Go for the Absolute Beginners – Hands-On

- Goroutines
- Go Modules
- Package Creation

Our Courses

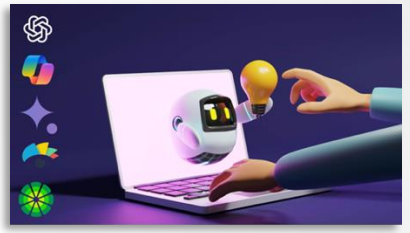
Puppet for the Absolute Beginners – Hands-On

Generative AI Essentials - Practical Use Cases

SaltStack for the Absolute Beginners – Hands-On

Infrastructure Automation with OpenTofu – Hands-On

AI Ecosystem for the Absolute Beginners - Hands-On



Mastering Prompt Engineering for GenAI

Mastering Docker Essentials - Hands-on

Unlocking Python for the Absolute Beginners

Podman for the Absolute Beginners - Hands-On

Practical Kubernetes – Beyond CKA and CKAD

Argo CD for the Absolute Beginners - Hands-On

Foundational Concepts

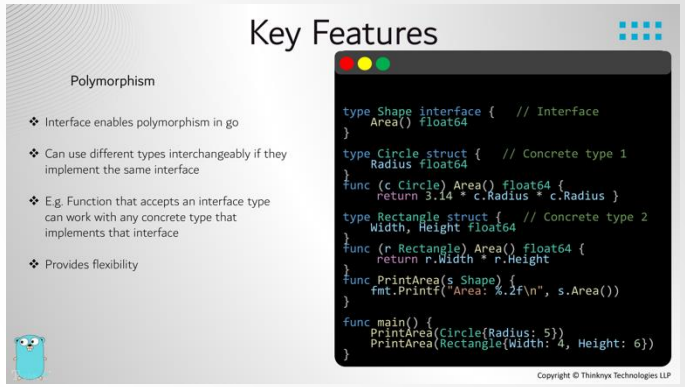
- Go Installation
- Variables & Data types
- Control structures
- Data Structures
- Functions
- Control structures
- Data Structures
- Functions

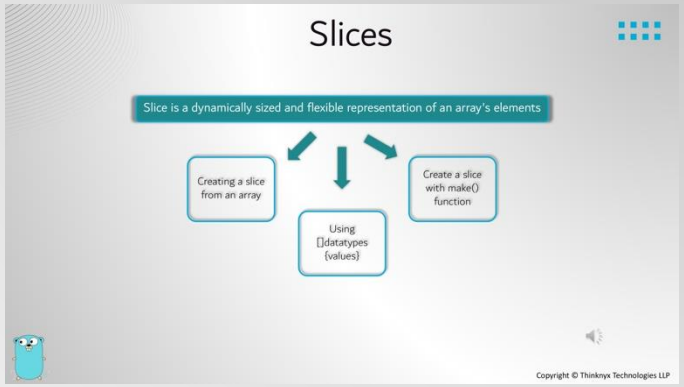
Course Workflow

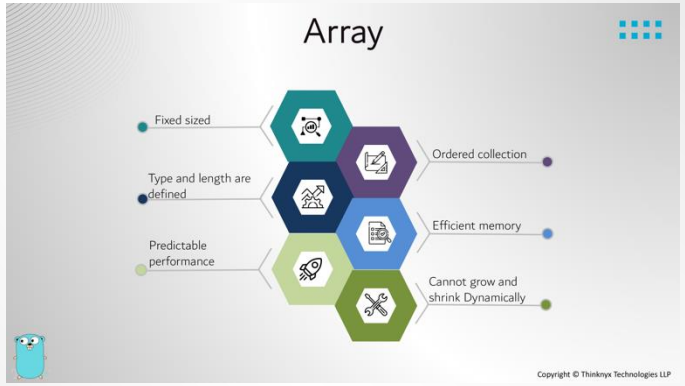
Lectures

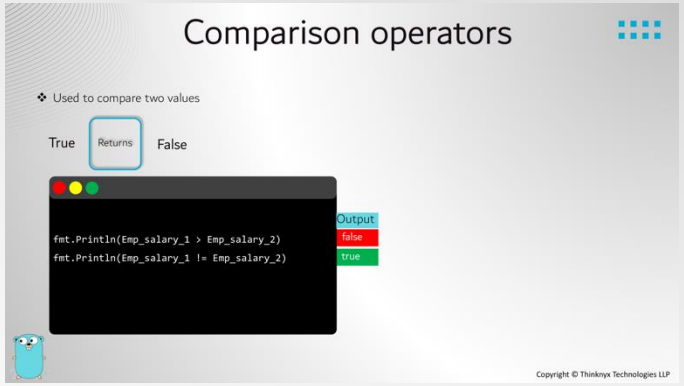
Live Demonstrations

Assignments









Course Objective

- Introduction to Golang
- Variables & Data Types
- Data Structure
- Control Structure
- Microservices using Go
- Go Interfaces
- Errors and Logging
- Core Go Packages
- Packages and Modules
- Concurrency in Go

Course Objective

Capstone Project

Section-1

Section Overview

- Introduction to Golang
- Installation and Set-up Go
- Writing first code

What is Golang?

An open-source, compiled programming language

Supported By	Developed by	Published in
		2009

Robert Griesemer

Rob Pike

Ken Thompson

2007

Primary goals

Speed

Simplicity

Ease of use

Why choose Golang?

Simple and Readable Syntax

- ✓ Simple and easily readable language
- ✓ Easier for developers to read and maintain the code

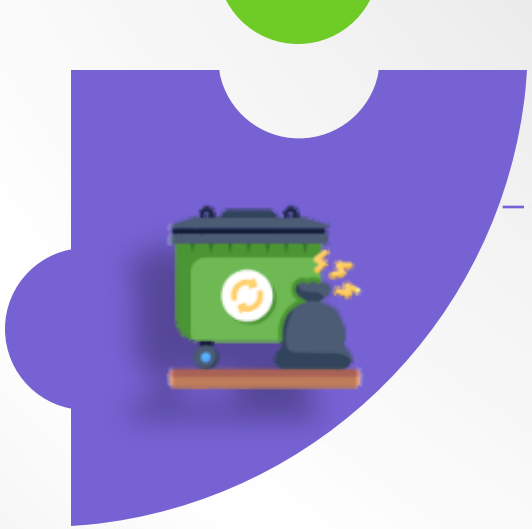
Concurrent Programming

- ✓ Built-in support for concurrency
- ✓ Enables developers to run multiple tasks
- ✓ Builds responsive applications

Cross-Platform

- ✓ Supports cross-compilation
- ✓ Compiles code on one platform, and run it on another

Garbage Collection

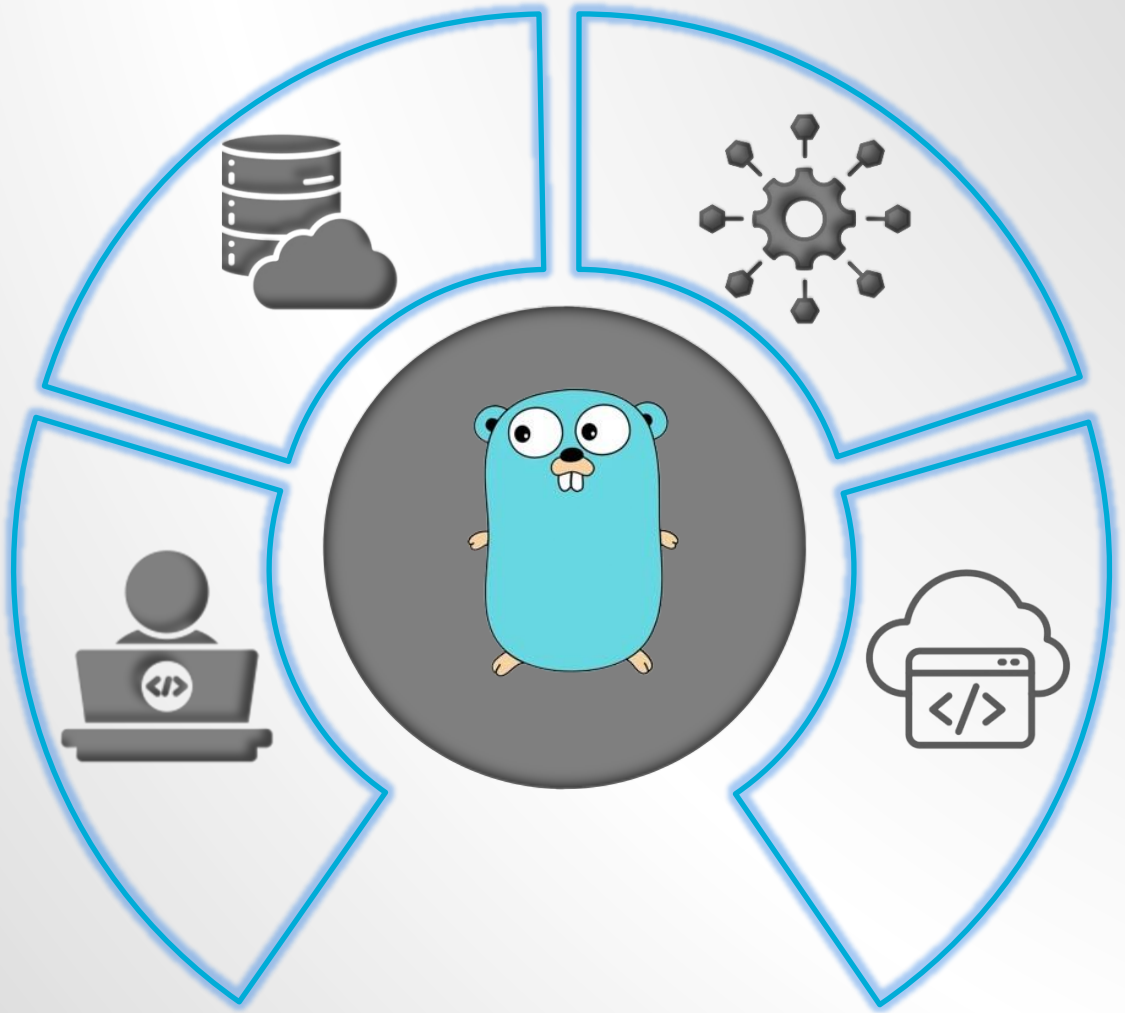


- ✓ Manages memory through built-in garbage collection
- ✓ Helps to free up memory occupied by unused objects
- ✓ Prevents memory leaks

Why choose Golang?

Backend Development

Microservices Architecture



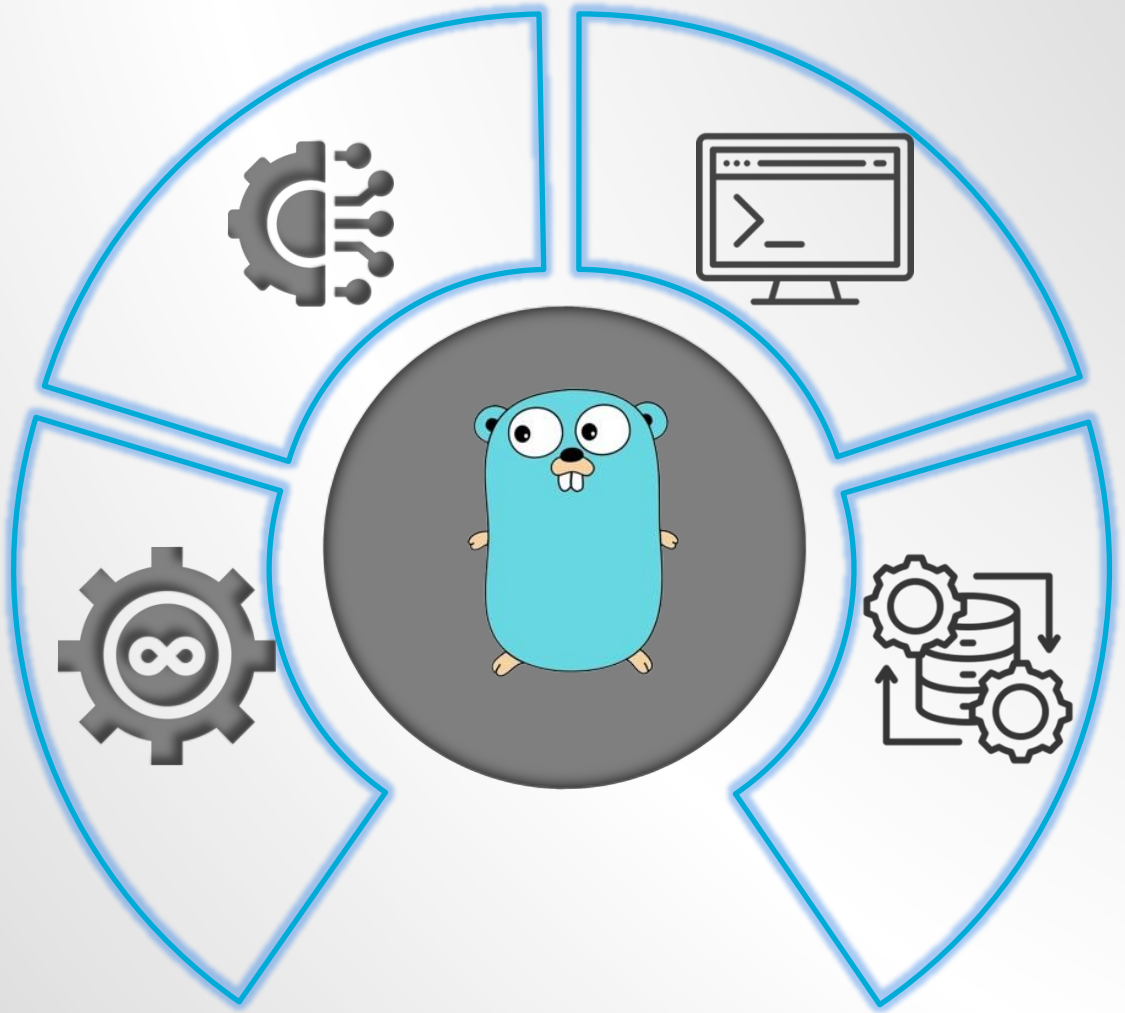
Web development

Cloud-Native Applications

Why choose Golang?

Networking Tools

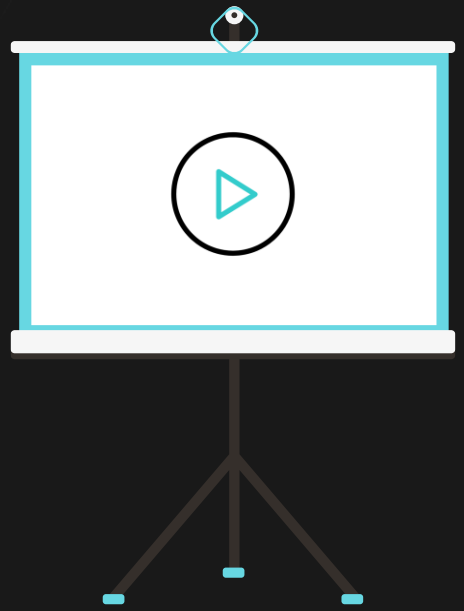
Command-Line Tools



DevOps Tools

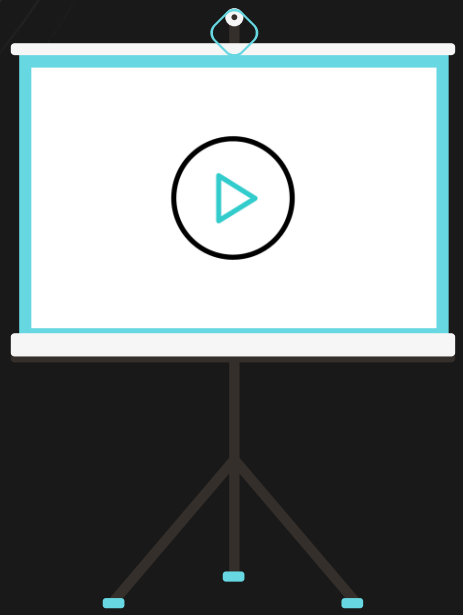
Data Processing

Demonstration



Installation and Set-up Go

Demonstration



Writing First Code

Section-2

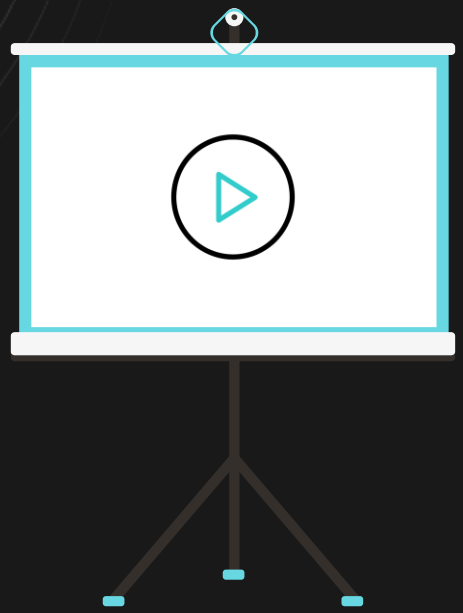
Section Overview

- Type of variables
- Variable declaration

Type of variables

Int	String	Float32	Bool
123 to -123	Stores text enclosed in " "	Stores decimal values	True or False

Demonstration



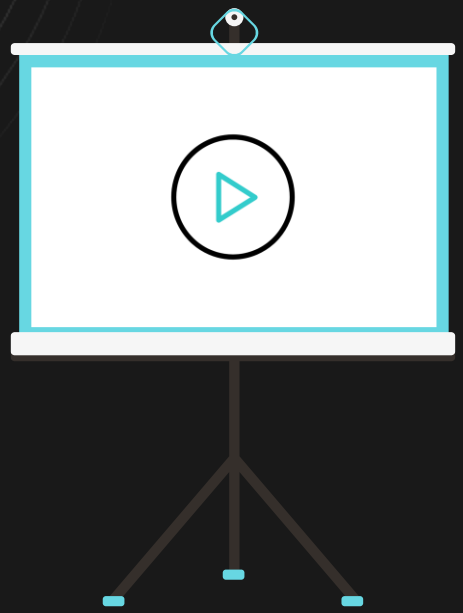
Declaring Variables

Default Values

Variables declared without initialization have default values.
e.g.

- An empty string for a string variable
- 0 for int
- false for bool

Demonstration



Default Values

Var v/s ':='

var age int = 24

- Can be declared inside and outside function
- Declaration and value assignment can be done separately
- Can declare multiple variables at once

age := 24

- Declaration possible only inside a function
- Declaration and value assignment is done at the same time
- Only single variable can be declared at one instance

```
var a, b, c, d int = 1, 2, 3, 4
```

Naming rules for variables

1. a,b,c or _
2. 1,2,3 and 4
3. a-z, A-Z, 0-9, and _
4. age = Age = AGE
5. Length of Variable name has no limit
6. Var name = 10
7. Go = 10

Section Overview

- What are constants
- Types of constants

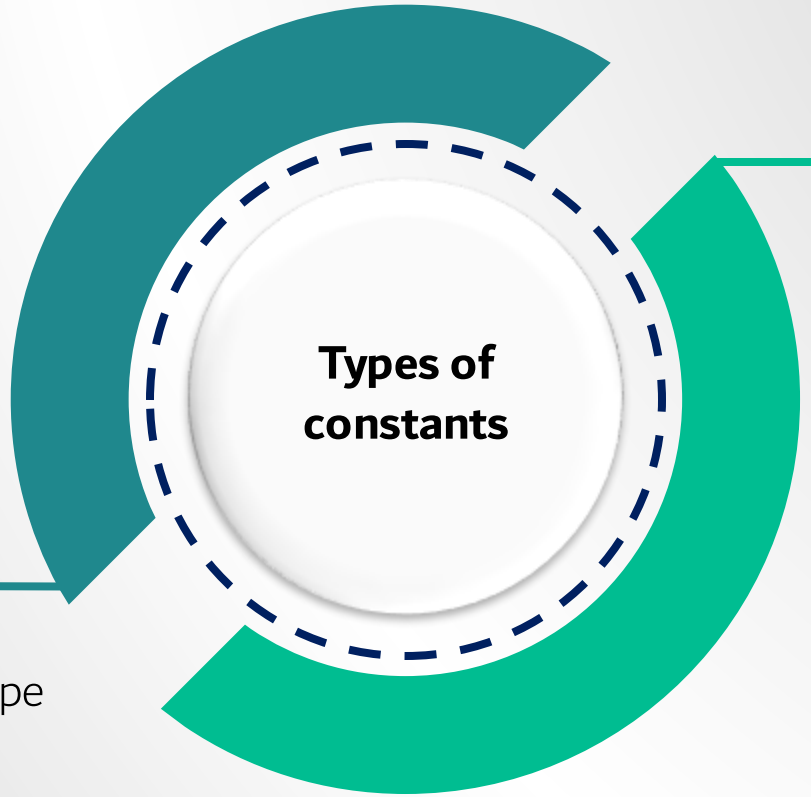
What are constants

- “**const**” keyword
- Declaring variable as constant makes it read-only and unmodifiable
- Constant must be assigned at the time of declaration

```
const EMPEXPERTIES int =
10
```

- ✓ Written in upper case
- ✓ Can be declared either inside or outside of a function

Types of constants



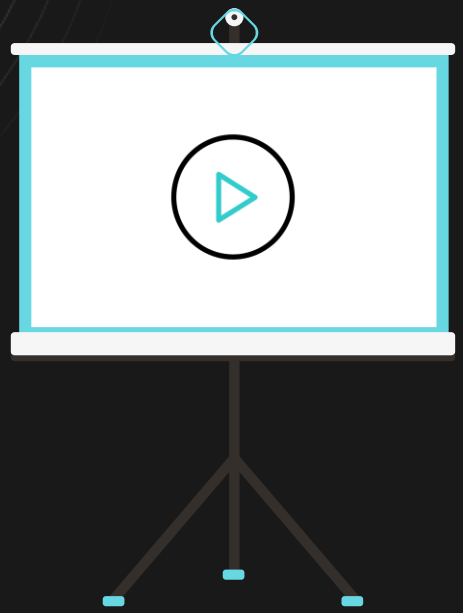
Typed constants

- ✓ Declared with a specific datatype
- ✓ int or string, along with their assigned values

Untyped constants

- ✓ Declared without specifying datatype

Demonstration



Constants

Section Overview

- Output Functions in Go

Output Functions in Go

fmt

- “fmt” used for output
- “Standard library package stands for format
- Formatted I/O operations:
  - Printing
  - Reading

Output Functions in Go

Print()

Displays text as-is, with no added spaces or line break

Println()

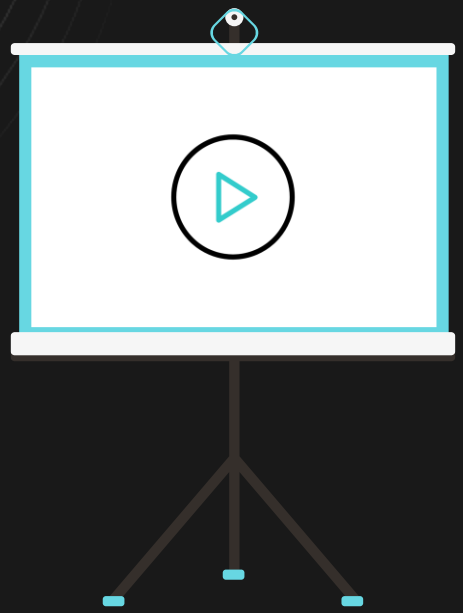
Similar to Print(), but adds a newline at the end

Printf()

Allows formatted output with placeholders

- %d for integers
- %s for strings
- %f for floats

Demonstration



Output Functions in Go

Section Overview

- Datatypes in Go

Data types

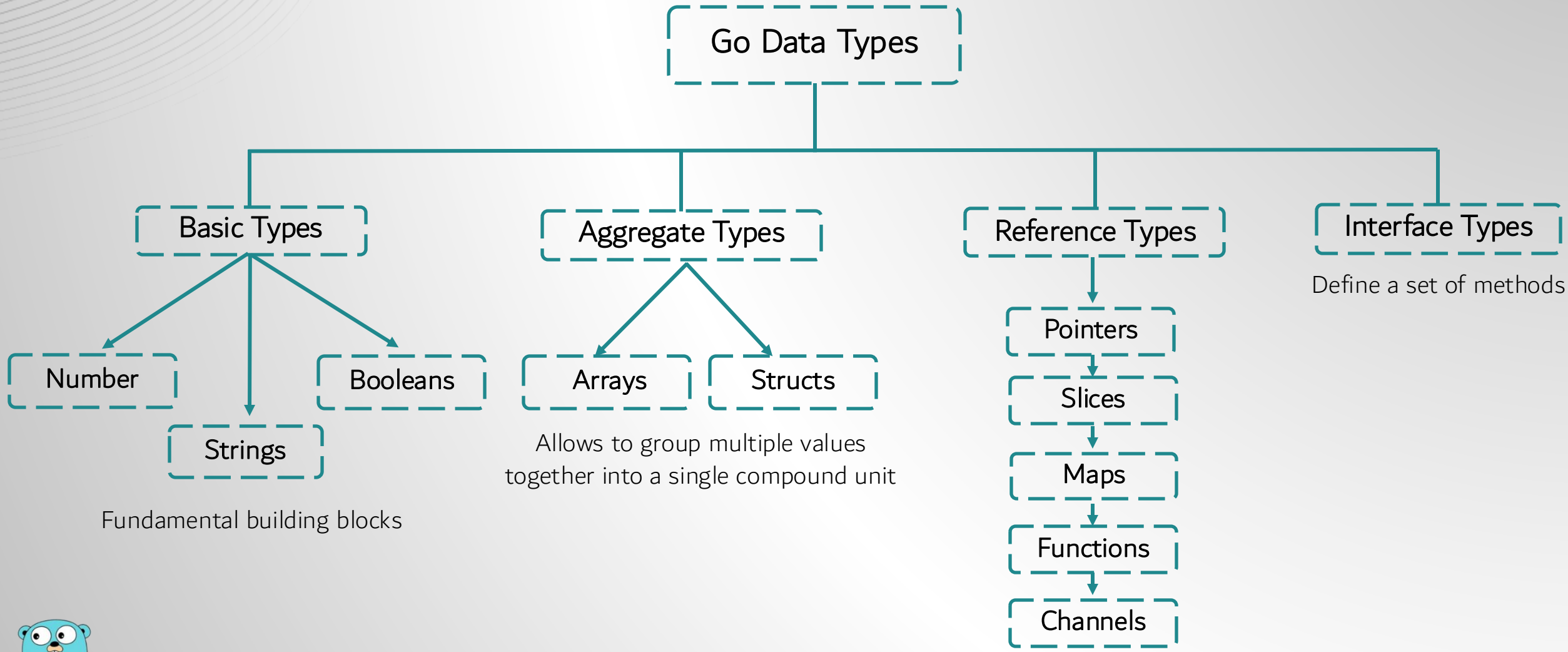
Determine its memory size

Data type is determined at compile time

Define the kind of data a variable can hold

Bounded to the variable

Data types



Overview of Basic Data Types

Strings

Enclosed characters in double quotes (“ ”)

Numbers

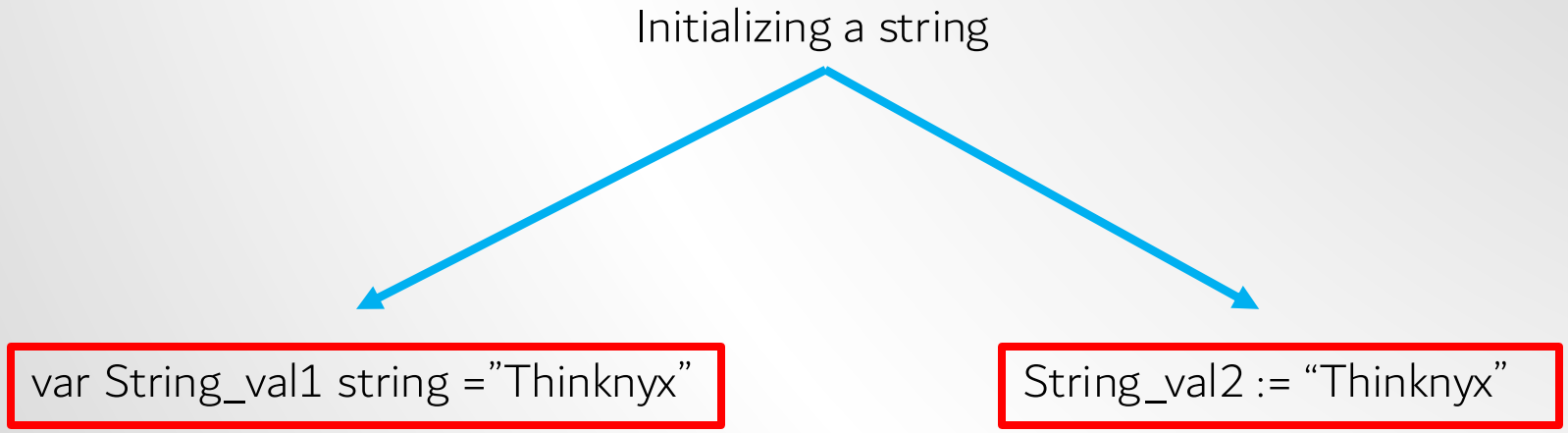
Include integers and floating-point numbers that represent numeric values

Booleans

Used to represent “**true** or **false**” values

Strings in Go

- Used to store sequence of characters
- Enclosed in double quotes

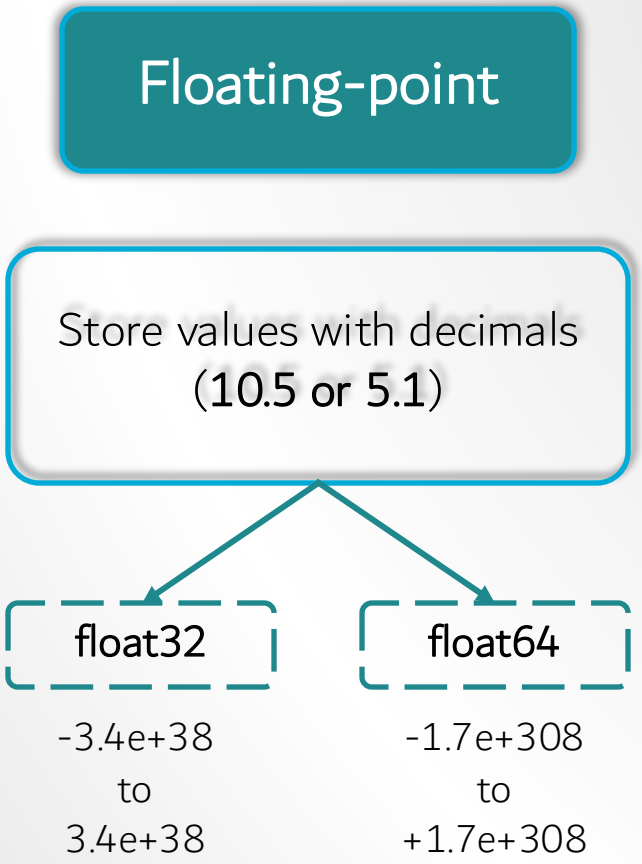


Overview of Numeric Data Types

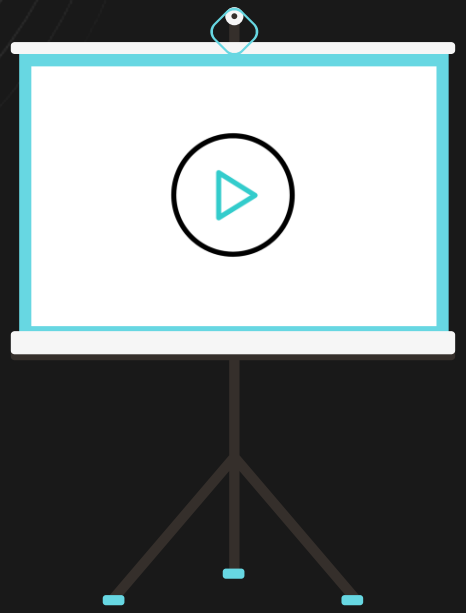
Floating-point

Integers

Overview of Numeric Data Types



Demonstration



Floating-point

Overview of Numeric Data Types

Integers

Store whole numbers
(**+** and **-**)

int: (32 bits on 32-bit systems, and 64 bits on 64-bit systems)

int8: Stores 8 bits (1 byte), ranging from -128 to 127

int16: Stores 16 bits (2 bytes), ranging from -32,768 to 32,767

int32: Stores 32 bits (4 bytes), ranging from -2,147,483,648 to 2,147,483,647

int64: Stores 64 bits (8 bytes), ranging from -9.2e18 to 9.2e18

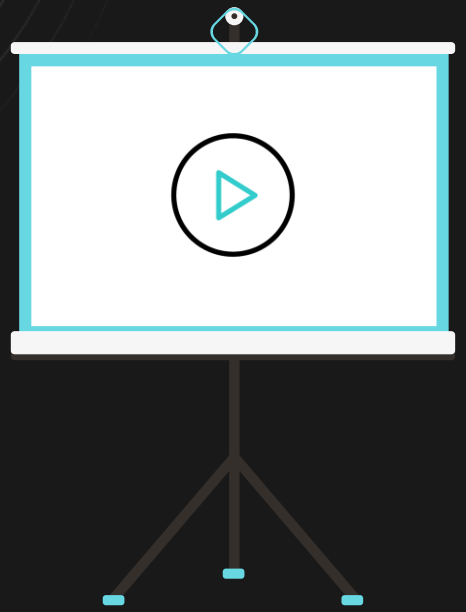
uint8: Stores 8 bits, ranging from 0 to 255

uint16: Stores 16 bits, ranging from 0 to 65,535

uint32: Stores 32 bits, ranging from 0 to 4.2e9

uint64: Stores 64 bits, ranging from 0 to 1.8e19

Demonstration



Integer

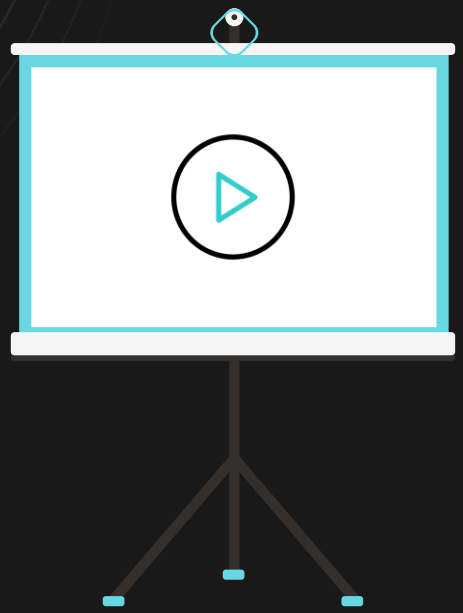
Overview of Basic Data Types

Boolean

Represent truth values:
true or false

Default value of boolean is false

Demonstration

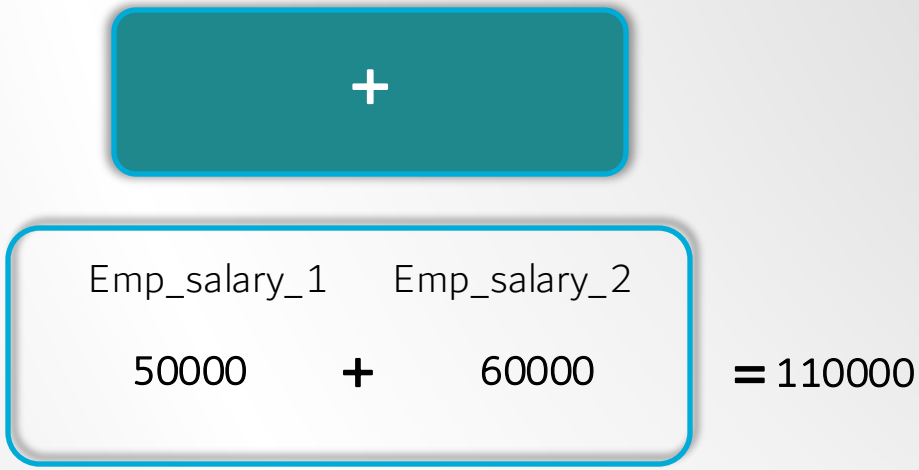


Boolean

Section Overview

- Operators in Go

Operators



Types of Operators

Comparison operators

Assignment operators

Logical operators

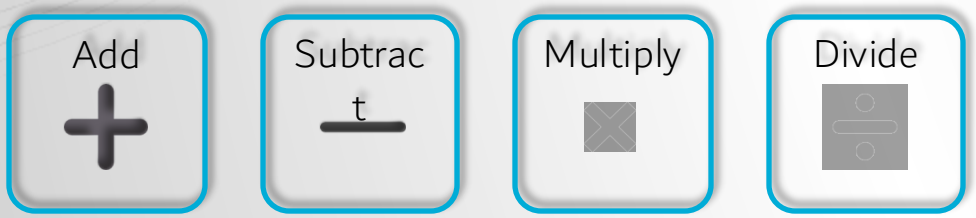
Arithmetic operators

Bitwise operators

Types of Operators

Arithmetic operators

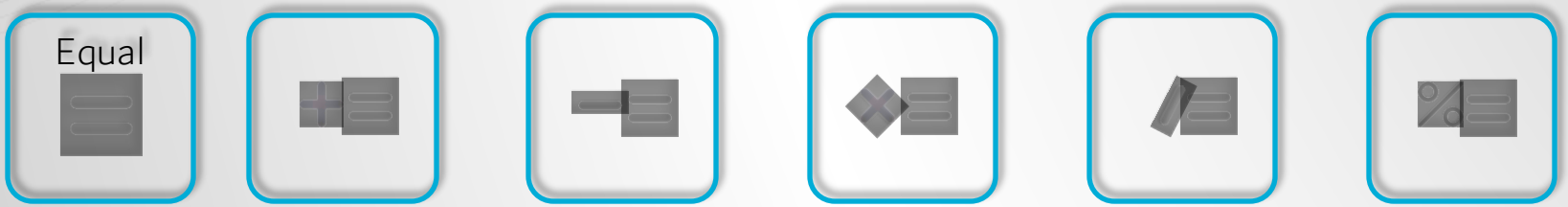
- Used to perform basic mathematical operations




```
EmpSal1 = 100
EmpSal2 = 200
EmpYearlysal1 = EmpSal1 * 12
EmpYearlysal2 = EmpSal2 * 12
fmt.Println(EmpYearlysal1)
fmt.Println(EmpYearlysal2)
```

Assignment operators

- Used to assign values to variables




```
Emp_salary_1 = 50000
Emp_salary_2 = 60000
Emp_salary_1 += Emp_salary_2
fmt.Println(Emp_salary_1)
```

Comparison operators

- Used to compare two values

True Returns False

```
fmt.Println(Emp_salary_1 > Emp_salary_2)
fmt.Println(Emp_salary_1 != Emp_salary_2)
```

Output

false

true

Logical operators

- Used to perform logical operations on variables

and

or

not

```
isSunny = true
isSunday = false
fmt.Println(isSunny && isSunday)
fmt.Println(isSunny || isSunday)
fmt.Println(!isSunny)
```

Output

```
false
true
false
```

Bitwise operators

- Used on binary numbers

bitwise and

bitwise or

bitwise xor

```
a = 5  // 0101
b = 3  // 0011
fmt.Println(a & b) // 0001
fmt.Println(a | b) // 0111
```

Output

1

7

Section Overview

- Input and output in go

Input & Output

- (I/O) relies on standard packages like “fmt”

Output Functions

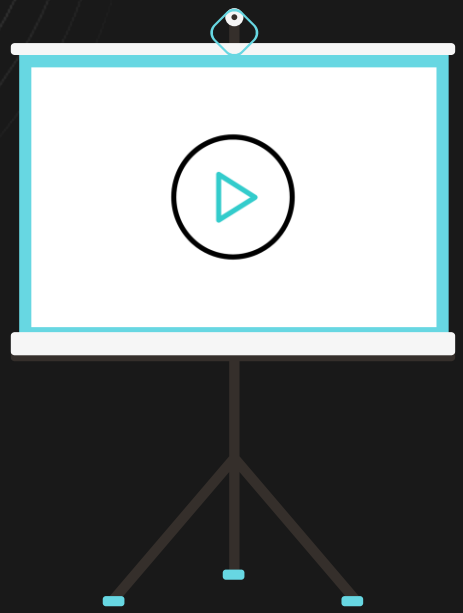
Print	Println	Printf
Print for normal printing of values	Break the line, so that the next value printed is in the next line, and last	Format printing with formatting verbs

Inputs in Go

Scan

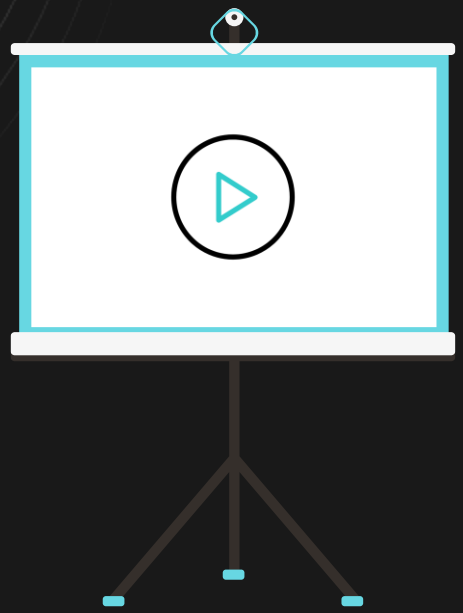
- Reads input values separated by whitespace
- Stops reading when all specified variables are filled
- Does not handle the newline character after input
- Remains in the input buffer

Demonstration



Scan

Demonstration



Use case – User Registration

Section-3

Section Overview

- What are Arrays
- What are slices

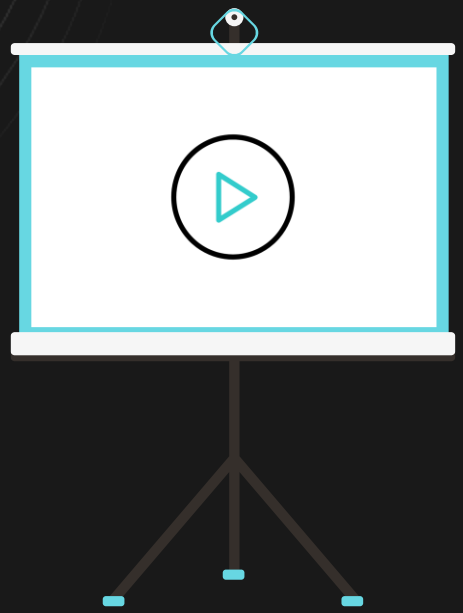
Array

Array

var a [5]int

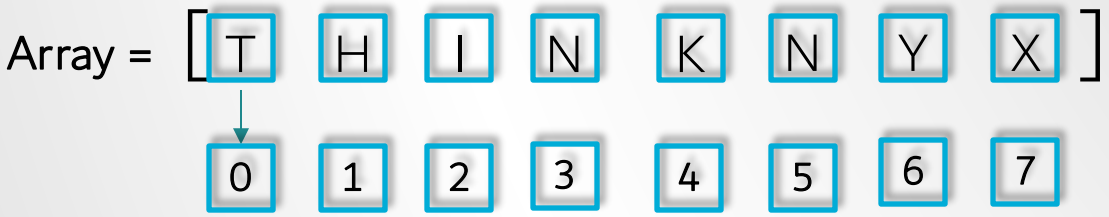
Store exactly five integers

Demonstration



Declaring Arrays

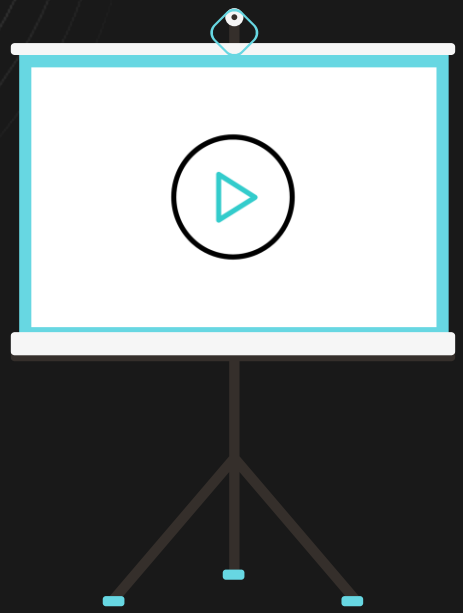
Accessing array elements




```
my_arr.  =“THINKNYX
arr[2]    = ‘I’
```

Len(arr)-1

Demonstration

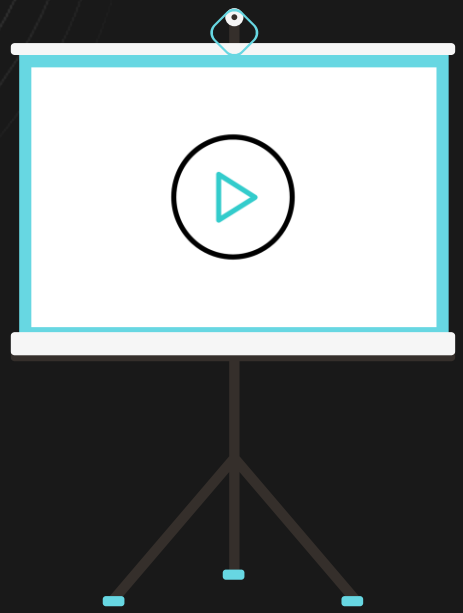


Accessing array elements

Modifying elements in array

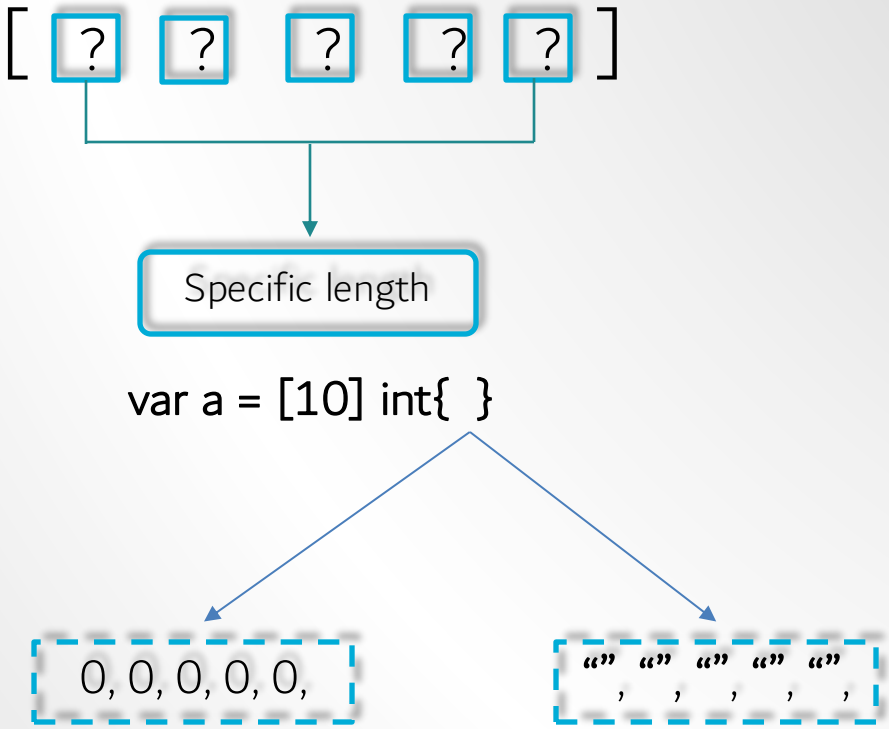
arr[2] = arr[6]

Demonstration

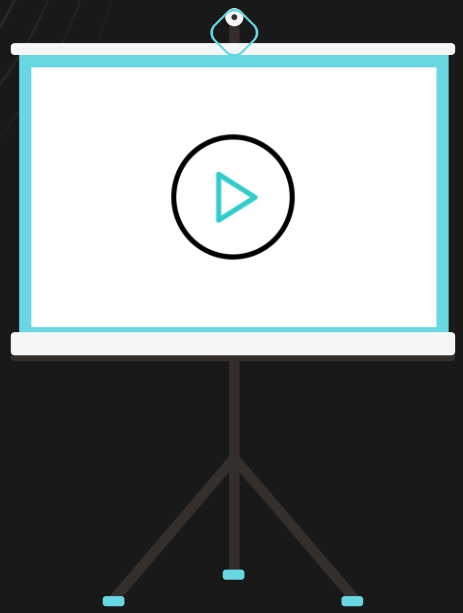


Modifying elements in array

Initialization of array



Demonstration

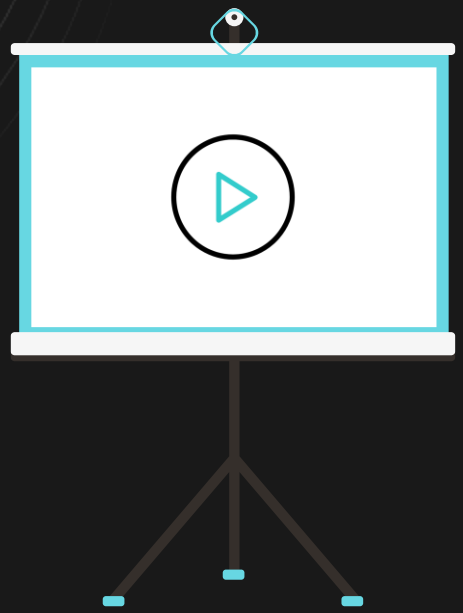


Initialization of array

Partially Initialized array

var a = [5] int{1,2}

Demonstration

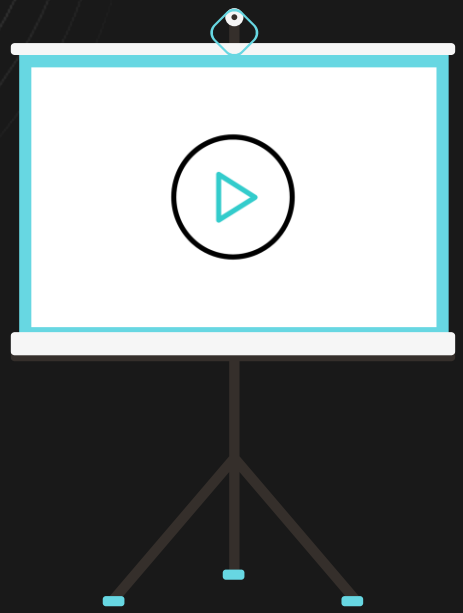


Partially Initialized array

Index based Initialized

var a = [5]int{1:10, 3:40}

Demonstration



Index based Initialized

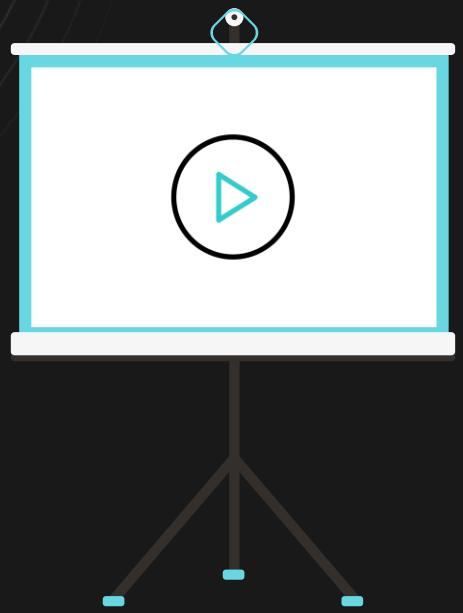
Finding length of an array

- To find the length of an array

  Len-function

- Returns total number of elements in the array

Demonstration



Finding length of an array

Section Overview

- What are slices

Slices

Slice is a dynamically sized and flexible representation of an array's elements

- Creating a slice from an array
- Using []datatypes {values}
- Create a slice with make() function

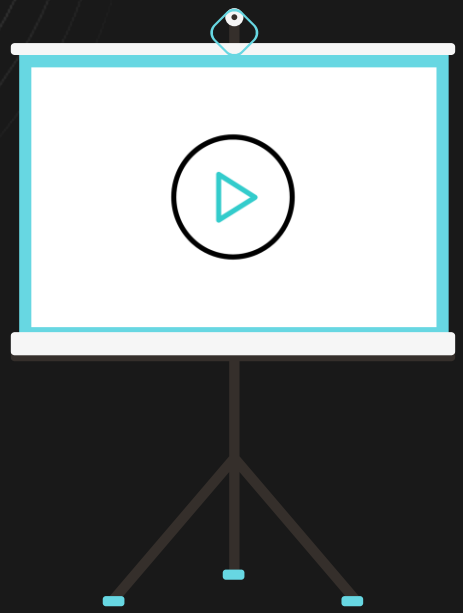
Creating a slice from an array

```
var s[]int := num[low:high]
```

Low bound
Starting index

High bound
End index

Demonstration



Creating a slice from an array

Using []datatypes{values}

Slice Literals

- Creating slices without referring to arrays
- Don't have to specify length

```
slice := []int{1, 2, 3, 4, 5}
```

- Does not store itself
- Reference of section of underlying array
- Modifying elements in slice, updates corresponding elements in array

Length -> fmt.Println(len(slice))

Capacity -> fmt.Println(cap(slice))

Create a slice with make() function

make() function

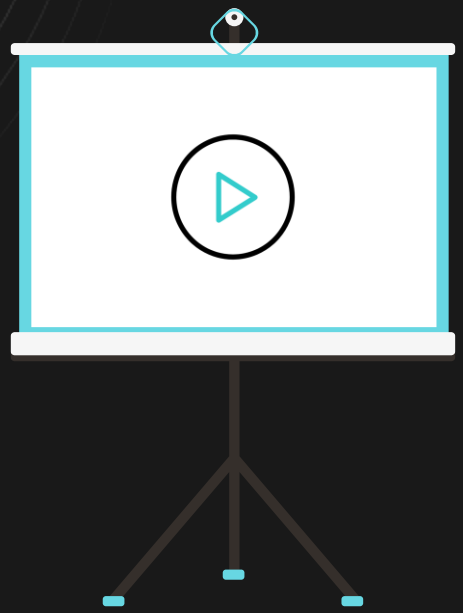
```
sliceName := make([]datatype, length, capacity)
```

int or string

If the capacity is omitted, it defaults to the length

This method provides flexibility and precise control over the slice's size and memory usage

Demonstration



Create a slice with make() function

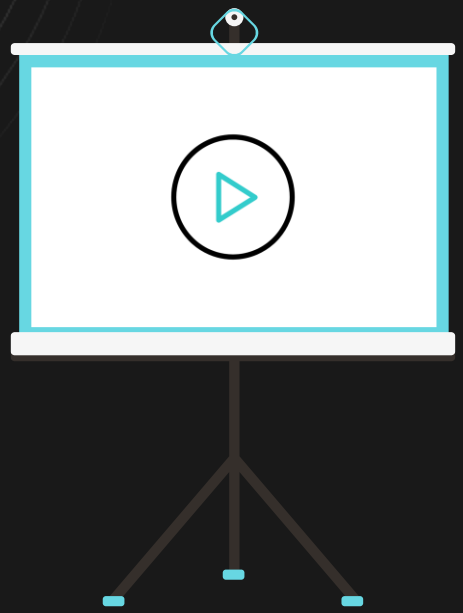
Section Overview

- Working with values in slices

Accessing values in slices

Access elements by referring to its index number

Also modify it by assigning a new value



Accessing values in slices

Section Overview

- What is maps

Maps

Each key is unique

Key-value pairs

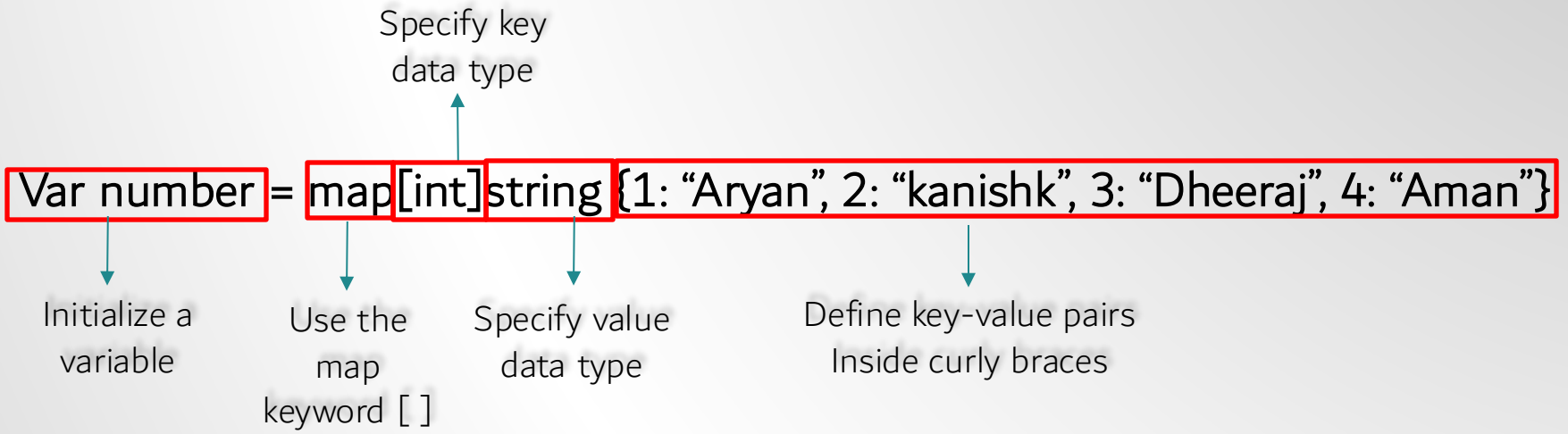
Quickly store and retrieve values

Unordered collection

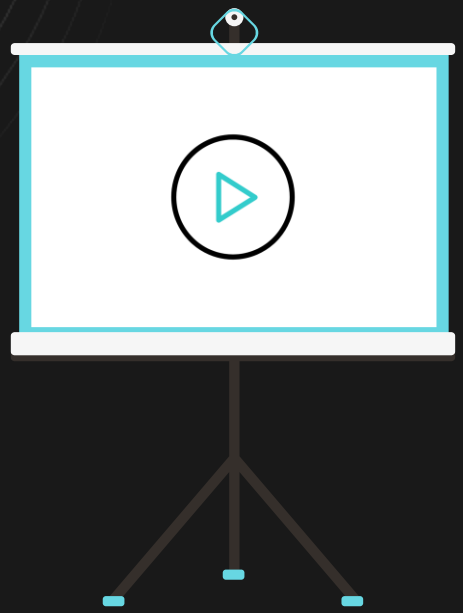
Making them like dictionaries

Maps

Maps



Demonstration



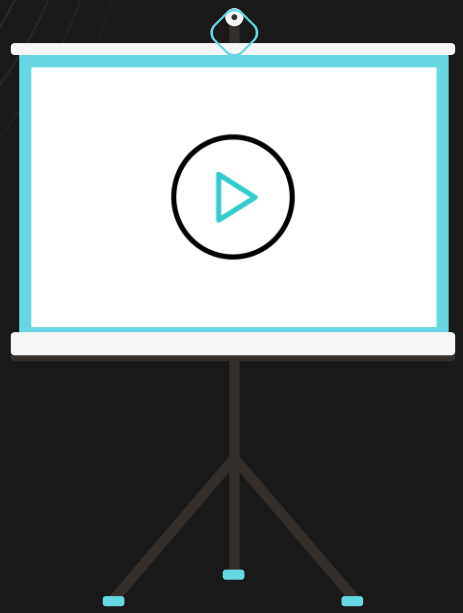
Method-1 (using var keyword)

Maps

Make() function

```
var variable_name := make()map[int] string)
variable_name[0] = “Dheeraj”
variable_name[1] = “Aryan”
```

Demonstration



Method-2 (using the make() function)

Maps

Important note:-

- Slices
- Functions
- Other maps

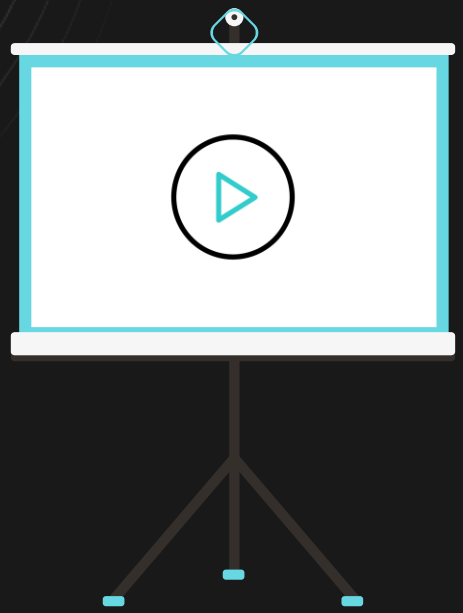
Maps

Important note:-

- Slices
- Functions
- Other maps

Data Type

Demonstration



Accessing, Modifying and Removing

Section Overview

- What is Struct

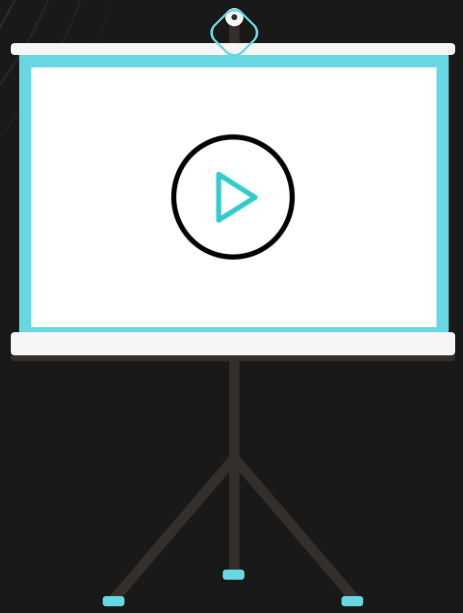
Structs

Structure or struct

- User-defined
- Allows to group members
- Enable the storage
- Useful for creating records

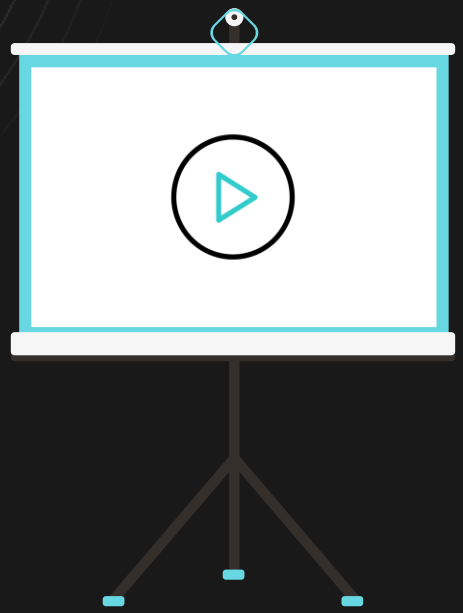
```
type name_of_struct struct {

        body of struct
}
```

Demonstration



Struct

Demonstration



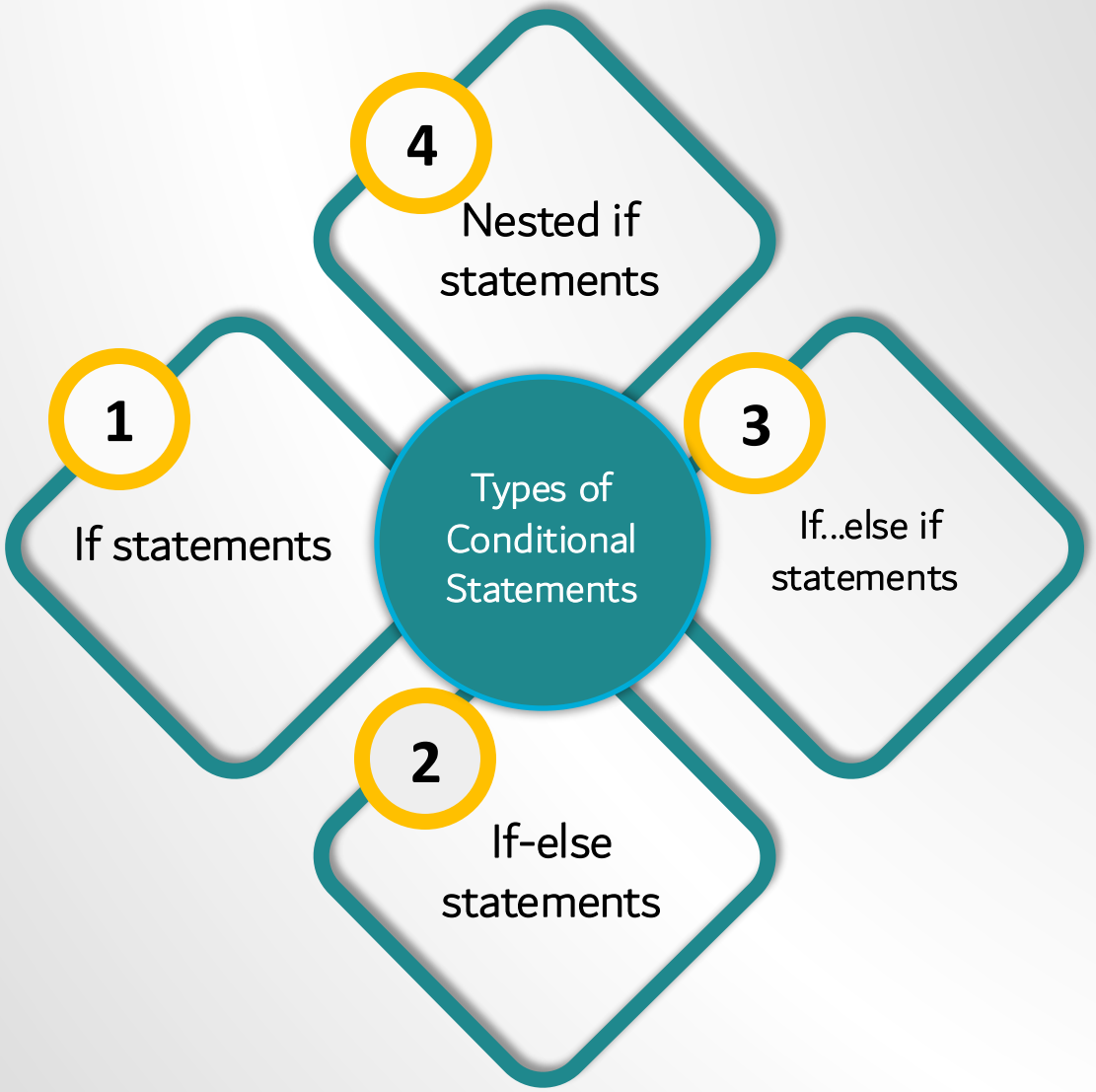
Use case – Creating Product Catalogue

Section-4

Section Overview

- Conditional Statements
- How to make decisions in programming

Type of Conditional Statements

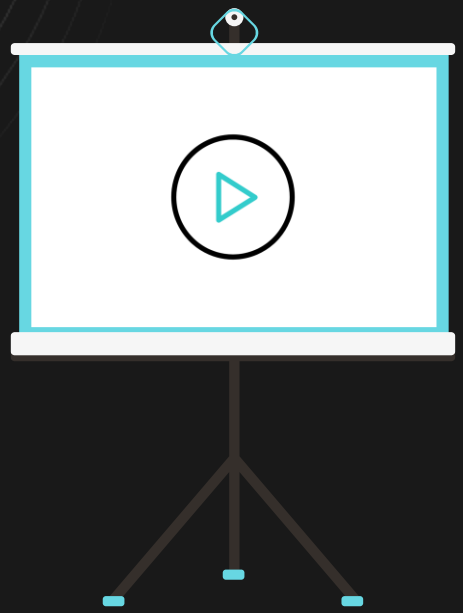


If Statements

- Checks a condition and executes a block of code
- If the condition is 'false', the code block is skipped

```
if condition { // code to execute if the condition is true }
```

Demonstration

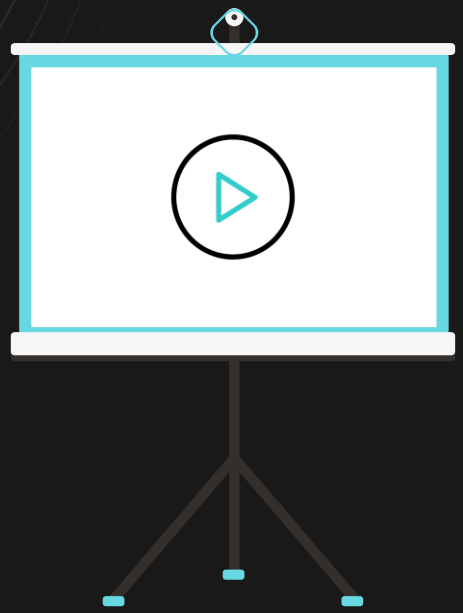


If Statements

If Else Statements

- Used to execute one block of code when a condition is true
- Execute alternate block when the condition is false
- Else block provides a fallback, running when the condition is false

Demonstration

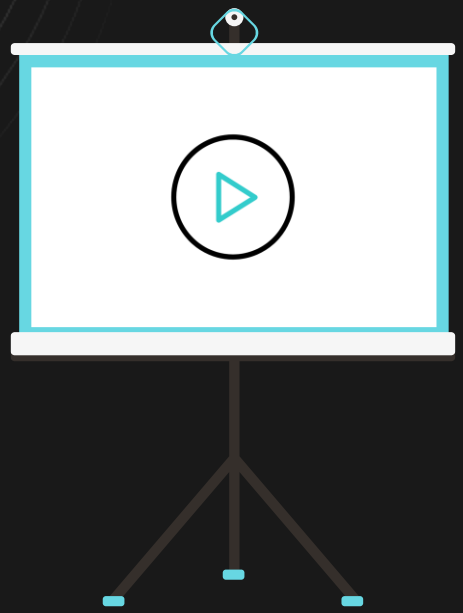


If Else Statements

Logical Operators with If Statements

- Logical operators like '&&' (AND) and '||' (OR) can combine multiple conditions in decision-making

Demonstration

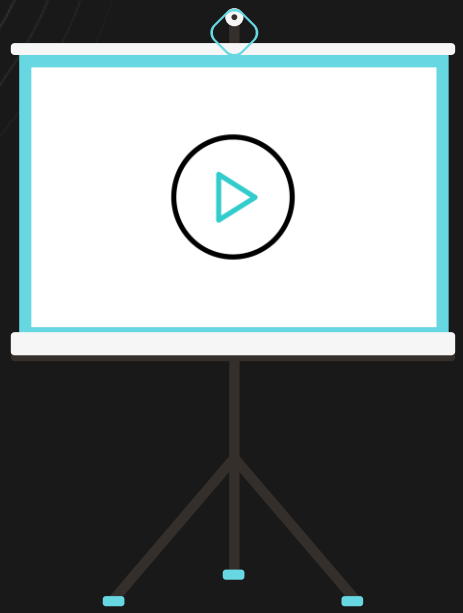


Using Logical Operators with If Statements

If...else if statements

- Helpful when there are multiple conditions to evaluate
- Evaluate multiple conditions sequentially
- If first condition is false, the program checks the next one until a true condition is found or defaults to the else block

Demonstration



Exploring Else If Statements

Section Overview

- Switch Statements

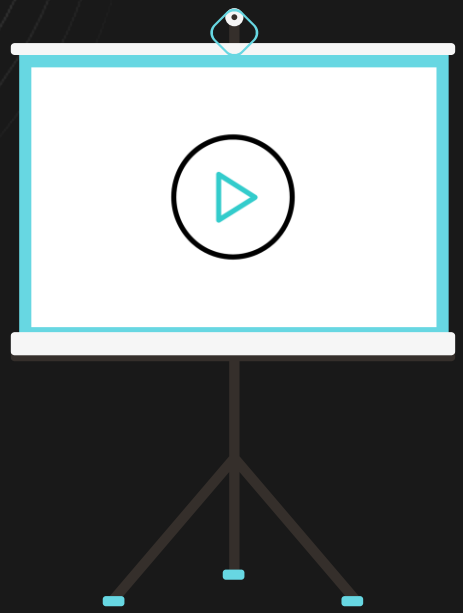
Switch

Switch statement

Powerful control flow mechanism for handling multiple cases based on a value

- Using values
- Using type switch

Demonstration



Using values

Why use Type Switch

Dynamic Type Handling

Useful when dealing with variables of interface type that can hold values of any type

Type-Specific Logic

Allows execution of different logic based on the actual type of the value

Processing data with unknown or mixed types, such as JSON decoding or implementing generic functionality

Section Overview

- What are Loops

Loops

Loops

Useful when you need to repeatedly execute the same code, each time with a different value

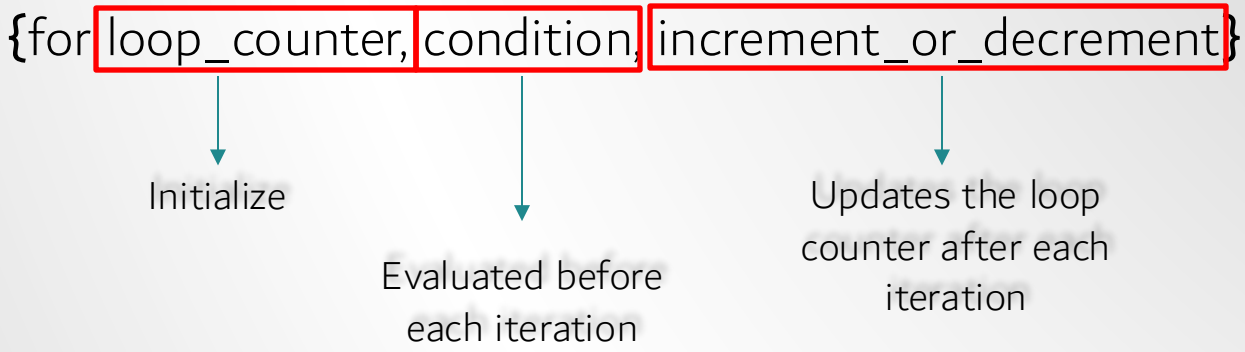
Iteration

For loop

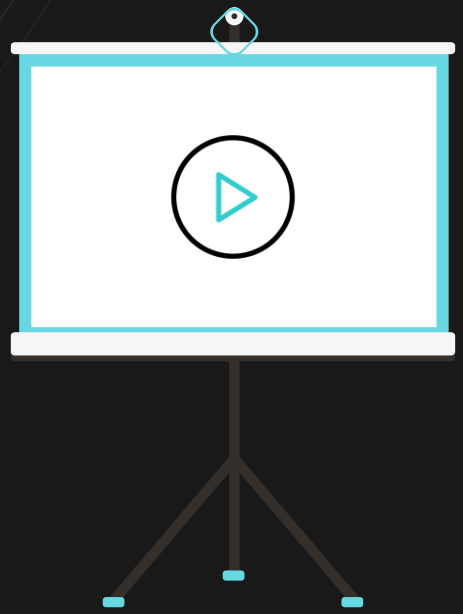
while Loop

do-while loop

Loops



Demonstration



Loops

Loops

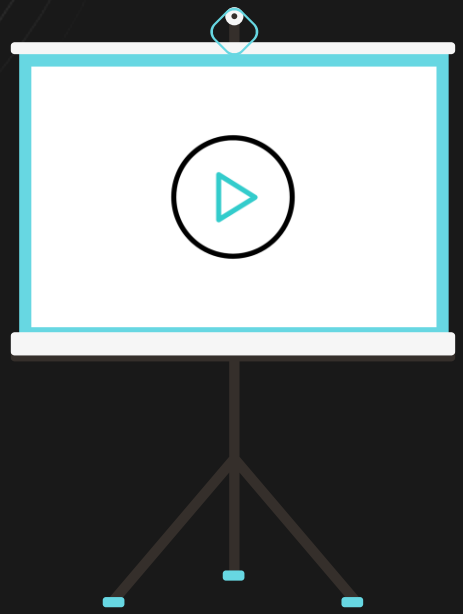
Jump statement

Continue Statement

Used to skip one or more iterations of a loop and proceed directly to the next iteration

Break Statement

Demonstration



Continue Statement

Loops

Jump statement

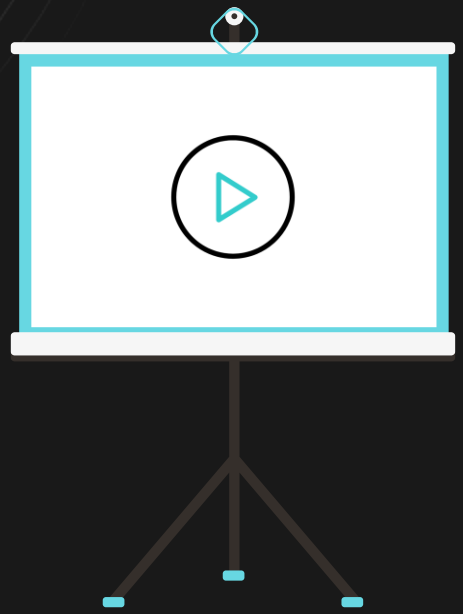
Continue Statement

Used to skip one or more iterations of a loop and proceed directly to the next iteration

Break Statement

Used to terminate the execution of a loop

Demonstration



Break Statement

For loop as while

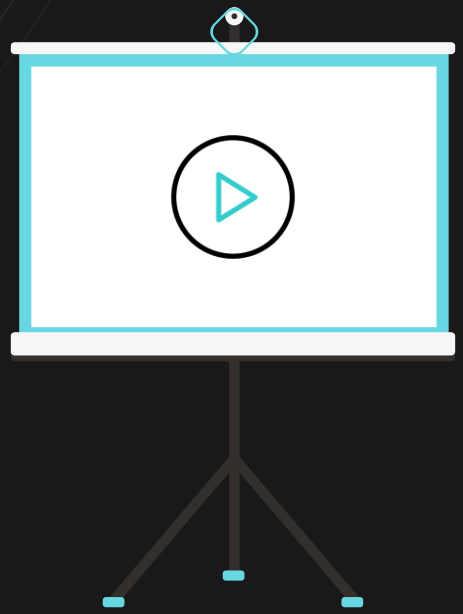
no explicit while loop

Provides a syntax within the for loop that can be used as a substitute for a while loop

Syntax

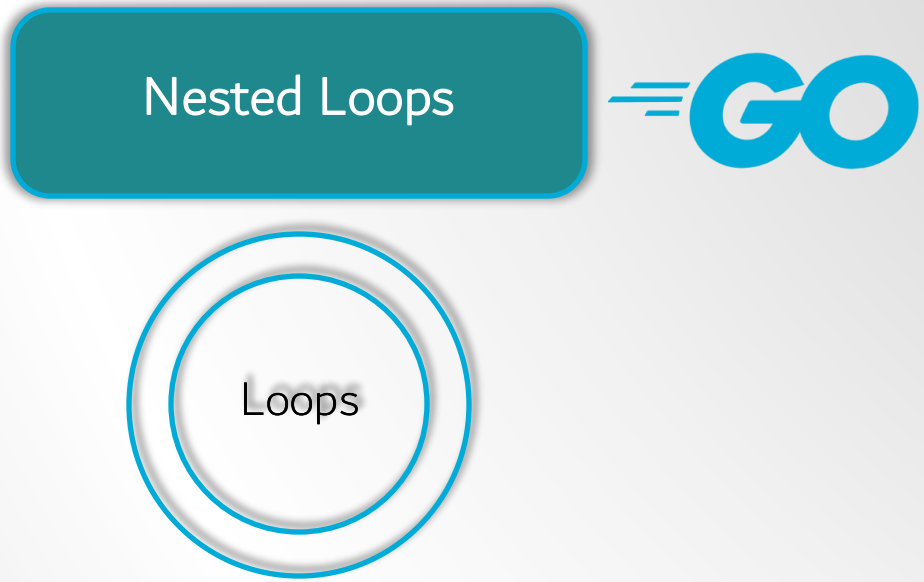
```
for (condition) {}
```

Demonstration



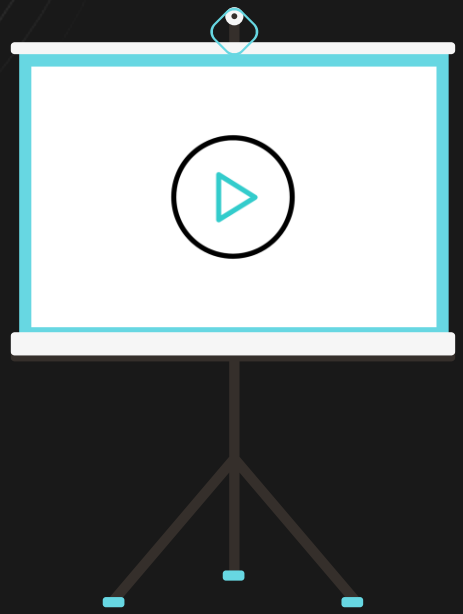
For loop as while

Nested Loops



Useful when dealing with multidimensional data structures or performing repetitive tasks within a loop

Demonstration



Nested Loops

Range Keyword

Range keyword

Used to simplify iteration over the elements of an

array	slice	map

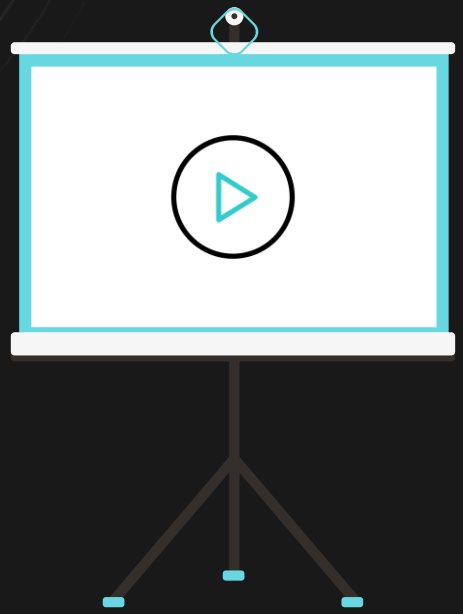
Provides both the index and the value of each element during the iteration

Syntax of range:

```
for idx, val := range variable_name{ }
```

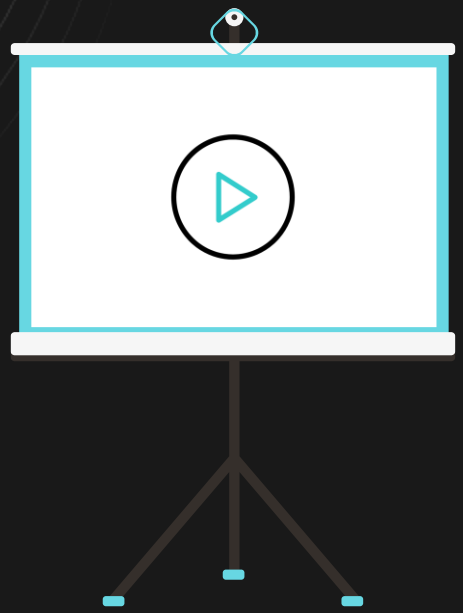
- `for` — Keyword to initiate the loop
- `idx` — Store the index
- `val` — Store the value

Demonstration



Range Keyword

Demonstration



Practical Use case of loops in go

Section Overview

- What is Function

Function

Block of code

Perform specific task

Reused multiple times

Function

Function

Inbuilt functions

User defined functions

len

For finding length

cap

For finding capacity

Defined by the developer

to make his/her work easy

Function

Inbuilt functions

User defined functions

len

For finding length

cap

For finding capacity

To create Function

Defined by the developer

to make his/her work easy

```
func function_name (parameter){}
```

Naming conventions

(A.....Z, a.....z)

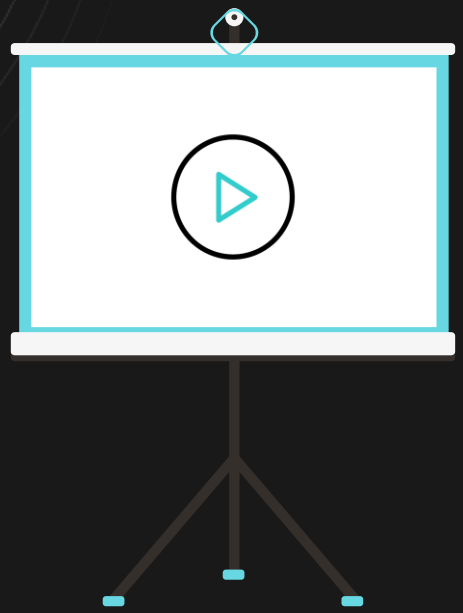
(A-Z, a-z, 0-9, _)

Case-sensitive

Thin knyx

camelCase or
snake_case

Demonstration



Functions

Parameters and Arguments

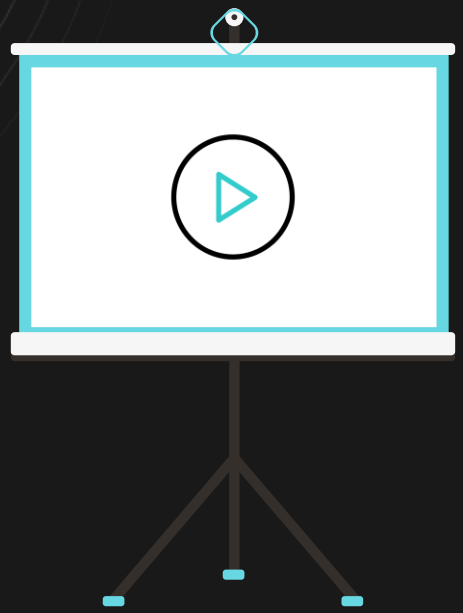
- Information can be passed to functions through parameters

  Act as variables within the function

- Parameters and their types are defined inside the parentheses()

  Include multiple parameters by separating them with(, ,)

Demonstration



Parameters and Arguments

Function return

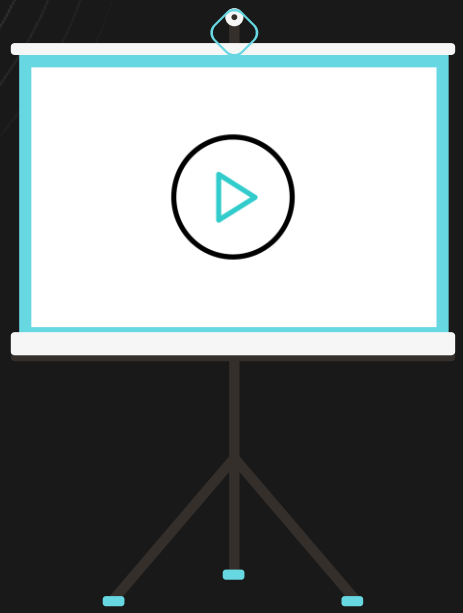
- To make a function return a value

  Specify the data type (such as int, string, etc.)

- Use the return keyword within the function to specify the value to be returned

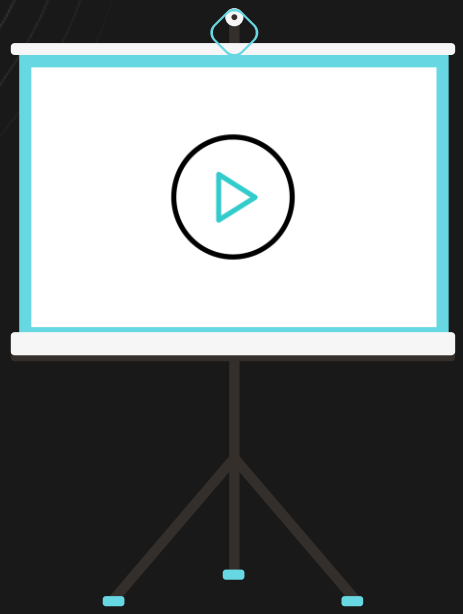
```
return name
```

Demonstration



Function return

Demonstration



Use Case: Order Placement

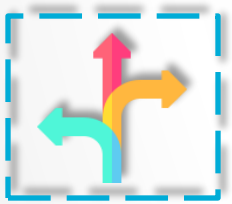
Section-5

Section Overview

- What is interface

Interfaces

Powerful feature in Go



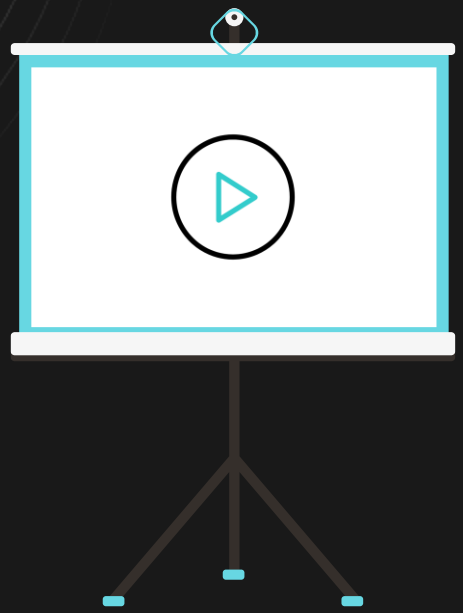
Flexible

Reusable code

- Allows us to define set of method signatures without implementing
- Defining a contract for behaviour

```
type shape interface {

  area() float64

}
```

Demonstration



Interface

Key Features

Decoupling and Abstraction

- Allows us to define methods without specifying exact type
- Leads to loose coupling between components
- Leads to loose coupling between components
- Easier to modify and extend code
- Without effecting related parts

```go
type Greeter interface {   // Interface
    Greet() string
}

type English struct{}   // Concrete type 1

func (e English) Greet() string {
    return "Hello"
}

type Spanish struct{}   // Concrete type 2

func (s Spanish) Greet() string {
    return "Hola"
}

func SayHello(g Greeter) {
    fmt.Println(g.Greet())
}

func main() {
    SayHello(English{})
    SayHello(Spanish{})
}
```

Key Features

Polymorphism

- Interface enables polymorphism in go
- Can use different types interchangeably if they implement the same interface
- E.g. Function that accepts an interface type can work with any concrete type that implements that interface
- Provides flexibility

```go
type Shape interface {   // Interface
    Area() float64
}

type Circle struct {   // Concrete type 1
    Radius float64
}
func (c Circle) Area() float64 {
     return 3.14 * c.Radius * c.Radius }

type Rectangle struct {   // Concrete type 2
    Width, Height float64
}
func (r Rectangle) Area() float64 {
     return r.Width * r.Height
}
func PrintArea(s Shape) {
    fmt.Printf("Area: %.2f\n", s.Area())
}

func main() {
    PrintArea(Circle{Radius: 5})
    PrintArea(Rectangle{Width: 4, Height: 6})
}
```

Key Features

Flexibility and Composition

- Composition rather than inheritance
- Can implement multiple interfaces, unlike OOPs
- Don't need to declare the implementation of interface
- Dose so by providing necessary methods

```go
type Writer interface {
    Write() string
}

type Reader interface {
    Read() string
}

// Concrete type implementing multiple interfaces
type File struct{}

func (f File) Write() string {
     return "Writing to file"
}

func (f File) Read() string  {
     return "Reading from file"
}

func main() {
    var w Writer = File{}
    var r Reader = File{}
    fmt.Println(w.Write())
    fmt.Println(r.Read())
}
```

Section Overview

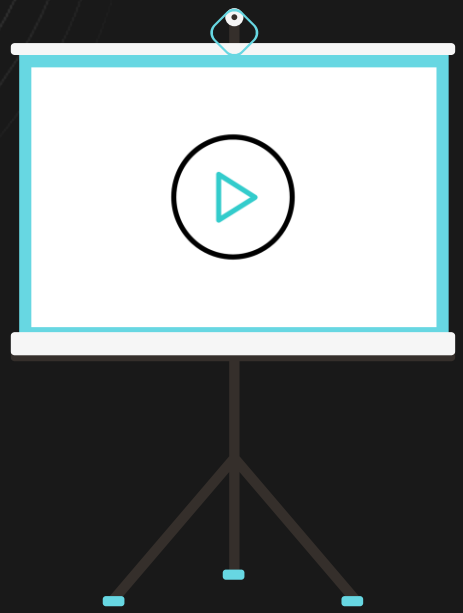
- Types of interface

Types of Interfaces

Type assertion

Type switch

Demonstration



Type assertion

Stringer

- Built-in interface in ‘fmt’ package
- Used for type representation as string
- If type implements stringer interface, string method is called

Stringer

```go
type Stringer interface {
    String() string
}
// Define a custom type
type Person struct {
    Name string
    Age  int
}
// Implement the Stringer interface
func (p Person) String() string {
    return fmt.Sprintf("Name: %s, Age: %d", p.Name,
p.Age)
}
func main() {
    p := Person{Name: "Alice", Age: 30}
// fmt.Println automatically calls the String method
    fmt.Println(p)
}
```

Benefits:

- ✓ Customization
- ✓ Integration
- ✓ Readability

Section-6

Section Overview

- What are goroutines

Goroutines

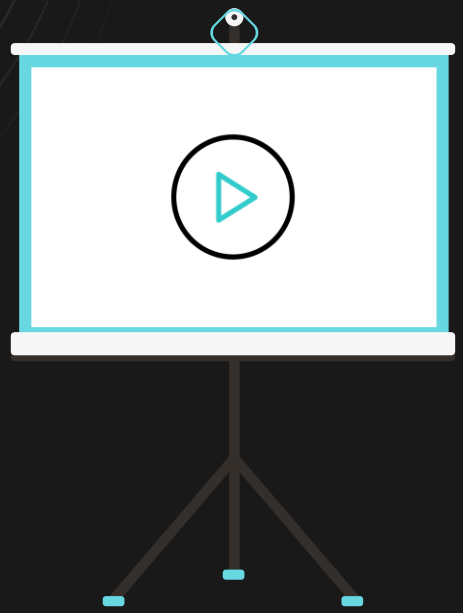
Goroutines

Lightweight thread execution

Achieve concurrency

"Go" Keyword

Demonstration



Goroutines

Advantages of Goroutines

Lightweight and Efficient

Simple Concurrency

Scalability

Safe Communication through Channels

Cost-Effective Parallelism

Section Overview

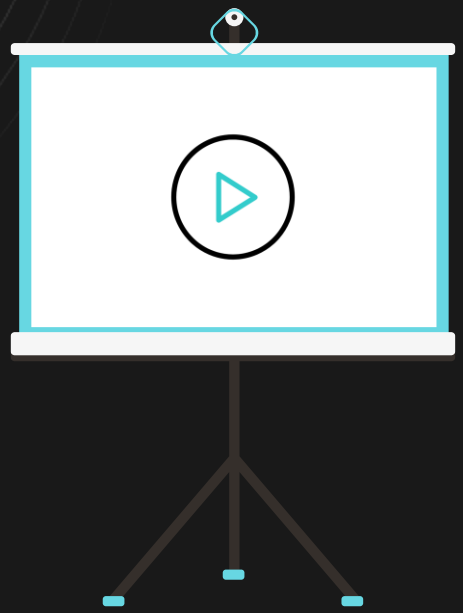
- Channels

Channels

Channels are a key feature used for communication between goroutines

- Enable concurrent programming
- Allows to send and receive values
- Created using make keyword or var keyword
- Creates a nil channel by default

Demonstration

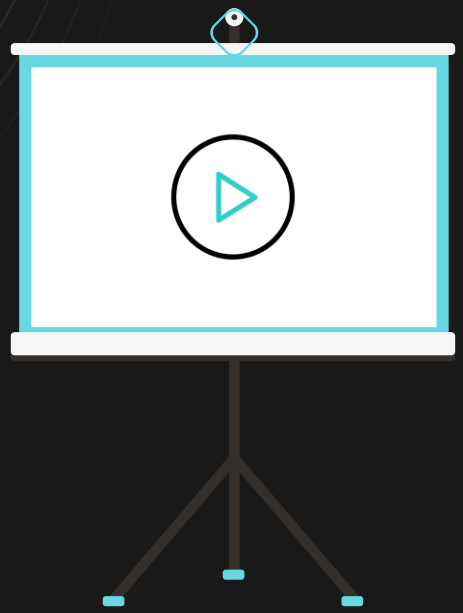


Channels

Section Overview

- Difference between goroutines & Sequential programming

Demonstration

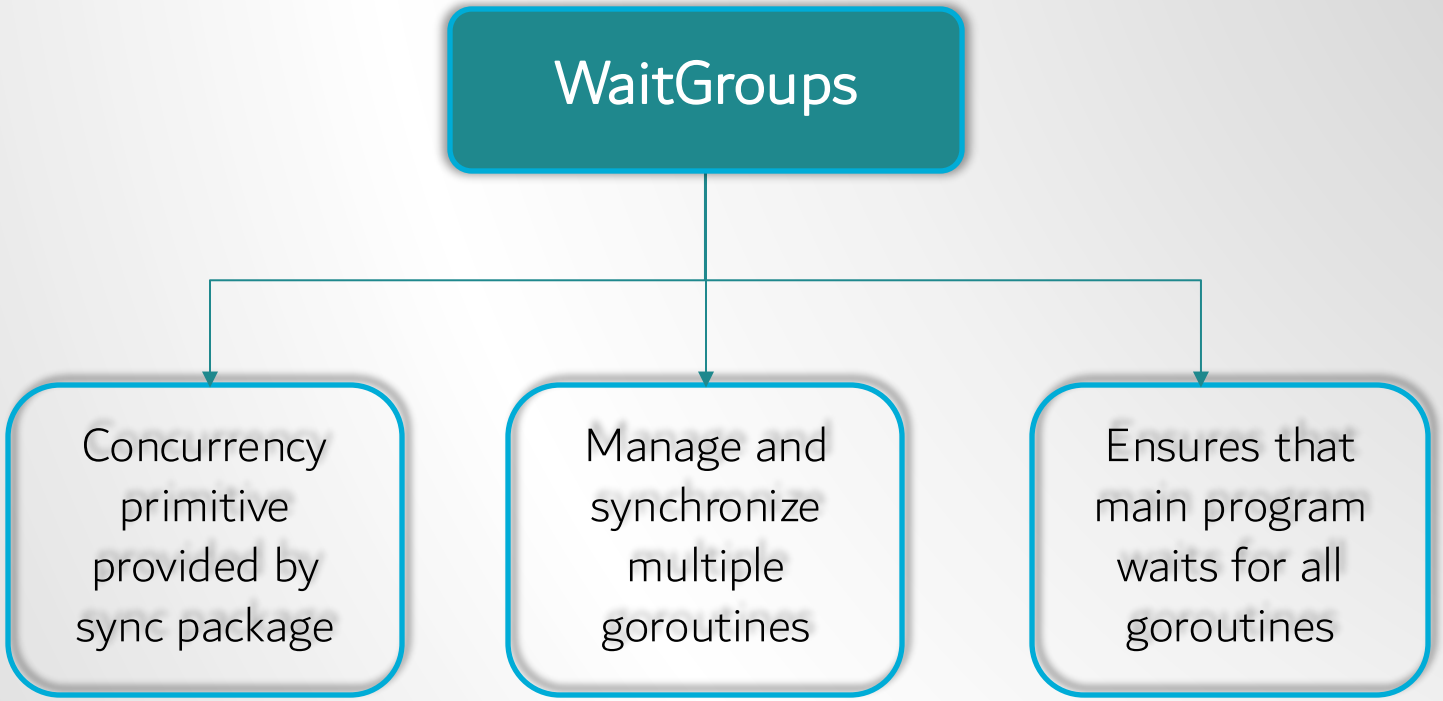


Difference between goroutines & Sequential programming

Section Overview

- What are wait groups

Wait groups



Wait groups

Add method

Specifies the number of goroutines the main function must wait for before proceeding

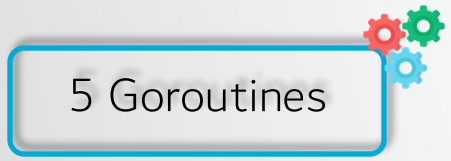
Wait method

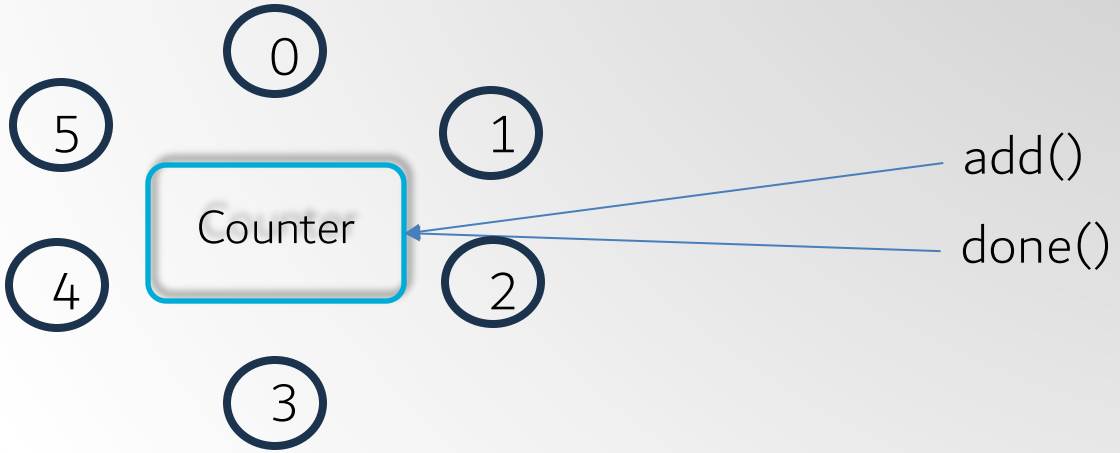
Blocks the execution of the code until the counter reaches zero

Done method

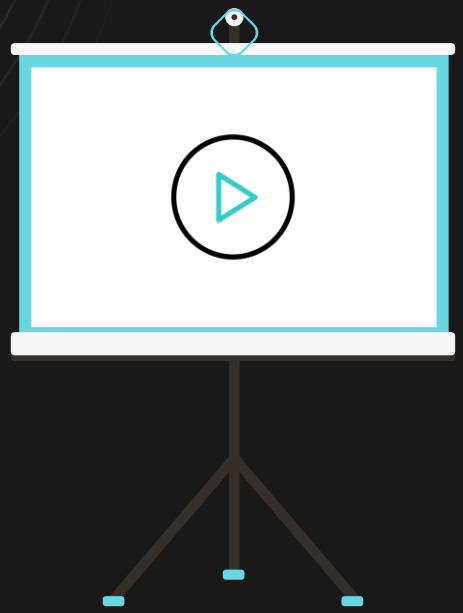
Decreases the counter by one after a goroutine finishes

Wait groups





Demonstration



Wait Groups

Section Overview

- Panic Situation

Panic Situation

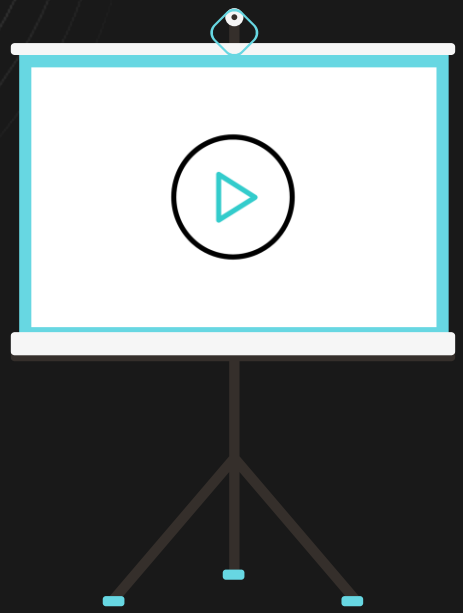
Panic

- Unexpected error or exceptional condition
- Halts the Execution of Program
- Used where program cannot continue to operate
- E.g. when critical errors occurs

Other reasons for occurrence of panic

- Sending data to closed channel
- Closing a closed channel

Demonstration



Struct

Section-7

Section Overview

- Library management

Library management

Modules

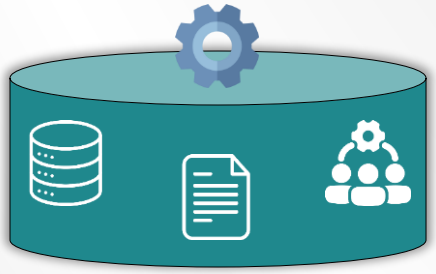
Repositories

Packages

Understanding Repositories

Repositories

"repo"



Associated with version control systems

Understanding Modules

Modules

Collection of related Go packages that are managed together

Organize and version code

Develop

Distribute

Depend

Module specifies the dependencies required to run your Go code

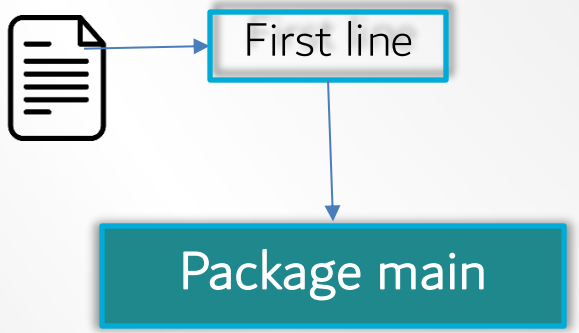
Version

Module

Understanding Packages

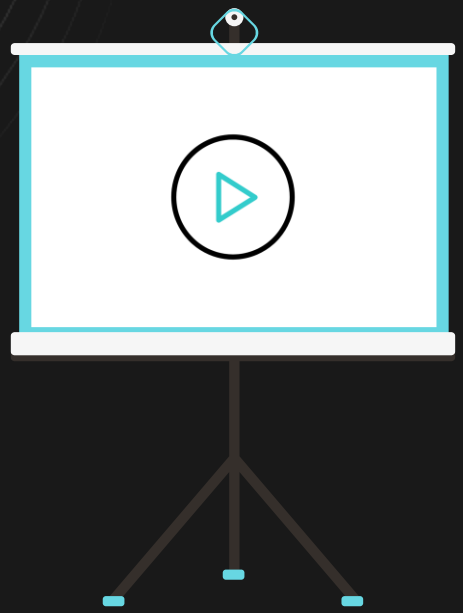
Packages

Way to group related Go source files together



Entry point of a Go application

Demonstration

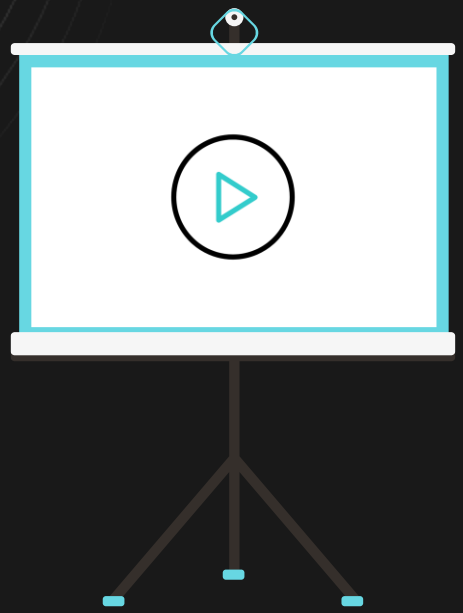


Creating a Go Module

Section Overview

- Create and Access Package

Demonstration



Create and access packages in go

Section Overview

- Commonly used Go commands

Go Commands

`go mod init`

Initializes and creates a `new go.`mod file in the current directory

Acts as the dependency manager for your module

`init`

Accepts an optional argument, which is the path or unique identifier of your new module

Go Commands

`go mod tidy`

Ensures that the **`go.mod`** file aligns perfectly with the source code of the module

Adds any missing module requirements necessary to build the current packages and dependencies

`go.sum`

Adding missing entries and cleaning up unnecessary ones

Essential for maintaining a clean and accurate dependency structure

Go Commands

```
go mod edit
```

Provides a command-line interface to `go.mode` file

Add, remove, or edit module dependency

Go Commands

```
go run <file>
```

Used to compile and run a Go program. Also used for executing the main Go package

```
.go
```

From a single directory, an import path, or even a pattern that matches a known package

```
go run .
```

```
main.go
```

Go Commands

```
go build
```

Compiles the Go packages specified by the import paths and their dependencies

```
.go
```

Arguments, the command treats them as source files for a single package and compiles them accordingly

Go Commands

```
go install
```

Compile and install packages specified by their paths

Executables, or **`main packages`**, are installed to the directory defined by the GOBIN environment variable

```
$GOPATH/bin or $HOME/go/bin
```

If GOPATH is not set

`go install` command caches them without installing them.

Go Commands

```
go get
```

Used to manage dependencies and updates the module requirements in the `go.mod` file for the main module

Builds and installs the packages listed on the command line

Section-8

Section Overview

- Inbuilt packages

Built-in packages

Inbuilt packages

- Predefined libraries — Go standard Library
- Ready-to-use functionality — Input/output operations, string manipulation, file handling, networking, cryptography, and more
- Designed to be — Efficient, reliable, and platform-independent

Packages can be used like as simple as, calling a function

Built-in packages

Packages are added as dependency to our project

Built-in packages

inbuilt packages come bundled together

Standard libraries

- Codes are bundled together to form a package
- Packages are bundled together to form a library
- Inbuilt packages provides us with pre-written code

Built-in packages

- Single package can be imported by using, **import <package name>**

```
import “fmt”

import “github.com/kansihk/v1”
```

- Multiple packages can be imported in a file by:

```
import (

“fmt”

“github.com/kansihk/v1”

)
```

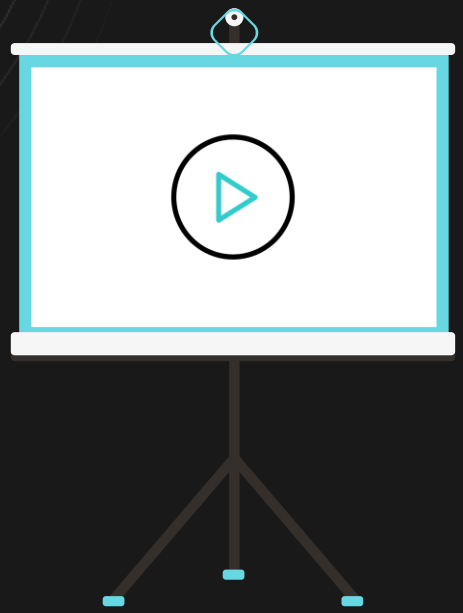
Built-in packages

- To avoid this error we can just place an underscore “_” before our package name

```
import _ “fmt”
```

- Defining a package

```
package “package_ name”
```

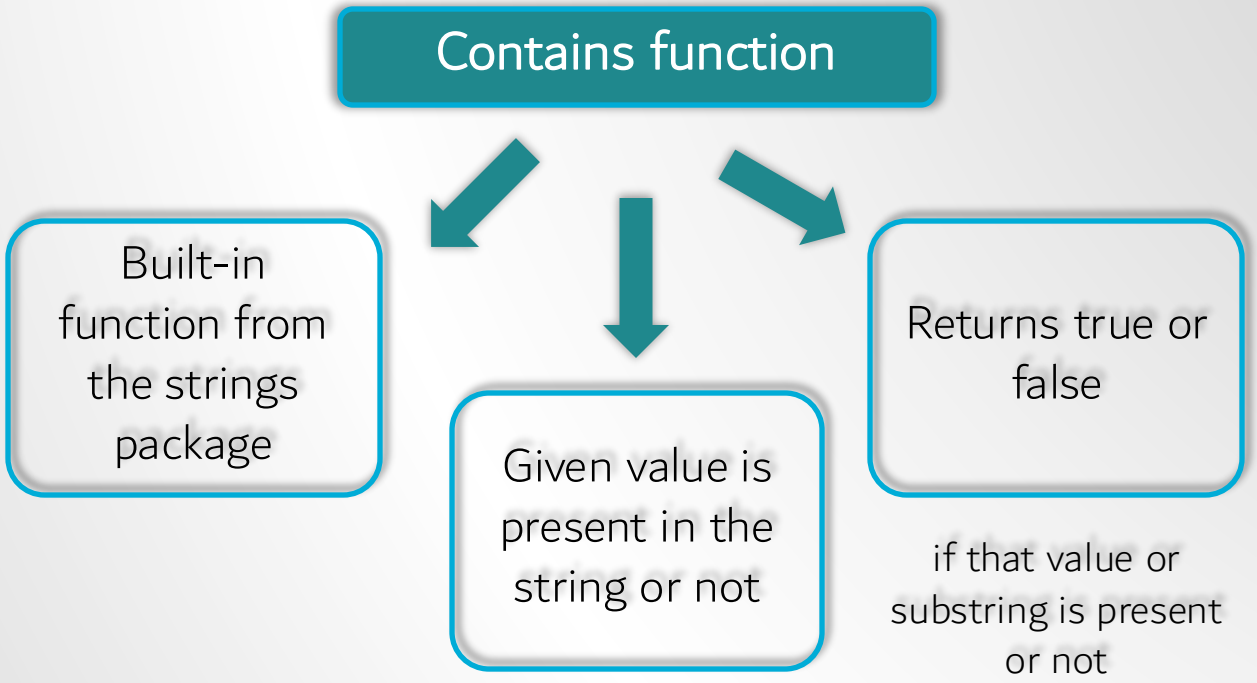


Built-In Packages

Section Overview

- Inbuilt package for strings

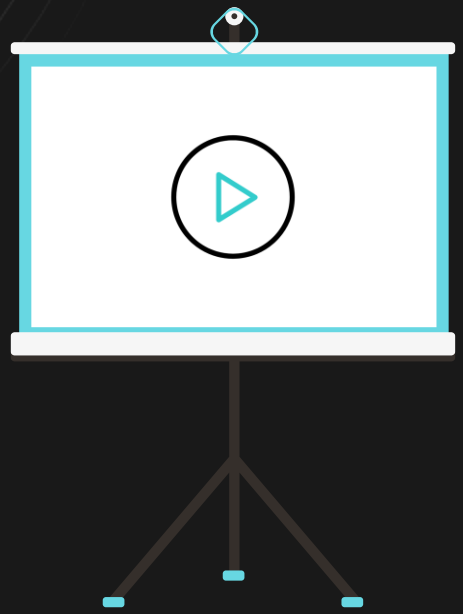
Contain



Contain

String.contains(value1, value2)

Demonstration



Contain

Count

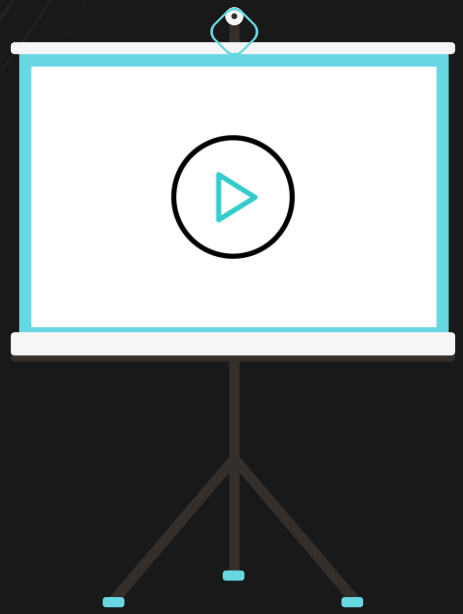
Count

Determines the number of occurrences of a specific word or character within a given string

String to search within

Word or character to count

Demonstration



Count

Section Overview

- File handling

File handling

OS library

- Provides API interface for file handling
- Creation, opening a file, modifying, deletion and many more

IO Package

- Provides basic interfaces for I/O primitives
- Handling file, network, and other I/O operations in Go programs

Filepath Package

- Utilizing the Filepath Package
- Helps parse and create file paths

File Handling

`filepath package`

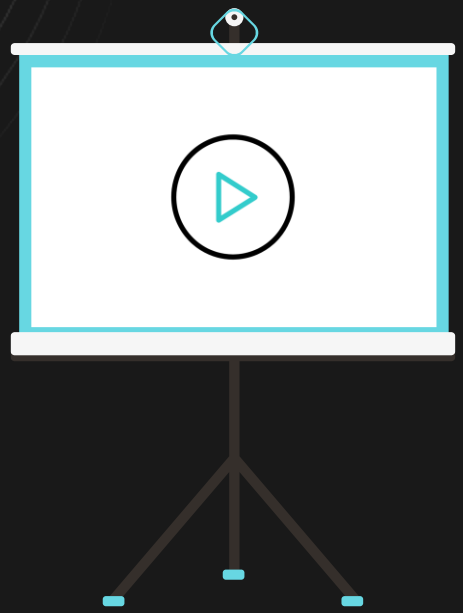
Join Function

Combines multiple path elements into a single path, using the appropriate separator for the operating system

Forward slashes

Backward slashes

Demonstration



Join

File Handling

Directory Function

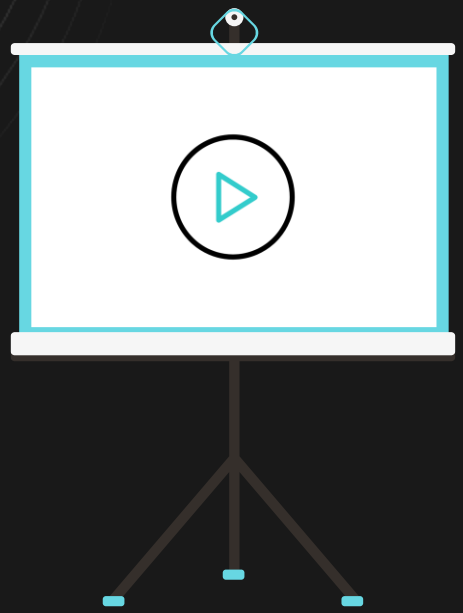
Returns all but the last element of a given path, which is typically the directory part of the path

it calls `Clean` on the path to remove any trailing slashes

Dir returns= `"."`

Dir returns= `single separator`

Demonstration



Directory Function

File Handling

`Dir Function`

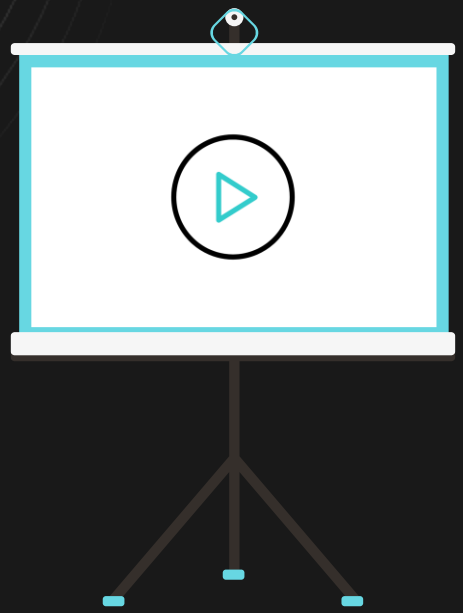
Base

Returns the last element of a given path. Before extracting the last element

Base returns= `"."`

Base returns= `single separator`

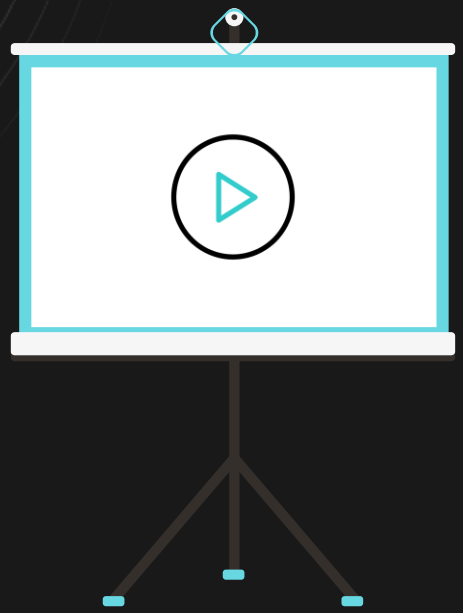
Demonstration



Base Function

Section Overview

- File handling in go



File Handling

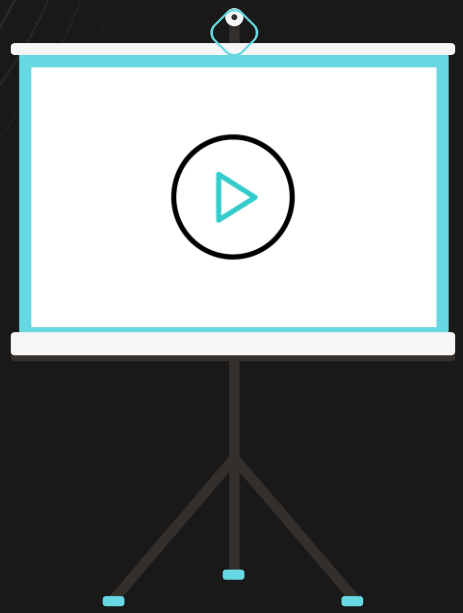
File Handling

`OpenFile function` Os packages

Creating and writing/appending to a file

Allows us to specify permission

- read-only
- write-only
- read-write



Creating and writing/appending to a file

Section-9

Section Overview

- Error Handling

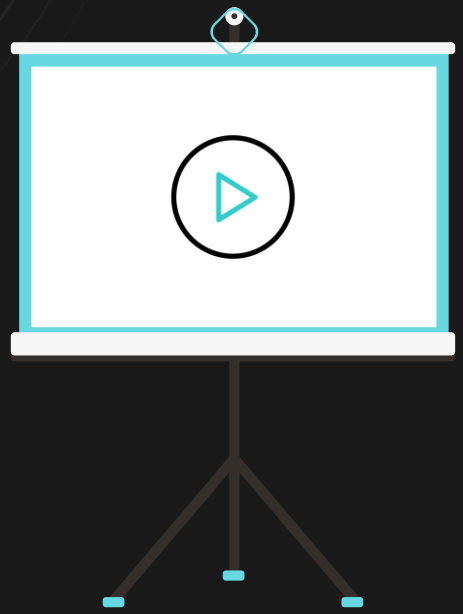
Error Handling

Halts execution and throws an error

errors.New()

fmt.Errorf()

Demonstration



Error Handling

Section Overview

- Enhanced Errors

Enhanced Errors

Basic error handling falls short with real-world scenarios

bufio.Scanner

- Reads entire line of text
- Suitable for handling dynamic or multiple inputs

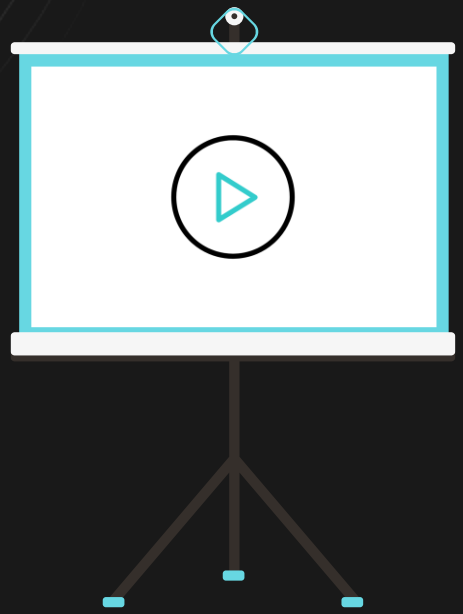
strings.Fields

- Split strings into smaller substrings
- Easier to handle multiple values

strconv.Atio

- Converts substrings into integers
- While managing errors for any valid inputs

Demonstration



Enhanced Errors

Section Overview

- Logging

Logging

Recording information about application's behaviour

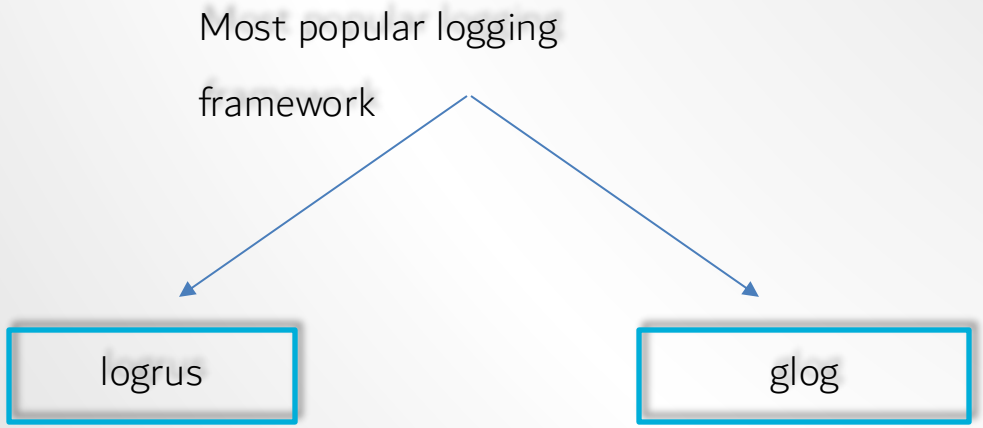
- Errors
- Events
- Operational Data

- Debugging
- Monitoring
- Maintaining Application

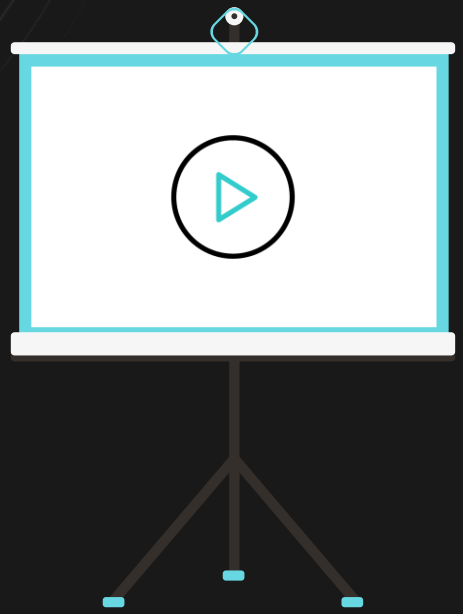
Logging

Log Package

- Provides simple and effective logging
- Useful for writing messages to console, file or other input streams
- Used for local development

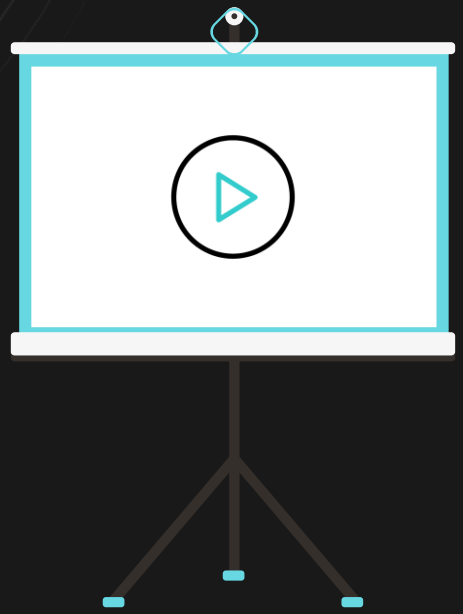


Demonstration



Logging

Demonstration



Use Case – File Handling

Section-10

Section Overview

- Microservices

Microservices

- Software Development Approach
- Each microservice focuses on specific function
- Communicates through lightweight protocols

HTTP

gRPC

Messaging Queues

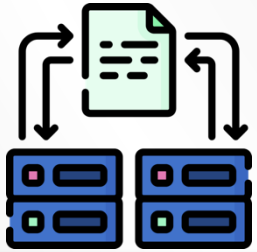
Microservices

Modularity

Decentralized Data Management

Independent Deployment

Technology Agnostic



Inter-Service Communication

Resilience

Microservices

Monolithic Applications

- Difficult to scale specific components
- Complex to maintain and update as the application grew
- Any change required redeploying the entire application
- Tightly coupled components led to high risk of system-wide failures

Microservices

Service Oriented Architecture

- Services were larger and typically communicated using enterprise service buses (ESBs)
- Emphasis on reusability and shared infrastructure

Challenges:

- Services were larger and typically communicated using enterprise service buses (ESBs)
- Emphasis on reusability and shared infrastructure

Microservices

Advent of Microservices

- Need for agility in software development (DevOps and CI/CD practices).
- Rise of containerization technologies (e.g., Docker, Kubernetes).
- Increased focus on scalability, resilience, and faster time-to-market.
- Cloud-native technologies enabling on-demand resource allocation

Microservices

Microservices in Modern Era

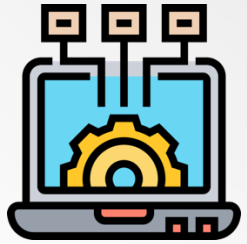
- Cloud platforms (AWS, Azure, Google Cloud) offering microservices-friendly infrastructure.
- Orchestration tools like Kubernetes for managing containerized services.
- API gateways for managing communication between services.

Advantages

Scalability

Agility

Resilience



Technology Flexibility

Section Overview

- Microservices in golang

Microservices

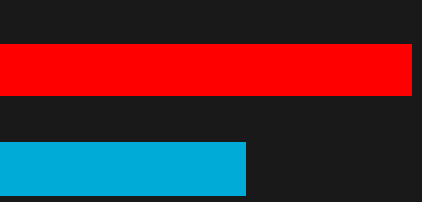
Some of the key features

- Lightweight and Fast
- Built-In Concurrency
- Scalable Architecture
- Rich Standard Library
- Ease of Deployment
- Inter-Service Communication

Microservices

Go frameworks for building microservices

Framework	Key Feature	Use Case
Gin	Fast and minimalist	Lightweight REST APIs
Go Micro	Distributed systems support	Complex service orchestration
Echo	High performance	Modular microservices
Kit	Toolkit for best practices	Reliable service design
Fiber	Express.js-like simplicity	High-performance APIs
Kratos	Enterprise-grade features	Scalable enterprise applications
gRPC-Go	RPC-based communication	Low-latency inter-service calls
Beego	All-in-one framework	Rapid development
Revel	Full-stack features	Structured microservices
Dapr	Event-driven runtime	Cloud-native, distributed systems



Thinknyx®
You Trust, We Deliver!

Go for the Absolute Beginners – Hands-On

Go programs

Data structures

Error Handling

Real-world use cases

Follow us on:

@thinknyx

@thinknyx

@thinknyx-technologies

@thinknyx

@thinknyx-technologies