



Go Language – Comprehensive Study Notes

1. Introduction to Go Language

Go, also known as Golang, is an open-source programming language developed by Google in 2009. It was designed by Robert Griesemer, Rob Pike, and Ken Thompson to improve productivity in modern software development. Go offers simplicity like C but comes with powerful built-in features for concurrency, garbage collection, and strong typing. It compiles to native machine code, resulting in fast execution and small binary sizes. Go is widely used in cloud computing, backend systems, microservices, and networking applications. Its design philosophy emphasizes clarity, maintainability, and scalability. With a concise syntax, Go is easy to learn for beginners and highly efficient for professionals. Popular platforms like Docker, Kubernetes, and Terraform are built using Go. This makes it a vital language for students aiming at system-level and large-scale application development.

Key Points:

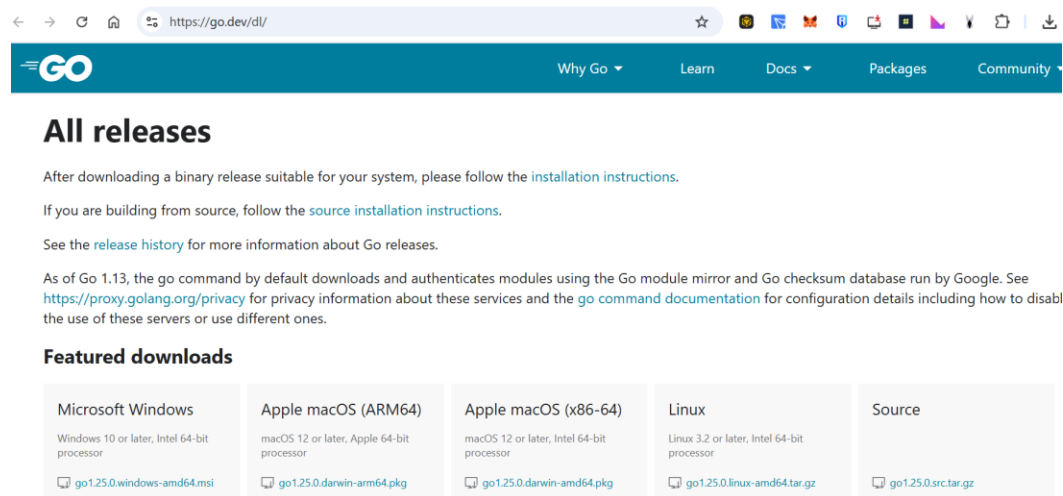
- Created in 2009 at Google.
- Statically typed, compiled language.
- Influenced by C but simpler.
- Has garbage collection.
- Strong concurrency support.
- Cross-platform compilation.
- Used in cloud-native tools.
- Fast compilation time.
- Minimalistic syntax.
- Supported by large community.

2. Setting Up Go Environment

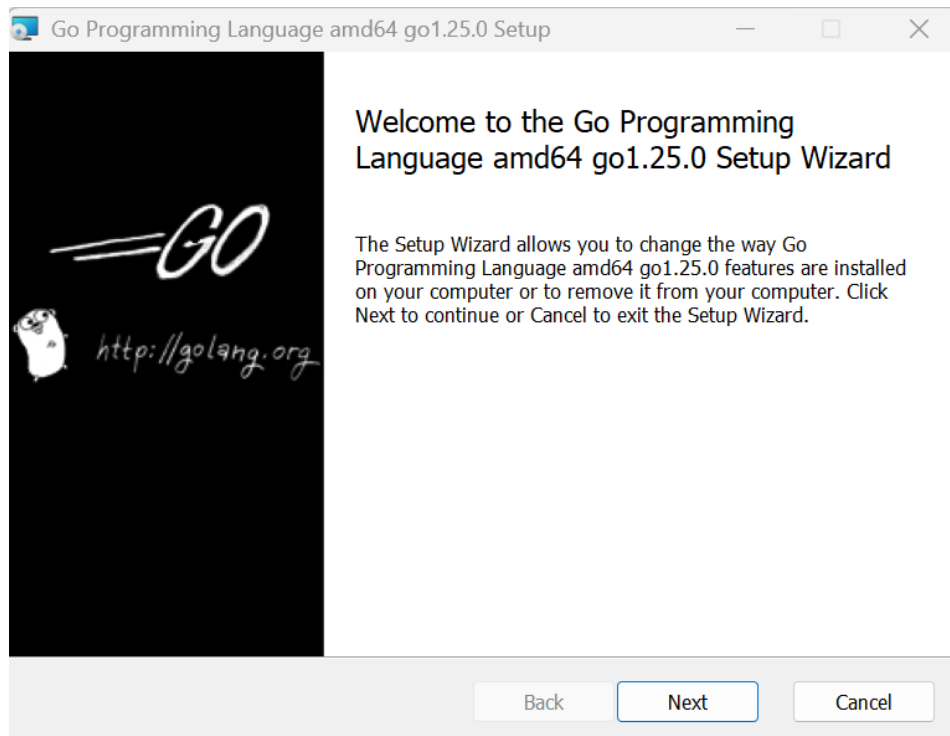
Before coding in Go, it's essential to set up the development environment properly. Go installation is simple and works on multiple operating systems like Windows, macOS, and Linux. The official Go distribution includes the compiler, standard libraries, and the go toolchain. Go uses an environment variable called GOPATH to locate source code, binaries, and packages. The modern Go version uses modules (go mod) for dependency management. Setting up an IDE or code editor such as Visual Studio Code, GoLand, or LiteIDE enhances productivity. Installing Go ensures that students can practice directly on their systems without relying on online compilers. Understanding the Go workspace structure helps in organizing code efficiently. This section will guide through installation, environment configuration, and testing setup.

Configuration process

- Download from golang.org.



- Install on Windows, Mac, Linux.



- Set GOROOT (Go installation directory).
- Set GOPATH (workspace location).
- Use go version to check installation.

```
Command Prompt
Microsoft Windows [Version 10.0.26100.4946]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP>go version
go version go1.25.0 windows/amd64

C:\Users\HP>
```

- Use `go env` to check variables.

```
C:\Users\HP>go env
set AR=ar
set CC=gcc
set CGO_CFLAGS=-O2 -g
set CGO_CPPFLAGS=
set CGO_CXXFLAGS=-O2 -g
set CGO_ENABLED=0
set CGO_FFLAGS=-O2 -g
set CGO_LDFLAGS=-O2 -g
set CXX=g++
set GCCGO=gccgo
set GOMODCACHE=C:\Users\HP\AppData\Local\go-build
set GOCACHE=C:\Users\HP\AppData\Local\go-build
set GOCACHEPROG=
set GODEBUG=
set GOENV=C:\Users\HP\AppData\Roaming\go\env
set GOEXE=.exe
set GOEXPERIMENT=
set GOFIPS140=off
set GOFLAGS=
set GOGCCFLAGS=-m64 -fno-caret-diagnostics -Qunused-arguments -Wl,--no-gc-sections -fmessage-length=0 -ffile-prefix-map=C:\Users\HP\AppData\Local\Temp\go-build2104061788=/tmp/go-build -gno-record-gcc-switches
set GOHOSTARCH=amd64
set GOHOSTOS=windows
set GOINSECURE=
set GOMOD=NUL
set GOMODCACHE=C:\Users\HP\go\pkg\mod
```

Create testing directory

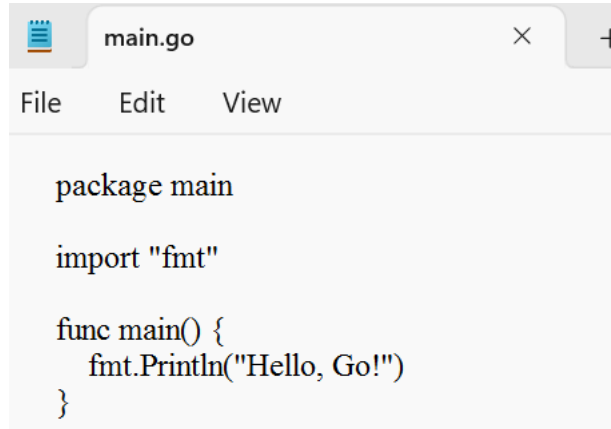
```
C:\Users\HP>md testing
```

Initialize a Go Module

Navigate into your project directory and run `go mod init <module_path>`. This creates a `go.mod` file for dependency management.

```
C:\Users\HP>go mod init testing
go: creating new go.mod: module testing
go: to add module requirements and sums:
    go mod tidy
```

Create following file in “testing” directory

A screenshot of a code editor window titled 'main.go'. The editor has a menu bar with 'File', 'Edit', and 'View'. The code content is as follows:

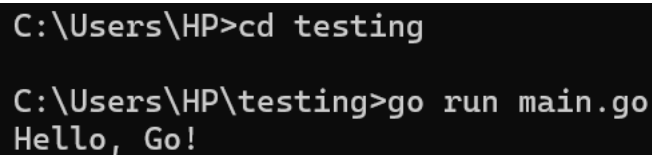
```
package main

import "fmt"

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

Test setup using `go run`.

Running main.go in Testing directory

A terminal window screenshot showing the following commands and output:

```
C:\Users\HP>cd testing

C:\Users\HP\testing>go run main.go
Hello, Go!
```

3. Go Syntax and Data Types

Go’s syntax is clean and easy to read, making it beginner-friendly yet powerful. Like C, it uses braces `{}` for blocks and semicolons are automatically inserted. Go supports basic data types like integers, floats, strings, and booleans, along with complex types like arrays, slices, maps, and structs. Type inference using `:=` allows Go to automatically determine variable types, making code concise. Constants are declared using the `const` keyword, while variables use `var`. Strings in Go are UTF-8 encoded, enabling internationalization. Understanding syntax and data types is crucial because they form the building blocks for functions, loops, and data structures. Students should practice declaring variables, using operators, and formatting output with `fmt`.

Key Points:

- `var` for variables, `const` for constants.
- Data types: `int`, `float`, `string`, `bool`.
- Type inference with `:=`.
- Arrays (fixed size), slices (dynamic).
- Maps (key-value pairs).
- Structs (custom data types).

- UTF-8 strings.
- Operators: arithmetic, logical, relational.
- `fmt.Println` for output.
- Zero values for uninitialized variables.

Example

```
package main

import (
    "fmt"
)

func main() {
    // ----- Variables -----
    var age int = 25
    var pi float64 = 3.14159
    var name string = "Alice"
    var isStudent bool = true

    // ----- Constants -----
    const country = "India"

    // ----- Type inference with := -----
    city := "New Delhi"
    year := 2025

    // ----- Zero values -----
    var zeroInt int    // 0
    var zeroFloat float64 // 0.0
    var zeroString string // ""
    var zeroBool bool   // false

    // ----- Arrays (fixed size) -----
    var numbers [3]int = [3]int{10, 20, 30}

    // ----- Slices (dynamic size) -----
    fruits := []string{"Apple", "Banana", "Cherry"}
    fruits = append(fruits, "Mango") // add new element

    // ----- Maps (key-value pairs) -----
    capitals := map[string]string{
        "India": "New Delhi",
        "USA":   "Washington D.C.",
        "Japan": "Tokyo",
    }
}
```

```
// ----- Structs (custom data types) -----
type Person struct {
    Name string
    Age  int
    City string
}
person1 := Person{Name: "Bob", Age: 30, City: "Mumbai"}

// ----- UTF-8 strings -----
utf8Str := "नमस्ते" // Hindi greeting

// ----- Operators -----
sum := age + 5           // arithmetic
isAdult := age >= 18     // relational
canDrive := isAdult && !false // logical

// ----- Output -----
fmt.Println("Name:", name)
fmt.Println("Age:", age)
fmt.Println("Pi:", pi)
fmt.Println("Is Student:", isStudent)
fmt.Println("Country:", country)
fmt.Println("City:", city, "Year:", year)
fmt.Println("Zero Values:", zeroInt, zeroFloat, zeroString, zeroBool)
fmt.Println("Numbers:", numbers)
fmt.Println("Fruits:", fruits)
fmt.Println("Capitals:", capitals)
fmt.Println("Person Struct:", person1)
fmt.Println("UTF-8 String:", utf8Str)
fmt.Println("Sum:", sum, "Is Adult:", isAdult, "Can Drive:", canDrive)
}
```

Result

```
C:\Users\HP\testing>go run main.go
Name: Alice
Age: 25
Pi: 3.14159
Is Student: true
Country: India
City: New Delhi Year: 2025
Zero Values: 0 0 false
Numbers: [10 20 30]
Fruits: [Apple Banana Cherry Mango]
Capitals: map[India:New Delhi Japan:Tokyo USA:Washington D.C.]
Person Struct: {Bob 30 Mumbai}
UTF-8 String: नमस्ते
Sum: 30 Is Adult: true Can Drive: true
```

Control structures guide the execution flow of a Go program. Go supports conditional statements like if, if-else, and switch for decision-making. Loops in Go are implemented using the for keyword, which can act like a while loop, a traditional for loop, or an infinite loop. The break and continue statements allow control over loop execution. The goto statement exists but is rarely used in modern practice. Switch cases in Go can be used with expressions other than integers, making them more versatile. Error handling in Go is done through explicit checks rather than exceptions. Understanding these control structures helps students write logic-based programs efficiently.

Key Points:

- if and if-else for decisions.

Example

```
package main
```

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

```
func main() {  
    age := 20  
    hasLicense := true  
  
    if age >= 18 && hasLicense {  
        fmt.Println("You are allowed to drive.")  
    } else if age >= 18 && !hasLicense {  
        fmt.Println("You are old enough but need a driving license.")  
    } else {  
        fmt.Println("You are not old enough to drive.")  
    }  
  
    // Example with nested if  
    number := -5  
    if number > 0 {  
        fmt.Println("Positive number")  
    } else {  
        if number == 0 {  
            fmt.Println("Number is zero")  
        } else {  
            fmt.Println("Negative number")  
        }  
    }  
}
```



```
C:\Users\HP\testing>go run main.go
You are allowed to drive.
Negative number
```

- switch supports multiple types.

Example

```
package main
```

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

```
func main() {
    // Example 1: Basic switch
    day := "Tuesday"
    switch day {
    case "Monday":
        fmt.Println("Start of the work week")
    case "Tuesday":
        fmt.Println("Second day of the week")
    case "Saturday", "Sunday": // multiple values in one case
        fmt.Println("Weekend!")
    default:
        fmt.Println("Midweek day")
    }

    // Example 2: Switch without expression (acts like if-else)
    age := 20
    switch {
    case age < 13:
        fmt.Println("Child")
    case age >= 13 && age < 20:
        fmt.Println("Teenager")
    default:
        fmt.Println("Adult")
    }

    // Example 3: Using fallthrough
    num := 1
    switch num {
    case 1:
```

```
        fmt.Println("Number is 1")
        fallthrough
    case 2:
        fmt.Println("Number is 1 or 2")
    default:
        fmt.Println("Other number")
    }

// Example 4: Switch on current day using time package
today := time.Now().Weekday()
switch today {
case time.Saturday, time.Sunday:
    fmt.Println("Today is weekend:", today)
default:
    fmt.Println("Today is a weekday:", today)
}
}
```

```
C:\Users\HP\testing>go run main.go
Second day of the week
Adult
Number is 1
Number is 1 or 2
Today is a weekday: Thursday
```

- for loop handles iteration.
- Range loop for slices, maps.
- break exits loops early.
- continue skips iteration.
- No while keyword, use for.

Example

```
package main
import "fmt"

func main() {
    // Example 1: Standard for loop (like C-style)
    for i := 1; i <= 5; i++ {
        fmt.Println("Count:", i)
    }

    // Example 2: While-style loop
    j := 1
```

```
for j <= 3 {  
    fmt.Println("While-style:", j)  
    j++  
}
```

```
// Example 3: Infinite loop with break  
count := 1  
for {  
    fmt.Println("Infinite loop count:", count)  
    count++  
    if count > 3 {  
        break // exit loop  
    }  
}
```

```
// Example 4: Loop with continue  
for k := 1; k <= 5; k++ {  
    if k%2 == 0 {  
        continue // skip even numbers  
    }  
    fmt.Println("Odd number:", k)  
}
```

```
// Example 5: Range loop for slices  
fruits := []string{"Apple", "Banana", "Cherry"}  
for index, value := range fruits {  
    fmt.Printf("Index: %d, Value: %s\n", index, value)  
}
```

```
// Example 6: Range loop for maps  
capitals := map[string]string{  
    "India": "New Delhi",  
    "USA": "Washington D.C.",  
    "Japan": "Tokyo",  
}  
for country, capital := range capitals {  
    fmt.Printf("Country: %s, Capital: %s\n", country, capital)  
}
```

```
}
```

```
C:\Users\HP\testing>go run main.go
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
While-style: 1
While-style: 2
While-style: 3
Infinite loop count: 1
Infinite loop count: 2
Infinite loop count: 3
Odd number: 1
Odd number: 3
Odd number: 5
Index: 0, Value: Apple
Index: 1, Value: Banana
Index: 2, Value: Cherry
Country: India, Capital: New Delhi
Country: USA, Capital: Washington D.C.
Country: Japan, Capital: Tokyo
```

- goto for jumping (not recommended).
- fallthrough in switch.

Example

```
package main
import "fmt"
func main() {
    // ----- goto example -----
    fmt.Println("Start of program")
    goto SkipSection // jump to label SkipSection
    fmt.Println("This line will be skipped") // skipped due to goto
}
```

SkipSection:

```
    fmt.Println("We jumped here using goto")
    // ----- fallthrough in switch -----
    num := 1
    switch num {
    case 1:
        fmt.Println("Case 1")
        fallthrough // forces execution of the next case
    case 2:
        fmt.Println("Case 2 (executed because of fallthrough)")
    case 3:
        fmt.Println("Case 3")
    default:
        fmt.Println("Default case")
    }
}
```

Result

```
C:\Users\HP\testing>go run main.go
Start of program
We jumped here using goto
Case 1
Case 2 (executed because of fallthrough)
```

- Error handling via if err != nil.

Example

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // Try to open a file
    file, err := os.Open("example.txt")

    if err != nil {
        // Error occurred → handle it
        fmt.Println("Error opening file:", err)
        return // exit the function early
    }
    defer file.Close() // ensure the file is closed when function ends

    fmt.Println("File opened successfully:", file.Name())

    // Another example: converting string to int
    var input string = "123a" // invalid integer
    value, err := fmt.Sscanf(input, "%d", new(int))
    if err != nil {
        fmt.Println("Error reading integer:", err)
    } else {
        fmt.Println("Successfully read integer:", value)
    }
}
```

```
C:\Users\HP\testing>go run main.go
Error opening file: open example.txt: The system cannot find the file specified.
```

5. Functions and Packages

Functions are the heart of Go programs, enabling modularity and reusability. Functions are declared using the `func` keyword, followed by parameters and return types. Go supports multiple return values, which is useful for returning results and errors together. Anonymous functions and closures allow functions to be treated as first-class citizens. Packages in Go help organize code into reusable modules. The `main` package defines the program entry point, while others can be imported for extra functionality. The Go standard library offers numerous packages for networking, file handling, math, and more. Creating and importing packages is essential for building scalable Go applications.

Key Points:

- `func` keyword to define functions.
- Multiple return values supported.
- Named return values.
- Anonymous functions.
- Closures store function state.
- Packages for modular code.
- `main` package for program start.
- `import` to include packages.
- Standard library is extensive.
- Custom packages possible.

Example

```
package main // main package is the entry point of a Go program
```

```
import (  
    "fmt" // import standard library package for formatted I/O  
)
```

```
// ----- Example of a function with multiple return values -----
```

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

```
// ----- Named return values -----
```

```
func rectangleArea(length, width float64) (area float64) {  
    area = length * width // named return variable assigned directly  
    return                // returns 'area' implicitly  
}
```

```
// ----- Function returning another function (Closure) -----
func counter() func() int {
    count := 0
    return func() int { // anonymous function capturing 'count'
        count++
        return count
    }
}

func main() {
    fmt.Println("=== Go Functions Demo ===")
    // Calling divide function with multiple return values
    result, err := divide(10, 2)
    if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("10 / 2 =", result)
    }
    // Calling named return value function
    fmt.Println("Area of rectangle 5x3 =", rectangleArea(5, 3))
    // Anonymous function assigned to a variable
    greet := func(name string) {
        fmt.Println("Hello,", name)
    }
    greet("Alice")
    // Using a closure to maintain state
    nextCount := counter()
    fmt.Println("Counter:", nextCount())
    fmt.Println("Counter:", nextCount())
    fmt.Println("Counter:", nextCount())
    // Showing package usage (standard library: fmt, custom possible)
    fmt.Println("We used 'fmt' from Go's extensive standard library.")
}
```

```
C:\Users\HP\testing>go run main.go
Error opening file: open example.txt: The system cannot find the file specified.

C:\Users\HP\testing>go run main.go
=== Go Functions Demo ===
10 / 2 = 5
Area of rectangle 5x3 = 15
Hello, Alice
Counter: 1
Counter: 2
Counter: 3
We used 'fmt' from Go's extensive standard library.
```

6. Concurrency in Go

One of Go's most powerful features is its built-in support for concurrency. Concurrency allows multiple tasks to run independently without blocking each other. Go achieves this using goroutines, which are lightweight threads managed by the Go runtime. Channels are used to communicate between goroutines safely. The select statement works like a switch but for channels, enabling multiple communication operations to be handled at once. Concurrency in Go is simpler compared to traditional threading models. This makes it ideal for building scalable, high-performance systems such as web servers and real-time applications. Students should understand synchronization techniques and avoid race conditions.

Key Points:

- Goroutines with go keyword.
- Channels for communication.
- Buffered and unbuffered channels.
- select for multiple channel ops.
- sync package for synchronization.
- Mutex for shared resource protection.
- WaitGroups for goroutine completion.
- Concurrency \neq parallelism.
- Ideal for network servers.
- Lightweight thread model.

Example

```
package main
```

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

// Function to simulate work
func worker(id int, wg *sync.WaitGroup, ch chan string, m *sync.Mutex) {
    defer wg.Done() // mark this goroutine as done when finished
    for i := 1; i <= 3; i++ {
        time.Sleep(time.Millisecond * 500) // simulate work
        // Protect shared output with Mutex
        m.Lock()
        msg := fmt.Sprintf("Worker %d: Step %d", id, i)
        ch <- msg // send to channel
        m.Unlock()
    }
}
```



```
func main() {
    fmt.Println("=== Go Concurrency Demo ===")

    var wg sync.WaitGroup // WaitGroup for goroutine completion
    var m sync.Mutex      // Mutex for shared resource protection

    // Unbuffered channel
    unbufferedCh := make(chan string)

    // Buffered channel (capacity 2)
    bufferedCh := make(chan string, 2)

    // Launch goroutines using `go` keyword
    wg.Add(2)
    go worker(1, &wg, unbufferedCh, &m)
    go worker(2, &wg, bufferedCh, &m)

    // Goroutine to read from both channels
    go func() {
        for {
            select {
            case msg := <-unbufferedCh:
                fmt.Println("Unbuffered:", msg)
            case msg := <-bufferedCh:
                fmt.Println("Buffered:", msg)
            case <-time.After(2 * time.Second): // timeout
                fmt.Println("No activity for 2 seconds, stopping listener.")
                return
            }
        }
    }()

    // Wait for all workers to complete
    wg.Wait()

    // Close channels to signal completion
    close(unbufferedCh)
    close(bufferedCh)

    fmt.Println("All workers done.")
}
```

Result

```
C:\Users\HP\testing>go run main.go
=== Go Concurrency Demo ===
Buffered: Worker 2: Step 1
Unbuffered: Worker 1: Step 1
Buffered: Worker 2: Step 2
Unbuffered: Worker 1: Step 2
Buffered: Worker 2: Step 3
Unbuffered: Worker 1: Step 3
Unbuffered:
Buffered:
Buffered:
Buffered:
All workers done.

C:\Users\HP\testing>go run main.go
=== Go Concurrency Demo ===
Unbuffered: Worker 1: Step 1
Buffered: Worker 2: Step 1
Buffered: Worker 2: Step 2
Unbuffered: Worker 1: Step 2
Buffered: Worker 2: Step 3
All workers done.
```

7. File Handling in Go

Go provides robust file handling through its `os`, `io`, and `bufio` packages. File operations like creating, opening, reading, writing, and deleting are straightforward. Error handling is mandatory for file operations, ensuring reliability. Reading can be done in small chunks or all at once, depending on the use case. Buffered reading improves efficiency for large files. Go also supports JSON, CSV, and XML file parsing through standard packages. Mastering file handling is important for students because almost every real-world application involves storing and retrieving data. This section will guide through safe file I/O with examples.

Key Points:

- Use `os.Open` for reading files.
- `os.Create` for creating files.
- `defer file.Close()` to close files.
- `bufio` for buffered I/O.
- `ioutil.ReadFile` for quick reads.
- `fmt.Fprintln` for writing.
- JSON handling with `encoding/json`.
- CSV handling with `encoding/csv`.
- Check errors with `if err != nil`.
- Delete files with `os.Remove`.

Technical work

File handling in Go is straightforward, with powerful standard library support. Common tasks like reading, writing, creating, and deleting files are built into the `os`, `bufio`, `ioutil`, `fmt`, and `encoding` packages.

1. Opening Files

- **os.Open** is used to open an existing file for reading.
- Returns a `*os.File` and an error value.
- Example:

```
file, err := os.Open("data.txt")
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

2. Creating Files

- **os.Create** creates a new file (or truncates if it already exists).
- Example:

```
file, err := os.Create("output.txt")
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

3. Closing Files

- Always **close the file** after operations using:

```
defer file.Close()
```

- This ensures resources are freed even if an error occurs.
-

4. Buffered I/O

- **bufio** improves performance by reading/writing in chunks.
- Example:

```
scanner := bufio.NewScanner(file)
```

```
for scanner.Scan() {  
    fmt.Println(scanner.Text())  
}
```

5. Quick File Reads

- **ioutil.ReadFile** reads the entire file content into memory.
- Example:

```
data, err := ioutil.ReadFile("data.txt")  
if err != nil {  
    log.Fatal(err)  
}  
fmt.Println(string(data))
```

6. Writing to Files

- **fmt.Fprintln** writes formatted text to a file.
- Example:

```
fmt.Fprintln(file, "Hello, Go File Handling!")
```

7. JSON File Handling

- **encoding/json** is used for reading/writing JSON data.
- Example (Writing):

```
data := map[string]string{"name": "John", "age": "25"}  
json.NewEncoder(file).Encode(data)
```

8. CSV File Handling

- **encoding/csv** helps read/write CSV files.
- Example:

```
writer := csv.NewWriter(file)  
writer.Write([]string{"Name", "Age"})  
writer.Write([]string{"John", "25"})  
writer.Flush()
```

9. Error Checking

- Always check for errors:

```
if err != nil {  
    log.Fatal(err)  
}
```

10. Deleting Files

- **os.Remove** deletes a file.
- Example:

```
err := os.Remove("oldfile.txt")  
if err != nil {  
    log.Fatal(err)  
}
```

✓ Key Takeaways:

- Always close files after use (defer is handy).
- For small files, use `ioutil.ReadFile` or `os.WriteFile`.
- For large files, use `bufio` for better performance.
- Handle errors properly to prevent runtime failures.

8. Error Handling in Go

Unlike many languages that use exceptions, Go uses explicit error handling. Functions often return an error type as their last return value. The programmer must check if the error is `nil` before proceeding. This approach encourages writing clear and predictable code. The `errors` package is used to create custom errors, and the `fmt.Errorf` function allows formatted error messages. Go also provides the `panic` and `recover` mechanisms for unexpected situations, though they should be used sparingly. Good error handling improves program stability, especially in production systems.

Key Points:

- `error` is an interface type.
- Check errors with `if err != nil`.
- Create errors with `errors.New`.
- Format errors with `fmt.Errorf`.
- `panic` stops normal execution.
- `recover` regains control.
- Avoid `panic` for normal errors.

- Use logging for error tracking.
- Graceful error handling improves UX.
- Errors are part of return values.

Technical work

Error Handling in Go

1. **error is an interface type**
 - In Go, errors are values. The error type is defined as:
 - type error interface {
 - Error() string
 - }
2. **Check errors with if err != nil**
 - Always check for errors after a function call that can fail:
 - if err != nil {
 - // handle the error
 - }
3. **Create errors with errors.New**
 - You can make a simple error:
 - err := errors.New("something went wrong")
4. **Format errors with fmt.Errorf**
 - Allows formatted error messages:
 - err := fmt.Errorf("file %s not found", filename)
5. **panic stops normal execution**
 - panic is used for unrecoverable errors (e.g., array index out of range).
 - panic("fatal error")
6. **recover regains control**
 - Used inside a defer function to stop a panic from crashing the program:
 - defer func() {
 - if r := recover(); r != nil {
 - fmt.Println("Recovered:", r)
 - }
 - }()
7. **Avoid panic for normal errors**
 - Reserve panic for unexpected, non-recoverable states (like corrupted memory), not user mistakes.
8. **Use logging for error tracking**
 - Example with log package:
 - log.Println(err)
9. **Graceful error handling improves UX**
 - Inform the user without crashing the program; allow retries or alternatives.
10. **Errors are part of return values**
 - In Go, multiple return values are common:
 - result, err := doSomething()