**Object-Oriented Programming in GoLang**

Object-oriented programming is a programming paradigm which uses the idea of “objects” to represent data and methods. Go does not strictly support object orientation but is a lightweight object oriented language

**Struct:**

Structs in Golang are user-defined types that hold just the state and not the behavior. Structs can be used to represent a complex object comprising more than one key-value pairs.

**Program:**

package main

import (

"fmt"

)

type Book struct{

name string

author string

pages int

}

func (book Book) print\_details(){

fmt.Printf("Book %s was written by %s.", book.name, book.author)

fmt.Printf("\nIt contains %d pages.\n", book.pages)

}

func main() {

book1 := Book{"Monster Blood", "R.L.Stine", 131}

book1.print\_details()

book1.name = "Vampire Breath"

book1.pages = 162

book1.print\_details()

}

**Output:**

Book Monster Blood was written by R.L.Stine.

It contains 131 pages.

Book Vampire Breath was written by R.L.Stine.

It contains 162 pages.

**Encapsulation:**

Go, encapsulation is implemented by capitalizing fields, methods, and functions which makes them public. When the structs, fields, or functions are made public, they are exported on a package level.

**Inheritance:**

When a class acquires the properties of its superclass then we can say it is inheritance. Here, subclass/child class are the terms used for the class which acquire properties. For this one, one must use a struct to achieve inheritance in Golang.

**Interfaces**

Interfaces are types that have multiple methods. Objects that implement all the methods of the interface automatically implement the interface, i.e., interfaces are satisfied implicitly. By treating objects of different types in a consistent way, as long as they stick to one interface

**Receiver in Golang:**

Receiver can be of struct type or non-struct type. When you create a method in your code the receiver and receiver type must be present in the same package.

And you are not allowed to create a method in which the receiver type is already defined in another package including inbuilt type like int, string, etc.

#### Method with struct type receiver

#### In Go language, you are allowed to define a method whose receiver is of a struct type. This receiver is accessible inside the method .

#### package main

#### import "fmt"

#### type author struct {

#### name string

#### branch string

#### particles int

#### salary int

#### }

#### func (a author) show() {

#### fmt.Println("Author's Name: ", a.name)

#### fmt.Println("Branch Name: ", a.branch)

#### fmt.Println("Published articles: ", a.particles)

#### fmt.Println("Salary: ", a.salary)

#### }

#### func main() {

#### res := author{

#### name: "Sona",

#### branch: "CSE",

#### particles: 203,

#### salary: 34000,

#### }

#### res.show()

#### }

#### Output:

#### Author's Name: Sona

#### Branch Name: CSE

#### Published articles: 203

#### Salary: 34000

#### Methods with Pointer Receiver

#### In Go language, you are allowed to create a method with a pointer receiver. With the help of a pointer receiver, if a change is made in the method,

#### it will reflect in the caller which is not possible with the value receiver methods.

#### Syntax:

#### func (p \*Type) method\_name(...Type) Type {

#### }

#### Program:

#### package main

#### import "fmt"

#### type author struct {

#### name string

#### branch string

#### particles int

#### }

#### func (a \*author) show(abranch string) {

#### (\*a).branch = abranch

#### }

#### func main() {

#### res := author{

#### name: "Sona",

#### branch: "CSE",

#### }

#### fmt.Println("Author's name: ", res.name)

#### fmt.Println("Branch Name(Before): ", res.branch)

#### p := &res

#### p.show("ECE")

#### fmt.Println("Author's name: ", res.name)

#### fmt.Println("Branch Name(After): ", res.branch)

#### }

#### Output:

#### Author's name: Sona

#### Branch Name(Before): CSE

#### Author's name: Sona

#### Branch Name(After): ECE

#### Method Can Accept both Pointer and Value

#### Go method can accept both value and pointer, whether it is defined with pointer or value receiver

#### Program:

#### package main

#### import "fmt"

#### type author struct {

#### name string

#### branch string

#### }

#### func (a \*author) show\_1(abranch string) {

#### (\*a).branch = abranch

#### }

#### func (a author) show\_2() {

#### a.name = "Gourav"

#### fmt.Println("Author's name(Before) : ", a.name)

#### }

#### func main() {

#### res := author{

#### name: "Sona",

#### branch: "CSE",

#### }

#### fmt.Println("Branch Name(Before): ", res.branch)

#### res.show\_1("ECE")

#### fmt.Println("Branch Name(After): ", res.branch)

#### (&res).show\_2()

#### fmt.Println("Author's name(After): ", res.name)

#### }

#### Output:

#### Branch Name(Before): CSE

#### Branch Name(After): ECE

#### Author's name(Before) : Gourav

#### Author's name(After): Sona

#### What is Docker?

#### Docker is an open source platform for building, deploying, and managing containerized applications.

#### Developers can create containers without Docker, but the platform makes it easier, simpler, and safer to build, deploy and manage containers.

#### Docker is essentially a toolkit that enables developers to build, deploy, run, update, and stop containers using simple commands and work-saving automation through a single API.

#### .

#### What is Docker daemon?

#### Docker daemon runs on the host operating system. It is responsible for running containers to manage docker services. Docker daemon communicates with other daemons. It offers various Docker objects such as images, containers, networking, and storage.

#### Docker architecture

#### Docker follows Client-Server architecture, which includes the three main components that are Docker Client, Docker Host, and Docker Registry.

#### 

1. **Docker Client**

Docker client uses commands and REST APIs to communicate with the Docker Daemon (Server). When a client runs any docker command on the docker client terminal, the client terminal sends these docker commands to the Docker daemon. Docker daemon receives these commands from the docker client in the form of command and REST API's request.

Docker Client uses Command Line Interface (CLI) to run the following commands -

docker build

docker pull

docker run

2. **Docker Host**

Docker Host is used to provide an environment to execute and run applications. It contains the docker daemon, images, containers, networks, and storage.

3. **Docker Registry**

Docker Registry manages and stores the Docker images.

There are two types of registries in the Docker -

Pubic Registry - Public Registry is also called as Docker hub.

Private Registry - It is used to share images within the enterprise.