STRUCTS

- Represents the structured data
- Syntax:

```
type  <Name_Of_Struct> struct {

Property1  <Respective_Type>
Property2  <Respective_Type>
Property3  <Respective_Type>
          .
          .
          .
PropertyN  <Respective_Type>

}
```

- Example: user information

```
type user struct {
     Name string
     Email string
     Age int

}
```

- The above snippet conveys that it's a struct type named user.
- Can be used to group different types of variables together.
- Assigning and / or Accessing Struct values:
- Accessible using dot (.) operator
- Example:

```
tempUser := user {
     Name = "manish"
     Email = "manish.abc@def.com"
}
```

- In code,
  fmt.Printf("name is % email is %s",tempUser.Name ,tempUser.Email)

NESTED STRUCTS:

- Used to handle more complex data models/ entities

- Examples for nested structs:

```
type user struct {
     Name string
     Email string
     Age int
     Address address

}
```

```
type address struct {
     State string
     Country string
     Zip int
}
```

ANONYMOUS STRUCTS:

- Struct without a name would generally be called as an anonymous struct
- Can be nested

- Must be instantiated immediately

- Example:

```
tempUser := struct {
     Name string
     Email string
} {
     Name : "manish"
     Email: "manish.abc@def.com"
}
```

- Should be used when you do not want to use the struct again like is only meant to be used once

EMBEDDED STRUCTS:

- Useful for sharing the fields between other structs

- Example:

```
type users struct {
    address
    Name  string
    Email string
    Age   int
}

type address struct {
    State   string
    Country string
    Zip     int
}
```

- Can be accessed in two ways:

```
userData := user {
     Name: "abc",
     Address: address {
          State: "xyz"
     }
}
```

```
fmt.Println(userData.Name)
fmt.Println(userData.Address.State); OR Fmt.Println(userData.State)
```

STRUCT METHODS:

- Go supports methods defined on structs
- Methods are indeed functions but with special parameter called as `receiver`
- Will be specified before the function name
- Allow us to define interfaces that our structs
- Example:

```
type circle struct {
    radius float
   }
// area has a receiver of (c circle) :
// receiver here (in blue box) is  a special type of
function parameter

func (c circle) area() float {
    return 3.14 * c.r * c.r
}
```