**Assignment 1, Go**

Put all deliverables into github repository in your profile. Share link to google form 24 hours before defense. Defend by explaining deliverables and answering questions.

Deliverables: code

Google form: <https://docs.google.com/forms/d/e/1FAIpQLSe0GyNdOYlvM1tX_I_CtlPod5jBf-ACLGdHYZq1gVZbUeBzIg/viewform?usp=sf_link>

### **Intro to Go: Basics of Syntax**

#### **Exercise 1: Hello, World!**

1. **Objective**: Write your first Go program.
2. **Steps**:
   * Set up your Go environment by installing Go from the [official website](https://golang.org/dl/).
   * Create a new file named main.go.
   * Write a simple program that prints "Hello, World!" to the console.
   * Run the program using the command go run main.go.
3. **Questions**:
   * What command did you use to compile and run your Go program?

go run main.go

* + What are the basic components of a Go program (e.g., package, import, function)?

main package, “fmt”, func main()

* + How does Go handle package imports?

We use **import** statement to bring built-in packages, and can install external packages by **go get**

#### **Exercise 2: Variables and Data Types**

1. **Objective**: Understand how to declare and use variables in Go.
2. **Steps**:
   * Declare variables of different data types (e.g., int, float64, string, bool).
   * Use both the var keyword and short declaration syntax (e.g., :=) to initialize variables.
   * Print the values and types of the variables using fmt.Printf.
3. **Questions**:
   * What is the difference between using var and := to declare variables?

**First one - Used to declare a variable with an explicit type or without initializing it**

**Second one - Used to declare and initialize a variable in one step, letting Go infer the type from the assigned value.**

* + How do you print the type of a variable in Go?

**fmt.Printf("Type of x: %T\n", x)**

* + Can you change the type of a variable after it has been declared? Why or why not?

**No, Go is a statically typed language,**

#### **Exercise 3: Control Structures**

1. **Objective**: Practice using basic control structures like if, for, and switch.
2. **Steps**:
   * Write a program that takes an integer input and prints whether the number is positive, negative, or zero using an if statement.
   * Implement a for loop that calculates the sum of the first 10 natural numbers.
   * Write a switch statement that prints the day of the week based on an integer input (1 for Monday, 2 for Tuesday, etc.).
3. **Questions**:
   * How does the if statement in Go differ from other languages like Python or Java?

**Unlike in Python or Java, Go does not require parentheses around the condition in an if statement. However, the braces {} are mandatory.**

* + What are the different ways to write a for loop in Go?

**For, while, infinite-loop**

* + How does a switch statement in Go differ from switch in languages like C or Java?

No need for break: In Go, the switch statement automatically breaks after the first matching case unless you explicitly use the fallthrough keyword.

Multiple cases in a single line: You can group cases together by separating them with commas.

Switch with no expression: You can write a switch without an expression to evaluate conditions within the cases.

#### **Exercise 4: Functions and Multiple Return Values**

1. **Objective**: Learn how to define and call functions, and work with multiple return values.
2. **Steps**:
   * Write a function add that takes two integers as arguments and returns their sum.
   * Create a function swap that returns the two input strings in reverse order.
   * Write a function that returns both the quotient and remainder of two integers.
3. **Questions**:
   * How do you define a function with multiple return values in Go?

**func divide(a, b int) (int, int) {**

* + **quotient := a / b**
  + **remainder := a % b**
  + **return quotient, remainder**
  + **}**
  + What is the significance of named return values in Go?

**func divide(a, b int) (quotient, remainder int) {**

* + **quotient = a / b**
  + **remainder = a % b**
  + **return // implicit return of quotient and remainder**
  + **}**
  + How can you ignore certain return values if you don't need them?
  + **result, \_ := divide(10, 3) // Ignore the remainder**
  + **fmt.Println("Quotient:", result)**

### **OOP in Golang**

#### **Exercise 1: Structs and Methods**

1. **Objective**: Understand the basics of OOP in Go using structs and methods.
2. **Steps**:
   * Define a struct called Person with fields Name and Age.
   * Write a method Greet for the Person struct that prints a greeting message.
   * Create an instance of Person, set its fields, and call the Greet method.
3. **Questions**:
   * How do you define a struct in Go?
   * How do methods differ from regular functions in Go?

**A method in Go is a function with a special receiver argument. The receiver can be any type, usually a struct, and the method operates on that receiver.**

**Regular functions don’t have a receiver and are not tied to any specific type.**

* + Can a method in Go be associated with types other than structs?

**Yes**

#### **Exercise 2: Embedding and Composition**

1. **Objective**: Learn how to achieve composition in Go through embedding.
2. **Steps**:
   * Define two structs: Employee with fields Name and ID, and Manager that embeds Employee and adds a field Department.
   * Create a method for Employee called Work that prints the employee's name and ID.
   * Create an instance of Manager, set its fields, and call the Work method.
3. **Questions**:
   * What is embedding in Go, and how does it relate to composition?

**Embedding in Go is a way to achieve composition by including a type within another struct. Unlike traditional inheritance, embedding allows you to compose behavior by directly incorporating fields and methods of one type into another.**

* + How does Go handle method calls on embedded types?

**When you embed a type, the fields and methods of the embedded type are promoted to the outer struct, meaning you can access them as if they were fields or methods of the outer struct.**

* + Can an embedded type override a method from the outer struct?

**No, Go does not have traditional overriding like in OOP languages. However, if the outer struct has a method with the same name as the embedded type’s method, the outer struct’s method takes precedence.**

#### **Exercise 3: Interfaces and Polymorphism**

1. **Objective**: Explore how Go uses interfaces to achieve polymorphism.
2. **Steps**:
   * Define an interface Shape with a method Area() that returns a float64.
   * Implement the Shape interface for two structs, Circle and Rectangle.
   * Write a function PrintArea that takes a Shape and prints its area.
3. **Questions**:
   * How do you define and implement an interface in Go?

**An interface in Go is a type that specifies a set of method signatures. Any type that implements these methods satisfies the interface implicitly (no need to declare that a type implements an interface).**

* + What is the role of interfaces in achieving polymorphism in Go?

**Interfaces enable polymorphism by allowing functions to accept different types that satisfy the same interface. This way, a function can work with any type that implements the required interface.**

* + How can you check if a type implements a certain interface?

**You can use a type assertion or type switch to check if a type implements a specific interface.**

#### **Exercise 4: Working with JSON using Structs**

1. **Objective**: Learn how to encode and decode JSON in Go using structs.
2. **Steps**:
   * Define a struct Product with fields Name, Price, and Quantity.
   * Write a function that converts a Product instance to JSON using json.Marshal.
   * Write another function that decodes a JSON string into a Product instance using json.Unmarshal.
3. **Questions**:
   * How do you work with JSON in Go?

**In Go, you work with JSON using the encoding/json package. This package provides functions like json.Marshal (to encode Go data structures to JSON) and json.Unmarshal (to decode JSON data into Go data structures).**

* + What role do struct tags play in JSON encoding/decoding?

**Struct tags in Go define how fields of a struct are mapped to JSON keys. They specify things like custom field names, omitting fields, and whether a field should be omitted if it's empty.**

* + How do you handle errors that may occur during JSON encoding/decoding?

**JSON encoding and decoding functions return an error, which you should handle appropriately. Errors can occur due to malformed JSON, type mismatches, or unsupported data structures.**