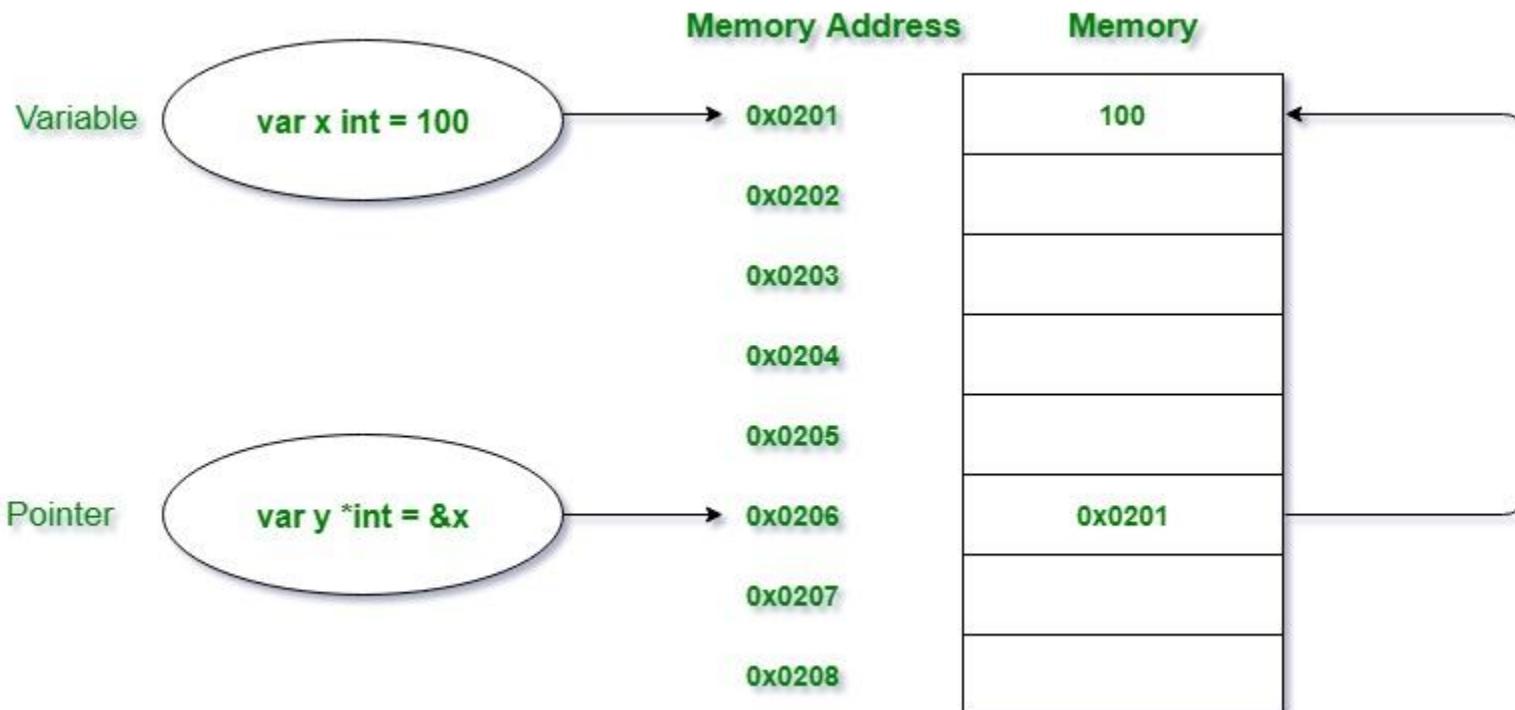
Stack Memory



- `x = 10` is stored directly in **stack memory**.
- `p = &x` is a **pointer** that stores the memory address of `x`.
- Dereferencing `*p` gives you the actual value (`10`).
- The **heap** (shown below) is where dynamically allocated values (via `new`, `make`, slices, maps, channels) live.



What is the need of the pointers?





Pointers

main.go

```
package main

import "fmt"

func main() {
    var creature string = "shark"
    var pointer *string = &creature

    fmt.Println("creature =", creature)
    fmt.Println("pointer =", pointer)
}
```



Structures

- **What is a Struct?**
- A **struct** is a collection of fields (variables) grouped together under one type.
Think of it like a **class without methods** in other languages.

1 Declaring a Struct

```
go

type Person struct {
    Name string
    Age  int
}
```



Structures

2 Creating Struct Instances

```
go

p1 := Person{"Alice", 25}           // positional
p2 := Person{Name: "Bob", Age: 30}   // named fields
p3 := Person{Name: "Charlie"}       // Age defaults to 0
var p4 Person                      // empty struct with zero values

fmt.Println(p1, p2, p3, p4)
```

3 Accessing & Modifying Fields

```
go

p := Person{Name: "Alice", Age: 25}
fmt.Println(p.Name) // Alice
p.Age = 26         // update field
```



Structures

4 Struct with Pointers

go

```
func birthday(p *Person) {  
    p.Age++ // modifies original  
}
```

```
p := Person{"Alice", 25}  
birthday(&p)  
fmt.Println(p.Age) // 26
```



Structures

5 Struct with Methods

```
func (v *Vehicle) Save() (bool, error) {
    vehicleMap[v.RegistrationNo] = v
    return true, nil
}
func (v *Vehicle) GetByID(id string) (*Vehicle, error) {
    if vehicle, exists := vehicleMap[id]; exists {
        return vehicle, nil
    }
    return nil, fmt.Errorf("vehicle not found")
}
func (v *Vehicle) GetAll() ([]*Vehicle, error) {      Parameswari Ettiapp
    vehicles := make([]*Vehicle, 0, len(vehicleMap))
    for _, vehicle := range vehicleMap {
        vehicles = append(vehicles, vehicle)
    }
    return vehicles, nil
}
func (v *Vehicle) Update(id string, color string) (*Vehicle, error) {
    if vehicle, exists := vehicleMap[id]; exists {
        vehicle.Color = color
        return vehicle, nil
    }
    return nil, fmt.Errorf("vehicle not found")
}
func (v *Vehicle) Delete(id string) (bool, error) {
    if _, exists := vehicleMap[id]; exists {
        delete(vehicleMap, id)
        return true, nil
    }
}
```



Structures - Anonymous

- Why use anonymous structs?
- **1. One-off / throwaway models**
- If you need a data structure **only once**, it's wasteful to create a named type.

```
response := struct {
    Status string `json:"status"`
    Data   any     `json:"data"`
}{  
    Status: "ok",
    Data:   "something",
}
```

This is common in **DTOs / API responses** — no need to define a full struct type.



Structures - Anonymous

- Why use anonymous structs?
- **2. Avoid clutter**
- If your domain already has many models, creating named types for small nested structs can bloat your codebase.

Instead:

```
type Policy struct {  
    ID   string  
    Type string  
  
    Coverage struct { // no separate Coverage type needed  
        MaxAmount float64  
        Deductible float64  
    }  
}
```

Here, `Coverage` exists **only inside `Policy`**, so we don't need to expose a separate type.



Structures - Anonymous

- Why use anonymous structs?
- **3. Encapsulation (locality of reference)**
 - By using anonymous structs inside a model:
 - You **signal** that the nested struct should not be reused elsewhere.
 - Keeps logic **tied closely** to the parent type.
 - For example, Policy.Holder only makes sense **inside Policy**, not as a standalone Holder type.



Structures - Anonymous

- Why use anonymous structs?
- **4. Quick prototyping**
- When you're just **testing an idea or writing quick code**, anonymous structs are faster than designing full models.
- **5. Anonymous fields (embedding)**
- You can embed a struct **without a field name**, promoting its fields:

```
type MotorClaim struct {
    Policy // anonymous embedded field
}
```

This lets you access `MotorClaim.Coverage` directly instead of `MotorClaim.Policy.Coverage`.



Structures - Anonymous

- **⚠ When NOT to use anonymous structs**
- When you need **reusability** (e.g., Customer used in many places).
- When you need **methods** on that type.
- When you want to export (public API).
- When clarity is more important than conciseness.



Structures - Anonymous

- ⚡ Rule of Thumb
- 🤝 Use **anonymous structs** when:
 - Data structure is **short-lived, local, or tightly coupled**.
 - It's only relevant **inside one type or one function**.
- 🤝 Use **named structs** when:
 - You expect **reuse** across packages.
 - You need **methods**.
 - You want to share/export in APIs.



Structures - Anonymous

```
// Policy model with anonymous struct fields
type Policy struct {
    ID      string
    Type   string // "HEALTH", "MOTOR", etc.

    // Inline anonymous struct for coverage details
    Coverage struct {
        MaxAmount  float64
        Deductible float64
        Copay      float64
    }

    // Inline anonymous struct for policy holder
    Holder struct {
        ID      string
        Name   string
        DOB    time.Time
    }

    // Inline anonymous struct for provider info
    Provider struct {
        ID      string
        Name   string
    }
}
```





When to Use Which?

Scenario	Anonymous Struct <input checked="" type="checkbox"/>	Named Struct <input checked="" type="checkbox"/>
Data is used only once, local to a parent	✓	✗
You want to reduce boilerplate	✓	✗
You need reusability across multiple models	✗	✓
You want to attach methods	✗	✓
Public API (exported types)	✗	✓
Quick prototyping	✓	✗



Structures

7 Nested Structs

```
go

type Address struct {
    City, Country string
}

type Employee struct {
    Name      string
    Address  Address
}

e := Employee{
    Name: "John",
    Address: Address{City: "Paris", Country: "France"},
}
fmt.Println(e.Address.City) // Paris
```



Structures – Embedding like Inheritance

```
// Base struct
type Policy struct {
    ID   string
    Type string
}

// Another base struct
type Audit struct {
    CreatedAt time.Time
    UpdatedAt time.Time
    Actor     string
}

// Child struct embeds Policy & Audit anonymously
type HealthClaim struct {
    Policy // embedded (anonymous field)
    Audit  // embedded

    ClaimID      string
    ClaimedAmount float64
    Status       string
}
```



Inline Structure

```
package main

import "fmt"

func main() {
    c := struct {
        Name string
        Type string
    } {
        Name: "Sammy",
        Type: "Shark",
    }
    fmt.Println(c.Name, "the", c.Type)
}
```



Methods in GO

```
func(receiver_name Type) method_name(parameter_list)(return_type){  
    // Code  
}
```

```
type Creature struct {  
    Name      string  
    Greeting string  
}
```

```
func (c Creature) Greet() {  
    fmt.Printf("%s says %s", c.Name, c.Greeting)  
}
```



Method vs Function

Feature	Function	Method
Receiver	✗ None	✓ Has a receiver (r Rectangle or r *Rectangle)
Tied to type	✗ Independent	✓ Belongs to a type (struct or custom type)
Call syntax	Add(3, 4)	rect.Area()
Use case	General-purpose logic	Behavior associated with a type (OOP-style)



Recursive Function

- **◆ What is a recursive function?**
- A recursive function is a function that **calls itself** until a **base condition** is met.
It's useful for problems that can be broken down into smaller sub-problems (factorial, Fibonacci, tree traversal, etc.).



Recursive Function

```
// LineItem represents one billed service, which may contain sub-services.  
type LineItem struct {  
    Code      string  
    Amount    float64  
    Children  []*LineItem  
}  
  
// Recursive function to calculate total cost  
func TotalCost(item *LineItem) float64 {  
    if item == nil {  
        return 0  
    }  
    total := item.Amount  
    for _, child := range item.Children {  
        total += TotalCost(child) // recursion here  
    }  
    return total  
}
```



Anonymous Function

- **◆ What is an Anonymous Function?**
- A function **without a name**.
- You can define it inline and assign it to a variable or call it immediately.
- Useful for **short logic, callbacks, goroutines**, etc.



Anonymous Function

1 Assign to a variable

```
go

package main

import "fmt"

func main() {
    add := func(a, b int) int {
        return a + b
    }

    fmt.Println(add(3, 4)) // 7
}
```



Anonymous Function

2 Immediate execution (IIFE: Immediately Invoked Function Expression)

```
go

package main

import (
    "fmt"
    "time"
)

func main() {
    go func(amount float64) {
        if amount > 1000 {
            fmt.Println("⚠️ High-value transaction alert:", amount)
        }
    }(1500)

    time.Sleep(time.Second) // wait for goroutine
}
```



Anonymous Function

3 Anonymous function as goroutine

```
go

package main

import (
    "fmt"
    "time"
)

func main() {
    go func(msg string) {
        fmt.Println("Hello,", msg)
    }("Banking User")

    time.Sleep(time.Second) // wait for goroutine
}
```



Anonymous Function

4 Passing anonymous function as argument

```
go

package main

import "fmt"

func operate(a, b int, op func(int, int) int) int {
    return op(a, b)
}

func main() {
    sum := operate(10, 20, func(x, y int) int {
        return x + y
    })
    fmt.Println(sum) // 30
}
```



Anonymous Function

5 Closures (anonymous function capturing variables)

```
go

package main

import "fmt"

func counter() func() int {
    count := 0
    return func() int {
        count++
        return count
    }
}

func main() {
    next := counter()
    fmt.Println(next()) // 1
    fmt.Println(next()) // 2
    fmt.Println(next()) // 3
}
```

Copy

Here, the anonymous function **remembers the variable** `count` even after `counter()` has returned.



Anonymous Function

```
go

package main

import "fmt"

func main() {
    rate := 0.05 // 5% interest

    applyInterest := func(balance float64) float64 {
        return balance * (1 + rate)
    }

    fmt.Println("New balance:", applyInterest(1000)) // 1050
}
```



Method with Pointer and Value

- Refer StructAuthorDemo



Interface

- In **Go**, an **interface** is a type that defines a set of methods but does not provide their implementation. Any struct (or type) that implements those methods *implicitly* satisfies the interface.



Interface

```
type PatientInfo interface {
    ShowPatientInfo(newMail string)
    DisplayPatientInfo(newMail string)
}

// call by value
func (patient Patient) ShowPatientInfo(newMail string) {
    println("Patient Information:")
    println("First Name:", patient.FirstName)
    println("Last Name:", patient.LastName)
    println("Date of Birth:", patient.DOB.Day, "/", patient.DOB.Month, "/", patient.DOB.Year)
    patient.Email = newMail // Modifying the email field
    println("Email:", patient.Email)
    println("Phone:", patient.Phone)
}

//call by reference
func (patient *Patient) DisplayPatientInfo(newMail string) {
    println("Patient Information:")
    println("First Name:", patient.FirstName)
    println("Last Name:", patient.LastName)
    println("Date of Birth:", patient.DOB.Day, "/", patient.DOB.Month, "/", patient.DOB.Year)
    patient.Email = newMail // Modifying the email field
    println("Email:", patient.Email)
    println("Phone:", patient.Phone)
}
```



Interface

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

The simplicity of the `error` interface makes writing logging and metrics implementations much easier. Let's define a struct that represents a network problem:

```
type networkProblem struct {
    message string
    code    int
}
```

Then we can define an `Error()` method:

```
func (np networkProblem) Error() string {
    return fmt.Sprintf("network error! message: %s, code: %v", np.m
}
```



Interface

```
type Modify interface {  
    changeName(name *string)  
}  
  
func receiveInterface(m Modify){  
    var name string ="Parameswari"  
    var ptrname *string=&name  
    m.changeName(ptrname)
```



Interface

Interfaces are **not** generic types

Other languages have 'generic' types - go (famously) does not.

Interfaces are 'implicit'

We don't manually have to say that our custom type satisfies some interface.

Interfaces are a contract to help us manage types

GARBAGE IN -> GARBAGE OUT. If our custom type's implementation of a function is broken then interfaces won't help us!

Interfaces are tough. Step #1 is understanding how to read them

Understand how to read interfaces in the standard lib. Writing your own interfaces is tough and requires experience



Interface keys and slices

```
1 package main
2
3 import "fmt"
4 import "reflect"
5
6 func main() {
7     data := []string{"one", "two", "three"}
8     test(data)
9     moredata := []int{1, 2, 3}
10    test(moredata)
11 }
12
13 func test(t interface{}) {
14     switch reflect.TypeOf(t).Kind() {
15     case reflect.Slice:
16         s := reflect.ValueOf(t)
17
18         for i := 0; i < s.Len(); i++ {
19             fmt.Println(s.Index(i))
20         }
21     }
22 }
23
24
25
```



Handling Errors

1. Returning and Checking Errors

```
go

package main

import (
    "errors"
    "fmt"
)

func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("cannot divide by zero")
    }
    return a / b, nil
}

func main() {
    result, err := divide(10, 0)
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Println("Result:", result)
}
```

👉 Every call must check `err`.



Custom Error Message

2. Creating Custom Errors

```
go

var ErrNotFound = errors.New("item not found")

func findItem(id int) (string, error) {
    if id != 1 {
        return "", ErrNotFound
    }
    return "Laptop", nil
}

func main() {
    item, err := findItem(2)
    if err != nil {
        if errors.Is(err, ErrNotFound) {
            fmt.Println("Custom Error:", err)
        }
        return
    }
    fmt.Println("Found:", item)
}
```



Wrapping Errors (Go 1.13+)

Use `fmt.Errorf` with `%w` to keep the original error for inspection.

```
go

func readFile() error {
    // simulate error
    return errors.New("disk not found")
}

func processFile() error {
    err := readFile()
    if err != nil {
        return fmt.Errorf("processFile failed: %w", err)
    }
    return nil
}

func main() {
    err := processFile()
    if err != nil {
        fmt.Println("Error:", err)
    }
}
```



Error Nil

```
package main

import (
    "errors"
    "fmt"
)

func boom() error {
    return errors.New("barnacles")
}

func main() {
    err := boom()

    if err != nil {
        fmt.Println("An error occurred:", err)
        return
    }

    fmt.Println("Anchors away!")
}
```



Error Along Value

```
package main

import (
    "errors"
    "fmt"
    "strings"
)

func capitalize(name string) (string, error) {
    if name == "" {
        return "", errors.New("no name provided")
    }
    return strings.ToTitle(name), nil
}
```



Error Along Value

```
func main() {
    name, err := capitalize("sammy")
    if err != nil {
        fmt.Println("Could not capitalize:", err)
        return
    }
    fmt.Println("Capitalized name:", name)
}
```



Error Propagation

- Refer amexws error propagation



Handling Panics in Go

- Errors that a program encounters fall into two broad categories: those the programmer has anticipated and those the programmer has not.
- The error largely deal with errors that we expect as we are writing Go programs.
- The error interface even allows us to acknowledge the rare possibility of an error occurring from function calls, so we can respond appropriately in those situations.



Handling Panics in Go

- Panics fall into the second category of errors, which are unanticipated by the programmer.
- These unforeseen errors lead a program to spontaneously terminate and exit the running Go program.
- Common mistakes are often responsible for creating panics.



Handling Panics in Go

- There are certain operations in Go that automatically return panics and stop the program.
- Common operations include indexing an array beyond its capacity, performing type assertions, calling methods on nil pointers, incorrectly using mutexes, and attempting to work with closed channels.
- Most of these situations result from mistakes made while programming that the compiler has no ability to detect while compiling your program.
- Since panics include detail that is useful for resolving an issue, developers commonly use panics as an indication that they have made a mistake during a program's development.



Handling Panics in Go

```
go

func safeDivide(a, b int) int {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }
    }()
}

return a / b // panic if b == 0
}

func main() {
    fmt.Println("Result:", safeDivide(10, 2)) // 5
    fmt.Println("Result:", safeDivide(10, 0)) // recovers instead of crashing
}
```



Panic and Recover in Go

```
func riskyOperation() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from panic:", r)
        }
    }()
    panic("something went very wrong")
}

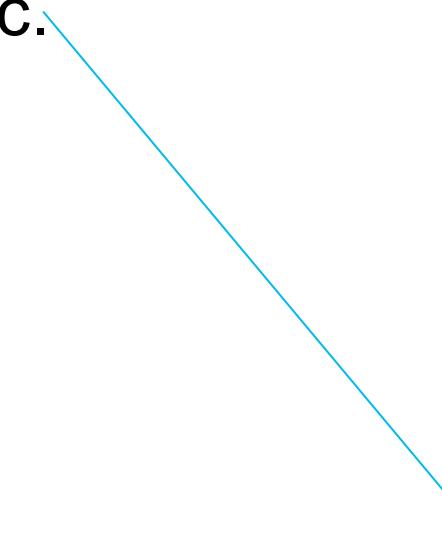
func main() {
    riskyOperation()
    fmt.Println("Program continues...")
}
```



Handling Panics in Go

- **Out of Bounds Panics**
- When you attempt to access an index beyond the length of a slice or the capacity of an array, the Go runtime will generate a panic.

```
func main() {  
    names := []string{  
        "lobster",  
        "sea urchin",  
        "sea cucumber",  
    }  
    fmt.Println("My favorite sea creature is:", names[len(names)])  
}
```





Nil Receivers

- The Go programming language has pointers to refer to a specific instance of some type existing in the computer's memory at runtime.
- Pointers can assume the value nil indicating that they are not pointing at anything.
- When we attempt to call methods on a pointer that is nil, the Go runtime will generate a panic.

```
func main() {  
    s := &Shark{"Sammy"}  
    s = nil  
    s.SayHello()  
}
```



RealTime Examples

```
func main() {
    data, err := os.ReadFile("missing.txt")
    if err != nil {
        if os.IsNotExist(err) {
            fmt.Println("File not found!")
        } else {
            fmt.Println("Other file error:", err)
        }
        return
    }
    fmt.Println("File content:", string(data))
}
```



Deferred Functions

- Our program may have resources that it must clean up properly, even while a panic is being processed by the runtime.
- Go allows you to defer the execution of a function call until its calling function has completed execution.
- Deferred functions run even in the presence of a panic and are used as a safety mechanism to guard against the chaotic nature of panics.
- Functions are deferred by calling them as usual, then prefixing the entire statement with the `defer` keyword, as in `defer sayHello()`.



Deferred Function

```
func main() {
    resp, err := http.Get("https://golang.org")
    if err != nil {
        fmt.Println("Network error:", err)
        return
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        fmt.Printf("HTTP Error: %d %s\n", resp.StatusCode, resp.Status)
        return
    }

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("Error reading body:", err)
        return
    }
    fmt.Println("Response:", string(body[:100]), "...")
}
```



Deferred Function

```
func main() {
    db, err := sql.Open("sqlite3", "test.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    var name string
    err = db.QueryRow("SELECT name FROM users WHERE id = ?", 1).Scan(&name)
    if err != nil {
        if err == sql.ErrNoRows {
            fmt.Println("No record found")
        } else {
            fmt.Println("Query error:", err)
        }
        return
    }

    fmt.Println("User:", name)
}
```



Go Workspace

Go module 02

Go Workspaces

Go module 3

Go module 01



Go Workspace

```
C:\Windows\System32\cmd.e + ^ Microsoft Windows [Version 10.0.22621.1555] (c) Microsoft Corporation. All rights reserved. G:\gowsdemo>mkdir newProject1 && cd newProject1 && go mod init newProject1 go: creating new go.mod: module newProject1 G:\gowsdemo\newProject1>cd .. G:\gowsdemo>mkdir newProject2 && cd newProject2 && go mod init newProject2 go: creating new go.mod: module newProject2 G:\gowsdemo\newProject2>cd .. G:\gowsdemo>go work init newProject1 newProject2 G:\gowsdemo>
```



Go Workspace

The screenshot shows a Go workspace interface. At the top, there's a toolbar with a file icon, the title "go.work", a close button ("X"), and a plus sign ("+"). Below the toolbar is a menu bar with "File", "Edit", and "View". The main area contains the following Go code:

```
go 1.19

use (
    ./newProject1
    ./newProject2
)
```



File Handling

- Intro
 - Everything is a File
- Basic Operations
 - Create Empty File
 - Truncate a File
 - Get File Info
 - Rename and Move a File
 - Delete Files
 - Open and Close Files
 - Check if File Exists
 - Check Read and Write Permissions
 - Change Permissions, Ownership, and Timestamps
 - Create Hard Links and Symlinks



File Handling

- Reading and Writing
 - Copy a File
 - Seek Positions in File
 - Write Bytes to a File
 - Quick Write to File
 - Use Buffered Writer
 - Read up to n Bytes from File
 - Read Exactly n Bytes
 - Read At Least n Bytes
 - Read All Bytes of File



File Handling

- Quick Read Whole File to Memory
 - Use Buffered Reader
 - Read with a Scanner
 - Archiving(Zipping)
 - Archive(Zip) Files
 - Extract(Unzip) Archived Files
 - Compressing
 - Compress a File
 - Uncompress a File
- Misc
 - Temporary Files and Directories
 - Downloading a File Over HTTP
 - Hashing and Checksums



Embedding interfaces in Go

- In Go language, the interface is a collection of method signatures and it is also a type means you can create a variable of an interface type.
- As Go language does not support inheritance, but the Go interface fully supports embedding.
- In embedding, an interface can embed other interfaces or an interface can embed other interface's method signatures in it.
- Any number of interfaces can be embedded in a single interface.
- Day3/Embeddingdemo.go



Bit Vector Type

- A bit vector is an array data structure that compactly stores bits.
- This library is based on 5 static different data structures:
- 8-bit vector: relies on an internal uint8
- 16-bit vector: relies on an internal uint16
- 32-bit vector: relies on an internal uint32
- 64-bit vector: relies on an internal uint64
- 128-bit vector: relies on two internal uint64 (for ASCII problems)
- The rationale of using a static integer compared to a dynamic []byte is first of all to save memory. There is no structure and/or slice overhead.



Bit Vector Type

- import "github.com/dropbox/godropbox/container/bitvector"
- Package bitvector provides the implementation of a variable sized compact vector of bits which supports lookups, sets, appends, insertions, and deletions.
- type BitVector
 - func NewBitVector(data []byte, length int) *BitVector
 - func (vector *BitVector) Append(bit byte)
 - func (vector *BitVector) Bytes() []byte
 - func (vector *BitVector) Delete(index int)
 - func (vector *BitVector) Element(i int) byte
 - func (vector *BitVector) Insert(bit byte, index int)
 - func (vector *BitVector) Length() int
 - func (vector *BitVector) Set(bit byte, index int)



Encapsulation in Golang

- Encapsulation is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.





Encapsulation in Golang

- Go language, encapsulation is achieved by using packages.
- Go provides two different types of identifiers, i.e. exported and unexported identifiers.
- Encapsulation is achieved by exported elements(variables, functions, methods, fields, structures) from the packages, it helps to control the visibility of the elements(variables, functions, methods, fields, structures).
- The elements are visible if the package in which they are defined is available in your program.



Exported Identifiers

- 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.
- This capital letter indicates that the given identifier is exported identifier.
- Exported identifiers are always limited to the package in which they are defined.
- When you export the specified identifier from the package you simple just export the name not the implementation of that identifier.
- This mechanism is also applicable for function, fields, methods, and structures.
- **// Exported Method**
- **res := strings.ToUpper(slc[x])**



Unexported Identifiers

- Unexported identifiers are those identifiers which are not exported from any package.
- They are always in lowercase.
- **// Unexported function**

```
func addition(val ...int) int {  
    s := 0  
    for x := range val {  
        s += val[x]  
    }  
    fmt.Println("Total Sum: ", s)  
    return s  
}
```



What Are Interface Types?

- An interface type specifies a collection of method prototypes.
- In other words, each interface type defines a method set.
- In fact, we can view an interface type as a method set. For any of the method prototype specified in an interface type, its name can't be the blank identifier `_`.
- We also often say that each interface type specifies a behavior set (represented by the method set specified by that interface type).



Interface Satisfaction

- A type satisfies an interface if it possesses all the methods the interface requires.
- For example, an `*os.File` satisfies `io.Reader`, `Writer`, `Closer`, and `ReadWrite`.
- A `*bytes.Buffer` satisfies `Reader`, `Writer`, and `ReadWrite`, but does not satisfy `Closer` because it does not have a `Close` method.
- As a shorthand, Go programmers often say that a concrete type “is a” particular interface type, meaning that it satisfies the interface.
- For example, a `*bytes.Buffer` is an `io.Writer`; an `*os.File` is an `io.ReadWriter`.



Parsing flags by flag value

- Using the flag package involves three steps:
- Define variables to capture flag values.
- Define the flags your Go application will use
- Finally, parse the flags provided to the application upon execution.
- Most of the functions within the flag package are concerned with defining flags and binding them to variables that you have defined.
- The parsing phase is handled by the Parse() function.
- `go run colortext.go -color`

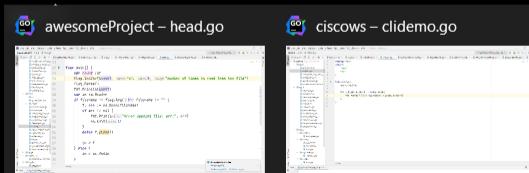
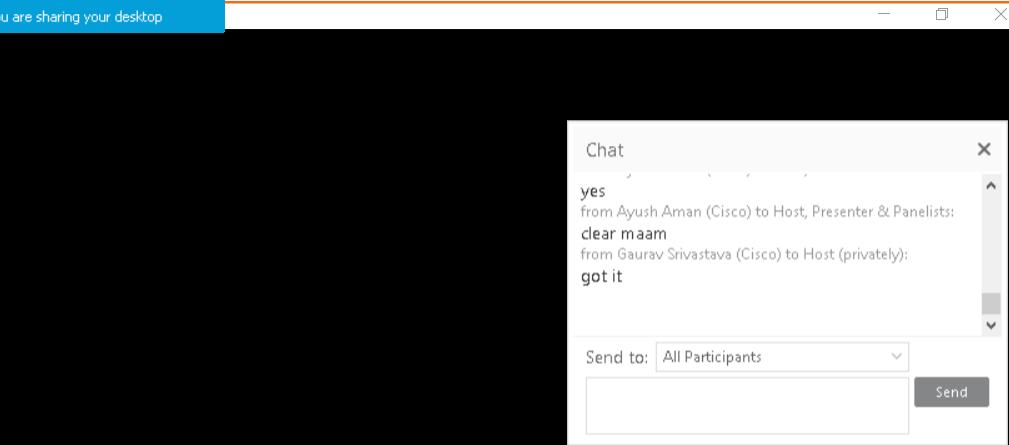


Parsing flags by flag value

```
c:\ Administrator: Command Prompt
import (
    "bufio"
    "flag"
)
F:\go\src\awesomeProject\Day3>go run head.go --n 25 -- head.go
25
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
//var fileName *string
//var count *int

func main() {
    var count int
    flag.IntVar(&count, "n", 5, "number of lines to read from the file")
    flag.Parse()
    fmt.Println(count)
    var in io.Reader
    if filename := flag.Arg(0); filename != "" {
        f, err := os.Open(filename)
        if err != nil {
            fmt.Println("error opening file: err:", err)
            os.Exit(1)
        }
        defer f.Close()
    }
}
F:\go\src\awesomeProject\Day3>
```





Parsing flags by flag value

```
F:\go\src\awesomeProject\Day3>go run head.go -- head.go
package main

import (
    "bufio"
    "flag"

F:\go\src\awesomeProject\Day3>go run flagstringint.go
env: development
port: 3000

F:\go\src\awesomeProject\Day3>go run flagstringint.go --port 8000
env: development
port: 8000

F:\go\src\awesomeProject\Day3>
```



The 3 ways to sort in Go

- Slice of ints, float64s or strings
- Custom comparator
- Custom data structures
- Bonus: Sort a map by key or value
- Performance and implementation



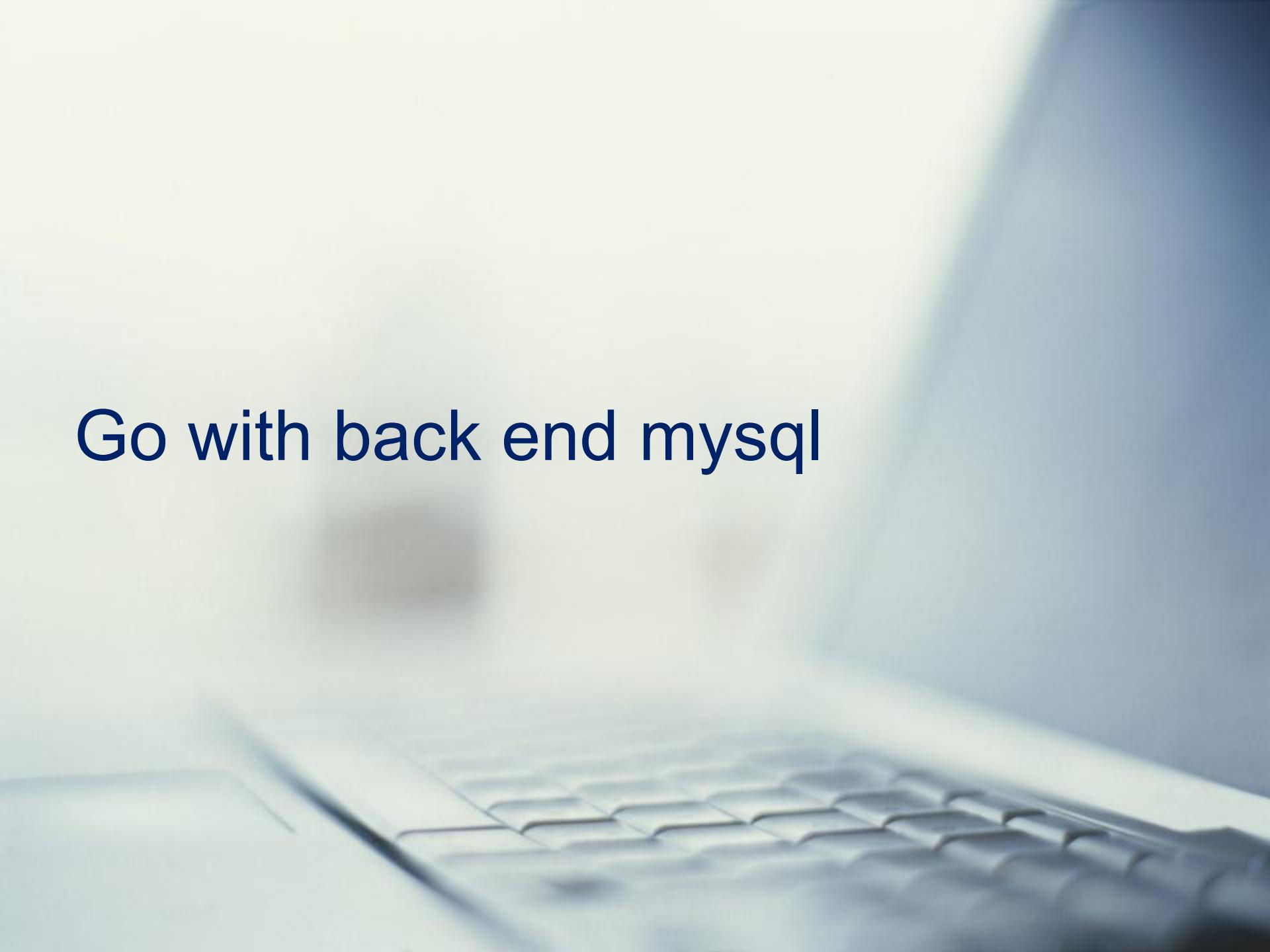
The 3 ways to sort in Go

```
// --- Custom Sort by Type ---
type ByPolicyType []models.Policy

func (p ByPolicyType) Len() int          { return len(p) }
func (p ByPolicyType) Swap(i, j int)     { p[i], p[j] = p[j], p[i] }
func (p ByPolicyType) Less(i, j int) bool { return p[i].Type < p[j].Type }

func main() {
    // Available policy types
    types := []string{"HEALTH", "MOTOR", "LIFE", "TRAVEL"}

    // Generate random policies
    var policies []models.Policy
    for i := 1; i <= 10; i++ {
        var p models.Policy
        p.Type = types[rand.Intn(len(types))]
```

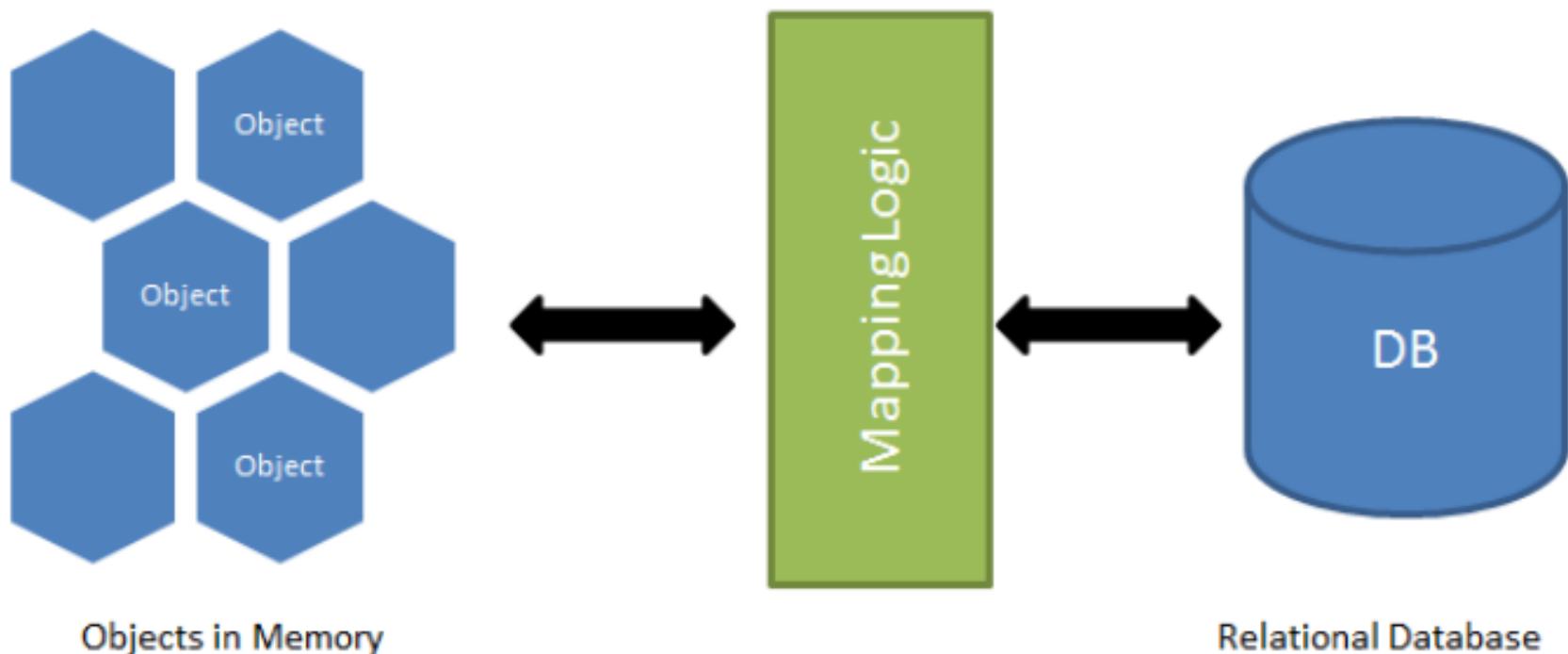
The background of the slide is a soft-focus photograph of a laptop computer. The keyboard is visible at the bottom, and the screen above it displays a light blue or white interface, likely a presentation slide. The overall aesthetic is clean and professional.

Go with back end mysql



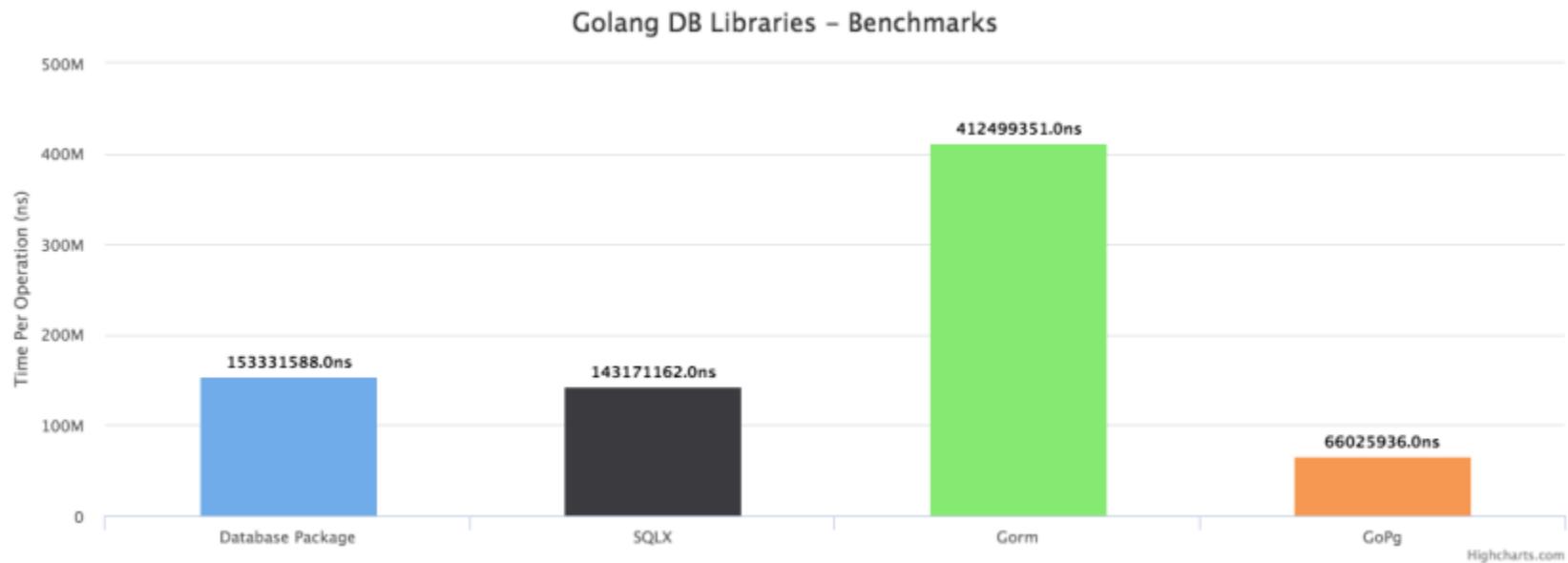
Go ORM

O/R Mapping





Go ORM





Go Regular Expressions

```
import (
    "fmt"
    "regexp"
)

func main() {
    // Compile regex (must escape backslashes)
    re := regexp.MustCompile(`\d+`)

    text := "Order123, Invoice456, Ref789"

    // Find all matches
    matches := re.FindAllString(text, -1)

    fmt.Println(matches) // [123 456 789]
}
```



Regular Expressions

Choice and grouping

Regexp	Meaning
xy	x followed by y
$x y$	x or y, prefer x
$xy z$	same as $(xy) z$
xy^*	same as $x(y^*)$

Repetition (greedy and non-greedy)

Regexp	Meaning
x^*	zero or more x, prefer more
$x^*?$	prefer fewer (non-greedy)
x^+	one or more x, prefer more
$x^+?$	prefer fewer (non-greedy)
$x^?$	zero or one x, prefer one
$x^{??}$	prefer zero
$x^{\{n\}}$	exactly n x



Regular Expressions

Character classes

Expression	Meaning
.	any character
[ab]	the character a or b
[^ab]	any character except a or b
[a-z]	any character from a to z
[a-zA-Z0-9]	any character from a to z or 0 to 9
\d	a digit: [0-9]
\D	a non-digit: [^0-9]
\s	a whitespace character: [\t\n\f\r]
\S	a non-whitespace character: [^\t\n\f\r]
\w	a word character: [0-9A-Za-z_]
\W	a non-word character: [^0-9A-Za-z_]
\p{Greek}	Unicode character class*
\pN	one-letter name
\P{Greek}	negated Unicode character class*
\PN	one-letter name



Regular Expressions

Special characters

To match a **special character** `\^$. | ?*+-[]{}()` literally, escape it with a backslash. For example `\{` matches an opening brace symbol.

Other escape sequences are:

Symbol	Meaning
<code>\t</code>	horizontal tab = <code>\011</code>
<code>\n</code>	newline = <code>\012</code>
<code>\f</code>	form feed = <code>\014</code>
<code>\r</code>	carriage return = <code>\015</code>
<code>\v</code>	vertical tab = <code>\013</code>
<code>\123</code>	octal character code (up to three digits)
<code>\x7F</code>	hex character code (exactly two digits)



Regular Expressions

Text boundary anchors

Symbol	Matches
\A	at beginning of text
^	at beginning of text or line
\$	at end of text
\z	
\b	at ASCII word boundary
\B	not at ASCII word boundary



Regular Expressions

- `MatchString` – checks if text matches

```
go
```

```
matched, _ := regexp.MatchString(`foo`, "seafood")
fmt.Println(matched) // true
```

- `FindString` – returns first match

```
go
```

```
re := regexp.MustCompile(`\d+`)
fmt.Println(re.FindString("abc123xyz")) // 123
```



Regular Expressions

- `FindAllString` – returns all matches

```
go
```

```
fmt.Println(re.FindAllString("a1 b22 c333", -1)) // [1 22 333]
```

- `ReplaceAllString` – replace matches

```
go
```

```
fmt.Println(re.ReplaceAllString("a1 b22 c333", "#"))
// a# b# c#
```



Regular Expressions

- Groups (capturing parentheses)

```
go
```

```
re := regexp.MustCompile(`(\w+)@(\w+)\.(\w+)`)  
match := re.FindStringSubmatch("test@example.com")  
  
fmt.Println(match[0]) // test@example.com  
fmt.Println(match[1]) // test  
fmt.Println(match[2]) // example  
fmt.Println(match[3]) // com
```



Regular Expressions

Pattern	Meaning	Example Match
\d+	One or more digits	123
[a-zA-Z]+	Letters only	Hello
^abc	Starts with abc	abcdef
xyz\$	Ends with xyz	abcxyz
\w+@\w+\.\w+	Simple email format	a@b.com



Regular Expressions

📌 5. Compile Once for Efficiency

If used often, compile regex once with `MustCompile` (panic on error) or `Compile` (returns error).

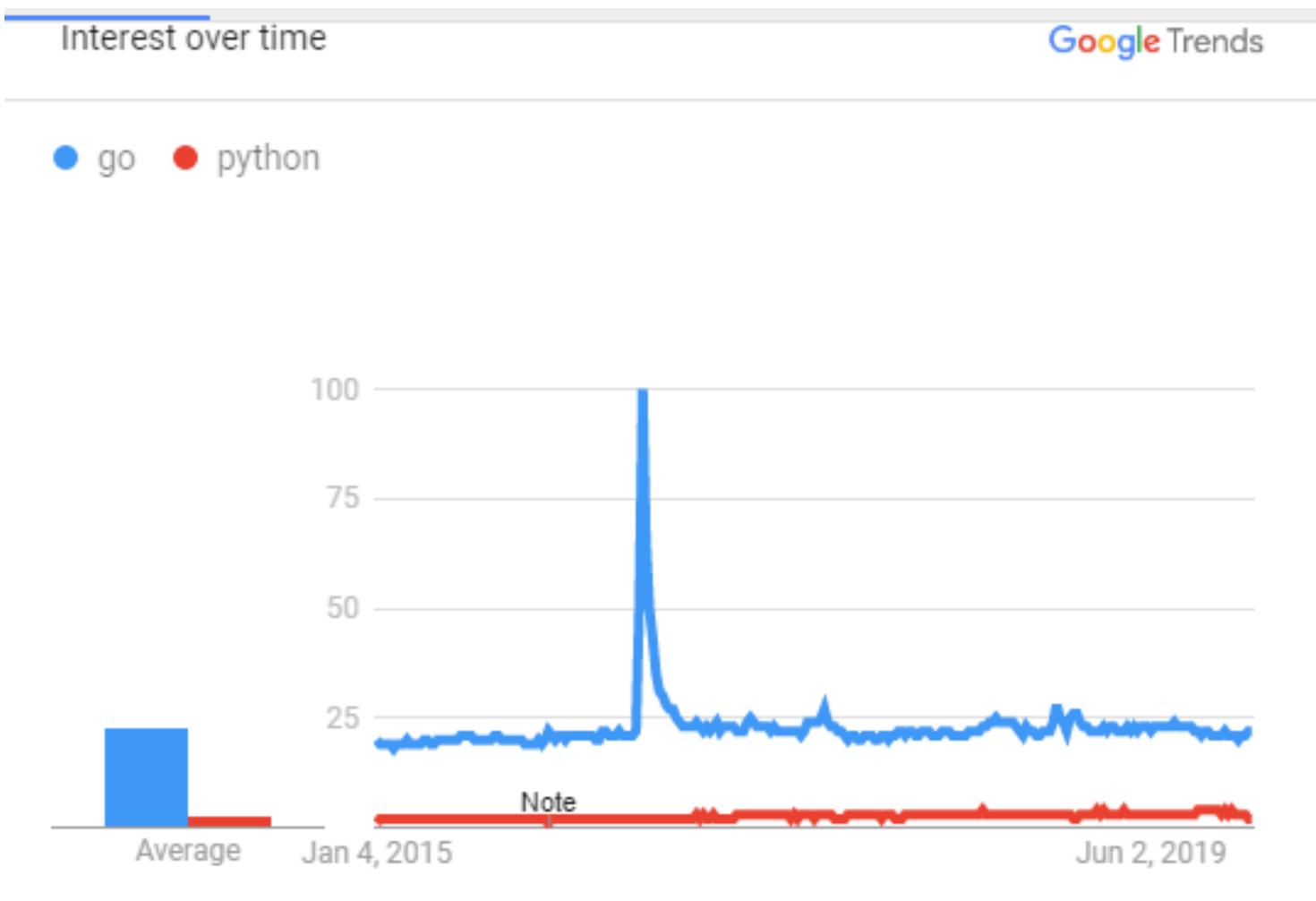
```
go

var re = regexp.MustCompile(`\d{4}-\d{2}-\d{2}`)

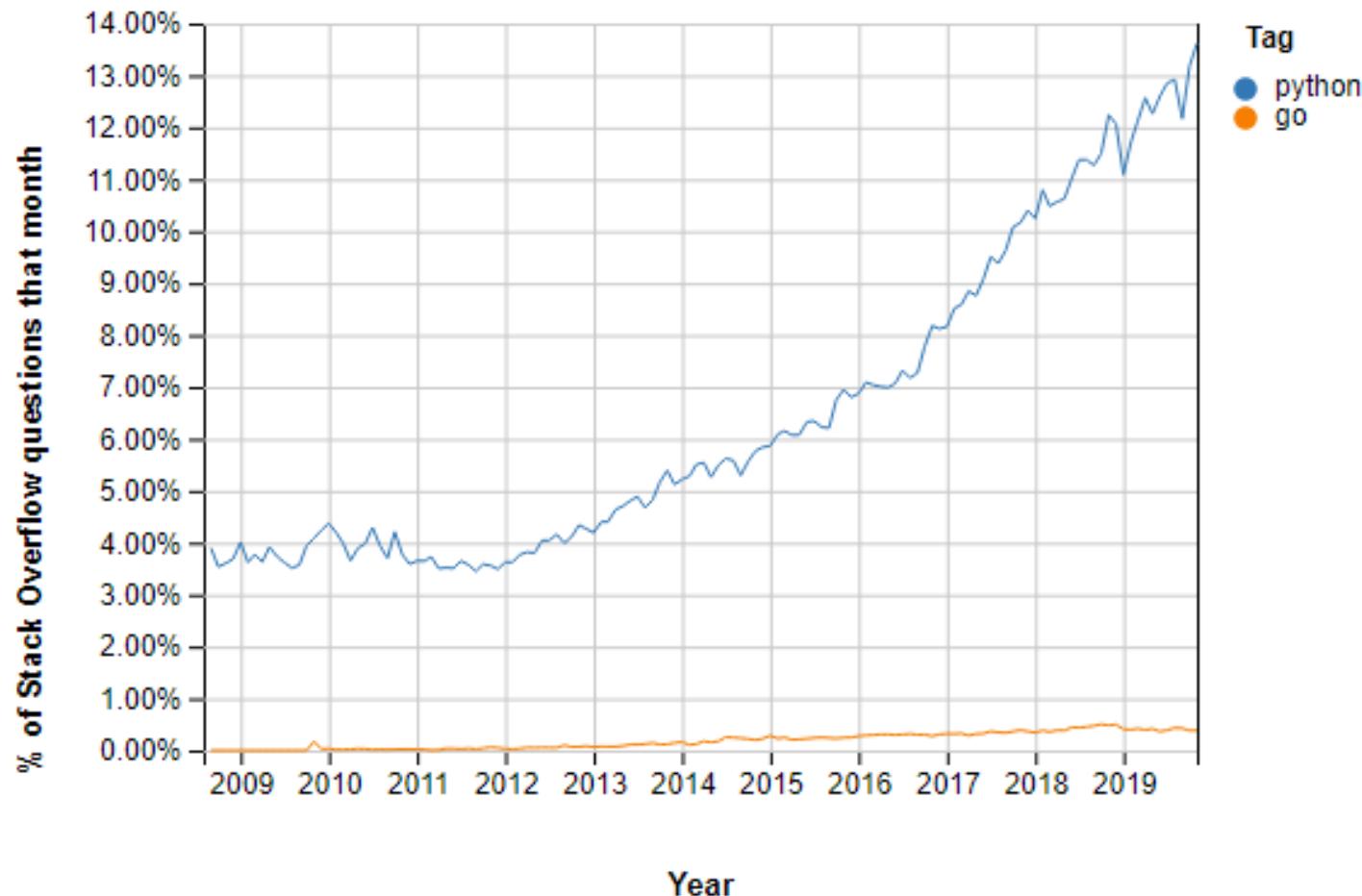
func isDate(s string) bool {
    return re.MatchString(s)
}
```



GO Community vs Python Community

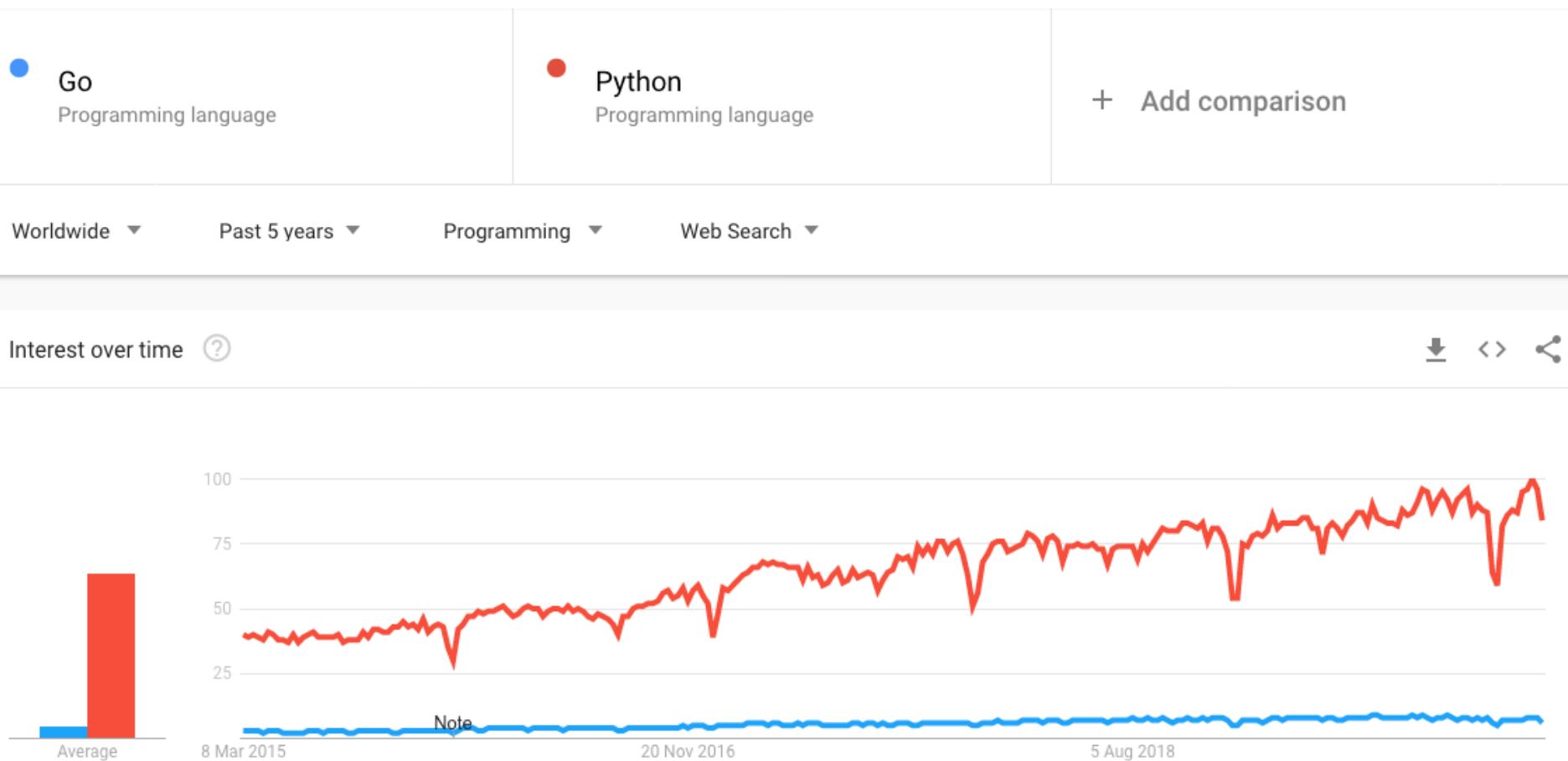


GO Community vs Python Community





GO Community vs Python Community





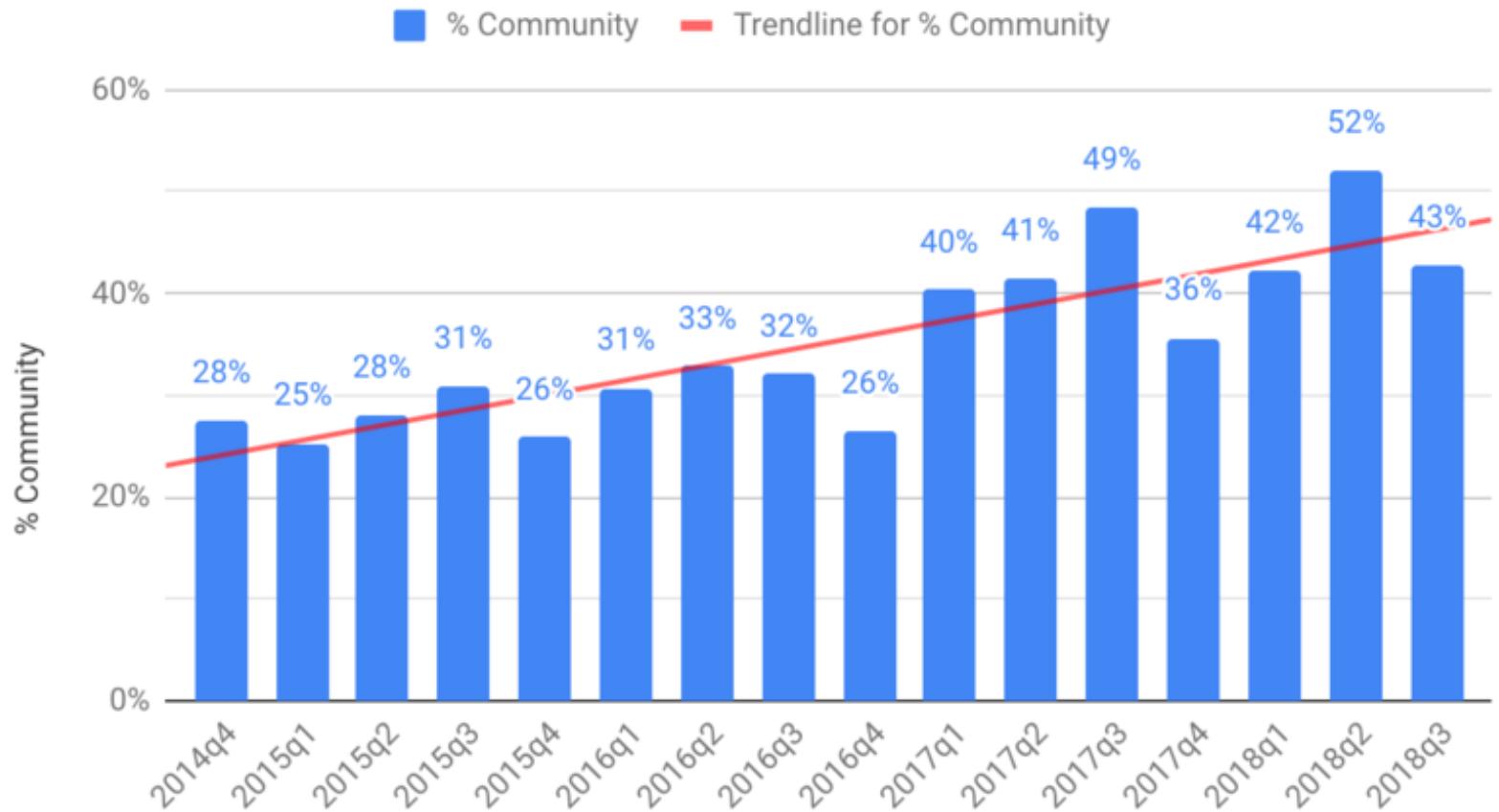
GO Community vs Python Community





GO Community vs Python Community

% of commits from the community





Go charts

```
> -eo main.go > ⚙ main

func main() {

    // Claim types
    types := []string{"HEALTH", "MOTOR", "LIFE", "TRAVEL"}

    // Generate fake claims
    var claims []Claim
    for i := 1; i <= 50; i++ {
        c := Claim{
            ID:      fmt.Sprintf("C%03d", i),
            Type:   types[rand.Intn(len(types))],
            Amount: gofakeit.Float64Range(1000, 90000), // 1k - 90k
        }
        claims = append(claims, c)
    }

    // Count claims per type
    counts := make(map[string]int)
    for _, c := range claims {
        counts[c.Type]++
    }

    // Prepare chart data
    var xAxis []string
    var yAxis []opts.BarData
    for _, t := range types {
        xAxis = append(xAxis, t)
        yAxis = append(yAxis, opts.BarData{Value: counts[t]})
    }

    // Create bar chart
    bar := charts.NewBar()
```

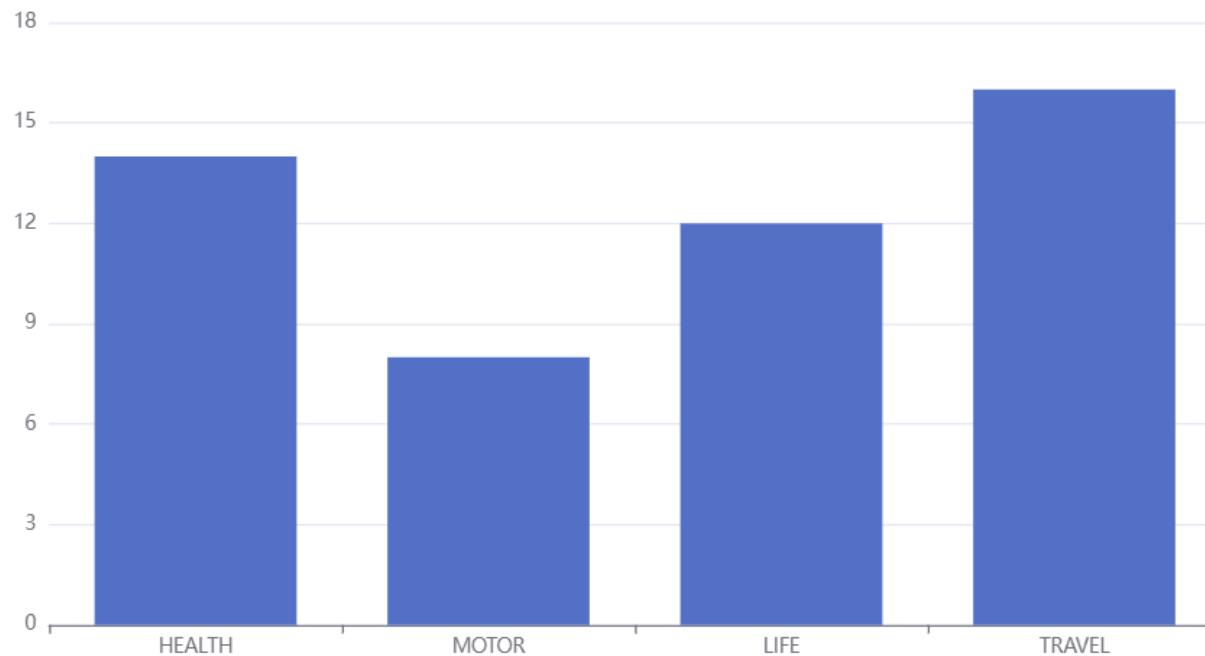


Go charts

Claims by Type

Generated using faker & random

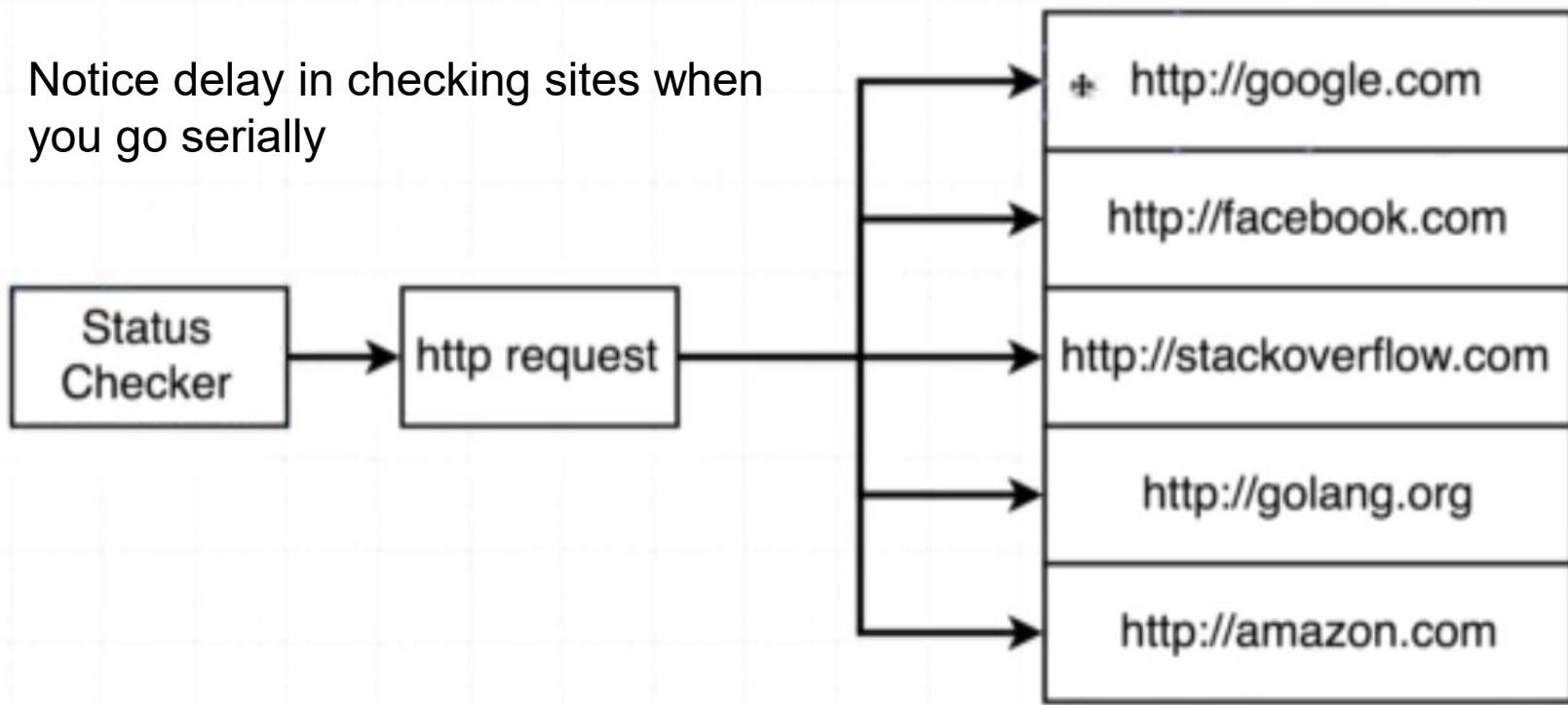
Claims



Website Status Checker

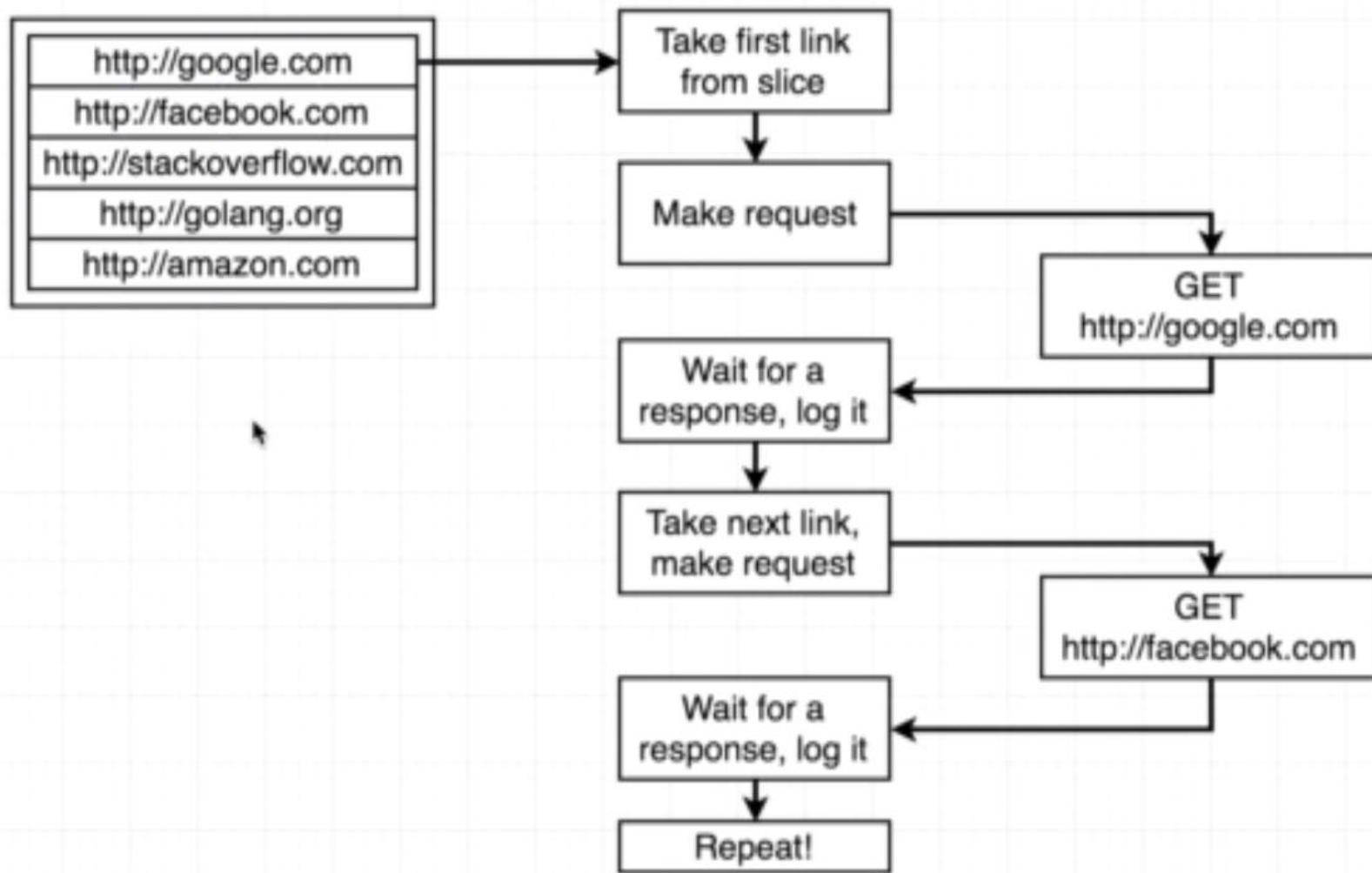
Serialchecker Ref

Notice delay in checking sites when you go serially

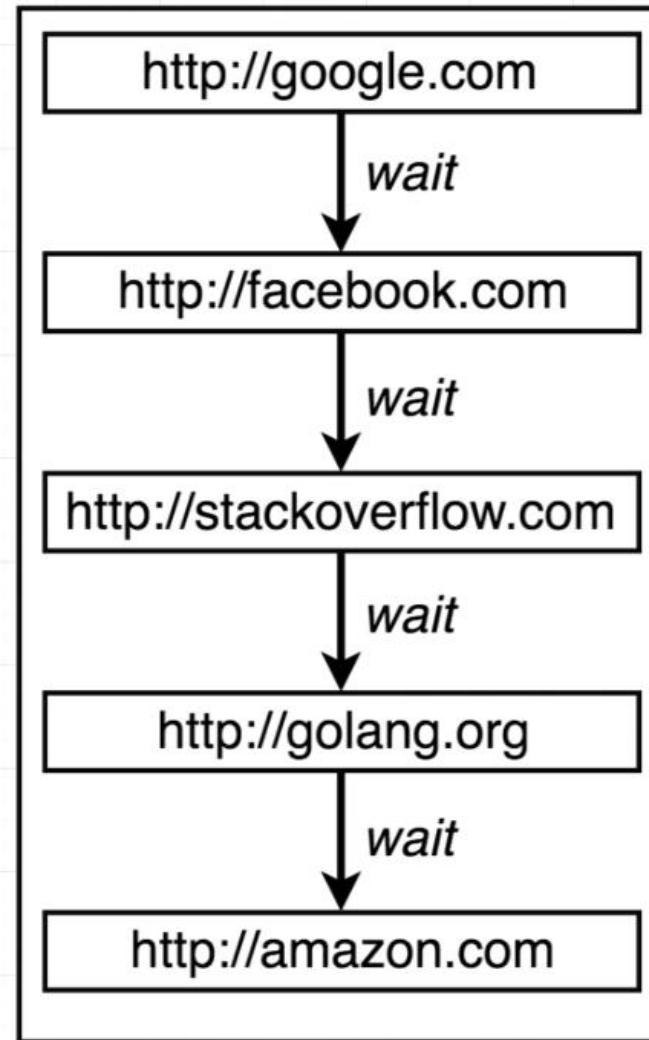




Website Status Checker

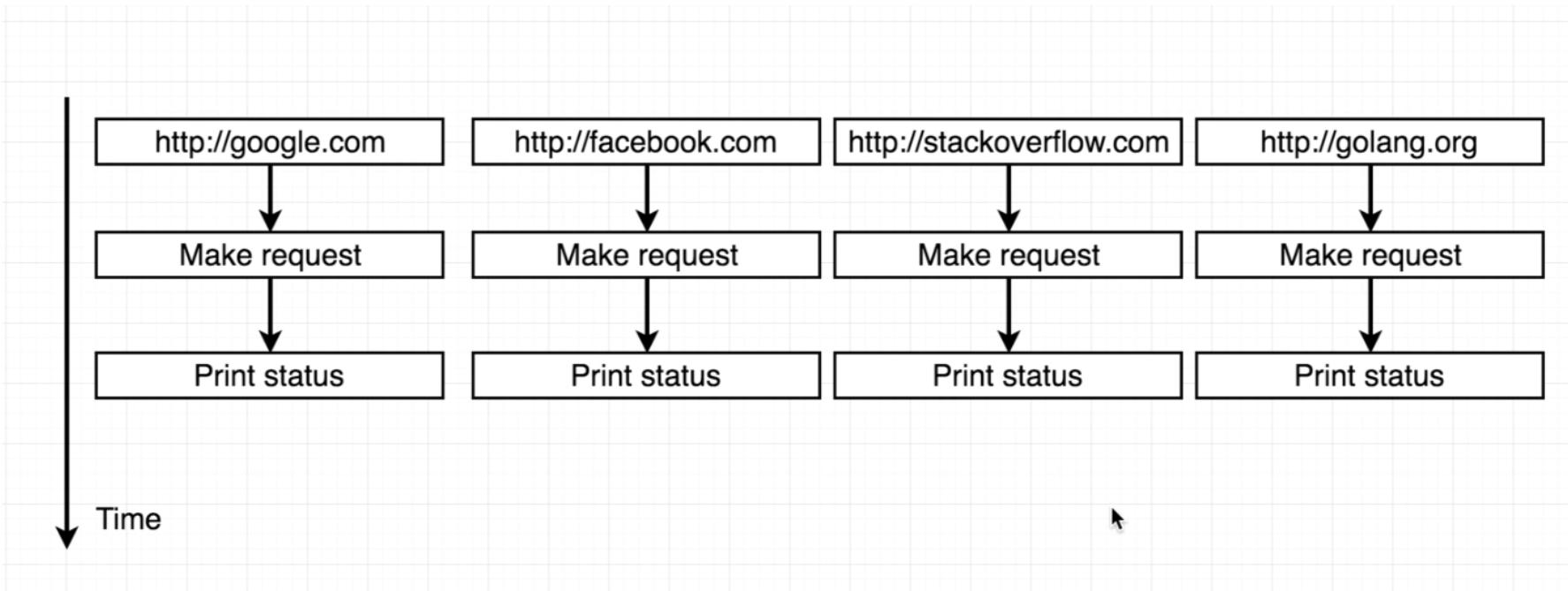


Website Status Checker





Go Routine Solution





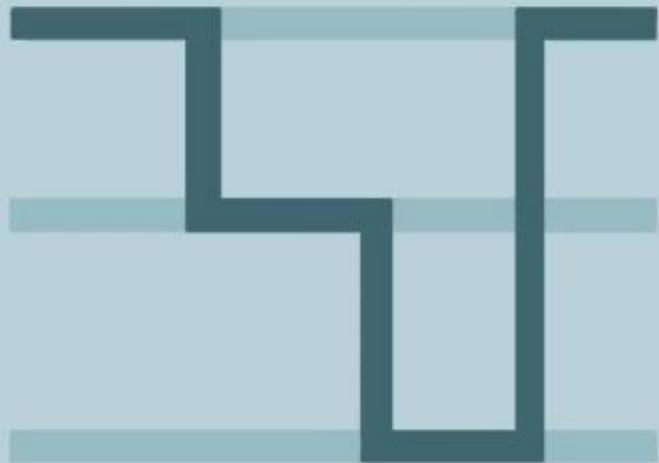
Understanding Concurrency vs Parallelism

Feature	Concurrency	Parallelism
Definition	Handling multiple tasks at once	Executing multiple tasks simultaneously
CPU cores needed	Works even with 1 core	Requires ≥ 2 cores
How it works	Task switching / scheduling	True parallel execution
Analogy	1 chef multitasking	Many chefs working at the same time
Go relation	Goroutines with scheduler	Goroutines + multiple OS threads/cores

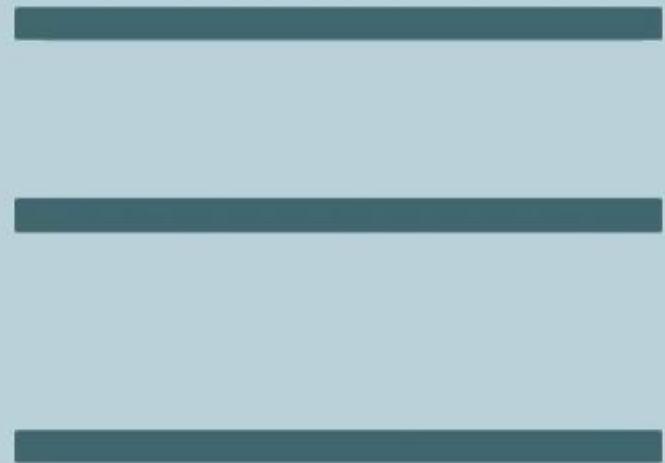


Understanding Concurrency

Concurrent

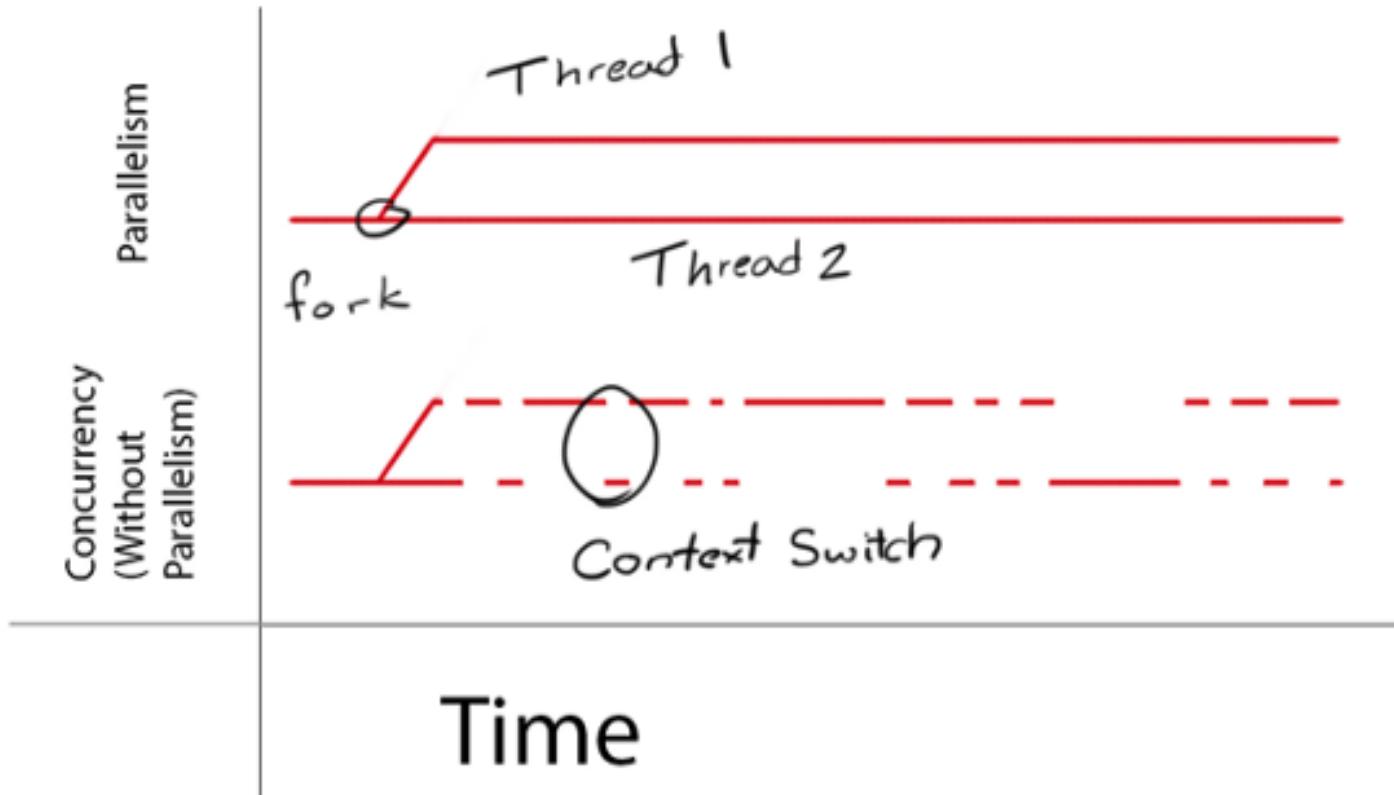


Parallel



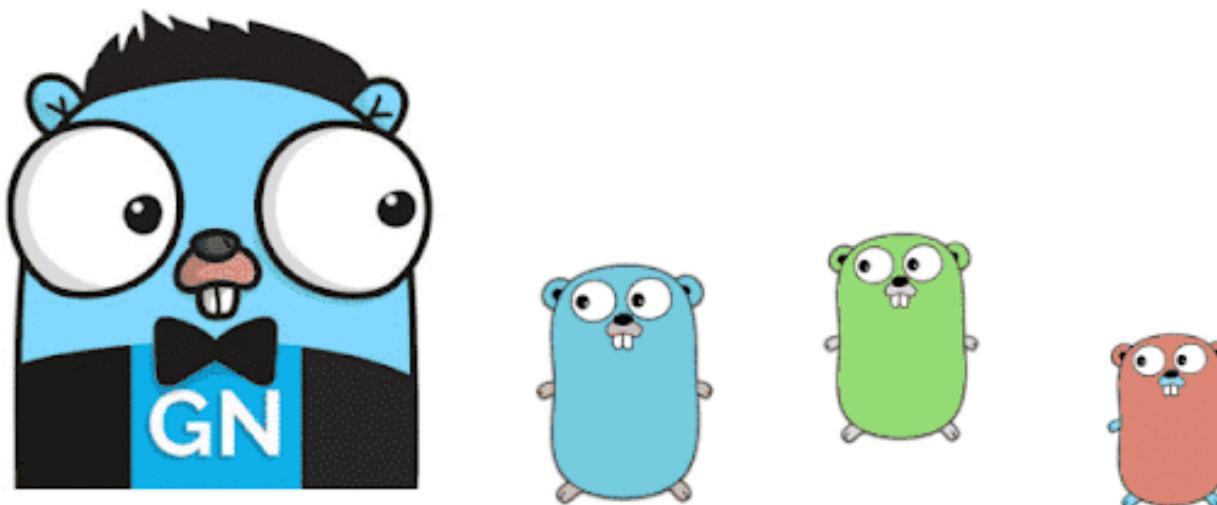


Understanding Concurrency



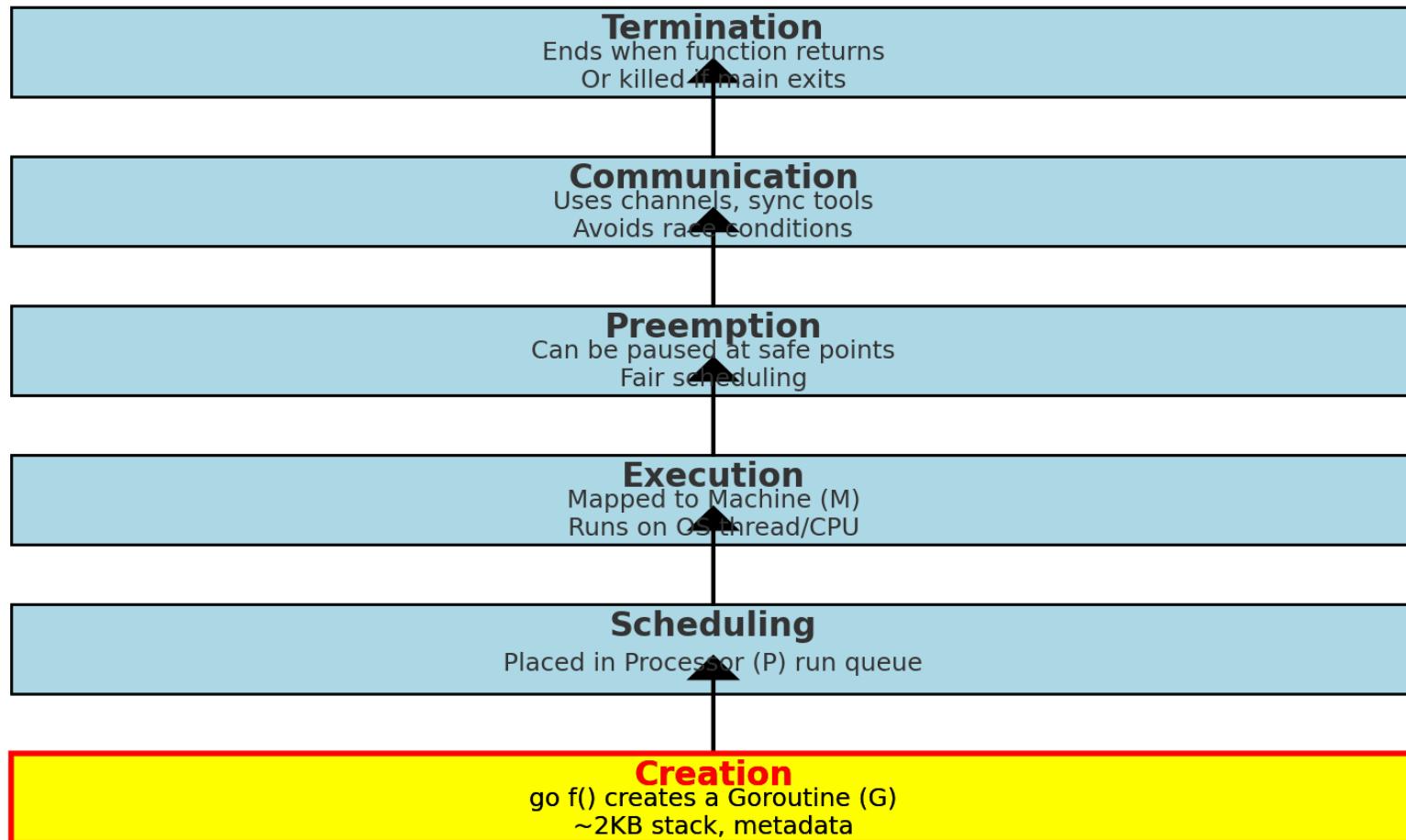


Go Routines





Goroutine





Go Routines

- Go Routines are a Golang wrapper on top of threads and managed by Go runtime rather than the operating system.
- Go runtime has the responsibility to assign or withdraw memory resources from Goroutines.
- A Goroutine is much like a thread to accomplish multiple tasks but consumes fewer resources than OS threads.
- Goroutine does not have a one-to-one relationship with threads.



Go Routines

-  **Goroutines**
- **Managed by:** Go runtime scheduler (G-M-P model).
- **Dependency:** Not hardware-dependent; multiplexed onto OS threads.
- **Communication:** Channels (built-in CSP style, low-latency sync).
- **ID:** No stable exposed ID (runtime avoids TLS overhead).
- **Cost:** Very cheap (~2–4 KB initial stack, growable).
- **Scheduling:** Cooperatively at safe points + async preemption.
- **Startup:** Fast creation (microseconds).
- **Stack:** Segmented & growable (shrinks too).
- **Scaling:** Millions possible in one process.



Go Routines

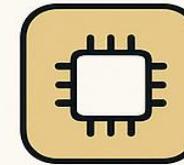
- **Threads**
- **Managed by:** OS kernel.
- **Dependency:** Hardware-dependent (OS/hardware threads).
- **Communication:** Shared memory, mutexes, condition vars (more boilerplate, higher latency).
- **ID:** Each thread has a unique OS ID (used with TLS).
- **Cost:** Expensive (MB-sized stack reserved).
- **Scheduling:** Preemptive (by OS kernel).
- **Startup:** Slower (tens/hundreds of μs).
- **Stack:** Fixed (usually 1–8 MB).
- **Scaling:** Thousands practical, not millions.



Go Routines



Goroutine



Thread

Managed by Go runtime	Managed by OS kernel
Not hardware-dependent	Hardware-dependent
Uses channels	Uses shared memory
Smaller stacks (~2 KB)	Larger stacks (MB)



Go Routines

Aspect	Goroutines	Threads
Management	Go runtime	OS Kernel
Stack	~2KB growable	MB fixed
Count	Millions possible	Thousands practical
Communication	Channels	Shared memory / sync primitives
Scheduling	G-M-P model	OS scheduler
Startup	Very fast	Slower
Cost	Very low	Higher

Questions





Module Summary

- In this module we discussed
 - Go Modules
 - Goroutines
 - Go Tools

