Go Bootcamp: The Ultimate Guide

Basics -> Intermediate -> Advanced

by CodeOvation

- Go is a statically typed, compiled high-level programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson.
- Syntactically similar to C.
- Also has memory safety and garbage collection.
- One of the most efficient language and runtime available currently.
- Competitor to Rust, which is also a memory safe language and runtime.

- Concurrency
- Performance
- Easy syntax
- Go runtime
- Type Safety

- IDE (Integrated Development Environment)
- VS Code
- Atom
- Vim / NeoVim
- Sublime Text
- Notepad++

GIT

- Git is a distributed version control system that helps track changes in files and coordinate work among multiple people.
- It allows for collaboration on projects, maintaining a history of changes, and reverting to previous versions if needed.
- Git was created by Linus Torvalds, who is also known for creating the Linux kernel. He developed Git in 2005 to manage the source code of the Linux kernel project more efficiently. Git quickly gained popularity due to its speed, flexibility, and powerful branching and merging capabilities, becoming one of the most widely used version control systems in the world.
- Repository: A repository (or repo) is a collection of files and their revision history. It's where all project files and version history are stored.
- Commit: A commit is a snapshot of your repository at a specific point in time. It records changes to files and includes a commit message describing the changes.
- Branch: A branch is a separate line of development. It allows you to work on new features or fixes without affecting the main codebase (often called the master branch).

GIT (contd.)

- Merge: Merging combines changes from different branches into one branch. It's used to integrate code changes made in different branches.
- Clone: To clone a repository means to create a local copy of a remote repository (typically hosted on a server like GitHub or GitLab) on your computer.
- Pull: Pulling updates your local repository with changes from a remote repository.
- Push: Pushing sends your committed changes from your local repository to a remote repository.
- Fetch: Fetching updates your local repository with changes from a remote repository without merging them into your working files.
- git init: Initializes a new Git repository in the current directory.
- git clone: Clones a remote repository into a new directory on your local machine.
- git add: Adds files or changes to the staging area for the next commit.
- git commit: Records changes to the repository with a commit message.
- git push: Sends committed changes to a remote repository.
- git pull: Fetches and merges changes from a remote repository to your local repository.
- git branch: Lists, creates, or deletes branches.
- git merge: Merges changes from different branches.
- git checkout: Switches branches or restores working files.

GIT (contd.)

- Remote Repositories:
    1. Platforms like GitHub, GitLab, and Bitbucket host remote repositories.
    2. They facilitate collaboration by providing features like pull requests, issue tracking, and code reviews.
- Commands we will use frequently in our course
    - git init
    - git add
    - git commit
    - git push
    - git remote -v
    - git status

‘import’ statement

- integrates necessary libraries into the source code impacting the resulting executable binary
- Importing a library in Go ensures that only the functions actually used in the program are included in the executable, optimizing its size and performance
- Tree Shaking: Go's toolchain performs a form of tree shaking, where unused code and functions that are imported but never referenced in your program are omitted from the final executable. This optimization helps in keeping the executable size minimal.

Tree Shaking

- Tree shaking works by analyzing the static structure of the code, typically during the build process. It traces which modules and functions are imported and used directly.
- React: React, along with tools like Webpack and Rollup, leverages tree shaking to eliminate unused React components and utility functions from the final bundled JavaScript. This optimization is crucial in large React applications to keep the bundle size manageable.
- Angular: Angular uses tree shaking to remove unused parts of its framework and third-party libraries, ensuring that only the necessary parts are included in the final build.
- Modules or functions that are imported but never actually used in the application are identified as dead code. Tree shaking then removes these dead code segments from the final output bundle.
- Tree shaking is a powerful optimization technique that helps in delivering leaner, faster, and more efficient JavaScript applications by removing unused code during the build process. It plays a critical role in modern web development, particularly in frameworks and libraries where modularization and performance are paramount.

Data Types

- Integers
- Floating Point Numbers
- Complex Numbers
- Booleans
- Strings
- Constants
- Arrays
- Structs
- Pointers
- Maps
- Slices
- Functions
- Channels
- JSON
- Text and HTML Templates

Loop - For Loop

- Unlike other languages which have for, for while, for each, for in, while loops, etc., Go has only one keyword for loops which is “For”.
- The for statement allows us to repeat a list of statements (a block) multiple times.
- This makes it easier and extremely convenient for programmers to use it as they please.
- The versatility of For loop can as far as your imagination.
- for initialization; condition; post { // zero or more statements }
- Any of these parts may be omitted from the for loop.
- We will see various ways of using the for loop as we go along in the course.
- Example:

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

Conditions - If else

- It is the sheer beauty and simplicity of Go language that it has broken tradition and simplified programming.
- ‘if else’ condition is no exception to this statement.
- if statements have an optional else part.
- While we see if elif and some other kind of variations of conditionals in other programming languages, Go has kept it simple.
- We get a simple straightforward ‘if else’ statement for applying conditions, whether we want to chain the conditions or nest them, ‘if else’ does everything.

Arrays

- An array is a fixed length sequence of zero or more elements of a particular type.
- Arrays are rarely used directly in Go.
- Slices, which can grow and shrink, are much more versatile, but to understand slices we must underst and arrays first.
- Arrays are declared using the syntax: var arr [size]Type, where size is a constant expression.
- Elements of an array are automatically initialized to their zero value (0 for numeric types, empty string for strings, etc.).
- Elements of an array are accessed using zero-based indexing: arr[index].
- Index must be within the bounds of the array (0 to size-1).
- Arrays can be initialized with values using an array literal syntax: arr := [size]Type{elem1, elem2, ...}.
- The size can be omitted if it's determined by the number of elements in the literal.

Slices

- Slices, can grow and shrink, and are much more versatile than arrays.
- Slices are an important data type in Go, giving a more powerful interface to sequences than arrays.
- Slices represent variable length sequences whose elements all have the same type.
- A slice type is written [ ]T, where the elements have type T. It looks like an array type without a size.
- A slice is a lightweight datastructure that gives access to a subsequence (or perhaps all) of the elements of an array, which is known as the slice’s underlying array.
- Arrays have a fixed size determined at compile-time, whereas slices are dynamically sized and more flexible.
- Slices are more commonly used in Go for variable-length collections of data.

Maps

- Maps are Go's built-in Associative array type (also called hashes/hash table or dicts, dictionaries in other languages).
- The hash table is one of the most ingenious and versatile of all data structures.
- It is an unordered collection of key/value pairs in which all the keys are distinc t, and the value associated with a given key can be ret rieved, updated, or removed using a constant number of key comparisons on the average, no matter how large the hash table.
- In Go, a map is a reference to a hash table, and a map type is written map[K]V, where K and V are the types of its keys and values.
- Keys cannot be changed once set, only values associated with those keys can be modified/ changed.
- New keys can be added dynamically but keys can NEVER be modified as they are constants.
- All of the keys in a given map are of the same type, and all of the values are of the same typ e, but the keys need not be of the same typ e as the values.
- The key type K must be comparable (important term, when used as generic, an example will come later) using ==, so that the map can test whether a given key is equal to one already within it.

Range

- Range iterates over anything iterable.
- It an be elements in a variety of data structures.
- Range can also iterate over just the keys of a map.
- Range on map iterates over key/value pairs.
- Range on arrays/slices provides both the index and value for each entry.
- Range on strings iterates over Unicode code points.
- Range can be used on Arrays, Slices, Maps, Array of Structs, etc.

Example:

```
nums := [ ]int{2, 3, 4}

for i, num := range nums {
    if num == 3 {
        fmt.Println("index:", i)
    }
}
```

Multiple Return Values

- The most common use of multiple return values in Go, is to return error when the function fails to return it's desired value.
- Multiple return value from a function is not limited to just 2 values but any number of values that you want returned from a function.
- If you only want a subset of the returned values, use the blank identifier _ to receive those values which you do not want.

Defer

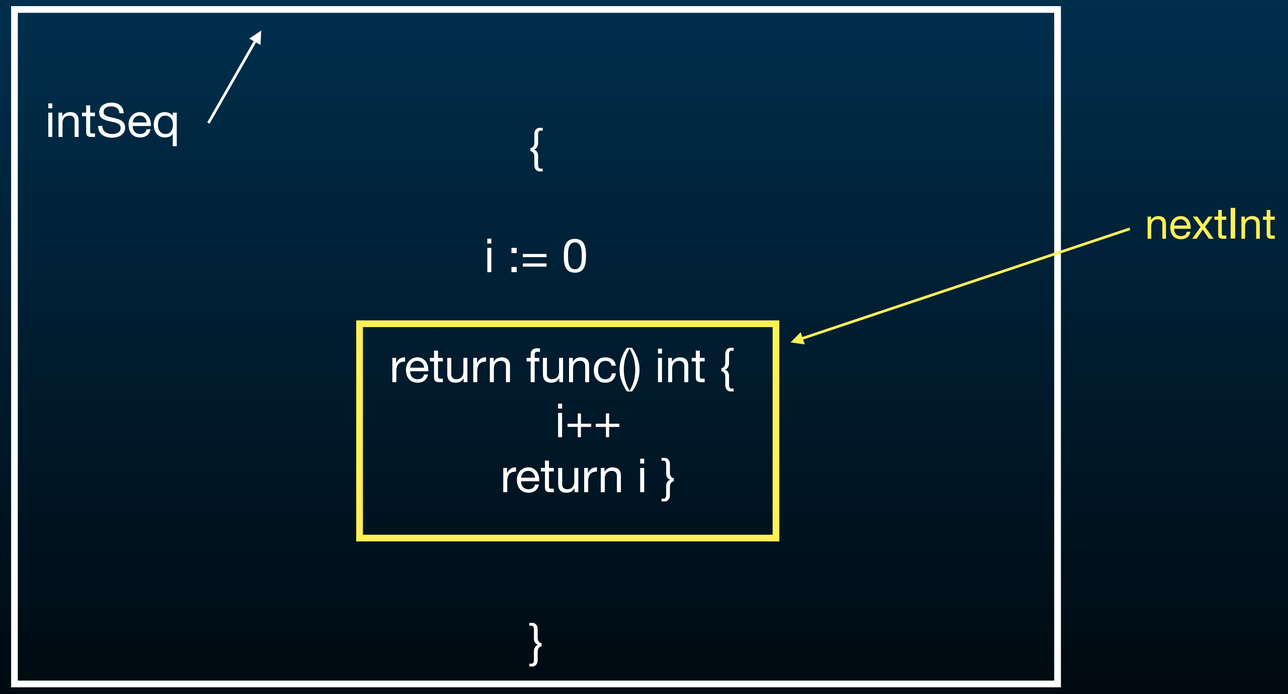
- Whenever we want a code block to run right before the function is about to return, we use the defer keyword to declare a function.
- Clean up actions and closing of any open files, or any other task.
- Works like finally clause in other languages (eg. try…catch…finally in Javascript)
- Can be declared anywhere in the function, even at the start but will always run at the end but right before the return statement execution.
- If a function does not return anything, then defer will run at the end.

Closures

- Go supports anonymous functions which can form closures.
- Anonymous functions are useful when you want to define a function inline without having to name it.
- A closure in Go is a function value that references variables from outside its body.
- It "closes over" these variables, capturing their current state.
- Closures are created by defining a function inside another function and accessing variables from the outer function.
- The inner function retains access to these variables even after the outer function has finished executing.
- Inside the closure, you can access and modify variables declared in the outer function.
- Changes made to these variables persist across multiple calls to the closure.
- Closures in Go are lexically scoped, meaning they retain the scope of the variables they reference when they are created.

Closures and Scope

- intSeq is a function that is returning another function.
- nextInt becomes the returned function without the initial value of i.
- value of i is stored in RAM/memory and now that value will keep getting updated as the returning function is programmed to access it and increase it.



Pointers

- Because calling a function makes a copy of each argument value, if a function needs to update a variable, or if an argument is so large that we wish to avoid copying it, we must pass the address of the variable using a pointer.
- pointer only stores memory address and not any other value (hex format)
- p *int stores address value of an integer type variable (pointer to int type)
- f *float will store address value of float type variable (pointer to float type)
- adding an asterisk to a pointer will dereference (result in value stored at the memory address)
- & denotes the address of the variable.

Example:

a := 90

var x *int = &a or x := &a

b := 18.18

var y *float = &b or y := &b

Runes

- In Go, a rune is an alias for the rune type, which represents a Unicode code point.
- A Unicode code point is a unique number assigned to each character in the Unicode standard.
- Go uses runes to represent individual characters, including those outside the ASCII range.
- Runes are encoded using UTF-8, which is the default encoding for Go source files and string literals.
- Rune literals are expressed using single quotes, like 'a' or '世'.
- Example: `var r rune = '😊'`

  `fmt.Println(r) // Output: 128522`
- When iterating over strings in Go, you iterate over runes rather than bytes to correctly handle multi-byte UTF-8 characters.

```
str := "Hello, 世界"
for _, r := range str {
    fmt.Printf("%c ", r) // Prints each rune (character) in the string
}
```

Strings

- In Go, a string is a sequence of bytes (or runes) with a fixed length.
- Strings are immutable, meaning once created, their contents cannot be changed.
- Strings are declared using double quotes ("):

```
var str string = "Hello, Go!"
```

- String literals can include escape sequences like \n for newline or \t for tab.

```
var multilineString = "First line\nSecond line\nThird line"
```

- Strings can be concatenated using the + operator or fmt.Sprintf function:

```
str1 := "Hello"
str2 := "Go!"
result := str1 + " " + str2 // Result: "Hello Go!"
```

Structs

- Structs allow you to create custom data types by grouping together multiple fields of different data types. This forms a single entity. They are like classes in other languages.
- Fields: Each field within a struct has a name and a type, defining the structure's components.
- Composition: Structs can be composed of other structs or data types, enabling complex data structures and relationships.
- Initialization: Structs can be initialized with values for their fields during declaration or later using field names.
- Accessing Fields: Fields within a struct can be accessed using dot notation (structName.fieldName).
- Value Semantics: Structs are passed by value, meaning that copies of structs are made when passed to functions, allowing for isolation of data.
- Mutability: Fields of a struct can be modified, allowing for state changes within the struct instance.
- Struct fields cannot be added or removed dynamically at runtime. The fields of a struct are defined statically at compile-time and cannot be changed dynamically during program execution.
- Structs in Go provide a fixed blueprint or template for creating instances, and all instances of a particular struct type have the same set of fields. Once a struct type is defined, it remains immutable in terms of its structure throughout the execution of the program.

Interfaces

- Contract: Interfaces define a set of methods that a type must implement to satisfy the interface.
- Polymorphism: Allows different types to be treated interchangeably if they satisfy the same interface.
- Flexibility: Enables writing code that can work with a variety of types, without needing to know their specific implementations.
- Composition: Types can satisfy multiple interfaces by implementing the required methods for each interface.
- Decoupling: Helps in decoupling code by programming to interfaces rather than concrete implementations.

Interfaces (contd.)

- geometry is an interface that declares two methods area( ) and perim( ) respectively.
- Now any struct that implements these methods will be accepted as geometry type because it has implemented the methods defined in geometry interface.
- In gRPC, we have interfaces that we implement in our API Code.
- In our example, rect and and circle both implemented the methods defined in geometry interface and hence were accepted as geometry type into a function that only accepted geometry type argument.
- rect1 did not implement both methods hence it was rejected.
- Usually we don’t create as many interfaces in our day to day programming as much as we use them. Specially when making gRPC API, we use a lot of interfaces.
- When working on gRPC API, you will get more practice and confidence in using interfaces.
- gRPC is all about predefined interfaces that we use throughout our API.

Methods

- Definition: Methods are functions associated with a specific type (usually a struct).
- Although they can also be associated with other types, such as interfaces or basic types like integers or strings.
- Receiver: Methods are defined with a receiver, which specifies the type the method operates on.
- Syntax: Method declaration syntax is func (receiver Type) methodName(parameters).
- Receiver Types: Receivers can be either a value receiver ((t Type)) or a pointer receiver ((t *Type)).
- Value Receiver: Value receivers operate on a copy of the struct, making modifications to the copy.
- Pointer Receiver: Pointer receivers operate directly on the struct, allowing modifications to the original struct.
- Accessing Fields: Methods can access and modify fields of the receiver struct.
- Encapsulation: Methods enable encapsulation by allowing the definition of behavior specific to a type.

Generics

- Generics are a programming language feature that allows the creation of functions, data structures, and algorithms that operate on types without specifying those types beforehand.
- Parameterized Types: With generics, you can define parameterized types and functions that can work with any data type.
- Type Safety: Generics provide type safety by allowing you to specify constraints on the types used with generic functions and types.
- Code Reusability: Generics promote code reusability by enabling the creation of generic data structures and algorithms that work with multiple types.
- Avoiding Code Duplication: Generics help avoid code duplication by eliminating the need to write separate functions or data structures for each data type.
- Constraints: Go's approach to generics includes type constraints, allowing you to specify requirements for the types used with generic functions or types.
- Syntax: Generics in Go are introduced using the type keyword followed by a type parameter in angle brackets (<T>).

Regular Expressions (Regex)

- Regular expressions are patterns used to match character combinations in strings.
- Package regexp: Go provides the regexp package for working with regular expressions.
- Matching: Regex allows you to find patterns within strings, such as specific characters or sequences of characters.
- Syntax: Regex patterns consist of normal characters, which match themselves, and special characters called metacharacters, which represent classes of characters or behavior.
- Usage: Regex is commonly used for tasks such as string validation, text search and replacement, and data extraction.
- Compile and Match: To use a regex pattern, you compile it into a regexp object using regexp.Compile( ) or regexp.MustCompile( ), then use methods like MatchString() or FindString( ) to apply the pattern to strings.
- Capture Groups: Regex allows for the capture of specific parts of matched strings using parentheses ( ) to create capture groups.
- Examples: Common regex patterns include matching email addresses, URLs, phone numbers, and extracting data from structured text.

Goroutine

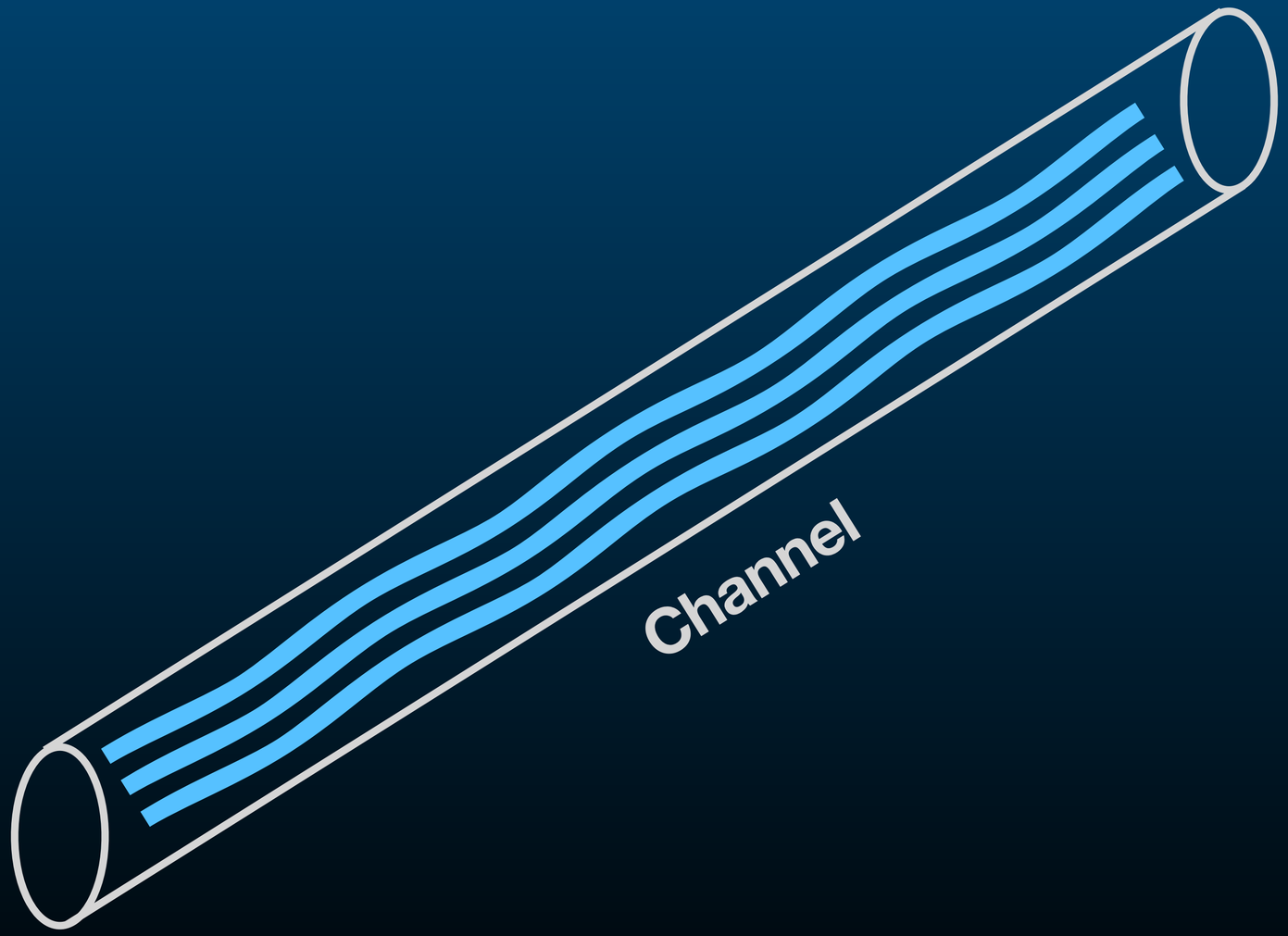
- Concurrency: Goroutines are lightweight threads managed by the Go runtime, allowing concurrent execution of code.
- Syntax: Goroutines are created using the go keyword followed by a function call or anonymous function.
- Non-Blocking: Goroutines enable non-blocking concurrent execution, allowing programs to perform multiple tasks simultaneously.
- Asynchronous: Goroutines execute asynchronously, meaning that they run independently of the main program flow.
- Channel Communication: Communication between goroutines is achieved using channels, facilitating safe data sharing and synchronization.
- Synchronization: Goroutines can be synchronized using synchronization primitives like channels, mutexes, and wait groups to coordinate execution.
- Scalability: Goroutines are highly scalable, allowing programs to create thousands or even millions of concurrent tasks efficiently.
- Context Switching: Goroutines are managed by the Go runtime, which handles context switching between them, making it efficient and lightweight.
- Example: Common examples of goroutine usage include concurrent I/O operations, parallel processing of data, and handling asynchronous tasks.

Channels

- Communication: Channels are a built-in feature of Go for communication and synchronization between goroutines.
- FIFO: Channels are first-in-first-out (FIFO) queues used to pass data between goroutines.
- Synchronization: Channels facilitate safe communication and synchronization by providing blocking operations for sending and receiving data.
- Unbuffered Channels: By default, channels in Go are unbuffered, meaning that the sender will block until the receiver is ready to receive the data.
- Buffered Channels: Buffered channels allow you to specify a capacity, allowing the sender to send multiple values without blocking until the buffer is full.
- Directionality: Channels can be defined with specific send-only (chan<-) or receive-only (<-chan) directions to enforce communication patterns.
- Select Statement: The select statement allows you to wait on multiple channel operations simultaneously, enabling non-blocking communication and synchronization.
- Close: Channels can be closed by the sender to signal that no more values will be sent. Receivers can use the comma-ok idiom to check if a channel is closed.
- Usage: Channels are commonly used for coordinating concurrent tasks, passing data between goroutines, and implementing patterns like producer-consumer and worker pools.

Channels (continued)

Send data into channel
ch <- initial value



Receive data from channel
result := <-ch

Select (for working on channels)

- Multiplexing: The select statement allows you to wait on multiple channel operations simultaneously.
- Syntax: The syntax of the select statement resembles a switch statement but is used for choosing between multiple communication operations.
- Cases: Each case in a select statement represents a channel operation (send, receive, or close) or a default case.
- Non-Blocking: If any of the channel operations in the select statement can proceed without blocking, that operation will be executed.
- Blocking Behavior: If all channel operations are blocked, the select statement will block until at least one of them becomes unblocked.
- Priority: When multiple channels are ready, the select statement chooses one at random. There is no guaranteed priority order.
- Timeouts: select can be combined with the time.After() function to create timeouts for channel operations.
- Concurrency Patterns: select is commonly used in conjunction with channels to implement concurrency patterns such as fan-in, fan-out, and rate limiting.
- Cancellation: select can be used to implement cancellation by combining it with a cancellation channel or context.
- Error Handling: select allows for graceful error handling by incorporating error channels alongside data channels.

Wait Groups

- Synchronization Mechanism: Wait Groups are a synchronization primitive in Go used to wait for a collection of goroutines to finish executing.
- Usage: Wait Groups are particularly useful when you need to coordinate the execution of multiple goroutines and wait for all of them to complete before continuing.
- Package sync: Wait Groups are provided by the sync package in Go.
- Add, Done, Wait: Wait Groups are managed using three main methods:
- Add(n int): Adds n to the Wait Group's internal counter, representing the number of goroutines to wait for.
- Done(): Signals that a goroutine has finished its execution. It decrements the internal counter of the Wait Group.
- Wait(): Blocks until the internal counter of the Wait Group becomes zero, indicating that all goroutines have finished.
- Concurrency Safety: Wait Groups are safe for concurrent use. Multiple goroutines can call Add, Done, and Wait methods concurrently.
- Deferred Done() Call: It's a common practice to defer the Done() method call in each goroutine to ensure that it's called even if the goroutine panics.

Atomic Counters

- Thread-Safe Counters: Atomic counters are a way to manage shared state across multiple goroutines in a thread-safe manner.
- Package sync/atomic: Atomic counters are provided by the sync/atomic package in Go.
- Operations: Atomic counters support atomic read-modify-write operations on shared integer variables.
- Functions:

  AddInt32, AddInt64: Atomically adds the given value to the variable and returns the new value.

  LoadInt32, LoadInt64: Atomically loads the current value of the variable.

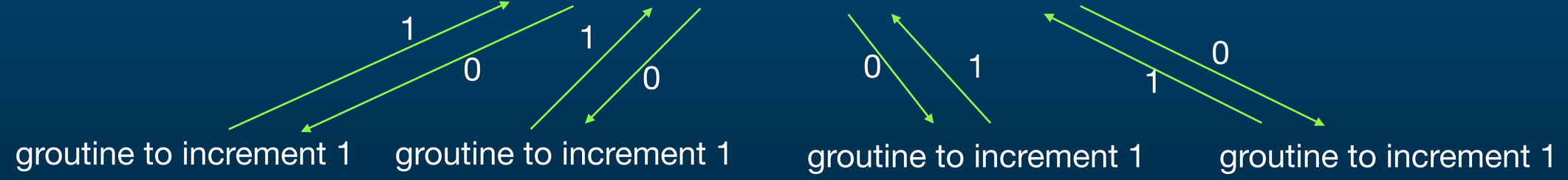
  StoreInt32, StoreInt64: Atomically stores the given value into the variable.

  SwapInt32, SwapInt64: Atomically swaps the value of the variable with the given value and returns the previous value.

  CompareAndSwapInt32, CompareAndSwapInt64: Atomically compares the variable to old and, if equal, swaps the variable with the given value.

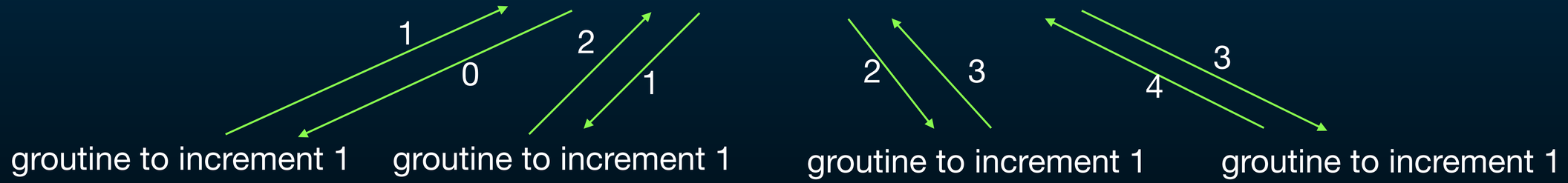
Atomic Counters (continued)

var ops Uint64 (initialized to 0)



They all retrieve 0 because the variable can be accessed by multiple go routines at the same time and it's value never gets locked to stop any other go routine from accessing it. All go routines increment it by 1 and their final out put is 1 because they retrieved the same value 0. They then store the same value (1) in the variable 'ops' one by one. Hence 4 increments still result in final output of 1 instead of 4.

var ops atomic.Uint64 (initialized to 0)



Because ops is now an atomic Uint64, as soon as it's value is accesses, it is locked and unaccessible by any other go routine until previous go routine is finished. Once the previous go routine is finished and increments the value by 1, only then the next go routine is able to access the variable and hence receives the updated value of the variable.

Mutexes (Mutual Exclusion)

- Mutexes are a synchronization primitive used to protect shared resources from concurrent access by multiple goroutines.
- Package sync: Mutexes are provided by the sync package in Go.
- Locking and Unlocking: Mutexes allow you to lock and unlock access to critical sections of code using the Lock() and Unlock() methods, respectively.
- Exclusive Access: While a goroutine holds a mutex lock, all other goroutines attempting to acquire the lock will block until the lock is released.
- Preventing Data Races: Mutexes help prevent data races by ensuring that only one goroutine can access a shared resource at a time.
- RWMutex: Go also provides a read-write mutex (sync.RWMutex) that allows multiple readers or a single writer to access a resource concurrently.
- Deadlocks: Care must be taken to avoid deadlocks when using mutexes. Deadlocks occur when goroutines end up waiting indefinitely for each other to release locks.
- Lock contention: High lock contention can lead to performance issues, as goroutines may spend significant time waiting for locks to be released.

Context

- Purpose: Context provides a way to pass cancellation signals and deadlines to functions and goroutines.
- Package context: Context is provided by the context package in Go.
- Cancellation: Context allows for the propagation of cancellation signals across the call stack, allowing for graceful termination of operations.
- Deadline: Context allows for the specification of deadlines, after which operations should be considered timed out.
- Value propagation: Context allows for the propagation of request-scoped values across API boundaries.
- WithCancel: The context.WithCancel function creates a new context with a cancellation function that can be used to cancel the context.
- WithTimeout/WithDeadline: The context.WithTimeout and context.WithDeadline functions create a new context with a deadline after which the context is automatically canceled.
- Background Context: The context.Background() function returns a background context, typically used as the root of a context tree.
- Context Tree: Contexts can be organized into a tree structure, with child contexts inheriting cancellation signals and deadlines from their parent.
- Propagation: Contexts are passed explicitly as arguments to functions and goroutines, allowing for easy propagation of cancellation signals and deadlines.
- Contexts can also carry metadata, which is key-value pairs associated with the context. This metadata can be used to propagate request-scoped information, such as authentication tokens, request IDs, and tracing/span information, across API boundaries.

REST API

- Representational State Transfer (REST) API is an architectural style for designing networked applications.
- RESTful principles emphasize a stateless client-server communication where each request from the client to the server must contain all necessary information.
- Represent entities (e.g., users, products) that the API interacts with, identified by URLs.
- HTTP Methods: CRUD operations mapped to HTTP methods: GET (Read), POST (Create), PUT, PATCH (Update), DELETE (Delete)
- Uniform Interface: Use of standard HTTP methods and status codes (e.g., 200 OK, 404 Not Found) for communication.
- Statelessness: Each request from a client to the server must contain all necessary information to understand the request, and the server should not store session state between requests.
- Client-Server Architecture: Separation of concerns between client and server, allowing them to evolve independently.
- Cacheability: Responses must define whether they can be cached or not to improve performance.
- Layered System: Allows for intermediaries such as load balancers or proxies to be inserted between clients and servers.

REST API (contd.)

- Endpoint URL: https://api.example.com/users, HTTP Method: GET, Response: JSON array of user objects
- Use nouns to represent resources (e.g., /users, /products) rather than verbs.
- Use HTTP methods correctly and consistently.
- Provide clear and consistent error handling using appropriate HTTP status codes.
- Examples: Postman, cURL, Swagger (OpenAPI), Insomnia
- Use HTTPS to secure data transmission.
- Authentication methods (e.g., OAuth, API keys).
- Rate limiting and throttling to protect against abuse.
- REST API simplifies communication between systems by using standard HTTP methods and leveraging stateless communication.
- Understanding REST principles helps in designing scalable, maintainable, and interoperable web services.

Protocol Buffers

- Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.
- Example: message

```
Person {
    string name = 1;
    int32 id = 2;
    string email = 3;
}
```

- Protocol buffers support generated code in C++, C#, Dart, Go, Java, Kotlin, Objective-C, Python, and Ruby. With proto3, you can also work with PHP.
- The current version of Protocol Buffers is Proto 3, which is more versatile and employs more flexibility.

Protocol Buffers (contd.)

- Protocol Buffers are similar to XML or JSON but are smaller, faster, and simpler.
- They define a language to describe structured data in a way that is both human-readable and machine-readable.
- Protocol Buffers define a data structure using a .proto file, which specifies the fields and their types.
- Once defined, the .proto file is compiled using the Protocol Buffer compiler (protoc) into language-specific classes.
- These classes can then be used to serialize data into a binary format (encoding) that can be efficiently transmitted or stored.
- Protocol Buffers also support backward and forward compatibility, allowing new fields to be added and old fields to be ignored by older code.
- They are widely used in Google's internal systems and are open-sourced for use in other projects.

Advantages of Proto Buf

- **Efficiency:**
  - **Protocol Buffers are binary encoded, resulting in smaller message sizes compared to text-based formats like JSON and XML.**
  - **Smaller sizes lead to faster data transmission and reduced storage requirements.**
- **Speed:**
  - **Binary encoding and streamlined parsing make Protocol Buffers faster to serialize and deserialize than text-based formats.**
  - **This efficiency is particularly advantageous in high-throughput systems or environments with limited bandwidth.**
- **Schema Enforcement:**
  - **Protocol Buffers use a defined schema (.proto file) that enforces data structure and type consistency.**
  - **This ensures that data conforms to expectations, reducing the likelihood of runtime errors due to unexpected data formats.**
- **Compatibility:**
  - **Protocol Buffers support backward and forward compatibility.**
  - **Changes to the data schema can be managed without breaking existing code, allowing for easier system evolution over time.**

Advantages of Proto Buf (contd.)

- Language Neutrality:
  - Protocol Buffers support multiple programming languages, enabling interoperability between different systems and languages.
  - Generated code for serialization and deserialization is available for a wide range of languages.
- Code Generation:
  - Protocol Buffers use code generation to create data access classes from the schema.
  - This simplifies integration into application code and ensures type-safe access to serialized data.
- Extensibility:
  - Protocol Buffers support adding new fields to messages without breaking existing code.
  - Old binaries can ignore new fields, maintaining compatibility with older versions of software.
- Tooling Support:
  - There are robust tooling and libraries available for working with Protocol Buffers, including validation tools and integration with popular development environments.
- Security:
  - Binary format and well-defined schemas can help prevent injection attacks and ensure data integrity during transmission.

Proto Buf in Action

- Defining A Message Type

syntax = “proto3"; (if we don’t do this, compiler will assume we are using proto2

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
}
```

- SearchRequest is the message definition which specifies three fields, one for each piece of data that you want to include in this type of message.
- Each field has a unique name and a type
- Each field in your message definition must be assigned a number between 1 and 536,870,911, but has to follow some rules:
    - The given number must be unique among all fields for that message.
    - Field numbers 19,000 to 19,999 are reserved for the Protocol Buffers implementation. The protocol buffer compiler will complain if you use one of these reserved field numbers in your message.
    - You cannot use any previously reserved field numbers or any field numbers that have been allocated to extensions.

Proto Buf in Action (contd.)

- You should use the field numbers 1 through 15 for the most-frequently-set fields. Lower field number values take less space in the wire format. For example, field numbers in the range 1 through 15 take one byte to encode. Field numbers in the range 16 through 2047 take two bytes.
- Field Labels:

  repeated: this field type can be repeated zero or more times in a well-formed message. The order of the repeated values will be preserved.

  map: this is a paired key/value field type. See Maps for more on this field type.
- To add comments to your .proto files, use C/C++-style // and /* ... */ syntax.
- Types:

ProtoBuf Type	Corresponding type in Go
double	float64
float	float32
int32	int32
int64	int64
uint32	uint32
uint64	uint64
sint32	int32
sint64	int64
fixed32	uint32
fixed64	uint64
sfixed32	int32
sfixed64	int64
bool	bool
string	string
bytes	[ ]byte

Protoc compiler

- Protocol buffer compiler generates the code from the .proto file in your chosen language.
- For Go, the compiler generates a .pb.go file with a type for each message type in your file.
- For Dart, the compiler generates a .pb.dart file with a class for each message type in your file.
- For C++, the compiler generates a .h and .cc file from each .proto, with a class for each message type described in your file.
- For Java, the compiler generates a .java file with a class for each message type, as well as a special Builder class for creating message class instances.
- Python is a little different — the Python compiler generates a module with a static descriptor of each message type in your .proto, which is then used with a metaclass to create the necessary Python data access class at runtime.

gRPC

- gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.
- It uses Protocol Buffers as its interface definition language (IDL) for describing both the service interface and the structure of the payload messages.
- gRPC enables efficient communication between distributed systems and microservices.
- IDL-Driven Development: Uses Protocol Buffers to define services and messages, promoting strong typing and code generation.
- Cross-Platform: Supports multiple programming languages, allowing services written in different languages to seamlessly communicate.
- Streaming: Supports both unary RPCs (request/response) and streaming RPCs (server-streaming and client-streaming).
- gRPC uses HTTP/2 for transport, providing benefits such as multiplexing, header compression, and server push.
- Suitable for IoT applications where low bandwidth and efficient communication are crucial.

gRPC (contd.)

```
// The login service definition.
service LoginService {
  // Sends a greeting
  rpc Login (LoginRequest) returns (LoginResponse) {}
}

// The request message containing the username and password.
message LoginRequest {
  string username = 1;
  string password = 2;
}

// The response message containing the boolean result if login was successful or not
message LoginResponse {
  bool success = 1;
}
```

gRPC Service Methods

gRPC lets you define four kinds of service method:

- **Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call.**

```
rpc Login(LoginRequest) returns (LoginResponse);
```

- **Server streaming RPCs where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. gRPC guarantees message ordering within an individual RPC call.**

```
rpc OnlineStatus(StatusRequest) returns (stream StatusResponse);
```

- **Client streaming RPCs where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response. Again gRPC guarantees message ordering within an individual RPC call.**

```
rpc SendChatMessages(stream ChatMessages) returns (ChatMessageConfirmation);
```

- **Bidirectional streaming RPCs where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved.**

```
rpc IncomingChatMessage(stream ChatMessage) returns (stream SendReadReceipts);
```

gRPC Status Codes

- gRPC defines its own set of status codes to communicate the status of RPC calls between the client and server.
- Some common gRPC status codes include OK, CANCELLED, UNKNOWN, INVALID_ARGUMENT, DEADLINE_EXCEEDED, NOT_FOUND, etc.
- These status codes are used to indicate the success or failure of an RPC call.
- Error handling in gRPC revolves around using these status codes to convey specific information about the result of an RPC call.
- It can be done by returning specific status codes and optional details in response to RPC calls.
- When a server encounters an error, it returns an appropriate status code along with optional metadata and details.
- Clients can handle these errors programmatically based on the returned status codes and messages.
- gRPC allows passing metadata alongside RPC calls, which can include additional information such as authentication tokens, tracing information, etc.

All About Concurrency

- Race conditions in Go
- Deadlocks
- Livelocks
- Starvation
- Concurrency vs Parallelism
- CSP (Communicating Sequential Processes)
- The Sync package
- Mutex, RW Mutex
- sync.NewCond
- sync.pool
- Channels
- Select statement
- GOMAXPROCS
- for select loop
- The or-channel
- Fan-Out, Fan-In
- The or-done channel

MongoDb

- A popular, high-performance, and scalable NoSQL database.
- Stores data in a flexible, JSON-like format.
- Uses a document-oriented approach.
- Ideal for handling large volumes of unstructured or semi-structured data.

MongoDb

- 1. SQL Databases:
  - - Structure: Data is stored in tables with predefined schemas (rows and columns).
  - - Schema: Requires a fixed schema, meaning the structure of data is defined in advance.
  - - Relationships: Typically use joins to link related data across multiple tables.
  - - Examples: MySQL, PostgreSQL, Oracle, SQL Server.

- 2. NoSQL Databases:
  - - Structure: Data is stored in various formats, such as documents, key-value pairs, wide-column stores, or graphs.
  - - Schema: Schema-less or flexible schema, allowing for dynamic and unstructured data.
  - - Relationships: Often designed to avoid complex joins, using nested or embedded structures instead.
  - - Examples: MongoDB, Cassandra, Redis, Neo4j.

Key Features of MongoDB

- Document-Oriented Storage
- Dynamic Schema
- Scalability
- High Performance
- Indexing
- Aggregation Framework
- Replication

Drawing Parallels(SQL-NoSQL)

- Database
- Table vs. Collection
- Row vs. Document
- Column vs. Field
- Schema
- Primary Key vs. _id
- Indexes
- Joins vs. Embedding/Referencing
- SQL Queries vs. MongoDB Queries
- Transactions
- Aggregation
- Foreign Keys vs. Manual Referencing: (Table Relations)
- ACID Compliance

How is MongoDB Special?

- Flexibility
- Ease of Use
- Fast Development Cycle
- Big Data Handling
- Versatility
- Community and Ecosystem