The Go Programming Language

What is Go?

- **Also known as:** Golang (due to its domain name, golang.org).
- **Type:** Open-source, statically typed, compiled programming language.
- **Creators:** Developed by **Robert Griesemer**, **Rob Pike**, and **Ken Thompson** at Google.
- **First released:** November 10, 2009.

Purpose & Vision

- Designed to combine the ease of development of **interpreted languages** (like Python) with the efficiency of **compiled languages** (like C++).
- Emphasis on:

1. **Concurrency**
2. **Simplicity**
3. **Performance**

Variables in Go

What Are Variables?

- Containers for storing data values in memory.
- Each variable has:
  - **Name (Identifier):** Used to refer to the variable.
  - **Type:** Determines the kind of data the variable can store.
  - **Value:** The actual data stored.

Variables in Go

Variable Binding

- **Binding:** The association between a variable and its data (value).
- **Static Binding:** Go is statically typed, meaning the type of a variable is determined at compile time.

```
1 var x int = 10
```

Variables in Go

Variable Binding

- **Binding:** The association between a variable and its data (value).
- **Static Binding:** Go is statically typed, meaning the type of a variable is determined at compile time.
- **Dynamic typing:** It is a programming language feature that determines variable types during runtime, rather than at compile time

```
1 y := 20 // Type inferred as int
```

Variables in Go

Scope

- **Local Scope:** Declared within a function or block.
- **Global Scope:** Declared outside functions, accessible throughout the package.

```
def my_function():
    local_var = 10
    # Local variable
    print(local_var)

my_function()
# print(local_var)
# Error: local_var is not accessible
```

```
global_var = 20  # Global variable

def my_function():
    print(global_var)
    # Accessible inside the function

my_function()
print(global_var)
# Accessible outside the function
```

Variables in Go

Memory Allocation

- **Static Memory Allocation:** Space for variables like globals is allocated at compile time.

```c
int global_var;  // Allocated statically
```

- **Dynamic Memory Allocation:** Space for heap variables is allocated at runtime using.

```c
int *ptr = (int *)malloc(sizeof(int) * 10);  // Allocates memory dynamically
free(ptr);  // Frees the allocated memory
```

Arithmetic Expressions in Go

What Are Arithmetic Expressions?

- A combination of variables, constants, and operators to perform mathematical computations is an arithmetic expression.

Arithmetic Expressions in Go

Operators in Go:

- **Basic Arithmetic Operators:**
  - Addition (+), Subtraction (–), Multiplication (*), Division (/), Modulus (%)
- **Unary Operators:**
  - Unary + and – for positive and negative numbers.

Arithmetic Expressions in Go

Evaluation Rules

Precedence order:

1. `*`, `/`, `%` (Multiplication/Division/Modulus)
2. `+`, `–` (Addition/Subtraction)

```
result := 10 + 5 * 2
// Evaluates to 20, not 30
```

Arithmetic Expressions in Go

Evaluation Rules

Associativity:

1. Determines evaluation order when operators have the same precedence.
2. Left-to-right for most operators

```
result := 10 - 5 + 3
// Evaluates as (10 - 5) + 3 = 8
```

Arithmetic Expressions in Go

Evaluation Rules

Associativity:

1. Determines evaluation order when operators have the same precedence.
2. Left-to-right for most operators

```
result := 10 - 5 + 3
// Evaluates as (10 - 5) + 3 = 8
```

Selection Statements in Go

What Are Selection Statements?

- **Definition:** Control structures that execute code blocks based on conditions.
- **Purpose:** Introduce decision-making logic into programs.

Selection Statements in Go

Types of Selection Statements in Go

`if` Statement:

- Executes a block of code if a condition is true.

```
1 if condition {
2 //Code to execute if condition is true
3 }
```

Selection Statements in Go

Types of Selection Statements in Go

`if-else` Statement:

- Adds an alternative block for when the condition is false.

```
1 if condition {
2       // True block
3 } else {
4       // False block
5 }
```

Selection Statements in Go

Types of Selection Statements in Go

`else if` Ladder Statement:

- Checks multiple conditions sequentially.

```
1 if condition1 {
2       // Code for condition1
3 } else if condition2 {
4       // Code for condition2
5 } else {
6       // Default code
7 }
```

Selection Statements in Go

Types of Selection Statements in Go

`switch` Statement:

- Simplifies multiple conditional checks.

```
switch expression {
case value1:
    // Code for value1
case value2:
    // Code for value2
default:
    // Default code
}
```

Subprograms in Go

What Are Subprograms?

- **Definition:** Reusable blocks of code designed to perform specific tasks.
- Commonly implemented as **functions** in Go.

Subprograms in Go

Key Characteristics of Subprograms?

- Functions can take **parameters** and return **values**.
- Support for **variadic parameters** (variable-length arguments).
- Can be assigned to variables, passed as arguments, and returned by other functions.

Subprograms in Go

Basic Function

```go
package main

import "fmt"

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

func main() {
    result := add(10, 5)
    fmt.Println("Sum:", result)
}
```

Subprograms in Go

Multiple Return Values

```go
package main

import "fmt"

func divide(a, b int) (int, int) {
    return a / b, a % b
}

func main() {
    quotient, remainder := divide(10, 3)
    fmt.Println("Quotient:", quotient, "Remainder:", remainder)
}
```

Object-Oriented Programming in Go

Does Go Support Object-Oriented Programming (OOP)?

- Go is **not a purely object-oriented language**, but it supports key principles of OOP:
  - **Encapsulation**
  - **Composition** (Preferred over classical inheritance)

Object-Oriented Programming in Go

Key Differences from Traditional OOP

- **No classes or objects**: Instead, Go uses **structs**.
- **No inheritance**: Go relies on **composition** for code reuse.
- **No polymorphism via classes**: Achieved through **interfaces.**

Object-Oriented Programming in Go

Encapsulation

- Achieved in Go using:
- **Structs** for data encapsulation.
- **Exported and unexported identifiers**:
  - Identifiers starting with an uppercase letter are **exported** (public).
  - Identifiers starting with a lowercase letter are **unexported** (private).

Object-Oriented Programming in Go

Encapsulation

```go
package main

import "fmt"

// Struct with encapsulated fields
type Person struct {
    Name string // Public field
    age  int    // Private field
}

// Public method to access private field
func (p *Person) GetAge() int {
    return p.age
}

func (p *Person) SetAge(newAge int) {
    p.age = newAge
}

func main() {
    p := Person{Name: "Alice"}
    p.SetAge(25)
    fmt.Println(p.Name, "is", p.GetAge(), "years old.")
}
```

Object-Oriented Programming in Go

Inheritance

- **Go does not support classical inheritance** but achieves similar behavior through **composition**.
- **Composition**: Embedding one struct into another to reuse functionality.

Object-Oriented Programming in Go

Inheritance

```go
package main

import "fmt"

// Base struct
type Animal struct {
    Name string
}

func (a Animal) Speak() {
    fmt.Println(a.Name, "makes a sound.")
}

// Derived struct using composition
type Dog struct {
    Animal // Embedded struct
}

func (d Dog) Speak() {
    fmt.Println(d.Name, "barks.")
}

func main() {
    d := Dog{Animal{Name: "Buddy"}}
    d.Speak()        // Dog-specific behavior
    d.Animal.Speak() // Access base behavior
}
```

Object-Oriented Programming in Go

Why Composition Over Inheritance?

- Promotes **flexibility** and **loose coupling**.
- Avoids complexity of deep inheritance hierarchies.

Go vs. Python vs. Java - Overview

Aspect	**Go**	**Python**	**Java**
Type System	Statically typed	Dynamically typed	Statically typed

Go vs. Python vs. Java - Overview

Aspect	**Go**	**Python**	**Java**
Type System	Statically typed	Dynamically typed	Statically typed
Concurrency Model	Built-in support (goroutines, channels)	Requires external libraries (e.g., asyncio)	Thread-based, requires boilerplate

Go vs. Python vs. Java - Overview

Aspect	Go	Python	Java
Type System	Statically typed	Dynamically typed	Statically typed
Concurrency Model	Built-in support (goroutines, channels)	Requires external libraries (e.g., asyncio)	Thread-based, requires boilerplate
Performance	High (close to C/C++)	Lower (due to interpretation overhead)	High (optimized via JVM)

Go vs. Python vs. Java - Overview

Aspect	Go	Python	Java
Type System	Statically typed	Dynamically typed	Statically typed
Concurrency Model	Built-in support	Requires external libraries	Thread-based, requires boilerplate
Performance	High	Lower	High
Main Use Cases	System programming, cloud services	Scripting, data science, web development	Enterprise applications, Android development

Thank You!

Done by:

Group-3

Sai Vijay Nagarur

Sahith Reddy