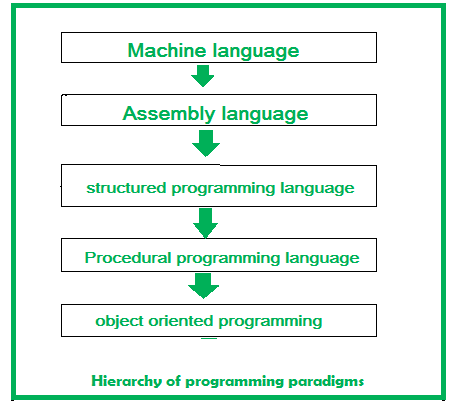
**1.26.2023 Golang Lesson instruction**

**OOP | Object Oriented Programming**

**Object-oriented design** started right from the moment computers were invented. Programming was there, and programming approaches came into the picture. Programming is basically giving certain instruction to the computer.

At the beginning of the computing era, programming was usually limited to machine language programming. Machine language means those sets of instructions that are specific to a particular machine or processor, which are in the form of 0’s and 1’s. These sequences of bits (0100110…). But it’s quite difficult to write a program or develop software in machine language. It’s actually impossible to develop software used in today’s scenarios with sequences of bits. This was the main reason programmers moved to the next generation of programming languages, developing assembly languages, which, were near enough to the English language to easily understand.



**Is Go an OOP Language?**

Go is not an OOP language-that much is for sure. But, can we use go language features to meet the principles of the object-oriented methodology? Yes, developers can, to some extent. For example, Go has no support of inheritance, but the idea can be compensated for with its support of composition. Similarly, a kind of polymorphism ca be created using interfaces. Understand that, if one truly wants to develop an application according to the object-oriented methodologies, Go is definitely not a good choice. Rather, one should look for languages like Java, C++, Python or any other high functioning language that supports object-oriented features.

// declaring a struct

type Book struct{

    // defining struct variables

    name string

    author string

    pages int

}

// function to print book details

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)

}

// main function

func main() {

    // declaring a struct instance

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

    // printing details of book1

    book1.print\_details()

    // modifying book1 details

    book1.name = "Vampire Breath"

    book1.pages = 162

    // printing modified book1

    book1.print\_details()

}

**Encapsulation**

It means hiding sensitive data from user. In 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 the package level. Some examples of public and private members are:

package gfg

// this function is public as

// it begins with a capital letter

func Print\_this(){

        // implementation

}

// public struct

type Book struct{

        // public field

        Name string

        // private field, only

        // available in gfg package

        author string

}

**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, Golang implements polymorphism.

// Golang program to illustrate the

// concept of interfaces

package main

import (

    "fmt"

)

// defining an interface

type Sport interface{

    // name of sport method

    sportName() string

}

// declaring a struct

type Human struct{

    // defining struct variables

    name string

    sport string

}

// function to print book details

func (h Human) sportName() string{

    // returning a string value

    return h.name + " plays " + h.sport + "."

}

// main function

func main() {

    // declaring a struct instance

    human1 := Human{"Rahul", "chess"}

    // printing details of human1

    fmt.Println(human1.sportName())

    // declaring another struct instance

    human2 := Human{"Riya", "carrom"}

    // printing details of human2

    fmt.Println(human2.sportName())

}

**Inheritance**

In OOP, computer programs are designed in such a way where everything is an object that interacts with one another. Inheritance is an integral part of OOP languages which lets the properties of one class to be inherited by the other. It basically helps in reusing the code and establish a relationship between different classes.

type Vehicle struct {

    Seats int

    Color string

}

type Car struct {

    Vehicle

}

type MotorCycle struct {

    Base Vehicle

}

In the above example, car struct is being embedded by another structure anonymously.

This means that we have direct access to the fields. This method is similar to what we are used to one the OOP side.

func main() {

    car := &Car{

        Vehicle{

            Seats: 4,

            Color: "blue",

        },

    }

    fmt.Println(car.Seats)

    fmt.Println(car.Color)

}

**Polymorphism in Go**

Polymorphism is another key feature of *object oriented programming*. It provides the ability to write code through the implementation of types that can take on different behavior at runtime. In Go, Polymorphism is achieved through interfaces and interfaces only. Once an interface implements a type, the functionality defined within it is open to any values of that type. The standard library is replete with this implementation. For example, the io package has an extensive set of interfaces and functions to stream data efficiently in our code. Here is a quick example to illustrate how polymorphic behavior is achieved through interfaces in Go:

package polymorph

import (

    "fmt"

    "math"

)

type Shape interface {

    area()

}

type Rectangle struct {

    X1, Y1, X2, Y2 float64

}

type Circle struct {

    Xc, Yc, Radius float64

}

func (r \*Rectangle) area() {

    fmt.Println("Rectangle Area : ", (r.X2-r.X1)\*(r.Y2-r.Y1))

}

func (c \*Circle) area() {

    fmt.Println("Circle Area: ", math.Pi\*math.Pow(c.Radius, 2))

}

func GetArea(s Shape) {

    s.area()

}

//--------------------------------------------------

package main

import "oop\_prog/polymorph"

func main() {

    r := polymorph.Rectangle{10, 10, 20, 20}

    polymorph.GetArea(&r)

    c := polymorph.Circle{5, 5, 30}

    polymorph.GetArea(&c)

}