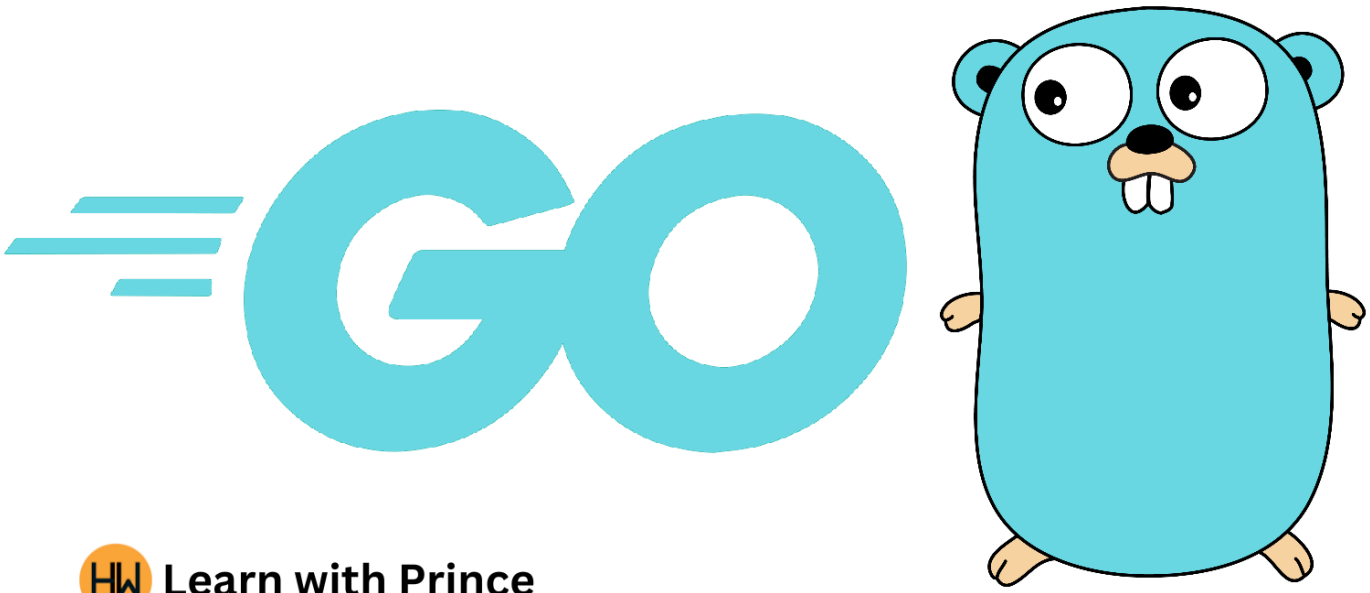


Learn with Prince

Learn Golang

- ***Why should you Learn to Go Programming Language?***

We are living in an era where technology continues to emerge at a breakneck speed. As 21st-century developers, it is crucial for us to learn new coding languages to keep pace with this rapid change as it increases our versatility, widens our competencies, and brightens our career prospects.

Go is a simple, fast, and concurrent programming language.

Go is a strong and statically typed programming language which essentially means each variable has a type that cannot be changed over time and must be defined at compile time. This would help us to capture errors easily at compilation time.

As Go is developed by Google, we are assured of a strong community supporting this language.
Google created the Go programming language, also known as Golang, to address some of the shortcomings and inefficiencies they were experiencing with existing languages and tools.

Why Google Created Go Language

1) **Compile Time**: Google has historically struggled with long compile times for its projects having large codebases.
   Go is designed for faster compilation without the need for dependency checking.

2) **String processing**: Google frequently processes and analyzes large amounts of text data in the form of web pages, requiring efficient manipulation of strings.

   Go programming language has a comprehensive set of string functions. which also uses garbage collection to make working with strings more efficient compared to some other languages like C++.

3) **Concurrency**: Concurrency in Go refers to the ability of a program to execute multiple tasks independently but not necessarily simultaneously. It's about structuring your program in a way that allows different parts of it to run independently, making efficient use of available resources.

   While concurrency is about managing many tasks, **parallelism** is about executing tasks simultaneously.

   In Go, a goroutine is a lightweight thread managed by the Go runtime.

4) **Learning curve:** Go is a relatively simple language with a straightforward syntax and a small set of core features. This makes it easy for programmers to learn and use, even if they are new to programming.

> *Go = C + strings + garbage collection + concurrency.*

Key Features of Go Language

1. ***Concurrency***: Go is built with concurrency in mind and provides several features to make it easy to write concurrent code.

    a. It's the idea that different tasks can be performed independently and simultaneously.
    b. Each task doesn't wait for the other to finish before starting their work.
2. ***Garbage collection***: Go includes a garbage collector that automatically manages memory, making it easier for developers to write code without having to worry about memory management.
3. ***Static typing:*** Go is a statically typed language, which means that variables are explicitly declared with a specific type and the type of a variable cannot be changed during its lifetime. This can help catch errors at compile time and improve the overall reliability of the code.
4. ***Lightweight: G***oroutines take only 8 kilobytes and you can have thousands of them.
5. ***Fast compilation:*** Go has a fast compiler that can quickly build large programs, making it suitable for building scalable applications.
6. ***Zero dependencies:*** Since the language does not rely on any external libraries or frameworks, you do not need to worry about installing those dependencies on the target machine. This can simplify the deployment process and reduce the risk of issues caused by missing or incompatible dependencies. This can be particularly useful for building applications that need to be deployed in a variety of different environments.
7. ***Built-in support for testing:*** Go includes built-in support for writing and running tests, making it easy to test and verify code.
8. ***Strong community:*** Go has a strong and active community of developers who contribute to the language and its ecosystem, including libraries and tools.

- *Go* is currently gaining a lot of popularity, and a lot of organizations now prefer to write their backend in *Go*.
- More importantly, all Cloud Native and some Blockchain projects are written or being written in *Go*, some popular tools are Kubernetes, Prometheus, and Docker.

- **Overall, Go is a versatile and powerful programming language that can be used in a wide range of projects. Whether you're building a web application, a network server, or a command-line tool, Go is a good language to consider.**

As more and more companies and people began to realize Go's potential, it became a mainstream language to build the following kinds of products.

- World-class system tools like Docker and Kubernetes
- Advanced databases like CockroachDB and InfluxDB
- Decentralized Blockchain platforms like Ethereum

- To separate configuration into infrastructure layers like Istio
- Faster continuous deployment like Drone.
- More performant messaging systems like NATS.
- Widely used CLI tools like Cobra

Here are some good resources to learn Golang.

- **The official Golang website** provides comprehensive documentation, tutorials, and other resources for learning the language.
- "**Go by Example**" website provides a collection of short, easy-to-understand examples of how to use various Golang features.
- "**A Tour of Go**" website provides an interactive, in-browser tutorial that teaches the basics of Golang.
- *"Effective Go"* is a free resource to learn the Go programming language and it is available on the official website of Golang. This course provides an explanation of all the key concepts in the Go programming language, how to use them, and their syntax.

Install GoLang

- Go to https://go.dev/doc/install
- Normally Google: "Download Go language", and you will get the link.
- Open Command Prompt and Run this command → go version

```
prince@Princes-MacBook-Air ~ % go version
go version go1.21.1 darwin/arm64
```

- Once you've done that you'll have to *Go* available in your command line. Typing go will show the available commands.

```
Go is a tool for managing Go source code.

Usage:
        go <command> [arguments]

The commands are:

        bug         start a bug report
        build       compile packages and dependencies
        clean       remove object files and cached files
        doc         show documentation for package or symbol
        env         print Go environment information
        fix         update packages to use new APIs
        fmt         gofmt (reformat) package sources
        generate    generate Go files by processing source
        get         add dependencies to current module and install them
        install     compile and install packages and dependencies
        list        list packages or modules
        mod         module maintenance
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
```

- *Go* enforces a structure to your projects, the idea is that all of your projects should live under one roof to access all of *Go's* features.
- By default, this is under `$HOME/go`, you can print to the console by using `go env GOPATH`.
- You can set this location to be anywhere on your computer.
- the default structure of the directory will be

```
$HOME/
└── go/
    ├── bin/
    │   └── (executable binaries)
    ├── pkg/
    │   └── (compiled package files)
    └── src/
        └── myproject/
            ├── main.go
            └── other files
```

- `bin/`: This directory contains the executable binaries after you build your Go programs. When you install a Go package with the `go install` command, its binary goes here.
- `pkg/`: This directory contains compiled package files. When you compile a Go package, its object files go here.
- `src/`: This is where your source code resides. Each project or package has its own directory under `src/`.

Creating a Project

- If your GOPATH is set to `$HOME/go`, then your project should be located under `$HOME/go/src/`. This ensures that your project is part of the **Go workspace**.
- This is a common practice to maintain a structured workspace that aligns with Go's package system.

Go Modules

- If you want to organize your Go projects outside the default GOPATH, you can use **Go Modules**. Go Modules allows you to create and manage projects outside of the GOPATH.

  Setting Up Go Modules:

  Create a new directory for your project:

```
mkdir $HOME/Desktop/mylearning
```

```
cd $HOME/Desktop/mylearning
```

Initialize Go Modules:

```
go mod init mylearning
```

- This command creates a `go.mod` file that tracks the dependencies of your project.

```
$HOME/Desktop/mylearing/
├── main.go
└── go.mod
```

- Inside your `main.go` file, you can write your Go code as usual.

```
// main.go
package main
import "fmt"
func main() {
    fmt.Println("Hello, Go Modules!")
}
```

```
go run main.go
```

This command will automatically download any necessary dependencies, build, and run your program. The `go.mod` file ensures that your project is self-contained.

- Go Modules simplify the management of dependencies and project structure.
- The `go.mod` file contains information about your project, including its name and dependencies.
- You can use any directory structure you prefer within your project.

- ***Let's build the First Project***
- Create a Folder, Outside the Project Folder
- And Create a file, print "Hello World by Prince"

Hello World by Prince

```
package main
import "fmt"
func main() {
     fmt.Println("Hello, world! by Prince")
}
```

Packages and Imports

- In Go, packages are used instead of classes, There are no concepts like OOPs in Java languages.
- Each package in Go is essentially a directory in your workspace. **Each Go file must belong to some package,** and it should start with the keyword `package` followed by the package name.
- The `main` package is a special package in Go. An executable program must contain a `main` package.
- Go uses relative imports to bring packages into the current file. The relative path is usually `$GOPATH/src` since most packages are stored in the `pkg` directory.

- Importing a package is done using the `import` keyword followed by the list of packages inside parentheses. For example:

```
go

import (
    "fmt"
)
```

- The standard library in Go comes preinstalled and contains essential and useful packages. `"fmt"` is one such package, and it's used for printing to the console.

In Go, it's a common convention to organize your projects within the `src/` directory under your GOPATH.

```
$HOME/go/
└── src/
    └── myproject/
        └── main.go
```

```go
// main.go
package main

import "fmt"

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

- Here, `package main` declares that this file belongs to the `main` package, and it contains the `main` function, which is the entry point for the executable program.

- Let's say you have another file in your project, `util.go`, with a custom package named `myutil`

- In Go, the concept of a package is closely tied to the directory structure.
- All Go files in a single directory are considered part of the same package.
- The package is declared at the top of each Go file with the `package` keyword.

```go
// util.go
package myutil

import "fmt"

func PrintMessage(message string) {
    fmt.Println(message)
}
```

- Now, you can use this custom package in your `main.go` file:

```
// main.go
package main

import "myproject/myutil"
import "fmt"

func main() {
    fmt.Println("Hello, Go!")
    myutil.PrintMessage("Hello from myutil package!")
}
```

- Here, `import "myproject/myutil"` tells Go to look for the `myutil` package in the `myproject` directory. The `PrintMessage` function from the `myutil` package is then used in the `main` function.

- **A package is a way to organize code in Go.**
- **The `import` keyword is used to include external packages.**

- *In Go, for a file to be the entry point of an executable program, it must belong to the `main` package and contain a `main` function*. This is a convention set by the Go programming language. The main package is special and is used for executable programs.
- When you use the `go run` command to execute a Go program, the Go runtime looks for the `main` function in the `main` package and executes it.

- you can absolutely give your Go file a name other than `main.go`. The naming of the file doesn't have any impact on whether it contains the `main` function or not. However, the name of the file must reflect the package name it belongs to.

- The name of the file doesn't have to be `main.go`. It could be any valid Go identifier. For example, it could be `myprogram.go`.

```go
// myprogram.go
package main

import "fmt"

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

- The key is that the package name (`package main`) and the presence of the `main` function in the file determine whether it's meant to be the entry point for a standalone executable program. The name of the file is up to you and does not affect the program's behavior.

Variables in GoLang

- Creating Variables

```go
package main
import "fmt"
func main() {
    // Variables with different data types
    var age int = 25
    var height float64 = 5.9
    var name string = "John"
    var isAdult bool = true

    // Print the values of variables
    fmt.Println("Name:", name)
    fmt.Println("Age:", age)
    fmt.Println("Height:", height)
    fmt.Println("Is Adult:", isAdult)
}
```

- A **constant** is similar to a variable, but its value cannot be changed once it's assigned. It's like a fixed value that remains constant throughout the program.

```
const pi float64 = 3.14159
```

In summary:

- **Variables** are containers for holding values, and you can change their values during the program.
- **Data types** define the kind of values a variable can hold (integers, floats, strings, booleans, etc.).
- **Constants** are like variables, but their values cannot be changed once assigned. They are useful for fixed values that shouldn't be altered during program execution.

- In Go, you can use the `:=` notation to declare and initialize a variable in a single line. This is particularly useful when Go can infer the variable type based on the assigned value
- The short variable declaration is often used when you want to declare and initialize a variable in a single line

```
package main
import "fmt"
func main() {
    // Shorthand variable initialization
    age := 25
    fmt.Println("Age:", age)
}
```

- In this example, `age` is declared and initialized using `:=`, and Go infers that it's of type `int` based on the assigned value `25`.
- Unlike some other programming languages, Go doesn't require semicolons to end statements. The end of a line is automatically treated as the end of a statement. This makes Go code look cleaner and less cluttered:

Variable and Function Visibility

- In Go, the visibility of a variable or function outside its package is determined by the **capitalization of its name.**
- If the first letter of a name is uppercase, it's exported (public). If it's lowercase, it's

unexported (private) and only visible within the same package.

```
// Public variable (exported)
var PublicVariable int = 42

// Private variable (unexported)
var privateVariable int = 10
```

- In this example, `PublicVariable` is visible and can be accessed from other packages, while `privateVariable` is only visible within the same package.

- The same rule applies to functions:

```
// Public function (exported)
func PublicFunction() {
    fmt.Println("This is a public function.")
}

// Private function (unexported)
func privateFunction() {
    fmt.Println("This is a private function.")
}
```

- The `Println` function in Go is used for printing output. It automatically adds spaces between the provided arguments and a newline character at the end. It is suitable for simple, unformatted output.

```go
package main

import "fmt"

func main() {
    age := 25
    name := "Alice"
    height := 5.8234567

    fmt.Println("hello world")
    fmt.Println("Name:", name, "Age:", age, "Height:", height)
    // Output:
    // hello world
    // Name: Alice Age: 25 Height: 5.8234567
}
```

- The `Printf` function (print formatted) is used for formatted printing. It allows you to control the output format by using **format specifiers** similar to those in the C programming language. It provides more control over how values are displayed.
- There are multiple format specifiers
    - `%d`: Integer
    - `%s`: String
    - `%T`: Type of the value
    - `%.3f`: Float Values (*The number after the dot in %f specifies the precision for floating-point numbers.*)

```go
package main
import "fmt"
func main() {
     age := 25
     name := "Alice"
     height := 5.8234567
```

```
    fmt.Printf("Age: %d\n", age)
    fmt.Printf("Name: %s\n", name)
    fmt.Printf("Height: %.2f\n", height)
    fmt.Printf("Type of age: %T\n", age)

    fmt.Println("hello world")
    fmt.Printf("Name: %s, Age: %d, Height: %.2f\n", name, age, height)
    // Output: Name: Alice, Age: 25, Height: 5.80
}
```

User Input

- Now, Let's suppose we want to store the Name of some person, then we need to take input from the user
- The `fmt` package provides the `Scan` family of functions for reading input.

```
package main
import "fmt"
func main() {
    var name string

    fmt.Println("Enter your name: ")
    fmt.Scan(&name)

    fmt.Printf("Hello, %s!\n", name)
}
```

In this example:

1. We declare a variable `name` of type string to store the user's input.
2. We use `fmt.Println` to display a prompt asking the user to enter their name.
3. We use `fmt.Scan(&name)` to read the user's input and store it in the `name` variable. The `&` symbol is used to get the memory address of the variable for `Scan` to populate.

4. Finally, we use `fmt.Printf` to print a greeting with the user's name.

- Keep in mind that `fmt.Scan` reads until the first whitespace character, so if you want to read a whole line, you might want to use `bufio` package's `NewScanner` or `ReadString` functions for more complex scenarios.

Bufio Package for User Input

```
package main
import (
     "bufio"
     "fmt"
     "os"
)
func main() {
     fmt.Print("Enter your name: ")

     reader := bufio.NewReader(os.Stdin)
     name, _ := reader.ReadString('\n')
     fmt.Printf("Hello, %s", name)
}
```

In this example:

- `bufio.NewReader(os.Stdin)` creates a new buffered reader that reads from the standard input (`os.Stdin`).
- `reader.ReadString('\n')` reads a line from the input until it encounters a newline character (`'\n'`). This allows the program to read the entire line of input, including spaces.
- The input is stored in the `name` variable, and any potential errors are handled.

- A buffered reader is a type of reader that reads data from an underlying source, such as a file or standard input (keyboard), and stores that data in a buffer. The buffer is a temporary storage area in memory. Buffered readers are commonly used to improve the efficiency of input operations.

Functions

- Functions are an essential part of *Go.*
- *In Go, functions are declared using the `func` keyword, followed by the function name, parameters (if any), return type (if any), and the function body.*

```
package main
import "fmt"

// Function without parameters and return type
func simpleFunction() {
    fmt.Println("This is a simple function.")
}

// Function with parameters and return type
func add(a, b int) int {
    return a + b
}

// Function with named return variable
func multiply(x, y int) (result int) {
    result = x * y
    return
}

// Main function (entry point of the program)
func main() {
```

```
    simpleFunction()

    sum := add(3, 5)
    fmt.Println("Sum:", sum)

    product := multiply(4, 6)
    fmt.Println("Product:", product)
}
```

Remember, when you specify a return type for a function, ensure that the function returns a value of that type. If a function does not return anything, the return type should be omitted, or you can explicitly use `func functionName()`.

In a Go application, the `main` function is the entry point of the program. It doesn't take any arguments and doesn't return anything. The execution of the program starts from the `main` function.

- The function body is enclosed in curly braces `{}`.
- It's a Go convention that the opening curly brace `{` should be on the same line as the function declaration.

```
// Correct way
func example() {
    // Function body
}

// Incorrect way
func example()
{
    // Function body
}
```

- This Below Piece of code is Wrong

```
func add(a int, b ) (result int) {
     result = a + b
     return
```

```
}
```

The issue in this code is that when we are declaring the parameters for the function `add`, we haven't specified the type for parameter `b`. In Go, you must provide a type for each parameter. Here's the corrected version:

```
func add(a, b int) (result int) {
    result = a + b
    return
}
```

In Go, when declaring parameters in a function signature, if multiple parameters share the same type, you can specify the type only once at the end of the list. This applies to both parameters and return variables.

```
func add(a int, b int) (result int) {
    result = a + b
    return
}
```

The above code is on the safe side If you specify the type on every parameter

Error Handling in Go

- Hey, In Go we Normally handle Errors as we do in other Programming Language, but again Go Introduced a unique concept of _ *(underscore)*
- In Go, the underscore (_) is used as a blank identifier. It serves as a write-only variable that allows you to discard values returned by a function or to ignore specific values when you are not interested in using them.

```
package main
import "fmt"

func divide(a, b float64) (float64, error) {
     if b == 0 {
          return 0, fmt.Errorf("denominator must not be zero")
```

```
    }
    return a / b, nil
}

func main() {
    // data := divide(67, 21)
    data, _ := divide(10, 0)
    fmt.Println(data)
    fmt.Printf("Data is  %.4f\n", data)
}
```

In this example:

- The `divide` function returns two values: the result of the division and an error (if any).
- In the `main` function, when calling `divide(10, 2)`, we are only interested in the result, so we use the blank identifier (`_`) to discard the error.
- When calling `divide(10, 0)`, we again use the blank identifier to discard the error, as we are not interested in handling it in this specific scenario.

- Absolutely, you can use any valid variable name in place of the underscore (`_`). The use of the underscore is a convention in Go to indicate that the variable is intentionally being ignored, and it's a way to make it clear to both the compiler and other developers that you are not planning to use that variable.
- If you don't use `fmt.Errorf` and simply return a string as an error, **it will still work**, but you lose some benefits that come with using `fmt.Errorf`.
- The use of `error` as a return type is a standard convention in Go for functions that may produce an error.

Array in Golang

- Arrays in Go provides a simple and efficient way to work with a fixed collection of elements.

- Suppose you want to store a List of fruits in the array.

- **Array Declaration:**
  - Arrays are declared using the syntax `var name [size]type`, where `name` is the array variable, `size` is the number of elements, and `type` is the data type of the elements.
  - Arrays have a fixed size, meaning you must specify the number of elements the array can hold at the time of declaration.
  - Once the array is created, its size cannot be changed.
  - Example: `var numbers [5]int` declares an integer array with five elements.
- **Array Initialization:**
  - Arrays can be initialized at the time of declaration or later in the program.
  - Example: `var numbers [5]int` creates an array of integers with zero values. You can also initialize it with specific values: `var numbers = [5]int{1, 2, 3, 4, 5}`.
- **Accessing Array Elements:**
  - Array elements are accessed using square brackets `[]` and the index of the element.
  - The index starts from 0, so `numbers[2]` refer to the third element in the array.
  - Example: `numbers[2]` returns the value at index 2 in the array.
- **Array Length:**
  - The length of an array is fixed at the time of declaration and cannot be changed.
  - The length is part of the array's type, so `[5]int` and `[10]int` are distinct types.

```go
package main

import "fmt"

func main() {
    // Declare and initialize an array to store a list of fruits
    var fruits [5]string

    // Assign values to array elements
    fruits[0] = "Apple"
```

```
    fruits[1] = "Banana"
    fruits[2] = "Orange"
    fruits[3] = "Grapes"
    fruits[4] = "Mango"

    // Access and print array elements
    fmt.Println("List of Fruits:", fruits)

    // Access and print a specific fruit
    fmt.Println("Fruit at index 2:", fruits[2])
}
```

```
package main

import "fmt"

func main() {
    // Initialize and assign values to an array in a single line
    numbers := [5]int{1, 2, 3, 4, 5}

    // Access and print array elements
    fmt.Println("Array:", numbers)
}
```

- *In Go, when you declare an array or a slice, the elements are initialized to their zero values. The zero value is the default value assigned to variables of a certain type when no explicit value is provided.*
- For numeric types (int, float, etc.), the zero value is `0`. For strings, it is an empty string (`""`). For boolean types, it is `false`, and for pointers or complex types, it is `nil`.

Let's look at an example with an array:

```
package main
import "fmt"
func main() {
    // Declare an array of integers
    var numbers [5]int
    // Print the array
    fmt.Println("Array:", numbers)
}
```

The output will be:

```
Array: [0 0 0 0 0]
```

Let's look at an example with an array of strings:

```
package main
import "fmt"
func main() {
     var numbers [5]string
     fmt.Println("Empty Array: ", numbers)

     // If you want to see the actual content of the array
     fmt.Printf("Array: %q\n", numbers)
}
```

- If you want to see the actual content of the array without the spaces, you can use `fmt.Printf` with the `%q` verb to print each string element with quotes
- the `%q` verb is a formatting directive used with the `fmt.Printf` and `fmt.Sprintf` functions to format strings for printing. It is often referred to as the "***quoted-string***" verb.

- In Go, you can use the `len` function to determine the length (number of elements) of an

array. Here's an example:

```
package main

import "fmt"

func main() {
    // Declare an array of integers
    var numbers [5]int

    // Print the length of the array
    fmt.Println("Length of the array:", len(numbers))
}
```

- Arrays in Go provide a simple and efficient way to work with a fixed collection of elements. However, in certain situations, slices (a more flexible and dynamic alternative) might be more suitable.

Slices in Golang

- In Go, a slice is a flexible and dynamic data structure that provides a more powerful alternative to arrays.
- Unlike arrays, slices have a dynamic size, and their length can be changed during the program's execution.
- Here's the basic syntax for creating a slice:

```
package main
```

```go
import "fmt"

func main() {
    // Declare and initialize a slice
    numbers := []int{1, 2, 3, 4, 5}

    // Access and print slice elements
    fmt.Println("Slice:", numbers)

    // Length of the slice
    fmt.Println("Length of the slice:", len(numbers))
}
```

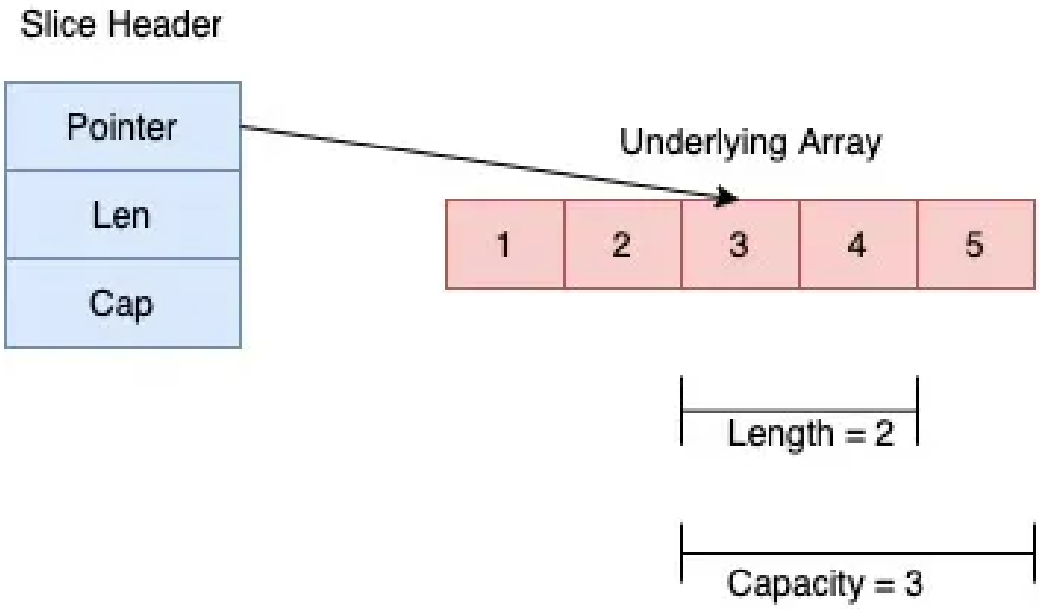
In this example:

- `numbers` is a slice of integers created using the slice literal `[]int{1, 2, 3, 4, 5}`.
- The length of the slice is determined by the number of elements in the literal.
- The `len` function is used to retrieve the length of the slice.

Key characteristics of slices:

1. **Dynamic Size:** Slices can grow or shrink dynamically during runtime.
2. **Reference to Underlying Array:** Slices are references to the underlying array, and modifying elements in the slice will affect the original array.
3. **Syntax:** The syntax for slices is similar to arrays, but the size is omitted, creating a slice. For example, `numbers := []int{1, 2, 3}`.
4. **Use of `make` Function:** You can use the `make` function to create a slice with a specific length and capacity.
5. **Appending Elements:** The `append` function is used to add elements to a slice.

- If you want to learn more details in Slices Please visit a really good article on this: https://golangforall.com/en/post/golang-slice.html



- When you create a slice without specifying the capacity explicitly, like `numbers := []int`, the capacity of the slice is initially set to the same value as its length. This means that the slice starts with a capacity equal to the number of elements it contains.

```go
package main

import "fmt"

func main() {
    // Creating a slice without specifying the capacity
    numbers := []int{1, 2, 3}

    // Printing the slice, length, and capacity
    fmt.Println("Slice:", numbers)
    fmt.Println("Length:", len(numbers))
    fmt.Println("Capacity:", cap(numbers))
```

```
    // output of this program is
    // Slice: [1 2 3]
    // Length: 3
    // Capacity: 3

}
```

```
package main

import "fmt"

func main() {
    // Creating a slice with make: length = 3, capacity = 5
    numbers := make([]int, 3, 5)

    // Printing the slice
    fmt.Println("Slice:", numbers)
    fmt.Println("Length:", len(numbers), "Capacity:", cap(numbers))
}
```

- In this example, `make` is used to create a slice of integers with an initial length of 3 and a capacity of 5. The length is the number of elements in the slice, and the capacity is the maximum number of elements it can hold without resizing the underlying array.
- The zero value of the element type is used to initialize the elements.
- The capacity will automatically grow if you append more elements to the slice, and the underlying array might be reallocated with a larger capacity as needed.

```
var numbers = []string
```

- You can't initialize the slice like this
- When you declare a slice without providing any initial values, you need to use the `make` function to create the slice with a specified length and capacity. The syntax

you provided (`var numbers = []string`) is an attempt to declare a slice without using `make`, but it will result in a compilation error.

```
package main

import "fmt"

func main() {
    // Creating an empty slice of strings with make
    var numbers = make([]string, 0)

    // Printing the slice, length, and capacity
    fmt.Println("Slice:", numbers)
    fmt.Println("Length:", len(numbers))
    fmt.Println("Capacity:", cap(numbers))
}
```

- In this example, `make([]string, 0)` creates an empty slice of strings with a length of 0 and an initial capacity of 0. The resulting slice can be dynamically resized by appending elements to it.
- Using `make` in this way is necessary when you want to create a slice without specifying initial values

If-else in Golang

- In Go, the syntax for the `if` statement is quite similar to other programming languages, but there is a key difference: the opening curly brace `{` must be on the same line as the `if` statement.
- Here's the basic structure of an `if` statement in Go:

```
package main

import "fmt"

func main() {
     // Example 1: Simple if statement
     x := 10
```

```
    if x > 5 {
        fmt.Println("x is greater than 5")
    }

    // Example 2: if-else statement
    y := 3
    if y > 5 {
        fmt.Println("y is greater than 5")
    } else {
        fmt.Println("y is not greater than 5")
    }

    // Example 3: if-else if-else statement
    z := 7
    if z > 10 {
        fmt.Println("z is greater than 10")
    } else if z > 5 {
        fmt.Println("z is greater than 5 but not 10")
    } else {
        fmt.Println("z is 5 or less")
    }

    // Example 4: Checking both conditions using &&
    if x > 5 && y < 10 {
        fmt.Println("Both conditions are true")
    } else {
        fmt.Println("At least one condition is false")
    }
}
```

Switch Case in Golang

- In Go, the `switch` statement provides a way to conditionally execute code based on the value of an expression. It's a more concise and flexible alternative to a sequence of `if-else` statements when dealing with multiple possible values for a variable.

```go
package main

import "fmt"

func main() {
    // Example 1: Basic switch statement
    day := 3

    switch day {
    case 1:
        fmt.Println("Monday")
    case 2:
        fmt.Println("Tuesday")
    case 3:
        fmt.Println("Wednesday")
    default:
        fmt.Println("Unknown day")
    }

    // Example 2: Switch statement with multiple values
    month := "January"

    switch month {
    case "January", "February", "March":
        fmt.Println("Winter")
    case "April", "May", "June":
        fmt.Println("Spring")
    default:
        fmt.Println("Other season")
    }

    // Example 3: Switch with expression
    temperature := 25

    switch {
    case temperature < 0:
        fmt.Println("Freezing")
    case temperature >= 0 && temperature < 10:
```

```
        fmt.Println("Cold")
    case temperature >= 10 && temperature < 20:
        fmt.Println("Cool")
    case temperature >= 20 && temperature < 30:
        fmt.Println("Warm")
    default:
        fmt.Println("Hot")
    }
}
```

In the examples above:

1. **In Example 1**, the `switch` statement checks the value of the `day` variable and executes the corresponding case. If no case matches, the code inside the `default` case is executed.
2. **In Example 2**, the `switch` statement checks the value of the `month` variable against multiple possible values. If the value matches any of the specified cases, the corresponding code is executed.
3. **In Example 3**, the `switch` statement uses an expression (`temperature`) rather than a specific value. Each case contains a condition, and the first case with a true condition is executed. The `switch` statement without an expression is equivalent to `switch true`.

The `switch` statement in Go is quite flexible and allows for concise and readable code, especially when dealing with multiple conditions.

For Loop in Golang

- In Go, the `for` loop is the primary way to create loops. The basic structure of a `for` loop is similar to other languages, but there's only one form of the `for` loop in Go, and it covers a variety of use cases.

```
package main
```

```go
import "fmt"

func main() {
	// Example 1: Basic for loop
	for i := 0; i < 5; i++ {
		fmt.Println(i)
	}

	// Example 2: Infinite loop with break statement
	counter := 0
	for {
		fmt.Println("Infinite Loop")
		counter++
		if counter == 3 {
			break
		}
	}
}
```

In the examples above:

1. **Basic for loop (Example 1):**
   - `for i := 0; i < 5; i++` initializes `i` to 0, checks if `i` is less than 5, and increments `i` after each iteration.
   - This loop prints the values of `i` from 0 to 4.
2. **Infinite loop with break statement (Example 2):**
   - `for { ... }` creates an infinite loop.
   - The loop can be exited using the `break` statement.
   - This loop prints "Infinite Loop" three times before breaking out.

Key points about `for` loops in Go:

- The `for` loop in Go is quite flexible and can be used for various purposes.
- You can use `break` to exit a loop prematurely or `continue` to move to the next iteration.
- There is no `do-while` loop in Go, but you can achieve similar behavior using a `for` loop with a conditional check at the end.
- Go doesn't have a `while` loop in the traditional sense that you might find in some other programming languages. In Go, the `for` loop is the only looping construct
- The `range` keyword simplifies looping over elements of a collection like slices, arrays, maps, and strings.

```
package main

import "fmt"

func main() {

     numbers := []int{1, 2, 3, 4, 5}
     for index, value := range numbers {
          fmt.Printf("Index: %d, Value: %d\n", index, value)
     }
}
```

- The `range` keyword in Go simplifies the process of iterating over elements in various types of collections, such as slices, arrays, maps, and strings. **It provides both the index** (or key in the case of maps) **and the value of each element** in the collection.
- In this example, `range numbers` return both the index and value of each element in the `numbers` slice. The loop will iterate five times (the length of the slice), and on each iteration, the `index,` and `value` will be set to the current index and value, respectively.

```go
package main

import "fmt"

func main() {
    message := "Hello, Golang!"

    // Looping over characters of a string using range
    for index, char := range message {
        fmt.Printf("Index: %d, Character: %c\n", index, char)
    }
}
```

- In this example, the `range message` returns both the index and the Unicode code point of each character in the `message` string.

Maps in Golang

- In Go, a map is a data structure that provides an unordered collection of key-value pairs, where each key must be unique.
- *It is similar to dictionaries or hash maps in other programming languages.*
- Maps are used to associate values with keys and allow for efficient retrieval of values based on those keys.

```go
package main

import "fmt"

func main() {
    // Creating a map
    studentGrades := make(map[string]int)

    // Adding key-value pairs
    studentGrades["Alice"] = 90
    studentGrades["Bob"] = 85
    studentGrades["Charlie"] = 95

    // Accessing values
    fmt.Println("Alice's Grade:", studentGrades["Alice"])

    // Modifying values
    studentGrades["Bob"] = 88

    // Deleting a key-value pair
    delete(studentGrades, "Charlie")

    // Checking if a key exists
    grade, exists := studentGrades["David"]
    fmt.Println("David's Grade Exists:", exists)
    fmt.Println("David's Grade:", grade)

    // Iterating over the map
    fmt.Println("Student Grades:")
    for name, grade := range studentGrades {
        fmt.Printf("%s: %d\n", name, grade)
    }
}
```

In this example:

- The `make` function is used to create an empty map called `studentGrades`.
- Key-value pairs are added to the map for students' grades.
- Values are accessed using the key, and modifications or deletions can be performed.
- The existence of a key is checked using the **second return value of a map lookup**.
- Iteration over the map is done using a `for` loop.
- In maps, we can use the `make` function to create an empty map with an initial capacity, but it won't allow you to specify initial key-value pairs directly.

Some key points about maps:

1. **Unordered:** Maps in Go are unordered, meaning there is no guaranteed order of key-value pairs when iterating over them.
2. **Dynamic Size:** The size of a map can grow or shrink dynamically as key-value pairs are added or removed.
3. **Keys and Values:** Keys and values can be of any comparable type, but all keys must be of the same type, and all values must be of the same type.
4. **Initialization:** Maps can be initialized using the `make` function, or using a map literal.

Here's an example using a map literal:

```
// Creating a map using a literal
studentGrades := map[string]int{
    "Alice":   90,
    "Bob":     85,
    "Charlie": 95,
}
```

- This is a concise way to create and initialize a map in one line.

Struct in Golang

- In Go, a struct (short for "structure") is a composite data type that groups together variables (fields or members) under a single name.
- Each field in a struct can have a different data type, and structs are used to create more complex data structures.

```go
package main

import "fmt"

// Define a struct named Person
type Person struct {
    FirstName string
    LastName  string
    Age       int
}

func main() {
    // Create an instance of the Person struct
    var person1 Person
    person1.FirstName = "John"
    person1.LastName = "Doe"
    person1.Age = 30

    // Access the fields of the struct
    fmt.Println("First Name:", person1.FirstName)
    fmt.Println("Last Name:", person1.LastName)
    fmt.Println("Age:", person1.Age)
}
```

In this example:

- The `Person` struct is defined with three fields: `FirstName`, `LastName`, and `Age`.
- An instance of the struct is created using `var person1 Person`.
- The fields of the struct are accessed and modified using dot notation (`person1.FirstName`, etc.).

Struct Initialization

- You can initialize a struct using various methods:

```
// Method 1: Using the var keyword
var person2 Person
person2.FirstName = "Alice"
person2.LastName = "Smith"
person2.Age = 25

// Method 2: Using a struct literal
person3 := Person{
    FirstName: "Bob",
    LastName:  "Johnson",
    Age:       35,
}

// Method 3: Using the new keyword (returns a pointer to the struct)
person4 := new(Person)
person4.FirstName = "Eve"
person4.LastName = "Taylor"
person4.Age = 28
```

Struct Embedding

- Structs can be embedded within other structs, allowing for a form of composition

```
type Address struct {
    Street  string
    City    string
```

```
    Country string
}

type Contact struct {
    Email string
    Phone string
}

type Employee struct {
    Person   // Embedded struct
    Address  // Embedded struct
    Contact  // Embedded struct
    Position string
}

// Creating an instance of the Employee struct
employee := Employee{
    Person: Person{
        FirstName: "Frank",
        LastName:  "Miller",
        Age:       45,
    },
    Address: Address{
        Street:  "123 Main St",
        City:    "Anytown",
        Country: "USA",
    },
    Contact: Contact{
        Email: "frank@example.com",
        Phone: "555-1234",
    },
    Position: "Manager",
}
```

- Structs are a fundamental building block in Go, and they are widely used to model data structures and organize data in a meaningful way. They play a crucial

role in creating complex types and managing relationships between different pieces of data.

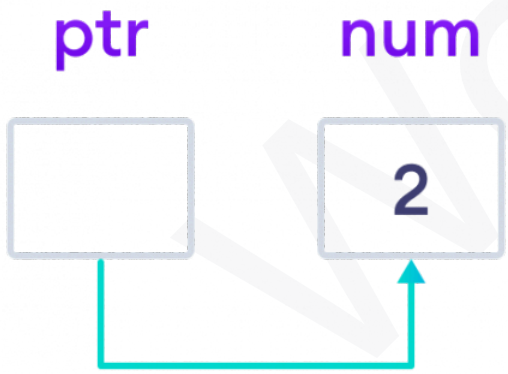
Pointers in Golang

- In Go, a pointer is a variable that stores the memory address of another variable.
- Pointers are used to indirectly refer to the value stored in a variable, rather than the value itself.
- They provide a way to work with the memory directly, which can be useful for various programming tasks, including efficient memory management and sharing data between functions.
- *To declare a pointer, you use the * (asterisk) symbol followed by the type of the variable it will point to. You can then initialize the pointer using the address-of (&) operator.*

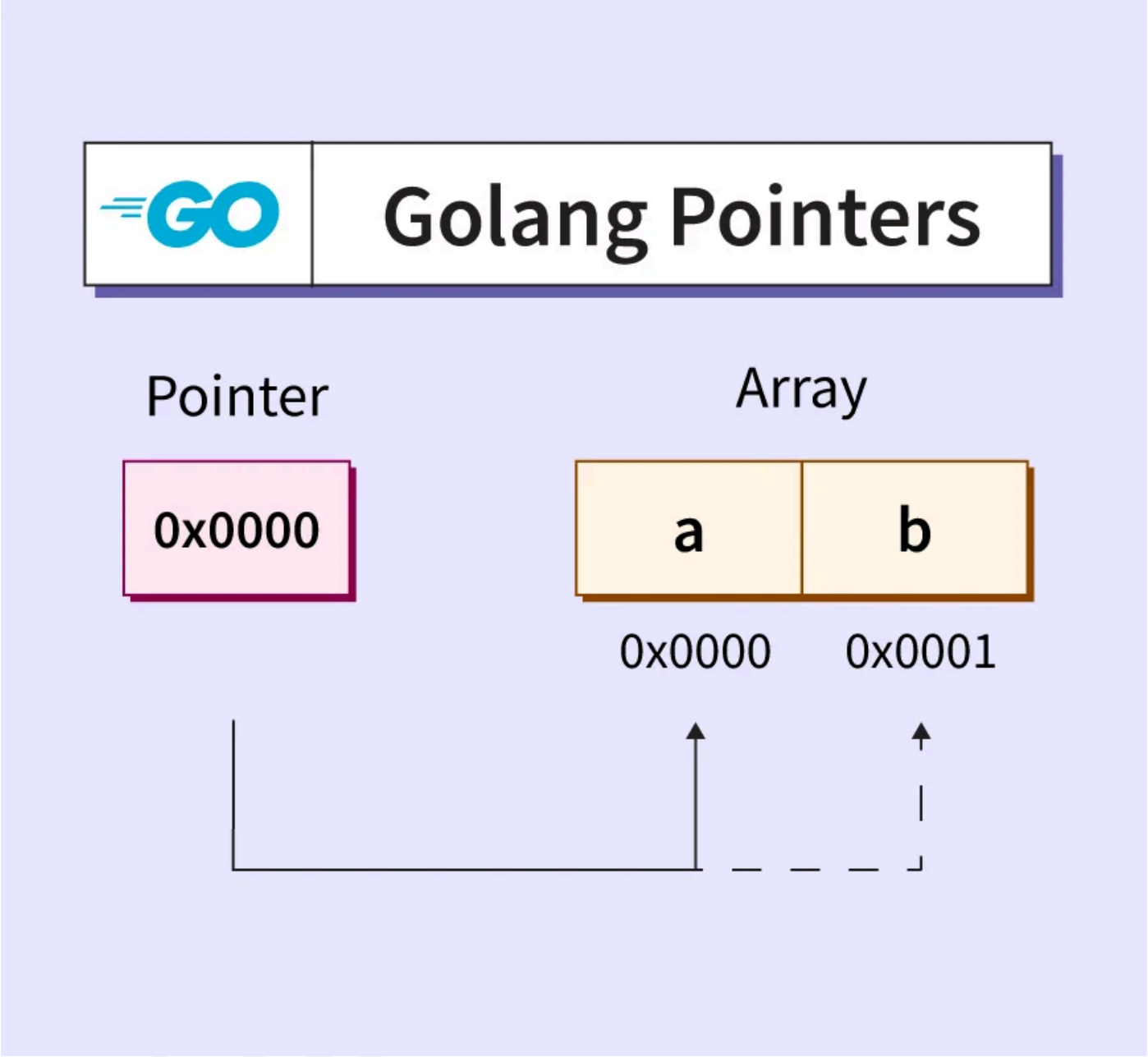
```
package main

import "fmt"

func main() {
    // Declare a variable and a pointer
    var num int = 2
    var ptr *int

    // Initialize the pointer with the address of the variable
    ptr = &num

    // Access the value through the pointer
    fmt.Println("Value of num:", num)
    fmt.Println("Value through pointer:", *ptr)
```

```
    // variable declaration and assignment on the same line.
    data := "prince"
    pointer := &data
    fmt.Println("Value through pointer: ", *pointer)
}
```

In this example, `ptr` is a pointer to an `int`, and it is initialized with the memory address of the `num` variable.



- Detail Pointer Video: https://youtu.be/-x0QnooY6O8



Nil Pointers

- In Go, pointers are initialized with nil by default if not explicitly set to point to a valid memory address. A nil pointer doesn't point to any valid memory location.

```go
package main

import "fmt"

func main() {
    var ptr *int

    // Check if the pointer is nil
    if ptr == nil {
        fmt.Println("Pointer is nil")
    }
}
```

Pointers are frequently used to pass variables by reference to functions, allowing the function to modify the original value. This is particularly useful when dealing with large data structures where passing by value would be inefficient.

```go
package main

import "fmt"

func modifyValueByReference(num *int) {
    *num = *num * 2
}

func main() {
    value := 10
    modifyValueByReference(&value)
    fmt.Println("Modified value:", value)
}
```

- We can modify the value by accessing the address

Data Conversion

- In Go, data conversion refers to the process of converting a value from one data type to another.

Numeric Type Conversion:

- When working with different numeric types (integers, floats), you may need to explicitly convert between them.

```
var integerNum int = 42
var floatNum float64 = float64(integerNum)
```

String Conversion:

- Converting numeric values or other types to strings and vice versa is a common operation.

```
var number int = 42
str := strconv.Itoa(number) // Integer to string

strNum := "123"
num, err := strconv.Atoi(strNum) // String to integer
```

- Convert string to float

```
 str := "3.14"
 num, err := strconv.ParseFloat(str, 64)
 if err == nil {
     fmt.Println(num)
 }
```

- These are just a few examples, and the `strconv` package offers more functions

for handling different types of conversions. Always check the Go documentation for the most up-to-date and comprehensive information

Strings package

- The `strings` package in Go provides a variety of functions for manipulating strings.

Splitting Strings:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str := "apple,orange,banana"
    parts := strings.Split(str, ",")
    fmt.Println(parts) // Output: [apple orange banana]
}
```

Counting Occurrences:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
```

```
    str := "one two three four two two five"

    count := strings.Count(str, "two")
    fmt.Println("Count:", count)
}
```

Trimming Whitespace:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str := "   Hello, Go!   "

    trimmed := strings.TrimSpace(str)
    fmt.Println("Trimmed:", trimmed)
}
```

Concatenating Strings:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    str1 := "Hello"
    str2 := "World"
```

```
    result := strings.Join([]string{str1, str2}, " ")
    fmt.Println(result)
}
```

These are just a few examples of the many functions provided by the `strings` package in Go. The package is quite versatile and allows you to perform various operations on strings, making it a valuable tool for string manipulation in Go programs.

Time & Date Conversion

- Go provides a powerful `time` package for handling time and date-related operations.
- the format "2006-01-02 15:04:05" might seem a bit strange at first, but it's chosen for a specific reason. In Go's time package, the reference date and time for formatting are fixed as "**January 2, 2006, 3:04:05 PM**", MST (Mountain Standard Time)
- This time is chosen because it is easy to remember and it is also the time when Go was officially announced.

Let's break down the format "2006-01-02 15:04:05" step by step:

- **2006:** This represents the year part of the date. The year is written as 2006, not the conventional YYYY format.
- **01:** This represents the month part of the date. The month is written as 01, not in the conventional MM format. January is represented by 01.
- **02:** This represents the day part of the date. The day is written as 02, not the conventional DD format.
- **15:** This represents the hour part of the time in a 24-hour format.
- **04:** This represents the minute part of the time.
- **05:** This represents the second part of the time.

```go
package main

import (
	"fmt"
	"time"
)

func main() {
	currentTime := time.Now()
	formattedTime := currentTime.Format("2006-01-02 15:04:05")
	fmt.Println("Formatted Time:", formattedTime)
}
```

`Format("2006-01-02 15:04:05")` tells Go to replace each part of the layout with the actual year, month, day, hour, minute, and second components from `currentTime`.

- This fixed reference date is a unique aspect of Go, and it helps to avoid confusion and errors in date and time formatting. Once you get used to it, it becomes a memorable and consistent way to work with time in Go.
- Format() will give a string as a returned value.
- In Go's time package, the reference time "2006-01-02 15:04:05" uses a 24-hour clock, so it doesn't directly represent AM or PM. However, you can use the "3:04 PM" format if you want to display the time in a 12-hour clock format with AM/PM.

```go
package main

import (
	"fmt"
	"time"
)

func main() {
	currentTime := time.Now()

	// Use the "3:04 PM" format for a 12-hour clock with AM/PM
	formattedTime := currentTime.Format("2006-01-02 3:04 PM")
	fmt.Println("Formatted Time:", formattedTime)
}
```

- In this example, the layout string "2006-01-02 3:04 PM" instructs Go to use a 12-hour clock with AM/PM. The "3" in the layout represents the hour in a 12-hour clock format.
- The `time.Parse` function in Go is used to parse a formatted string representing a time and convert it into a `time.Time` value.

```go
package main

import (
    "fmt"
    "time"
)

func main() {
```

```
    dateStr := "2023-11-25"
    parsedTime, err := time.Parse("2006-01-02", dateStr)

    if err == nil {
        fmt.Println("Parsed Time:", parsedTime)
    } else {
        fmt.Println("Error parsing time:", err)
    }
}
```

- We can also add or subtract Time

```
package main

import (
    "fmt"
    "time"
)

func main() {
    currentTime := time.Now()

    // Add 1 day
    newTime := currentTime.Add(24 * time.Hour)

    fmt.Println("Current Time:", currentTime)
    fmt.Println("New Time (after adding 1 day):", newTime)
}
```

- If you want to add the day in the format then “Monday” is the day which we have to use in the formatting

```
package main
```

```
import (
     "fmt"
     "time"
)

func main() {
     currentTime := time.Now()
     newTime := currentTime.Format("02.01.2006 Monday")
     fmt.Println("New Time:", newTime)
}
```

Defer in Golang

- In Go, The `defer` statement in Go is used to ensure that a function call is performed later in a program's execution.

```
package main

import "fmt"

func main() {
     fmt.Println("Start of the program")

     // The function inside defer will be executed when the surrounding
function (main in this case) exits
     defer fmt.Println("This will be executed at the end")

     fmt.Println("Middle of the program")
}
```

In this example, when you run the program, you'll see the output:

```
Start of the program
Middle of the program
This will be executed at the end
```

Even though the `defer` statement is encountered before the "Middle of the program" `Println`, the deferred function is executed after the surrounding `main` function finishes.

- When you have multiple `defer` statements in a function, they are executed in a last-in, first-out (LIFO) order. The last `defer` statement you encounter in the code will be the first one to be executed when the function exits.

```
package main

import "fmt"

func main() {
     fmt.Println("Start of the program")

     defer fmt.Println("This will be executed second")

     defer fmt.Println("This will be executed first")

     fmt.Println("Middle of the program")
}
```

In this example, when you run the program, the output will be:

```
Start of the program
Middle of the program
```

```
This will be executed second
This will be executed first
```

Even though the second `defer` statement appears earlier in the code, it will be executed before the first one because of the LIFO order.

Each `defer` statement is stacked onto a list, and when the function exits, the statements are executed in reverse order. This can be useful for cleanup tasks or for ensuring that resources are released in the reverse order in which they were acquired.

Files in Golang

- File operations in Go involve working with the `os` and `io/ioutil` packages. Here's a basic guide to common file operations like creating, reading, and writing files

```go
package main

import (
	"fmt"
	"io"
	"os"
)

func main() {
	// Create a new file or truncate an existing one
	file, err := os.Create("example.txt")
	if err != nil {
		fmt.Println("Error creating file:", err)
		return
	}
	defer file.Close()
```

```
    // Initial content to be added to the file
    initialContent := "Hello, this is the initial content."

    // Write the initial content to the file using io.WriteString
    _, err = io.WriteString(file, initialContent+"\n")
    if err != nil {
        fmt.Println("Error writing to file:", err)
        return
    }

    fmt.Println("File created with initial content.")
}
```

- we can use the `io` package in Go for writing to a file. The `io.WriteString` function is a convenient way to write a string to a file.
- `file.Close()` is important in many cases when you are done working with a file. When you open a file using functions like `os.Create`, `os.Open`, or others, you are acquiring system resources to interact with that file. Failing to close the file properly can lead to resource leaks and might cause issues like running out of file descriptors. The `Close()` function is responsible for releasing those resources.
- The `io.WriteString` function returns two values:
    1. **The number of bytes written:** This is an integer representing the number of bytes written to the file.
    2. **An error:** If the write operation encounters an error, it will be non-nil, indicating that something went wrong.

- Reading File Buffer

```
package main

import (
	"fmt"
	"io"
	"os"
)

func main() {
	// Open the file
	file, err := os.Open("example.txt")
	if err != nil {
		fmt.Println("Error opening file:", err)
		return
	}
	defer file.Close()
```

```
    // Create a buffer to read the file content
    buffer := make([]byte, 1024)

    // Read the file content into the buffer
    for {
        n, err := file.Read(buffer)
        if err == io.EOF {
            break // End of file reached
        }
        if err != nil {
            fmt.Println("Error reading file:", err)
            return
        }

        // Process the read content (in this example, just print it)
        fmt.Print(string(buffer[:n]))
    }
}
```

- `os.Open("example.txt")` attempts to open the file named "example.txt."
- If there's an error (e.g., the file doesn't exist), it prints an error message and exits the program.
- `defer file.Close()` ensures that the file is closed when the `main` function exits, regardless of how it exits.
- **Buffer Creation:**
  - `make([]byte, 1024)` creates a byte slice (buffer) with a capacity of 1024 bytes. This buffer will be used to read chunks of the file.
- **Read File Content:**
  - `file.Read(buffer)` reads content from the file into the buffer.
  - The loop continues until the end of the file (EOF) is reached.
  - If there's an error during the read operation (other than EOF), it prints an error message and exits the program.
  - `fmt.Print(string(buffer[:n]))` processes and prints the read content. In this example, it prints the content as a string.

- Reading File ioutil

- There is an easy way to read data from file
- There are other ways to read a file in Go, and one commonly used approach is to use the `ioutil.`
- `ReadFile` function from the `io/ioutil` package. This function simplifies the process of reading the entire contents of a file into a byte slice.

```
package main

import (
	"fmt"
	"io/ioutil"
)

func main() {
	// Read the entire file into a byte slice
	content, err := ioutil.ReadFile("example.txt")
	if err != nil {
		fmt.Println("Error reading file:", err)
		return
	}
```

```
    // Process the file content (in this example, just print it)
    fmt.Println(string(content))
}
```

1. `ioutil.ReadFile("example.txt")` reads the entire content of the file into a byte slice (`content`).
2. If there's an error during the read operation, it prints an error message and returns.
3. The file content is then processed (in this case, printed).

Using `ioutil.ReadFile` is convenient for scenarios where you want to read the entire content of a file into memory. However, keep in mind that it might not be suitable for very large files as it loads the entire content into memory at once. For larger files, reading in chunks as shown in the previous example may be more appropriate.

Web Request in Golang

- In Go, a ***web request*** refers to an HTTP request made to a web server. These requests are used to retrieve or send data over the internet, typically to interact with web applications or APIs.
- In Go, you can make web requests using the `net/http` package, which provides functions to create and send HTTP requests, as well as handle responses.

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
```

```
)

func main() {
	fmt.Println("Learning web request...")

	// Make a GET request

	response, err :=
http.Get("https://jsonplaceholder.typicode.com/todos/1")
	fmt.Printf("type of response: %T\n", response);
	if err != nil {
		fmt.Println("Error making GET request: ", err)
		return
	}
	defer response.Body.Close()

	// Read the response body
	body, err := ioutil.ReadAll(response.Body)
	if err != nil {
		fmt.Println("Error reading response body:", err)
		return
	}

	// Print the response body
	fmt.Println(string(body))
}
```

- `defer response.Body.Close()` statement is used to ensure that the response body is closed after you have finished reading from it. In Go, it's important to close resources like **network connections** and **file handles** to free up system resources.
- **Resource Management:** HTTP responses in Go are represented by `http.Response` objects, which have a `Body` field containing the response body. This body is a stream of data from the server, and it's important to close this stream once you're done reading from it to release the associated resources.
- ***`ioutil.ReadAll(response.Body)`***: This line reads the entire response body (`response.Body`) into a byte slice (`body`). `ioutil.ReadAll` reads from the `response.Body` until an EOF (end of file) is reached and returns the data read and an error. If there's an error reading the response body (e.g., connection closed, invalid

data), it will be stored in `err`.

- ***`fmt.Println(string(body))`***: This line converts the byte slice `body` to a string and prints it to the console. The response body typically contains the content (data) returned by the server in response to the request.

URL in Golang

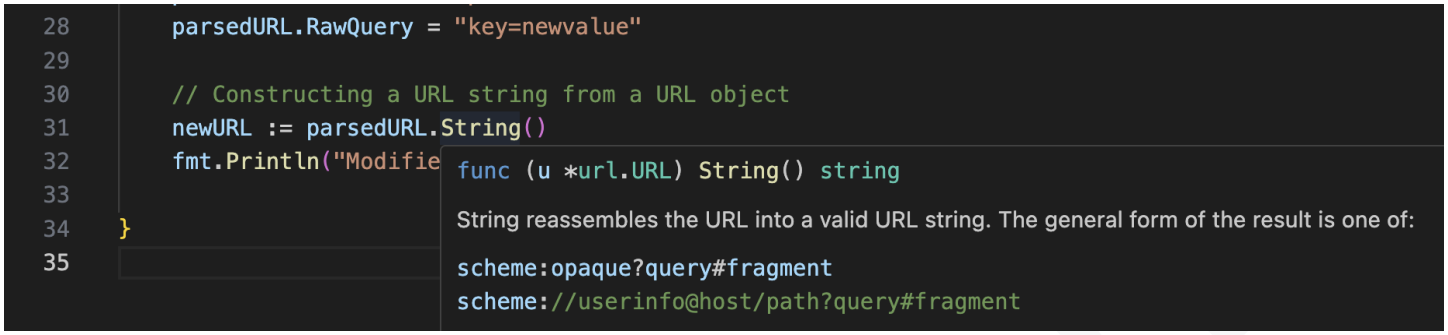
- In Go, The `net/url` package provides functionalities to parse, construct, and manipulate URLs (Uniform Resource Locators). URLs are used to identify resources on the internet, such as web pages, images, and files.
- Here's a beginner-friendly explanation of some key concepts in the `net/url` package:

- **Parsing URLs:** The `url.Parse` function is used to parse a **string into a URL object** (`url.URL` struct). This allows you to break down a URL into its individual components, such as scheme, host, path, and query parameters.
- **Accessing URL Components:** Once you have a URL object, you can access its components using various fields:
  - `Scheme`: Indicates the protocol used (e.g., "http", "https").
  - `Host`: Specifies the domain name and optionally the port number.
  - `Path`: Represents the path component of the URL, which specifies the resource's location on the server.
  - `RawQuery`: Contains the raw query string, including query parameters.
- **Query Parameters:** Query parameters are key-value pairs appended to the end of a URL, usually starting with a `?` and separated by `&`. The `url.Values` type is used to represent query parameters as a map. You can access and manipulate query parameters using methods like `Get`, `Set`, and `Add`.
- **Constructing URLs:** The `url.URL` struct provides methods to construct URLs:
  - `String()`: **Converts a URL object back to its string representation**.
  - `ResolveReference()`: Resolves a relative URL against a base URL to create an absolute URL.
- **Modifying URLs:** You can modify a URL object by directly setting its fields, such as `Scheme`, `Host`, `Path`, and `RawQuery`.

```go
package main

import (
	"fmt"
	"net/url"
)

func main() {
	// Parse a URL string into a URL object
	myURL :=
"https://example.com:8080/path/to/resource?key1=value1&key2=value2"
	parsedURL, err := url.Parse(myURL)
	if err != nil {
		fmt.Println("Error parsing URL:", err)
		return
	}

	// Accessing URL components
	fmt.Println("Scheme:", parsedURL.Scheme)
	fmt.Println("Host:", parsedURL.Host)
	fmt.Println("Path:", parsedURL.Path)
	fmt.Println("RawQuery:", parsedURL.RawQuery)

	// Modifying URL components
	parsedURL.Scheme = "http"
	parsedURL.Host = "newhost.com"
	parsedURL.Path = "/newpath"
	parsedURL.RawQuery = "key=newvalue"

	// Constructing a URL string from a URL object
	newURL := parsedURL.String()
	fmt.Println("Modified URL:", newURL)
}
```

- In this example, we start by parsing a URL string using `url.Parse`, which returns a `url.URL` object representing the parsed URL.
- We then access and print various components of the URL, such as the scheme, host, path, and query string.

- we demonstrate how to modify the URL components by directly setting the fields of the `url.URL` object.
- we use the `String` method to convert the modified `url.URL` object back to a string representation.



JSON in Golang

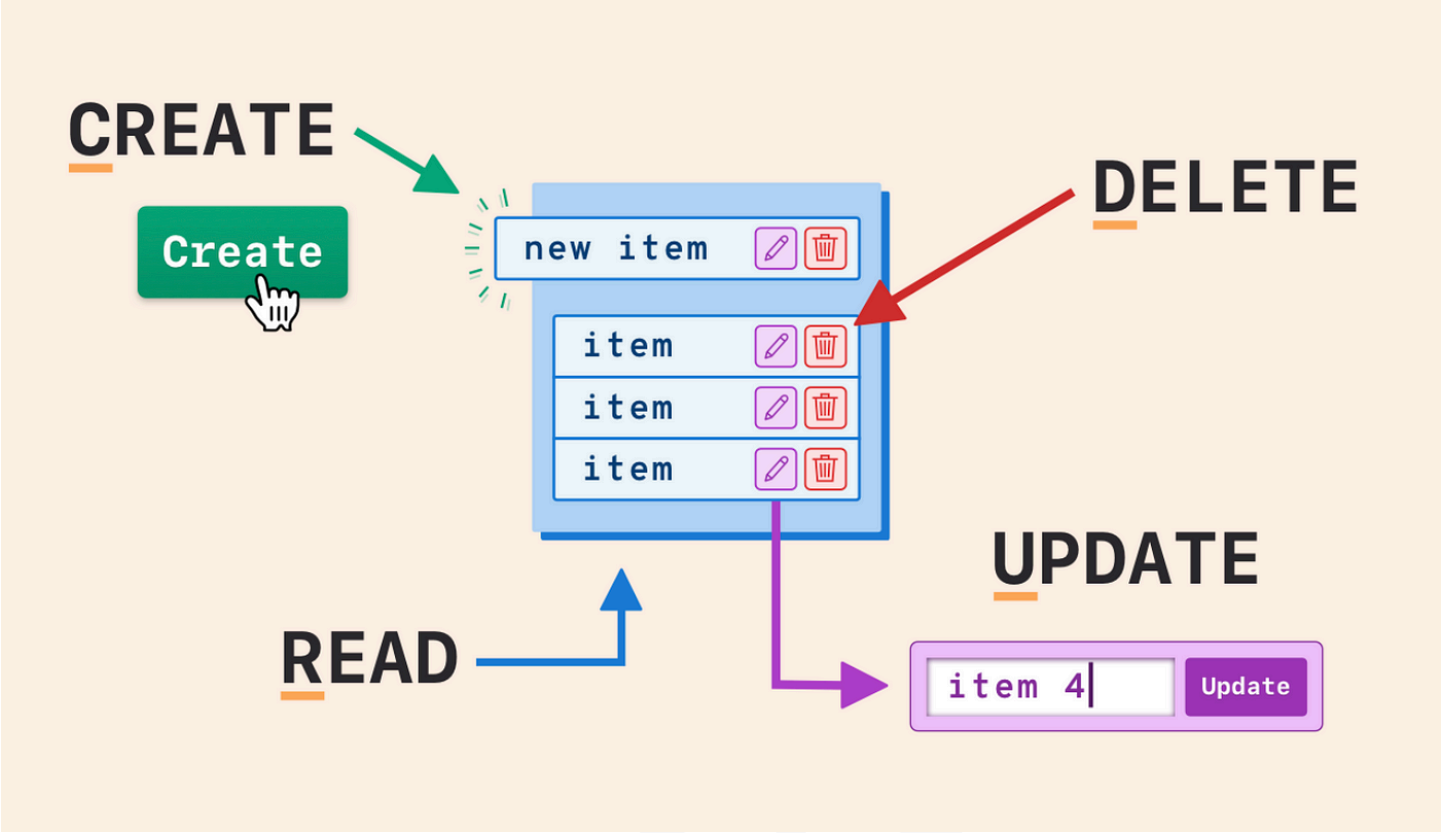
- In Go, the `encoding/json` package is used to **encode and decode JSON** (JavaScript Object Notation) data. JSON is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.

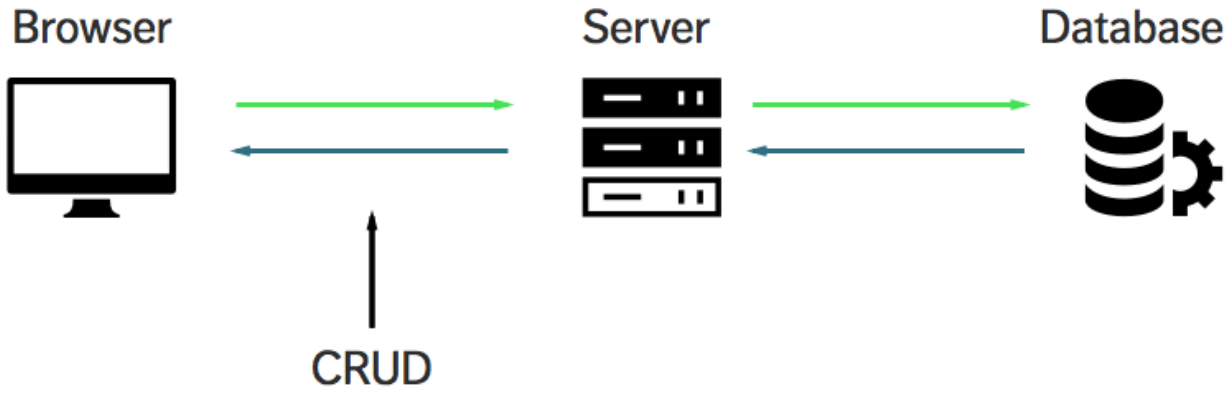
- Here's a basic example to demonstrate how to define and use JSON in Go:
- **Defining a Struct:** Define a struct that represents the JSON data structure. Each field in the struct should have a tag specifying the JSON key associated with it.
- **Marshalling (Encoding):** Use `json.Marshal` to convert a Go struct into a JSON-encoded byte array.
- **Unmarshalling (Decoding):** Use `json.Unmarshal` to convert a JSON-encoded byte array into a Go struct.

```
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
```

```
	Name    string `json:"name"`
	Age     int    `json:"age"`
	IsAdult bool   `json:"is_adult"`
}

func main() {
	// Encoding (Marshalling)
	person := Person{Name: "Alice", Age: 30, IsAdult: true}
	jsonData, err := json.Marshal(person)
	if err != nil {
		fmt.Println("Error marshalling JSON:", err)
		return
	}
	fmt.Println("JSON data:", string(jsonData))

	// Decoding (Unmarshalling)
	var newPerson Person
	err = json.Unmarshal(jsonData, &newPerson)
	if err != nil {
		fmt.Println("Error unmarshalling JSON:", err)
		return
	}
	fmt.Println("Decoded Person:", newPerson)
}
```

CRUD in Golang





- **GET Method**
- For This operation, we are going to use
- https://jsonplaceholder.typicode.com/

Try it

Run this code here, in a console or from any site:

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
      .then(response => response.json())
      .then(json => console.log(json))
```

Run script

```
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
```

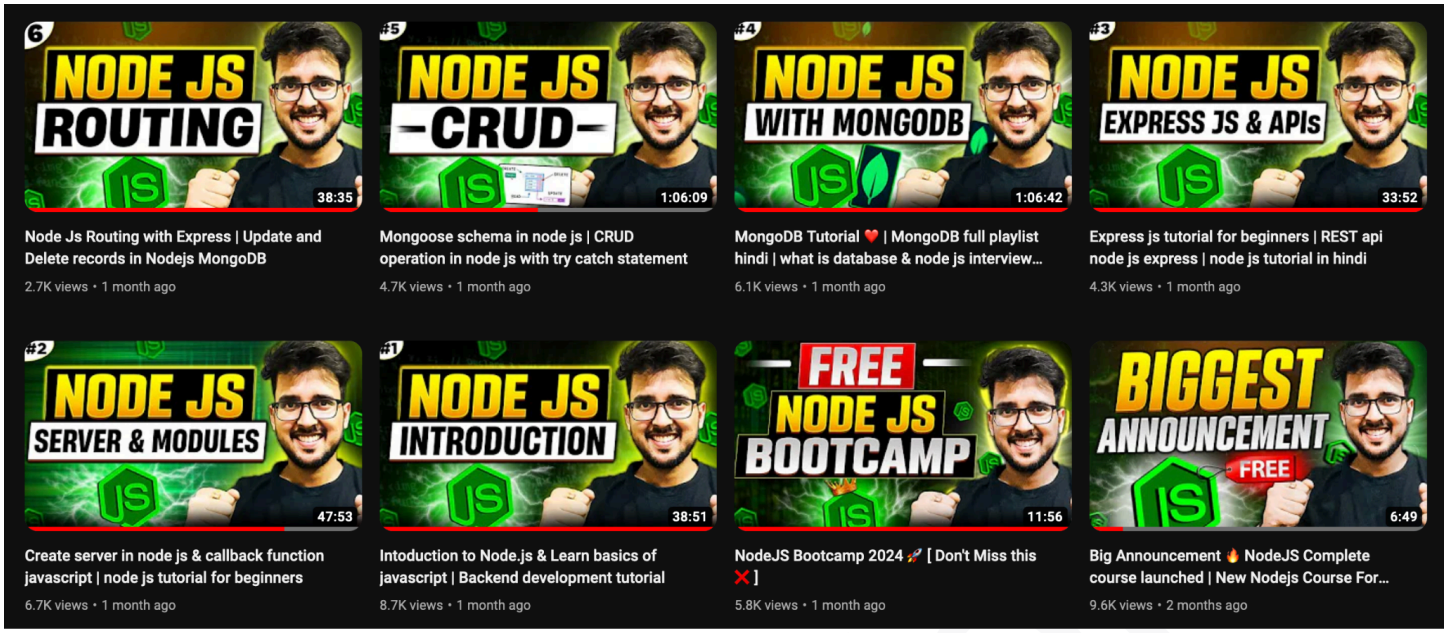
Routes

All HTTP methods are supported. You can use http or https for your requests.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

- Here, we can use any server API's. If you have your own then you can also use it
- E.g. we have created http://localhost:3000/
- In our NodeJS Hotel Management system

Hello World by Prince




```go
package main

import (
	"encoding/json"
	"fmt"
	"io/ioutil"
	"net/http"
)

type Todo struct {
	UserID    int    `json:"userId"`
	ID        int    `json:"id"`
	Title     string `json:"title"`
	Completed bool   // No `json` tag specified
}

func PerformGetRequest() {
	const myurl = "https://jsonplaceholder.typicode.com/todos/1"
	response, err := http.Get(myurl)
	if err != nil {
		panic(err)
	}
	defer response.Body.Close()
```

```
	if response.StatusCode != http.StatusOK {
		fmt.Println("Failed to fetch data:", response.Status)
		return
	}

	var todo Todo
	err = json.NewDecoder(response.Body).Decode(&todo)
	if err != nil {
		fmt.Println("Error decoding response:", err)
		return
	}

	fmt.Println("Todo ID:", todo.ID)
	fmt.Println("Todo Title:", todo.Title)
	fmt.Println("Todo Completed:", todo.Completed)
}

func main() {
	PerformGetRequest()
}
```

- Defines a struct named `Todo` with four fields: `UserID`, `ID`, `Title`, and `Completed`.
- Each field has a tag (e.g., `json:"userId"`) specifying the JSON key associated with it. This is used during JSON encoding and decoding to map struct fields to JSON keys.

In Go, when you use the `encoding/json` package to marshal or unmarshal structs to and from JSON, it follows certain default rules if you don't explicitly specify field tags. For example, when marshaling a struct to JSON, the default behavior is to use the struct field names as the JSON keys. Similarly, when unmarshaling JSON to a struct, the default behavior is to match JSON keys to struct field names.

*In this example, the `Completed` field in the `Todo` struct doesn't have a `json` tag, but the `encoding/json` package still correctly unmarshals the JSON string into the `Todo` struct because it uses the **default behavior of matching field names**.*

```
dataBytes, err := ioutil.ReadAll(response.Body)
if err != nil {
```

```
    fmt.Println("Error reading response body:", err)
    return
}

var todo Todo
err = json.Unmarshal(dataBytes, &todo)
if err != nil {
    fmt.Println("Error decoding response:", err)
    return
}
```

*You can use `ioutil.ReadAll(response.Body)` to read the response body into a byte slice (`[]byte`). However, using `json.NewDecoder(response.Body)` and `Decode` **are often preferred when working with JSON data** because they allow for streaming and efficient decoding of large JSON payloads without needing to load the entire payload into memory at once.*

- **POST Method**
- In Go, we can send data via the POST Method

```
func PerformPostRequest() {
     const myurl = "https://jsonplaceholder.typicode.com/todos"

     todo := Todo{
          UserID:    1,
          ID:        201,
          Title:     "Learn Go",
          Completed: false,
     }
```

```
    // Convert the Todo struct to JSON
    jsonData, err := json.Marshal(todo)
    if err != nil {
        fmt.Println("Error marshalling JSON:", err)
        return
    }

    // Convert the JSON byte slice to a string
    jsonStr := string(jsonData)

    // Create an io.Reader from the string
    jsonReader := strings.NewReader(jsonStr)

    // Send the POST request
    resp, err := http.Post(myurl, "application/json", jsonReader)
    if err != nil {
        fmt.Println("Error sending request:", err)
        return
    }
    defer resp.Body.Close()

    // Print the response status code
    fmt.Println("Response Status:", resp.Status)
}
```

- *In this case, `json.Marshal` returns a byte slice (`[]byte`) containing the JSON representation of the `todo` struct.*
- *`string(jsonData)` converts this byte slice to a string. However, since*
- *`strings.NewReader` expects a string, we use `string(jsonData)` to convert the byte slice to a string so that it can be passed to `strings.NewReader` to create an `io.Reader` for the request body.*
- *`http.Post` function, the third argument expects an `io.Reader` interface for the request body.*

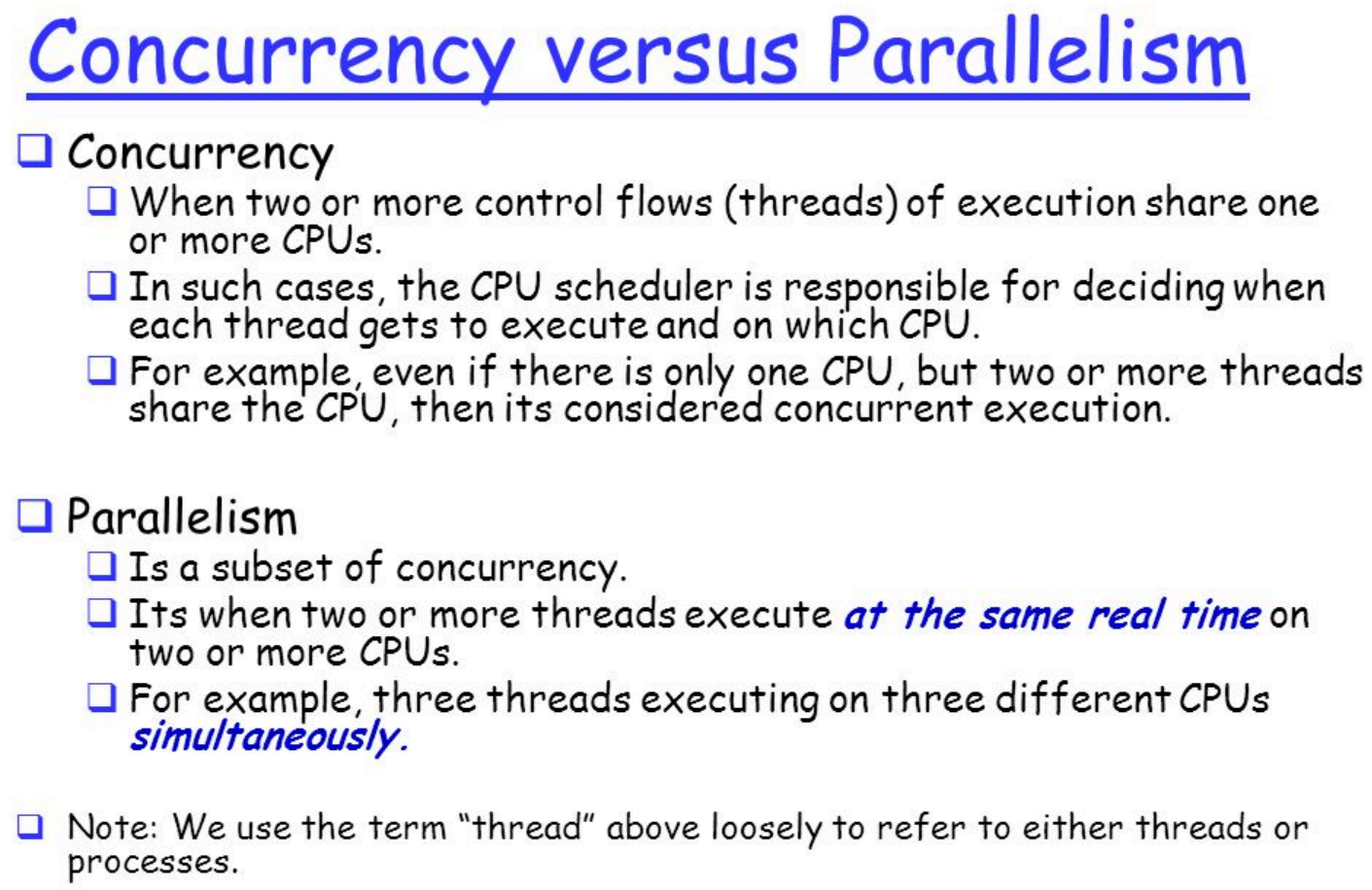
- **UPDATE Method**
- In Go, we can send data via the UPDATE Method

```go
func PerformUpdateRequest() {
	const myurl = "https://jsonplaceholder.typicode.com/todos/1"

	todo := Todo{
		UserID:    1,
		ID:        1,
		Title:     "Update Todo",
		Completed: true,
	}

	// Convert the Todo struct to JSON
	jsonData, err := json.Marshal(todo)
	if err != nil {
		fmt.Println("Error marshalling JSON:", err)
		return
	}

	// Create a PUT request
	req, err := http.NewRequest(http.MethodPut, myurl,
bytes.NewBuffer(jsonData))
	if err != nil {
		fmt.Println("Error creating request:", err)
		return
	}
	req.Header.Set("Content-Type", "application/json")

	// Send the request
	client := http.Client{}
	resp, err := client.Do(req)
	if err != nil {
		fmt.Println("Error sending request:", err)
		return
	}
	defer resp.Body.Close()

	fmt.Println("Response Status:", resp.Status)
}
```

- **DELETE Method**
- In Go, we can send data via the DELETE Method

```
func PerformDeleteRequest() {
     const myurl = "https://jsonplaceholder.typicode.com/todos/1"

     // Create a DELETE request
     req, err := http.NewRequest(http.MethodDelete, myurl, nil)
     if err != nil {
          fmt.Println("Error creating request:", err)
          return
     }

     // Send the request
     client := http.Client{}
     resp, err := client.Do(req)
     if err != nil {
          fmt.Println("Error sending request:", err)
          return
     }
     defer resp.Body.Close()

     fmt.Println("Response Status:", resp.Status)
}
```

Concurrency vs Parallelism

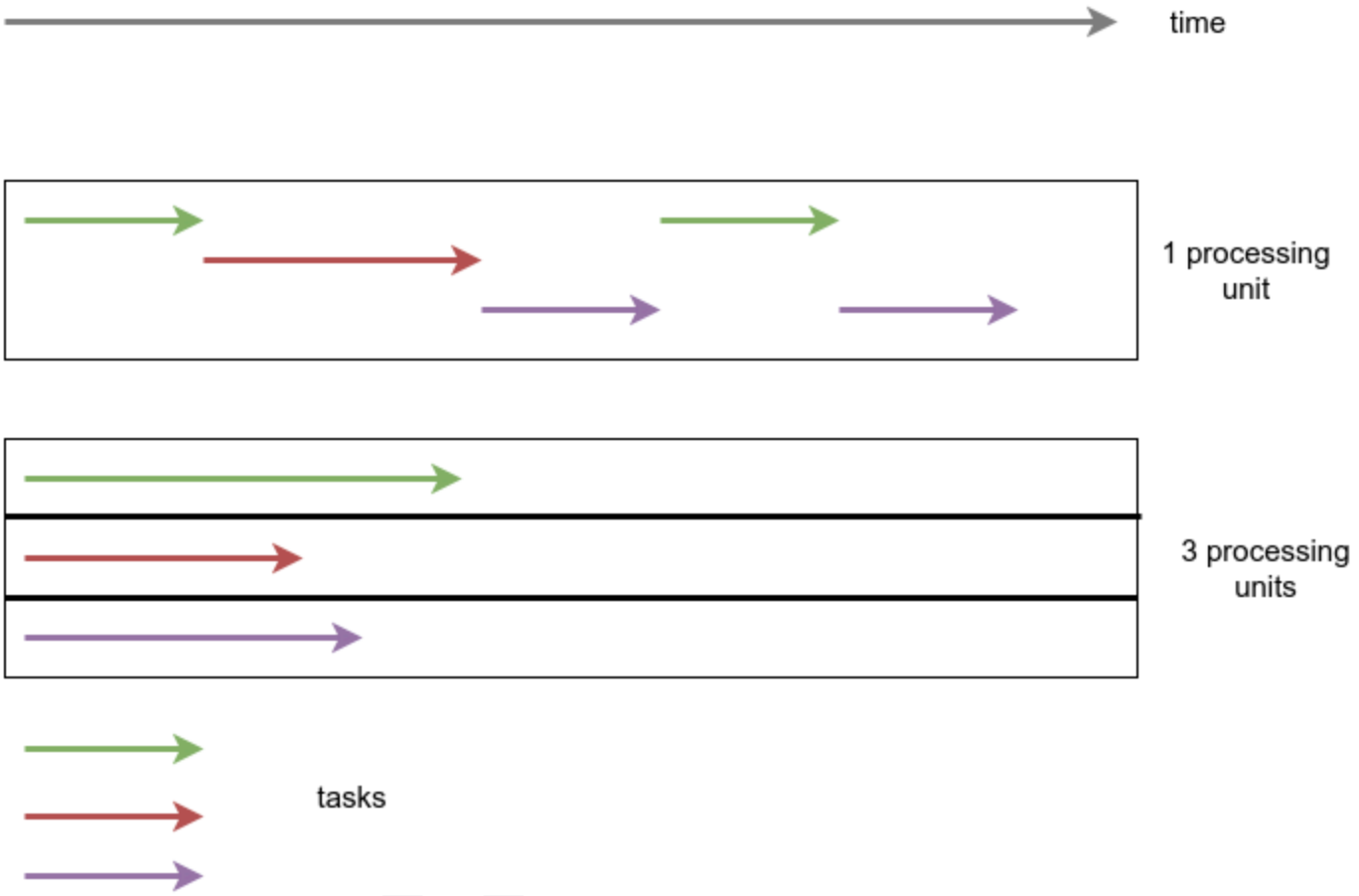


Concurrency: Think of concurrency as a way to structure your program so that it can handle multiple tasks at the same time. It's like a chef in a kitchen working on several dishes simultaneously. In programming, concurrency allows different parts of a program to be executed out of order or in partial order, enabling tasks to start, run, and complete in overlapping periods.

Parallelism: Parallelism, on the other hand, is about actually executing multiple tasks simultaneously. It's like having multiple chefs working on different dishes at the same time, each with their own stove and ingredients. In programming, parallelism involves utilizing multiple processors or cores to run different parts of a program at the same time, achieving true simultaneous execution.

Hello World by Prince

In essence, concurrency is a way to structure your program to make efficient use of available resources, allowing tasks to overlap in time. Parallelism, on the other hand, is about executing these tasks simultaneously to speed up overall execution. While related, they are not the same thing, and a language like Go uses goroutines and its scheduler to enable efficient concurrent execution.



- **processor** = chef
- **Task** = Vegetable cutting, Dishes cleaning, Cooking food

Goroutines in Golang

- Goroutines are a key feature of the Go programming language that allow you to run functions concurrently, or in parallel, with other parts of your program.
- Imagine you have a task that can be broken down into smaller sub-tasks that can be executed independently. Instead of waiting for each sub-task to finish before starting the next one,

- you can use goroutines to execute them concurrently. This can greatly improve the performance of your program, especially when dealing with tasks that involve I/O operations or other types of blocking operations.

```
package main

import (
	"fmt"
	"time"
)

func sayHello() {
	fmt.Println("Hello")
}

func sayHi() {
	fmt.Println("Hi Prince :)")
}

func main() {
	// Start a new goroutine
	go sayHello()
	sayHi()

	// Wait for a moment to allow the goroutine to finish
	time.Sleep(1000 * time.Millisecond)
}
```

- Do as much experiment on above code to make them understand how code execution works parallelly.

sync.WaitGroup in Golang

- `sync.WaitGroup` is a synchronization primitive in Go that is used to wait for a collection of goroutines to finish their execution.
- It allows you to coordinate the execution of multiple goroutines and ensure that they all complete before continuing with the rest of the program.

Here's a simple explanation of how `sync.WaitGroup` works:

1. You create a `sync.WaitGroup` variable to keep track of the number of goroutines you want to wait for.
2. For each goroutine you start, you increment the WaitGroup counter using the `Add` method.
3. Inside each goroutine, you call `Done` on the WaitGroup to signal that the goroutine has finished its work.
4. Finally, you call `Wait` on the WaitGroup to block the main goroutine until all other goroutines have called `Done`.

```
package main

import (
	"fmt"
	"sync"
)

func worker(id int, wg *sync.WaitGroup) {
	defer wg.Done() // Signal that this goroutine is done
	fmt.Printf("Worker %d starting\n", id)
	// Simulate some work
	fmt.Printf("Worker %d done\n", id)
}

func main() {
	var wg sync.WaitGroup

	// Start 3 worker goroutines
	for i := 1; i <= 3; i++ {
		wg.Add(1) // Increment the WaitGroup counter
		go worker(i, &wg)
```

```
    }

    // Wait for all workers to finish
    wg.Wait()
    fmt.Println("All workers done")
}
```

- In this example, we start three worker goroutines and use a `sync.WaitGroup` to wait for all of them to finish before printing "All workers done". The `Add` method is used to increment the WaitGroup counter for each goroutine, and `Done` is called in the `worker` function to signal that the goroutine has finished. The `Wait` method is called to block the main goroutine until all worker goroutines have completed.

Real Life Analogy of sync.Waitgroup

Let's consider a scenario where you are hosting a dinner party and you have multiple tasks to prepare before the guests arrive. Each task can be seen as a goroutine, and you want to ensure that all tasks are completed before the party starts.

In this analogy:

- Goroutines are like the individual tasks that need to be completed, such as setting the table, preparing the food, and decorating the room.
- `sync.WaitGroup` is like your checklist or to-do list for the party. You use it to keep track of the tasks that have been completed and to know when all tasks are done.
- Calling `Add` on the WaitGroup is like adding a task to your checklist.
- Calling `Done` is like checking off a task on your checklist when it's completed.
- Calling `Wait` is like waiting for all the tasks to be checked off before starting the party.

Using `sync.WaitGroup` is not mandatory, but it's a common and recommended way to synchronize goroutines when you need to wait for them to finish. However, there are other synchronization mechanisms in Go that you can use depending on your specific use case. you can also use **channels** to synchronize goroutines.

********** THE END **********