Go Notes Book

A concise guide to understanding and using Go (Golang) for efficient, concurrent programming.

Table of Contents

1. Introduction to Go
2. Core Concepts
3. Structs and Interfaces
4. Concurrency
5. Error Handling
6. Best Practices
7. Example: Simple Task API

Introduction to Go

Go, or Golang, is an open-source programming language developed by Google. Known for its simplicity, performance, and built-in concurrency, it's designed for scalable, modern applications.

- **Key Features**:
  - Statically typed with a simple syntax
  - Built-in concurrency with goroutines and channels
  - Fast compilation and execution
  - Standard library for networking, HTTP, and more
- **Use Case**: Cloud services, microservices, CLI tools, and web servers.

Core Concepts

Variables and Data Types

Go is statically typed; types are declared explicitly or inferred.

```
package main

import "fmt"

func main() {
    name := "Alice" // String (type inferred)
    age := 25       // Integer
    price := 19.99  // Float64
    isActive := true // Boolean
    fmt.Println(name, age, price, isActive)
}
```

Control Flow

Use `if`, `for`, and `switch` for decision-making and looping. No `while` loop; use `for` instead.

```
// If statement
if age >= 18 {
    fmt.Println("Adult")
} else {
    fmt.Println("Minor")
}

// For loop
for i := 0; i < 5; i++ {
    fmt.Println(i) // Outputs 0 to 4
}
```

Functions

Define functions with `func`. Multiple return values are common.

```
func greet(name string) string {
    return "Hello, " + name + "!"
}
```

Structs and Interfaces

Structs

Structs define custom data types.

```
type Dog struct {
    Name string
}

func (d Dog) Bark() string {
    return d.Name + " says Woof!"
}

func main() {
    dog := Dog{Name: "Buddy"}
    fmt.Println(dog.Bark()) // Outputs: Buddy says Woof!
}
```

Interfaces

Interfaces define behavior via method sets.

```
type Animal interface {
    MakeSound() string
}

func makeAnimalSound(a Animal) string {
    return a.MakeSound()
}
```

Concurrency

Go's concurrency model uses goroutines (lightweight threads) and channels.

Goroutines

Run functions concurrently with `go`.

```
func printNumbers() {
    for i := 1; i <= 5; i++ {
        fmt.Println(i)
    }
}

func main() {
    go printNumbers() // Runs concurrently
    fmt.Println("Main function")
    time.Sleep(time.Second) // Wait for goroutine
}
```

Channels

Synchronize and communicate between goroutines.

```
func main() {
    ch := make(chan string)
    go func() {
        ch <- "Hello from goroutine!"
    }()
    msg := <-ch
    fmt.Println(msg) // Outputs: Hello from goroutine!
}
```

Error Handling

Go uses explicit error returns instead of exceptions.

```
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return a / b, nil
}

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

Best Practices

1. **Keep Code Simple**: Follow Go's philosophy of simplicity and clarity.
2. **Use `go fmt`**: Standardize code formatting.
3. **Handle Errors Explicitly**: Always check error returns.
4. **Leverage Packages**: Organize code into modular packages.
5. **Use Interfaces Sparingly**: Prefer concrete types unless polymorphism is needed.
6. **Test Thoroughly**: Use the `testing` package (`go test`).

Example: Simple Task API

Below is a simple REST API for managing tasks using Go's standard library.

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "strconv"
    "sync"
)

type Task struct {
    ID        int    `json:"id"`
    Title     string `json:"title"`
    Completed bool   `json:"completed"`
}

type TaskStore struct {
    sync.Mutex
    tasks  map[int]Task
    nextID int
}

func NewTaskStore() *TaskStore {
    return &TaskStore{
        tasks:  make(map[int]Task),
        nextID: 1,
    }
}

func (ts *TaskStore) AddTask(title string) Task {
    ts.Lock()
    defer ts.Unlock()
    task := Task{ID: ts.nextID, Title: title, Completed: false}
    ts.tasks[ts.nextID] = task
    ts.nextID++
    return task
}

func (ts *TaskStore) GetTasks() []Task {
```

```
    ts.Lock()
    defer ts.Unlock()
    tasks := make([]Task, 0, len(ts.tasks))
    for _, task := range ts.tasks {
        tasks = append(tasks, task)
    }
    return tasks
}

func main() {
    store := NewTaskStore()

    http.HandleFunc("/tasks", func(w http.ResponseWriter, r *http.Request)
{
        switch r.Method {
        case "GET":
            tasks := store.GetTasks()
            json.NewEncoder(w).Encode(tasks)
        case "POST":
            var task struct{ Title string }
            json.NewDecoder(r.Body).Decode(&task)
            newTask := store.AddTask(task.Title)
            w.WriteHeader(http.StatusCreated)
            json.NewEncoder(w).Encode(newTask)
        default:
            http.Error(w, "Method not allowed",
http.StatusMethodNotAllowed)
        }
    })

    fmt.Println("Server running on :8080")
    http.ListenAndServe(":8080", nil)
}
```

Steps to Use

1. Save the code as `task_api.go`.
2. Run it: `go run task_api.go`.
3. Test the API:
    - GET `http://localhost:8080/tasks` to list tasks.
    - POST to `http://localhost:8080/tasks` with JSON `{"title": "Buy groceries"}` to add a task.
    - Use tools like `curl` or Postman.

Example Commands

```
# Add a task
curl -X POST -H "Content-Type: application/json" -d '{"title":"Buy
groceries"}' http://localhost:8080/tasks

# View tasks
curl http://localhost:8080/tasks
```

Additional Resources

- **Official Docs**: go.dev
- **Tutorials**: Go Tour, Go by Example, Practical Go (Dave Cheney)
- **Community**: Go Forum, Reddit r/golang