Not OOP

- While Go is not a traditional object-oriented language like `Java`, `C++` or `Python`, it allows you to apply some object-oriented principles in a different way.
- Let's explore how you can use Go in an object-oriented programming (OOP) manner, including using structs instead of classes and the `New()` function instead of constructors.

Structs Instead of Classes

In Go, you use structs to define data structures that encapsulate both data and methods (functions) that operate on that data. While Go doesn't have classes, structs serve a similar purpose and allow you to define types with fields and methods.

In a Dungeons & Dragons (D&D) context, let's create a struct to represent a D&D character:

```
package main

import "fmt"

// Character represents a D&D character.
type Character struct {
    Name    string
    Class   string
    Level   int
    HitDice int
}
```

Structs Instead of Classes

```
// Method associated with the Character struct
func (c Character) PrintInfo() {
    fmt.Printf("Name: %s\nClass: %s\nLevel: %d\nHit Dice: %d\n", c.Name, c.Class, c.Level, c.HitDice)
}

func main() {
    // Create a D&D character using a struct literal
    gandalf := Character{
        Name:    "Gandalf",
        Class:   "Wizard",
        Level:   10,
        HitDice: 6,
    }

    // Call a method on the character
    gandalf.PrintInfo()
}
```

In this example:

- We define a `Character` struct with fields for the character's name, class, level, and hit dice.
- We define a method `PrintInfo()` associated with the `Character` struct to print the character's information.
- We create a D&D character (`gandalf`) using a struct literal and call the `PrintInfo()` method on it.

Structs Instead of Classes

While Go doesn't have classes and inheritance in the traditional sense, structs provide a way to encapsulate data and behavior together, similar to objects in OOP.

by embedding other types (structs) or by implementing interfaces.
    - This simplifies code and avoids complex inheritance hierarchies.
3. **Interfaces for Polymorphism**:
    - Polymorphism in Go is achieved through interfaces, which define a set of method signatures that a type must implement.
    - Types in Go are not explicitly tied to interfaces but can satisfy interfaces implicitly based on method signatures.
    - This allows for more flexible and duck-typed polymorphism compared to statically declared inheritance hierarchies in C++.
4. **No Constructors or Destructors**:
    - Go does not have constructors or destructors like C++. Instead, you use factory functions to create instances and rely on garbage collection for resource management.
    - Initialization methods, such as `func NewType(...)` conventions, are often used in Go to initialize struct instances.