struct چیست؟

پیچیده‌ترین ساختار داده‌ای که تا حالا باهاش کار کردیم، map بود. اگه بخوایم یه مرور سریع بکنیم، map به ما اجازه می‌داد یه کلید (key) رو به یه مقدار (value) وصل کنیم. مثلاً تو برنامه‌ی دفترچه تلفن، اسم هر نفر رو با کمک map به شماره تلفنش وصل می‌کردیم.

اما بعضی وقتا دلمون می‌خواد آزادی عمل بیشتری داشته باشیم. مثلا فقط اسم و شماره تلفن کافی نیست؛ می‌خوایم آدرس ایمیلش رو هم داشته باشیم. شاید حتی تاریخ تولدش، شهر محل سکونتش یا هر چیز دیگه‌ای.

اینجاست که struct به کمکمون میاد!

چرا struct؟

تو map فقط می‌تونستیم نوع کلید و نوع مقدار رو مشخص کنیم. یعنی همه‌ی کلیدها باید از یه نوع باشن، همه‌ی مقدارها هم از یه نوع مشخص.

اما تو struct:

- دیگه محدود به دو داده (کلید و مقدار) نیستیم
- به هر تعداد که دلمون بخواد میتونیم فیلد (هر داده ای که تو struct ذخیره میشه) تعریف کنیم
- حتی نوع هر فیلد رو هم می‌تونیم جداگونه تعیین کنیم!

تعریف یه struct

برای تعریف یه struct، از کلمه‌ی کلیدی type استفاده می‌کنیم. شکل کلیش اینه:

```
type StructName struct {
    FieldName1 FieldType1
    FieldName2 FieldType2
    // ...
}
```

بیاید خطبه‌خط بررسی کنیم:

- type: قبلاً باهاش آشنا شدیم؛ یعنی داریم یه نوع جدید تعریف می‌کنیم.
- Name: اسم نوع جدیده. هر اسمی می‌تونه باشه (معمولاً با حرف بزرگ شروع می‌کنیم).
- struct { ... }: کلمه‌ی کلیدی برای تعریف ساختار داده‌ای struct
- FieldName: نام هر فیلد (field) داخل struct
- FieldType: نوع هر فیلد (مثل string, int, bool, []string, map[string]int)

مثال:

```
type Contact struct {
    Name  string
    Phone string
    Email string
}
```

خب اینجا یه نوع جدید به اسم Contact ساختیم. این نوع از سه تا فیلد تشکیل شده:

- Name برای ذخیره کردن نام هر مخاطب، از نوع string
- Phone برای ذخیره کردن شماره تلفن هر مخاطب، از نوع string
- Email برای ذخیره کردن ایمیل هر مخاطب، از نوع string

استفاده از struct تعریف‌شده

حالا که یه struct ساختیم، چطوری ازش استفاده کنیم؟ خیلی ساده:

```
var nedaContact Contact
```

اینجا یه متغیر جدید به اسم nedaContact تعریف کردیم که از نوع Contactـه. یعنی می‌تونه سه تا داده‌ی مربوط به یه مخاطب رو تو خودش نگه‌داره.

مقداردهی به struct

برای مقداردهی به struct دو روش کلی داریم:

1-مقداردهی هنگام تعریف (literal)

روش اول: با نام داده‌ها (Named)

```
nedaContact := Contact{
    Name:  "Neda",
    Phone: "+989051010202",
    Email: "neda@gopher.com",
}
```

اینجا خیلی واضح گفتیم که هر مقدار مربوط به کدوم داده‌ست. این روش خواناتر و امن‌تره، مخصوصاً وقتی تعداد داده‌ها زیاد می‌شه یا ترتیب‌شون مهم نیست.

مزایا:

- خوانایی بالا
- لازم نیست ترتیب داده‌ها رعایت بشه

معایب:

- نوشتنش یه‌ذره طولانی‌تره

روش دوم: بدون نام داده‌ها (Positional)

```
arianContact := Contact{"Arian", "+989051020304", "arian@gopher.com"}
```

اینجا مقادیر رو به همون ترتیبی که تو تعریف struct اومدن نوشتیم.

مزایا:

- کوتاه‌تر و سریع‌تر

معایب:

- ترتیب خیلی مهمه!
- اگه اشتباه کنی، برنامه ممکنه کامپایل بشه ولی کدت درست کار نکنه و نتیجه اشتباهی بده (خطای منطقی)

2-مقداردهی بعد از تعریف (با انتساب | assignment)

مقداردهی به روش Zero Value

```
var nargesContact Contact
```

اینجا هیچ مقداری ندادیم، پس Go خودش به هر فیلد مقدار پیش‌فرض (zero value) می‌ده:

- string → ""
- int → 0
- bool → false
- ...

بعداً اگه خواستیم، می‌تونیم با عملگر . مقدار بدیم:

```
nargesContact.Name = "Narges"
nargesContact.Phone = "+9890510102020"
nargesContact.Email = "narges@gopher.com"
```

مزایا:

- ساده و قابل گسترش
- خوبه وقتی قراره struct رو از ورودی یا دیتابیس پر کنیم

معایب:

- اگه فراموش کنی یه فیلد رو مقداردهی کنی، ممکنه نتیجه اشتباهی بگیری چون zero value ها خودشون بی‌صدا عمل می‌کنن!

تاثیر اضافه شدن field در آینده

ما تا اینجا سه تا متغیر از نوع Contact ساختیم و به سه روش مختلف مقداردهی کردیم. حالا فرض کن درست بعد از این‌که این کارا رو کردیم، تصمیم می‌گیریم یه داده‌ی جدید هم به struct اضافه کنیم برای ذخیره تاریخ تولد هر مخاطب.

```
type Contact struct {
	Name      string
	Birthdate string
	Phone     string
	Email     string
}
```

خب بریم برنامه رو همون‌طوری که هست اجرا کنیم و ببینیم چی میشه:

```
nedaContact := Contact{
    Name:  "Neda",
    phone: "+989051010202",
    Email: "neda@gopher.com",
}

arianContact := Contact{"Arian", "+989051020304", "arian@gopher.com"}

var nargesContact Contact
nargesContact.Name = "Narges"
nargesContact.Phone = "+9890510102020"
nargesContact.Email = "narges@gopher.com"

fmt.Printf("%+v\n", nedaContact)
fmt.Printf("%+v\n", arianContact)
fmt.Printf("%+v\n", nargesContact)
```

خروجی

```
application\main.go:27:71: too few values in struct literal of type Contact
```

این خطا مربوط به مقدار دهی arianContact هست. چون اون رو با روش positional مقداردهی کرده بودیم، الان دیگه تعداد داده‌هایی که براش نوشتیم با تعداد داده‌هایی که struct انتظار داره نمی‌خونه. کامپایلر هم گیر میده.

تو روش positional باید **تک‌تک داده‌ها رو به همون ترتیبی که تو تعریف struct اومدن**، و **به همون تعداد** مقداردهی کنیم. اگه یکی‌ش رو جا بندازی یا ترتیب رو رعایت نکنی، یا کامپایلر ارور میده، یا بدتر از اون... برنامه اجرا میشه ولی نتیجه‌اش اشتباهه!

حل مشکل

برای اینکه arianContact درست مقداردهی بشه باید براش چهار تا مقدار بدیم. مثلاً:

```
arianContact := Contact{"Arian", "+989051020304", "arian@gopher.com",
"2012-18-3"}
```

الان بریم خروجی رو ببینیم:

```
{Name:Neda Birthdate: Phone:+989051010202 Email:neda@gopher.com}
{Name:Arian Birthdate:+989051020304 Phone:arian@gopher.com Email:2012-3-18}
{Name:Narges Birthdate: Phone:+9890510102020 Email:narges@gopher.com}
```

دیدی چی شد؟

به جای اینکه مقدار "1990-1-1" بره توی فیلد Birthdate، به خاطر اشتباه تو ترتیب، رفته تو فیلد Email! چون فکر کردیم Birthdate آخرین فیلده، اما تو struct دومین داده‌ست. این یعنی مقادیر جابجا شدن و داده‌هامون به هم ریختن — بدون اینکه حتی یه اخطار از کامپایلر بگیریم. چون همه‌ی داده‌ها از نوع string بودن و کامپایلر به اشتباه حساس نبود.

نکته مهم:

این دقیقاً یکی از خطرهای استفاده از روش **positional** هست. تو ظاهرش آسونه و سریع می‌نویسی، ولی اگه یه روز فیلد جدیدی به struct اضافه بشه، باید با دقت بری تمام جاهایی که به روش positional مقداردهی شدن رو پیدا و اصلاح کنی. چون یه خطای کوچیک می‌تونه باعث بشه داده‌ها اشتباهی ذخیره یا پردازش بشن.

اما تو دو روش دیگه — یعنی روش named و روش zero value — هیچ مشکلی پیش نمیاد. چون اگه برای فیلد جدید مقدار تعیین نکنی، Go خودش بهش مقدار صفر (zero value) میده و بقیه چیزا سر جاشون می‌مونن.

پس همیشه حواست باشه:

اگه قراره struct در آینده تغییر کنه (که معمولاً همین‌طوره)، روش named یا zero value انتخاب‌های امن‌تری هستن. روش positional فقط وقتی خوبه که ساختار فیکس باشه و کد سریع و ساده بخوای.

دسترسی به فیلدها

برای خوندن یا تغییر دادن مقدار هر فیلد توی struct از اپراتور . استفاده می‌کنیم:

```
fmt.Println(nargesContact.Phone)
nargesContact.Phone = "+989051010202"
fmt.Println(nargesContact.Phone)
```

یعنی خیلی راحت می‌تونی یه داده‌ی خاص از یه struct رو بگیری، یا عوضش کنی. درست مثل کار کردن با متغیرهای معمولی.

تعریف متد بر روی struct

اگه از درس قبل یادت باشه، گفتیم می‌تونیم برای type هایی که خودمون تعریف کردیم یه‌سری متد (method) هم تعریف کنیم که فقط روی اون type در دسترس هستن. این متدها معمولاً یه عملیاتی روی داده‌ای که براش type تعریف کردیم انجام می‌دن.

حالا از اونجایی که struct ها معمولاً به شکل یه type جدید تعریف می‌شن، طبیعتاً می‌تونیم چندتا متد هم براشون تعریف کنیم.

ما اومدیم Birthdate رو خیلی ساده به شکل string تعریف کردیم داخل Contact ولی فرض کن حالا بخوای سن یه مخاطب رو حساب کنی! مثلاً بخوای از طریق SMS به مخاطبایی که سن‌شون زیر 40 ساله، پیام بدی و دعوتشون کنی به یه رویداد 3 روزه‌ی کوچینگ کسب‌وکار.

خب، اینجا می‌تونیم یه متد روی نوع Contact بنویسیم که مقدار Birthdate رو بخونه و سن رو برات حساب کنه:

```
func (c Contact) Age() int {
    layout := "2006-1-2"
    birthTime, err := time.Parse(layout, strings.TrimSpace(c.Birthdate))
    if err != nil {
        return 0
    }

    today := time.Now()

    age := today.Year() - birthTime.Year()

    if today.YearDay() < birthTime.YearDay() {
        age--
    }

    return age
}
```

چطور به Birthdate دسترسی پیدا کردیم؟

وقتی یه متد روی یه struct تعریف می‌کنی، اون struct به عنوان **گیرنده (receiver)** به متد پاس داده می‌شه.
تو مثال بالا، (c Contact) یعنی یه نسخه‌ی کپی‌شده از یه Contact به اسم c رو داریم. از طریق این متغیر می‌تونیم به همه‌ی فیلدهاش دسترسی داشته باشیم، مثل c.Name یا c.Birthdate یا c.Phone و غیره.

استفاده از متد Age

حالا که این متد رو داریم، چون روی Contact تعریف شده، می‌تونیم خیلی راحت ازش برای هر مخاطبی استفاده کنیم:

```
nedaContact := Contact{
    Name:  "Neda",
    Birthdate: "2009-9-7",
    phone: "+989051010202",
    Email: "neda@gopher.com",
}

arianContact := Contact{
    "Arian", "2012-3-18",
    "+989051020304", "arian@gopher.com",
}

var nargesContact Contact
nargesContact.Name = "Narges"
nargesContact.Birthdate = "2011-6-25"
nargesContact.Phone = "+989051010202"
nargesContact.Email = "narges@gopher.com"

fmt.Printf("Neda is %d years old\n", nedaContact.Age())
fmt.Printf("Arian is %d years old\n", arianContact.Age())
fmt.Printf("Narges is %d years old\n", nargesContact.Age())
```

خروجی

```
Neda is 15 years old
Arian is 13 years old
Narges is 14 years old
```

البته یه نکته رو یادت نره: اینکه هر کسی چند سالشه بستگی داره به اینکه این برنامه رو تو چه تاریخی اجرا کنی. من خودم الان که دارم اینو تست می‌کنم، تاریخ سیستمم 2025-08-04 هست. با این تاریخ، ندا 15 سالشه، آرین 13 سالشه و نرگس هم 14 سالش درمیاد.

Struct تو در تو (Nested Structs)

قبلاً گفتیم که هر فیلدی که داخل یه struct تعریف می‌کنیم، می‌تونه از هر نوعی باشه. یعنی چی؟ یعنی هم می‌تونه از نوع‌های ساده مثل string یا int باشه، هم می‌تونه خودش یه struct باشه! این یعنی ما می‌تونیم یه struct تو در تو بسازیم. به این کار می‌گن **nest کردن structها**.

حالا چرا ممکنه بخوایم همچین کاری کنیم؟

ما قبلاً فیلد Birthdate رو تو struct مربوط به مخاطب‌هامون (Contact) به صورت یه رشته (string) ساده تعریف کردیم. ولی حالا می‌خوایم یه قدم جلوتر بریم و برای Birthdate یه type اختصاصی به اسم Date بسازیم، و بعد اون فیلد رو از string به Date تغییر بدیم.

اما چرا اصلاً باید این کارو بکنیم؟ مگه همون string مشکلی داشت؟

بیا یه نگاهی به این مثال بنداز:

```
nimaContact := Contact{
    "Nima", "2012-18-4",
    "+989052020404", "nima@gopher.com",
}

fmt.Printf("Nima is %d years old\n", nimaContact.Age())
```

ما اینجا حواسمون نبود و تاریخ تولد نیما رو اشتباهی وارد کردیم. به جای اینکه year-month-dayباشه، به صورت year-day-month نوشتیمش. وقتی این کد رو اجرا کنیم:

```
Nima is 0 years old
```

در حالی که نیما واقعاً 13 سالشه! ولی برنامه چون نتونسته تاریخ رو درست بفهمه، سن رو زده 0 سال. خطایی هم بهت نشون نمی‌ده. توی سکوت کامل یه خروجی غلط تحویلت می‌ده!

خب حالا چی کار کنیم که این اشتباه پیش نیاد؟

تعریف type جدید برای تاریخ

با ساختن یه type جدید به اسم Date، خودمون ساختار تاریخ رو مشخص می‌کنیم. اینطوری دیگه مجبور نیستیم همیشه تاریخو به صورت رشته وارد کنیم و نگران اشتباه فرمت باشیم.

```
type Date struct{
    Year uint
    Month uint
    Day uint
}
```

حالا یه تابع هم تعریف می‌کنیم به اسم NewDate که سه تا ورودی می‌گیره: سال، ماه، روز. توی این تابع چک می‌کنیم که مقدارها معتبر باشن. مثلاً ماه بین 1 تا 12 باشه، روز بین 1 تا 31، و سال بین 1900 تا 2100.

```
func NewDate(year int, month int, day int) (Date, string) {
    if day < 1 || day > 31 {
        errorMessage := "Invalid day: must be between 1 and 31."
        return Date{}, errorMessage
    }

    if month < 1 || month > 12 {
        errorMessage := "Invalid month: must be between 1 and 12."
        return Date{}, errorMessage
    }

    if year < 1900 || year > 2100 {
        errorMessage := "Invalid year: must be between 1900 and 2100."
        return Date{}, errorMessage
    }

    return Date{Year: uint(year), Month: uint(month), Day: uint(day)}, ""
}
```

اگه تاریخ درست وارد بشه، یه Date معتبر بهمون می‌ده. اگه اشتباه باشه، بهمون یه پیام خطا می‌ده تا بفهمیم چی شده.

حالا Contact رو آپدیت می‌کنیم

```
type Contact struct {
	Name      string
	Birthdate Date
	Phone     string
	Email     string
}
```

خب، تا اینجا عالیه. ولی یه مشکلی داریم.

ما قبلاً یه متد به اسم Age روی Contact تعریف کرده بودیم که سن رو حساب می‌کرد. ولی الان که Birthdate دیگه یه Date هستپس باید متد رو اصلاح کنیم،از طرف دیگه، بهتر نیست Age رو ببریم روی خود Date؟ چون اون واقعاً داره روی یه تاریخ کار می‌کنه، نه روی یه مخاطب.

متد Age روی Date

```
func (d Date) Age() int {
    layout := "2006-1-2"
    formattedBirthdate := fmt.Sprintf("%d-%d-%d", d.Year, d.Month, d.Day)
    birthTime, err := time.Parse(layout, formattedBirthdate)
    if err != nil {
        return 0
    }

    today := time.Now()

    age := today.Year() - birthTime.Year()

    if today.YearDay() < birthTime.YearDay() {
        age--
    }

    return age
}
```

تست مجدد: آیا مشکلمون حل شد؟

حالا بریم ببینیم داستان نیما چی شد:

```
nimaBirthdate, errorMessage := NewDate(2012, 18, 4)
if errorMessage != "" {
    fmt.Println("Failed to create birthdate for Nima: invalid date provided.")
    fmt.Println(errorMessage)
}

nimaContact := Contact{
    "Nima", nimaBirthdate,
    "+989052020404", "nima@gopher.com",
}

fmt.Printf("Nima is %d years old\n", nimaContact.Birthdate.Age())
```

خروجی:

```
Failed to create birthdate for Nima: invalid date provided.
Invalid month: must be between 1 and 12.
Nima is 0 years old
```

هنوز خروجی "0 سال" رو می‌ده، ولی این بار حداقل بهمون گفته مشکل چیه! و حالا می‌دونیم باید مقدار ماه رو درست کنیم. این یعنی **عیب‌یابی راحت‌تر** و **کدنویسی مطمئن‌تر**.

یه نکته ظریف!

حتی اگه تابع NewDate رو هم استفاده نکنیم، بازم این ساختار جدید خیلی بهمون کمک می‌کنه.

مثلاً:

```
nimaContact := Contact{
    "Nima",
    Date{
        Year: 2018,
        Month: 18,
        Day: 4,
    },
    "+989052020404",
    "nima@gopher.com",
}
```

الان با یه نگاه راحت می‌فهمی که مقدار ماه (Month: 18) اشتباهه!

ولی اگه همون ورژن رشته‌ای رو داشته باشی چی؟

```
nimaContact := Contact{
    "Nima", "2012-18-4",
    "+989052020404", "nima@gopher.com",
}

fmt.Printf("Nima is %d years old\n", nimaContact.Age())
```

اینجا فقط یه رشته است. اصلاً نمی‌تونی مطمئن باشی ترتیبش درسته یا نه! شاید روز و ماه قاطی شدن، شاید صفر اول ماه افتاده، شاید... خلاصه هر چیزی ممکنه.

جمع‌بندی

پس کاری که کردیم چی بود؟ اومدیم یه struct جدید به اسم Date تعریف کردیم، و اونو **nest** کردیم توی Contact با این کار:

- ساخت تاریخ مطمئن‌تر شد
- خوانایی بهتر شد
- خطاها واضح‌تر شدن
- متد Age هم رفت سر جای درست خودش

این یعنی یه قدم اساسی به سمت کد تمیزتر و قابل‌اعتمادتر

لرن پث

nest کردن struct به روش Composition

خیلی وقت‌ها تو مدل‌سازی برنامه‌مون، با مفاهیمی برخورد می‌کنیم که یه جورایی به هم چسبیدن؛ جوری که انگار یکی‌شون نسخه‌ی ساده‌تر یا خالص‌تر اون یکیه.

مثلا تو یه قسمت از برنامه یه نوع داریم به اسم Person که شامل اطلاعات پایه مثل اسم و تاریخ تولده:

```
type Person struct {
    Name      string
    Birthdate Date
}

func (p Person) GetIntroduction() string {
    return fmt.Sprintf(
        "My name is %s and I'm %d years old.",
        p.Name, p.Birthdate.Age(),
    )
}
```

اگه اینو با **Contact** مقایسه کنی، می‌بینی که بخشی از اطلاعات Contact رو همینجا داریم.
پس می‌تونیم ادعا کنیم که Contact همون Person هست ولی با ویژگی‌ها و داده‌های بیشتر؟
و اگه اینطوری بهش نگاه کنیم، چرا بیایم توی Contact دوباره Name و Birthdate بذاریم؟
نه تنها الکی تکرار می‌شه، بلکه یه عالمه کد اضافه می‌نویسیم که نگهداریش سخته.

اینجاست که **Composition** به دادمون می‌رسه.

می‌تونیم توی تعریف Contact از Person به عنوان یه داده استفاده کنیم.
این‌طوری هم می‌تونیم داده‌های پایه‌ی Person رو داشته باشیم، هم از متدهایی مثل GetIntroduction استفاده کنیم، بدون اینکه حتی یه خط کد دوباره بنویسیم.

روش تعریف

برای اینکه بتونیم Person رو به عنوان یه داده جدید به Contact اضافه کنیم جوری که Contact در واقع همون Person باشه ولی فقط یه سری ویژگی‌های اضافه داشته باشه، به این صورت عمل می‌کنیم:

```
type Contact struct {
	Person
	Phone	string
	Email	string
}
```

حالا فرض کن می‌خوایم یه نمونه از Contact درست کنیم:

```
nedaBirthdate, errorMessage := NewDate(2009, 9, 7)
if errorMessage != "" {
    fmt.Println("Failed to create birthdate for Neda: invalid date provided.")
    fmt.Println(errorMessage)
}

nedaContact := Contact{
    Person: Person{
        Name:      "Neda",
        Birthdate: nedaBirthdate,
    },
    Phone: "+989051010202",
    Email: "neda@gopher.com",
}
fmt.Printf("%s is %d years old\n", nedaContact.Name, nedaContact.Birthdate.Age())
```

یعنی با اینکه ما برای فیلد Person تو Contact اسمی نذاشتیم، موقع مقداردهی به روش Named یا Zero Value می‌تونیم با اسم Struct ای که ازش compose کردیم مقدار بدیم.

یه نکته خیلی مهم:

به این بخش توجه کن:

```
fmt.Printf("%s is %d years old\n", nedaContact.Name, nedaContact.Birthdate.Age())
```

با اینکه Name و Birthdate زیرمجموعه‌ی Person هستن، ما تونستیم مستقیماً از طریق nedaContact بهشون دسترسی داشته باشیم. چطور این اتفاق افتاد؟
بخاطر همین ویژگی composition هست. وقتی ما یه struct رو به این روش nest کنیم (یعنی اسم براش نذاریم) می‌تونیم مستقیماً به متدها و داده‌های اون struct داخلی از طریق struct بیرونی دسترسی داشته باشیم.

حتی می‌تونیم الان به متد GetIntroduction از Person هم به طور مستقیم از Contact دسترسی داشته باشیم:

```
fmt.Println(nedaContact.GetIntroduction())
```

این قدرت و سادگی Composition تو زبان Go هست که باعث می‌شه کد تمیزتر، قابل نگهداری‌تر و خلاصه‌تر بشه.

بازنگری پروژه

عملگرهای مجاز

فرض کن می‌خوایم یه محدودیت جدید به پروژه‌مون اضافه کنیم:
وقتی یه شماره جدید رو از طریق phonebook.Store ذخیره می‌کنیم، باید اول چک کنیم که اون شخص قبلاً تو سیستم ثبت نشده باشه!

یعنی نه فقط اینکه شماره تکراری ذخیره نشه، بلکه حتی اگر یه شخص با **همون اسم** و **همون سن** هم قبلاً ذخیره شده باشه، دوباره ذخیره نشه.
(احتمالاً اون‌قدر تعداد دوست‌هامون کمه که محاله دو نفر با اسم مشترک دقیقاً هم‌سن باشن!)

قبل از اینکه این قابلیت رو اضافه کنیم، یه نگاهی بندازیم به **عملگرهایی که می‌تونیم تو زبان Go روی struct ها استفاده کنیم**، تا ببینیم برای این کار چه گزینه‌هایی داریم.

جدول عملگرهای مجاز

نوع عملگر	عملگرها	توضیح
مقایسه‌ای	==, !=	مقایسه‌ی فیلد به فیلد (در صورت امکان)
انتساب	=	کپی کردن struct
دسترسی	.	دسترسی به فیلدهای struct
اشاره‌گر	&	گرفتن آدرس struct
مقداردهی به اشاره‌گر	*	دسترسی به مقادیر از طریق آدرس

با عملگرهای **انتساب** و **دسترسی به فیلد** قبلاً آشنا شدیم.
عملگرهای **اشاره‌گر** و **دسترسی به مقدار اشاره‌گر** رو تو درس «اشاره‌گرها» بررسی خواهیم کرد.

اینجا اما می‌خوایم تمرکز کنیم روی **عملگرهای مقایسه‌ای**، چون لازمه برای جلوگیری از ذخیره‌سازی تکراری، structها رو مقایسه کنیم.

بررسی عملگر ==

این عملگر فقط زمانی true برمی‌گردونه که:

- هر دو struct از **یک نوع** باشن.
- تمام فیلدهاشون به‌صورت **نظیر به نظیر برابر** باشن.

به مثال زیر توجه کن

phonebook/application/main.go

```
var arianJahani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
}

var arianZamani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
}

fmt.Println(arianJahani == arianZamani)
```

خروجی

```
true
```

چرا؟ چون:

- هر دو از نوع person.Person هستن.
- مقادیر تمام فیلدهاشون یکیه (حتی فیلد تو در توی Birthdate)

لرن پث

حالا فرض کن یه فیلد Friends []string به struct اضافه کنیم:

phonebook/person/person.go

```
type Person struct {
	Name      string
	Birthdate date.Date
	Friends []string
}
```

phonebook/application/main.go

```
var arianJahani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
    Friends: []string{"Milad", "Bahram", "Saeed"},
}

var arianZamani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
    Friends: []string{"Milad", "Bahram", "Saeed"},
}

fmt.Println(arianJahani == arianZamani)
```

خروجی

```
invalid operation: arianJahani == arianZamani (struct containing []string cannot
be compared)
```

چرا این اتفاق افتاد؟ چون:

- نمی‌تونیم از == برای مقایسه‌ی string[] (یا هر نوع slice) استفاده کنیم.
- از اونجایی که مقایسه‌ی structها به‌صورت **فیلد به فیلد** انجام میشه، و یکی از فیلدها قابل مقایسه نیست، پس کل struct هم قابل مقایسه نیست.
- **کامپایلر با دادن خطا از این اشتباه جلوگیری می‌کنه.**

نکته کلی:

اگه یه struct شامل یکی از این نوع داده‌ها باشه:

- slice
- map
- function
- channel

هیچ‌جوره نمی‌تونی اون struct رو مستقیماً با == یا =! مقایسه کنی.
و اگه این کارو بکنی، تو **زمان کامپایل** با خطا مواجه می‌شی، نه در زمان اجرا!

حالا اگه بخوایم واقعاً این دو struct رو با هم مقایسه کنیم، باید چی کار کنیم؟

باید یه تابع دستی تعریف کنیم که مقایسه رو خودش انجام بده:

phonebook/application/main.go

```
func ArePersonsEqual(p1, p2 Person) bool {
    if p1.Name != p2.Name {
        return false
    }

    if p1.Birthdate != p2.Birthdate {
        return false
    }

    if len(p1.Friends) != len(p2.Friends) {
        return false
    }

    for i := range p1.Friends {
        if p1.Friends[i] != p2.Friends[i] {
            return false
        }
    }

    return true
}
```

و حالا این‌طوری ازش استفاده می‌کنیم:

phonebook/application/main.go

```
fmt.Println(ArePersonsEqual(arianJahani, arianZamani))
```

خروجی

```
true
```

لرن پث

بررسی عملگر =!

عملگر =! دقیقاً مثل == کار می‌کنه با این تفاوت که وقتی حداقل یکی از فیلدهای دو struct برابر نباشه، نتیجه‌ی مقایسه true میشه. و اگر تمام فیلدها باهم برابر باشن، میشه false.

به مثال زیر دقت کن:

phonebook/person/person.go

```
type Person struct {
	Name      string
	Birthdate date.Date
}
```

phonebook/application/main.go

```
var arianJahani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
}

var arianZamani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
}

fmt.Println(arianJahani != arianZamani)
```

خروجی

```
false
```

دلیلش واضحه: چون هر دو struct از یک نوع هستن و تمام فیلدهاشون مقدار یکسانی دارن، بنابراین =! مقدار false برمی‌گردونه.

حالا برگردیم سراغ اون محدودیتی که می‌خواستیم به phonebook.Store اضافه کنیم

می‌خواستیم بررسی کنیم که آیا یه Person خاص قبلاً تو مخزن ذخیره شده یا نه.
برای این کار یه تابع کمکی می‌نویسیم به اسم isPersonInRepo که کل repo رو بگرده و در صورت وجود اون Person، مقدار true برگردونه.

phonebook/phone_book/store.go

```
func isPersonInRepo(p person.Person) bool {
    for _, c := range repo {
        if c.Person == p {
            return true
        }
    }

    return false
}
```

حالا وقتشه از این تابع توی Store استفاده کنیم

phonebook/phone_book/store.go

```
func Store(c contact.Contact) bool {
    if !validator.IsNameValid(c.Name) {
        return false
    }

    if !validator.IsPhoneValid(c.Phone) {
        return false
    }

    allPhoneNumber := GetAllPhoneNumbers()

    if validator.IsPhoneAlreadyAdded(allPhoneNumber, c.Phone) {
        return false
    }

    if isPersonInRepo(c.Person) {
        return false
    }

    addToRepo(c)
    return true
}
```

تست کنیم ببینیم همه‌چی درست کار می‌کنه یا نه!

phonebook/application/main.go

```
arianBirthdate, errorMessage := date.New(2012, 3, 18)

if errorMessage != "" {

    fmt.Println("Failed to create birthdate for Arian: invalid date provided.")

    fmt.Println(errorMessage)

}

arianContact := contact.New("Arian", arianBirthdate, "+989051020304",
"arian@gopher.com")

phonebook.Store(arianContact)


arianJahaniBirthdate, errorMessage := date.New(2012, 3, 18)

if errorMessage != "" {

    fmt.Println("Failed to create birthdate for Arian: invalid date provided.")

    fmt.Println(errorMessage)

}

arianJahaniContact := contact.New("Arian", arianJahaniBirthdate, "+989051080304",
"arian@gopher.com")

phonebook.Store(arianJahaniContact)
```

خروجی

```
┌────────────────┬────────────────┐
│      NAME      │ PHONE NUMBER   │
├────────────────┼────────────────┤
│   Emergency    │      115       │
│     Police     │      110       │
│ FireDepartment │      125       │
│      Neda      │ +989051010202  │
│     Arian      │ +989051020304  │
│      Nima      │ +989052020404  │
└────────────────┴────────────────┘
```

همون‌طور که می‌بینی، arianJahaniContact با اینکه شماره‌ش فرق داشت، چون **اطلاعات شخصی‌اش** با یه شخص قبلاً ذخیره‌شده برابر بود، تو لیست نهایی اضافه نشد

بررسی مدل ارسال Struct به توابع

در زبان Go ، **structها به‌صورت پیش‌فرض به شکل call by value** به توابع ارسال می‌شن.
یعنی وقتی یه struct رو به عنوان ورودی به تابع می‌فرستیم، **یه کپی از اون struct ساخته می‌شه** و داخل تابع، ما داریم با اون کپی کار می‌کنیم.

هر تغییری که توی تابع روی struct ورودی انجام بدیم، **خارج از تابع تأثیری نداره**.

به مثال زیر توجه کن:

phonebook/application/main.go

```
func MarkAsDuplicated(p person.Person) {
    p.Name = p.Name + " (Duplicated)"
}
```

این تابع یه کار ساده انجام می‌ده: پسوند (Duplicated) رو به فیلد Name اضافه می‌کنه.

حالا بیایم تستش کنیم:

phonebook/application/main.go

```
var arianZamani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
}

MarkAsDuplicated(arianZamani)

fmt.Printf("%+v\n", arianZamani)
```

خروجی

```
{Name:Arian Birthdate:{Year:2012 Month:3 Day:18}}
```

اما چرا اسم همچنان Arian مونده و پسوند (Duplicated) اضافه نشده؟

چون همون‌طور که گفتیم، arianZamani از نوع struct هست و **structها به صورت call by value** به توابع ارسال می‌شن.

پس داخل تابع MarkAsDuplicated ما فقط با یه **کپی** از arianZamani کار کردیم. در نتیجه، تغییراتی که توی تابع انجام دادیم، **بیرون از تابع قابل مشاهده نیستن.**

راه‌حل: بازگردوندن struct اصلاح‌شده

تو چنین شرایطی، بهتره بعد از انجام تغییرات روی struct ، **همون struct اصلاح‌شده رو برگردونیم** و کسی که تابع رو صدا زده، بیاد مقدار جدید رو جایگزین مقدار قبلی کنه:

phonebook/application/main.go

```
func MarkAsDuplicated(p person.Person) person.Person{
    p.Name = p.Name + " (Duplicated)"

    return p
}
```

phonebook/application/main.go

```
var arianZamani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
}

arianZamani = MarkAsDuplicated(arianZamani)

fmt.Printf("%+v\n", arianZamani)
```

خروجی

```
{Name:Arian (Duplicated) Birthdate:{Year:2012 Month:3 Day:18}}
```

حالا یه سؤال مهم: اگه struct یه فیلدی مثل slice یا map داشت، بازم همین رفتار رو داره؟

بله، همچنان struct به صورت **call by value** به تابع فرستاده می‌شه.

اما اینجا یه نکته مهم وجود داره:

وقتی struct رو کپی می‌کنیم، فیلدهایی مثل slice یا map که خودشون **referential type** هستن (یعنی اشاره‌گر پشت‌صحنه دارن)، **کپی‌شون فقط یه کپی از خود pointer هست، نه داده‌ای که بهش اشاره می‌کنه!**

یعنی چی؟ یعنی هم نسخه اصلی، هم کپی، **به یه آرایه‌ی مشترک اشاره می‌کنن**.
پس هر تغییری روی اون slice داخل تابع انجام بدی، **روی نسخه اصلی هم تأثیر می‌ذاره.**

مثال با فیلد slice

فرض کن دوباره فیلد Friends رو به Person اضافه کردیم.
حالا یه تابع می‌نویسیم که مقادیر داخل Friends رو trim کنه، یعنی فاصله‌های اضافه اول و آخرشون رو حذف کنه:

phonebook/application/main.go

```
func trimFriends(p person.Person) {
    for i, f := range p.Friends {
        p.Friends[i] = strings.TrimSpace(f)
    }
}
```

phonebook/application/main.go

```
var arianZamani = person.Person{
    Name: "Arian",
    Birthdate: date.Date{
        Year:  2012,
        Month: 3,
        Day:   18,
    },
    Friends: []string{"Milad  ", "   Bahram", "Saeed"},
}

trimFriends(arianZamani)
```

خروجی

```
{Name:Arian Birthdate:{Year:2012 Month:3 Day:18} Friends:[Milad Bahram Saeed]}
```

با اینکه arianZamani به صورت call by value به تابع trimFriends ارسال شده،
اما چون فیلد Friends از نوع slice هست، تغییراتی که روی عناصرش انجام دادیم، روی **نسخه اصلی struct هم اعمال شده**.

نکته مهم

structها همیشه به صورت **کپی** (call by value) منتقل می‌شن.

اما اگه داخلشون فیلدهایی باشن مثل slice، map یا pointer، ممکنه تغییر روی این فیلدها، **مستقیم نسخه‌ی اصلی رو تحت تأثیر بذاره**. پس همیشه موقع کار با این نوع داده‌ها، باید بدونی دقیقاً داری چی رو تغییر می‌دی.

استفاده‌ی کاربردی از متد MarkAsDuplicated در phonebook

تا اینجا دیدیم که چطوری می‌تونیم یک مقدار از نوع Person رو به عنوان کپی به تابع بدیم و با یه تغییر کوچیک، یه نسخه‌ی جدید ازش بسازیم. حالا وقتشه از این قابلیت به‌صورت واقعی و کاربردی توی پروژه‌ی خودمون استفاده کنیم.

یادته یه محدودیت اضافه کرده بودیم که اگر یه Person تکراری بود، دیگه اجازه نده Contact جدید با اون Person ذخیره بشه؟

خب، **نظرمون عوض شد.**

حالا می‌خوایم **اجازه بدیم که اون Contact ذخیره بشه**، ولی با یه تفاوت کوچیک:

اگر Person تکراری باشه، پسوند (Duplicated) به اسمش اضافه کنیم تا بعداً متوجه بشیم که این دو نفر یکی هستن که با شماره‌های مختلف ثبت شدن.

برای این کار، بهتره تابع MarkAsDuplicated رو به عنوان یه **متد روی نوع Person** تعریف کنیم؛ چون داره روی خود Person عملیات انجام می‌ده.

phonebook/person/person.go

```
func (p Person) MarkAsDuplicated() Person {
    p.Name = p.Name + " (Duplicated)"

    return p
}
```

بروزرسانی تابع Store

حالا وقتشه تابع Store رو به‌روزرسانی کنیم تا از این متد جدید استفاده کنه:

phonebook/phone_book/store.go

```
func Store(c contact.Contact) bool {
    if !validator.IsNameValid(c.Name) {
        return false
    }

    if !validator.IsPhoneValid(c.Phone) {
        return false
    }

    allPhoneNumber := GetAllPhoneNumbers()

    if validator.IsPhoneAlreadyAdded(allPhoneNumber, c.Phone) {
        return false
    }

    if isPersonInRepo(c.Person) {
        c.Person = c.Person.MarkAsDuplicated()
    }

    addToRepo(c)
    return true
}
```

تا اینجای کار، تابع Store هنوز هم همه‌ی بررسی‌های اولیه رو انجام می‌ده، اما به‌جای اینکه جلوی ثبت افراد تکراری رو بگیره، **اسمشون رو تغییر می‌ده** و بعد ذخیره می‌کنه

تست نهایی

phonebook/application/main.go

```
arianBirthdate, errorMessage := date.New(2012, 3, 18)

if errorMessage != "" {

    fmt.Println("Failed to create birthdate for Arian: invalid date provided.")

    fmt.Println(errorMessage)

}

arianContact := contact.New("Arian", arianBirthdate, "+989051020304",
"arian@gopher.com")

phonebook.Store(arianContact)


arianJahaniBirthdate, errorMessage := date.New(2012, 3, 18)

if errorMessage != "" {

    fmt.Println("Failed to create birthdate for Arian: invalid date provided.")

    fmt.Println(errorMessage)

}

arianJahaniContact := contact.New("Arian", arianJahaniBirthdate, "+989051080304",
"arian@gopher.com")

phonebook.Store(arianJahaniContact)
```

خروجی

```
┌──────────────────────┬──────────────────┐
│         NAME         │   PHONE NUMBER   │
├──────────────────────┼──────────────────┤
│      Emergency       │       115        │
│       Police         │       110        │
│    FireDepartment    │       125        │
│        Neda          │  +989051010202   │
│       Arian          │  +989051020304   │
│ Arian (Duplicated)   │  +989051080304   │
│        Nima          │  +989052020404   │
└──────────────────────┴──────────────────┘
```

همه‌چیز درست کار می‌کنه.

Arian با دو شماره مختلف ثبت شده، ولی چون Person شون یکی بوده، برای دومی به‌صورت خودکار (Duplicated) به اسمش اضافه شده.

visibility در struct ها

همونطور که قبلاً یاد گرفتیم، تو زبان Go می‌تونیم با بزرگ یا کوچیک نوشتن حرف اول یک شناسه (مثل متغیر، تابع یا ثابت) **محدوده‌ی دسترسی یا همون visibility** اون رو مشخص کنیم.

اگه حرف اول **بزرگ باشه**، اون موجودیت **قابل دسترسی از بیرون پکیجه** (exported)، و اگه **کوچیک باشه**، فقط **داخل همون پکیج قابل استفادست** .(unexported)

خب، این قانون برای **structها هم صادقه** .هم خود struct می‌تونه exported یا unexported باشه، هم **هر کدوم از فیلدهای داخل struct** می‌تونن visibility جداگانه‌ای داشته باشن.

لرن پث

یه مثال از پکیج date

یه نگاهی به این تعریف struct توی پکیج date بنداز:

phonebook/date/date.go

```
package date


type Date struct {
    Year  uint
    Month uint
    Day   uint
}
```

فیلدهای Year، Month و Day چون با حرف بزرگ شروع شدن، از بیرون پکیج date هم قابل دسترسی هستن.

حالا بیایم فیلدها رو unexported کنیم

تصور کن فیلدهای struct رو اینجوری تعریف کنیم:

phonebook/date/date.go

```
package date


type Date struct {
    year  uint
    month uint
    day   uint
}
```

و خط آخر تابع New رو هم به‌روزرسانی کنیم:

phonebook/date/date.go

```
return Date{year: uint(year), month: uint(month), day: uint(day)}, ""
```

خب، بریم برنامه رو اجرا کنیم و ببینیم چی می‌شه...

```
utils\date.go:11:50: d.Year undefined (type date.Date has no field or method Year,
but does have unexported field year)

utils\date.go:11:58: d.Month undefined (type date.Date has no field or method
Month, but does have unexported field month)

utils\date.go:11:67: d.Day undefined (type date.Date has no field or method Day,
but does have unexported field day)
```

کامپایلر داره بهمون می‌گه که این فیلدها وجود ندارن چون اسمشونو عوض کردیم. یعنی شما می‌خوای به d.Year دسترسی پیدا کنی در صورتی که اسمش به year تغییر کرده و با این اسم جدید باید استفاده کنی.

بیایم جایی که باعث خطا شده رو اصلاح کنیم

تو فایل utils/date.go که خارج از پکیج date هست، قبلاً این خط رو داشتیم:

```
formattedBirthdate := fmt.Sprintf("%d-%d-%d", d.Year, d.Month, d.Day)
```

حالا اومدیم این رو کردیم

```
formattedBirthdate := fmt.Sprintf("%d-%d-%d", d.year, d.month, d.day)
```

دوباره اجرا کردیم...

خروجی

```
utils\date.go:11:50: d.year undefined (cannot refer to unexported field year)
utils\date.go:11:58: d.month undefined (cannot refer to unexported field month)
utils\date.go:11:67: d.day undefined (cannot refer to unexported field day)
```

این بار خطای کامپایل فرق کرده:

می‌گه "نمی‌تونی به یه فیلد unexported از بیرون پکیج دسترسی داشته باشی"

راه‌حل: تعریف getter و setter

وقتی فیلدهای struct رو private (یا همون unexported) می‌کنیم، برای اینکه بتونیم تو پکیج‌های دیگه ازشون استفاده کنیم، باید راه دسترسی کنترل‌شده‌ای براشون تعریف کنیم.

و اینجاست که **getter** و **setter** وارد ماجرا می‌شن.

- **Getter** فقط مقدار فیلد رو برمی‌گردونه.
- **Setter** به ما اجازه می‌ده مقدار فیلد رو تغییر بدیم.

مثلاً برای struct Date با 3 فیلد، می‌تونیم این 3 تا getter رو تعریف کنیم:

phonebook/date/date.go

```
func (d Date) GetYear() uint {
	return d.year
}

func (d Date) GetMonth() uint {
	return d.month
}

func (d Date) GetDay() uint {
	return d.day
}
```

چون این متدها خودشون داخل پکیج date تعریف شدن، اجازه دارن به فیلدهای private (year, month, day) دسترسی پیدا کنن و مقادیرشون رو برگردونن.

استفاده از getter در بیرون از پکیج

حالا میتونیم تو تابع Age به جای دسترسی مستقیم از این توابع getter برای دسترسی به فیلد های Date استفاده کنیم.

phonebook/date/date.go

```
formattedBirthdate := fmt.Sprintf("%d-%d-%d", d.GetYear(), d.GetMonth(),
d.GetDay())
```

اگه دوباره برنامه رو اجرا کنیم، بدون هیچ مشکلی کامپایل می‌شه و اجرا هم درست انجام می‌شه

جمع‌بندی:

می‌تونیم struct رو exported یا unexported تعریف کنیم.

- برای فیلدهای struct هم همین قانون صادقه.
- فیلدهایی که با حرف کوچیک شروع بشن، از بیرون پکیج قابل دسترسی نیستن.
- برای کنترل بهتر دسترسی به فیلدها، بهتره اون‌ها رو private تعریف کنیم و به‌جاش توابع getter و setter بنویسیم.
- اینطوری اگه بعداً اسم فیلدها رو تغییر دادیم، فقط کافیه تو همون getter و setter اصلاح کنیم، نه همه جای پروژه.

نتیجه؟ **کنترل بیشتر، کد تمیزتر، دردسر کمتر توی آینده.**

تمرین 1: مقایسه Struct

یک struct به نام **Book** با فیلدهای Title و Author بساز.
دو نمونه از این struct ایجاد کن و با استفاده از عملگر == بررسی کن که آیا کاملاً برابر هستند یا نه.

تمرین 2: مقایسه Struct با Slice

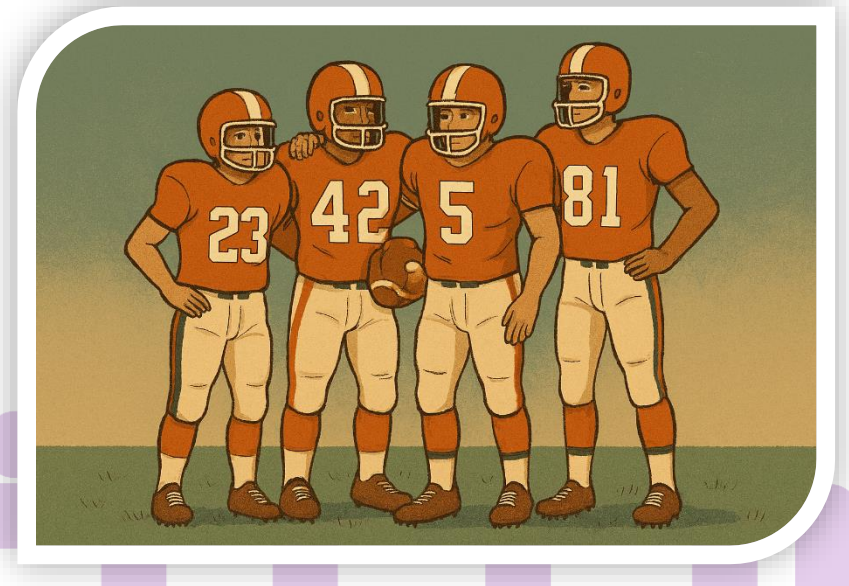
یک struct به نام **Playlist** با فیلدهای Name (string) و Songs ([]string) بساز.
دو Playlist با داده‌های یکسان ایجاد کن و سعی کن با == مقایسه کنی. ببین چه خطایی می‌گیری و بعد یک تابع برای مقایسه‌شون بنویس.

تمرین 3: مدل ارسال Struct به عنوان پارامتر

یک struct به نام **Contact** با فیلد Name و Phone بساز.

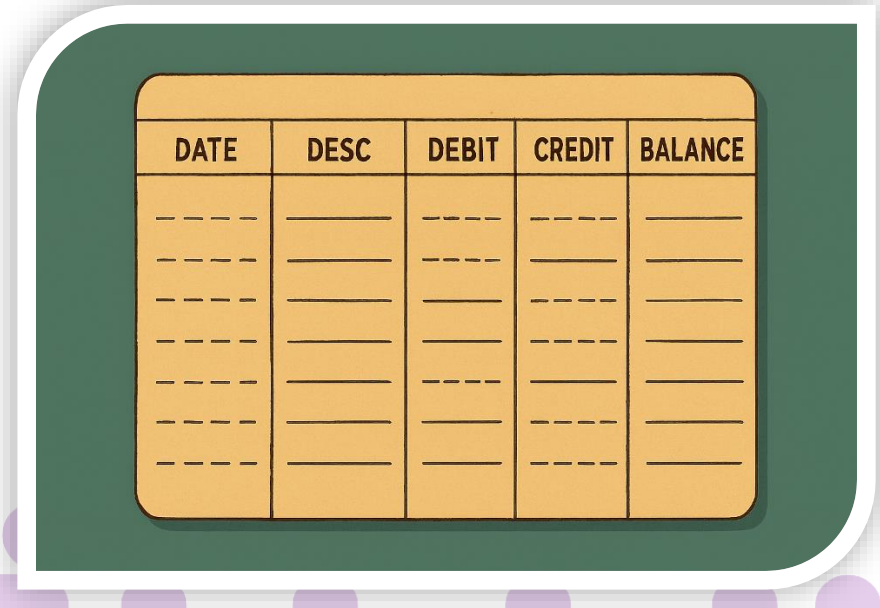
یک متد MaskPhone() براش بنویس که وسط شماره تلفن رو با * جایگزین کنه، اما چون struct به صورت پیش‌فرض call by value پاس داده میشه، این تغییر رو در بیرون از متد اعمال کن.

تمرین 4: Slice درون Struct



یک struct به نام **Team** با فیلدهای Name و Members ([]string) بساز.
یک تابع TrimMembers() بنویس که فاصله اضافی اول و آخر نام اعضا رو حذف کنه و ببین چرا با وجود پاس دادن struct به صورت value، تغییرات روی نسخه اصلی هم اعمال میشه.

تمرین 5: Visibility و Getter



یک struct به نام **Account** در یک پکیج جدا با فیلدهای private username و balance بساز. برای این struct متدهای Getter تعریف کن که این مقادیر رو برگردونه، و سپس در یک پکیج دیگه مقادیر رو نمایش بده.

تمرین 6: نقش عملگر = در Struct

یک struct به نام **Profile** با فیلدهای Name و Age بساز.
یک نمونه از این struct ایجاد کن (p1) و سپس با استفاده از عملگر = یک کپی از آن بساز.
سپس مقدار Name را در p2 تغییر بده و چاپ کن که آیا مقدار p1 تغییر کرده یا نه.

سؤال:

1. چرا وقتی مقدار p2تغییر می‌کنه، p1 تغییر نمی‌کنه؟
2. اگر یکی از فیلدهای struct از نوع **slice** باشه، آیا نتیجه متفاوت خواهد بود؟