Go Structs

The Blueprint of Data

```
type User struct { ... }
```

Part 5: Go Basics Series

1. What is a Struct?

In Go, we don't have Classes. We have **Structs**.

A Struct is a user-defined type that groups together different fields of data. Think of it as a custom container for related information.

Why use them?

- **Organization:** Keep related data (Name, Age, Email) in one place.
- **Type Safety:** Define strictly what data your application passes around.
- **Real World Modeling:** Represent Users, Orders, or Products.

```
type Vertex struct {
    X int
    Y int
}
```

2. Creating Structs

There are multiple ways to initialize a struct. This is called a "Struct Literal".

```
type User struct {
    Name string
    Age  int
}

func main() {
    // 1. Named fields (Recommended)
    u1 := User{Name: "Alice", Age: 30}

    // 2. Order based (Risky if fields change)
    u2 := User{"Bob", 25}

    // 3. Zero Value (Fields are set to "" and 0)
    var u3 User
}
```

3. Accessing Data

You access fields using the **dot . notation**.

```
func main() {
    v := Vertex{X: 1, Y: 2}

    // Get a value
    fmt.Println(v.X)

    // Set a value
    v.X = 4
    fmt.Println(v.X)
}
```

4. Pointers to Structs

Often, we pass a **pointer** to a struct (`*User`) instead of the struct itself. This avoids copying the data (performance) and allows us to modify the original.

Go Magic: You don't need to manually dereference (`(*p).X`). You can just type `p.X`.

```
func main() {
    v := Vertex{X: 1, Y: 2}
    p := &v   // p is a pointer to v

    p.X = 19  // Go understands this is (*p).X

    fmt.Println(v.X) // Prints 19
}
```

5. Embedding (Composition)

Go uses "Composition over Inheritance". You can embed one struct inside another to reuse fields.

```
type Address struct {
    City string
}

type User struct {
    Name    string
    Address // Embedded Struct (No name given)
}

func main() {
    u := User{
        Name: "Dave",
        Address: Address{City: "NYC"},
    }

    // Access embedded fields directly!
    fmt.Println(u.City) // Prints "NYC"
}
```

6. Struct Tags (Meta-Data)

You will often see backticks `` `...` `` after fields. These are **Tags**, used by libraries like `encoding/json` to know how to name fields.

```go
type Product struct {
    ID    int    `json:"id"`
    Name  string `json:"product_name"`
    Price int    `json:"-"` // Ignore this field
}

func main() {
    p := Product{ID: 1, Name: "Phone", Price: 500}

    b, _ := json.Marshal(p)
    fmt.Println(string(b))
    // Output: {"id":1, "product_name":"Phone"}
}
```

7. Summary

- **Definition:** A collection of fields. No classes, just Structs.
- **Initialization:** Use `Struct{Key: Val}` for clarity.
- **Pointers:** Use `*Struct` to modify data and avoid copying. Access is easy via dot notation `p.X`.
- **Composition:** Embed structs inside others to share behavior.
- **Tags:** Use backticks to control JSON/Database behavior.