**Structs and Interfaces in Go**

**What is a Struct?**

**Definition**: A struct in Go is a custom type that allows you to define a collection of fields (variables) that can hold different types of data.

**Syntax**:

type <TypeName> struct {

<FieldName1> <FieldType1>

<FieldName2> <FieldType2>

...

}

**Example**:

type gasEngine struct {

mpg uint8

gallons uint8

}

**Creating and Using Structs**

**Zero-valued Structs**: When a struct is created, its fields are initialized with default zero values (e.g., 0 for numbers).

**Example**:

var myEngine gasEngine

fmt.Println(myEngine.mpg, myEngine.gallons)

// Output: 0 0

**Struct Literals**: You can create a struct with specific values using literals.

**With field names**:

var myEngine2 gasEngine = gasEngine{mpg: 25, gallons: 15}

**Without field names** (values assigned in order):

var myEngine3 gasEngine = gasEngine{30, 15}

**Embedding Structs**

You can embed other structs inside a struct. This means that one struct can contain another.

**Example of owner struct**:

type owner struct {

name string

}

**Embedding the owner struct**:

type gasEngine struct {

mpg uint8

gallons uint8

owner // embedded struct, shorthand for `owner owner`

// or do this: ownerinfo owner

}

**Accessing Struct Fields**

You can access fields of an embedded struct directly (without needing to reference the nested struct).

var myEngine3 gasEngine = gasEngine{30, 15, owner{"Alex"}}

fmt.Println(myEngine3.mpg, myEngine3.gallons, myEngine3.name) // Access the name directly

// or do this: fmt.Println(myEngine3.mpg, myEngine3.gallons, myEngine3.ownerinfo.name)

**Updating a field**:

myEngine3.mpg = 20

fmt.Println(myEngine3.mpg, myEngine3.gallons) // Output: 20 15

**Anonymous Structs in Go**

**What is an Anonymous Struct?**

* An **anonymous struct** is a struct that does not have a name associated with it. It is defined and initialized at the same time, typically for one-off use or when you don't need to reuse the structure elsewhere.

**Defining and Initializing an Anonymous Struct**

* You can define an anonymous struct and initialize its fields at the same time using the struct keyword.

var myEngine4 = struct {

        mpg     uint8

        gallons uint8

    }{25, 15}

    fmt.Println(myEngine4.mpg, myEngine4.gallons)

**Not reusable**: The main drawback is that you cannot reuse this struct definition. If you need to create another struct with the same structure, you'll need to rewrite the entire definition.

var myEngine5 = struct {

        mpg     uint8

        gallons uint8

    }{30, 20}

    fmt.Println(myEngine5.mpg, myEngine5.gallons)

**Methods on Structs**:

* In Go, methods are just functions tied to a specific type.
* Methods allow the struct to access its own fields, and it can interact with other methods attached to that type.

**Example of milesLeft Method**:

func (e gasEngine) milesLeft() uint8 {

    return e.gallons \* e.mpg

}

* + This milesLeft method belongs to the gasEngine struct. It calculates how far the vehicle can travel based on the fuel (gallons) and fuel efficiency (mpg).

**Using Methods**:

You can call these methods by creating an instance of the struct and invoking the method on it.

var myEngine1 gasEngine = gasEngine{25, 1}

fmt.Println(myEngine1.mpg, myEngine1.gallons)

fmt.Printf("The miles left in tank: %v\n", myEngine1.milesLeft())

func canMakeIt(e gasEngine, miles uint8){

    if miles<=e.milesLeft(){

        fmt.Println("You can make it there!")

    }else{

        fmt.Println("Need to fuel up first!")

    }

}

type electricEngine struct {

    mpkwh uint8

    kwh   uint8

}

func (e electricEngine) milesLeft() uint8 {

    return e.kwh \* e.mpkwh

}

var myEngine2 electricEngine = electricEngine{25, 15}

    fmt.Println(myEngine2.mpkwh, myEngine2.kwh)

    fmt.Printf("The miles left in tank: %v\n", myEngine2.milesLeft())

**Interfaces in Go**

1. **Interfaces Overview**:
   * An interface in Go specifies a set of method signatures that a type must implement.
   * **Key Point**: A type doesn't have to declare that it implements an interface. As long as it has the required methods, it automatically satisfies the interface.
2. **Applying Interfaces to the Problem**:
   * You might have multiple engine types, such as a gasEngine and an electricEngine, each with a milesLeft method.
   * To generalize a function that works for both engine types, we can use an interface.
3. **Example of Interface Definition**:

**engine Interface**:

type engine interface {

    milesLeft() uint8

}

The interface specifies that any type implementing the milesLeft method with the signature func milesLeft() uint8 can be used in place of an engine.

1. **Using Interface in Functions**:
   * The function canMakeIt can now accept any type that satisfies the engine interface.
   * This makes the function more flexible and reusable for different engine types.

**Example**:

func canMakeIt(e engine, miles uint8) {

    if miles <= e.milesLeft() {

        fmt.Println("You can make it there!")

    } else {

        fmt.Println("Need to fuel up first!")

    }

}

canMakeIt can accept any engine type (gasEngine, electricEngine, etc.) and check if the vehicle can make it to the destination based on its milesLeft method.

**Difference Between Specific Type and Interface**

* **Specific Type (gasEngine)**:
  + The function canMakeIt can only accept a gasEngine.
  + This is restrictive and limits the flexibility of your code.
* **Using Interface**:
  + The canMakeIt function can now accept any type that has the milesLeft() method, including gasEngine and electricEngine.
  + This improves code flexibility and reusability.

**Why Interfaces are Useful in Go**

* **Flexibility**:
  + Interfaces allow you to write more general functions that can operate on different types as long as they implement the required methods.
* **Code Reusability**:
  + Instead of rewriting the function for each engine type, you can write the function once and use it for any type that satisfies the interface.