مفهوم Type

تا الان یکی‌دو بار برامون پیش اومده که برنامه‌هایی نوشتیم که کمی پیچیده بودن. پیچیدگی بیشتر یعنی سخت‌تر شدن درک و تغییر کدهای برنامه در آینده. زبان‌های برنامه‌نویسی باید تمام تلاششونو بکنن تا کدها جوری نوشته بشن که تو آینده هم خوندنشون راحت باشه، هم تغییر دادنشون. وگرنه برای فهمیدن یه تیکه کد ساده باید کلی کامنت و توضیح اضافه کنیم!

یکی از مهم‌ترین قابلیت‌هایی که Go بهمون داده، تعریف type هست. باهاش می‌تونیم برای داده‌هایی که مدام تو برنامه تکرار می‌شن، یه **معنای مشخص و قابل فهم** بسازیم.

احتمالاً یادت باشه که تو یکی از تمریانت آخر درسنامه «تابع» ازت خواستم همه حالت‌های تبدیل دما رو بنویسی.

توابع مستقیم تبدیل:

- از سلسیوس به فارنهایت
- از سلسیوس به کلوین
- از فارنهایت به سلسیوس
- از کلوین به سلسیوس

توابع ضمنی تبدیل:

- از فارنهایت به کلوین
- از کلوین به فارنهایت

حالا اگه یه نگاهی به این توابع داشته باشیم، بهتر می‌فهمیم منظورمون از **نبود معنا** چیه:

```
func celsiusToFahrenheit(c float64) float64 {
    return (c * 1.8) + 32
}

func celsiusToKelvin(c float64) float64 {
    return c + 273.15
}

func fahrenheitToCelsius(f float64) float64 {
    return 5/9 * (f - 32)
}

func kelvinToCelsius(k float32) float64 {
    return k – 273.15
}

func fahrenheitToKelvin(f float64) float64 {
    return celsiusToKelvin(fahrenheitToCelsius(f))
}

func kelvinToFahrenheit(k float64) float64 {
    return celsiusToFahrenheit(kelvinToCelsius(k))
}
```

حالا بیا یه لحظه خودتو بذار جای کسی که برای اولین بار این کدها رو می‌بینه. یا حتی خودت، وقتی چند هفته بعد برگردی سراغش. همه ورودی‌ها و خروجی‌ها float64 هستن! یعنی هیچ سرنخی از اینکه این عددها چه واحدی دارن، نداریم. فقط می‌دونیم عددن.

مثلا

```
fmt.Println(kelvinToFahrenheit(37.5))
```

از کجا معلوم این 37.5 کلوینه؟ نکنه اشتباهی مقدار سلسیوس دادی؟
همه چی خیلی خام و بی‌معناست و تو پروژه‌های واقعی، اینجور چیزا خیلی راحت باعث اشتباهات منطقی عجیب‌غریب می‌شن.

راه‌حل: تعریف type سفارشی

زبان Go بهمون اجازه می‌ده که برای داده‌ها، یه **اسم جدید با معنا** بسازیم.
کافیه طبق فرمول زیر جلو بریم:

```
type name TYPE
```

type کیوردیه که باهاش یه نوع جدید تعریف می‌کنیم. همیشه همینه.

Name اسم نوع جدیده که خودمون می‌ذاریم. این اسم باید توی سطح پکیج **منحصربه‌فرد** باشه (یعنی اسم متغیر یا تابع یا type دیگه نباشه).

Type نوع زیرساختیه که نوع جدیدمون از روش ساخته می‌شه. می‌تونه یه نوع پیش‌فرض Go باشه (مثل int, float64, []string) یا حتی یه نوعی که خودمون قبلاً تعریف کردیم.

مثلا:

```
type Celsius float64
type Fahrenheit float64
type Kelvin float64
```

با این کار، دیگه هر عددی که داریم فقط یه عدد خشک نیست. بلکه معلومه که اون عدد نماینده‌ی چه واحدیه.

حالا اگه بخوایم توابع قبلی رو بازنویسی کنیم، اینطوری می‌شن:

```
func CelsiusToFahrenheit(c Celsius) Fahrenheit {
    return Fahrenheit(float64(c)*1.8 + 32)
}

func CelsiusToKelvin(c Celsius) Kelvin {
    return Kelvin(float64(c) + 273.15)
}

func FahrenheitToCelsius(f Fahrenheit) Celsius {
    return Celsius((float64(f) - 32) * 5.0 / 9.0)
}

func KelvinToCelsius(k Kelvin) Celsius {
    return Celsius(float64(k) - 273.15)
}

func FahrenheitToKelvin(f Fahrenheit) Kelvin {
    return CelsiusToKelvin(FahrenheitToCelsius(f))
}

func KelvinToFahrenheit(k Kelvin) Fahrenheit {
    return CelsiusToFahrenheit(KelvinToCelsius(k))
}
```

حالا اگه کسی این کدها رو ببینه، دقیقاً می‌فهمه ورودی‌ها و خروجی‌ها چی‌ان.
یعنی دیگه نمی‌گی «یه عدد فلوت بده»، بلکه می‌گی: «یه دمای کلوین بده» — خیلی خواناتر و امن‌تره.

فراخوانی توابع چجوری میشه؟

شاید فکر کنی مثل قبل می‌تونی یه float64 پاس بدی و تموم. مثلاً:

```
var temp float64 = 37.5

fmt.Println(KelvinToFahrenheit(temp))
```

این کد **کامپایل نمی‌شه** و خطای زیر رو می‌گیری. چون ورودی تابع Kelvin هست، نه float64!

```
cannot use temp (variable of type float64) as Kelvin value in argument to
KelvinToFahrenheit
```

پس چاره‌ش چیه؟ باید مقدار float64 رو **صریحاً به Kelvin تبدیل کنیم**:

```
var temp float64 = 37.5

fmt.Println(KelvinToFahrenheit(Kelvin(temp)))
```

عبارت Kelvin(temp) در واقع یه **type casting** انجام می‌ده.

یعنی می‌گه: این عدد float64 رو به نوع Kelvin تبدیل کن. چون Kelvin بر پایه float64 ساخته شده، این تبدیل مشکلی نداره.

حتی می‌تونی مستقیم بنویسی:

```
fmt.Println(KelvinToFahrenheit(37.5))
```

نکته طلایی

اگه بنویسی:

```
fmt.Println(KelvinToFahrenheit(37.5))
```

ممکنه تعجب کنی که چرا **بدون خطا اجرا می‌شه!**

دلیلش اینه که **ثابت‌های عددی مثل 37.5 تو زبان Go در واقع untyped هستن**.

یعنی تا زمانی که نوع نگرفتن، می‌تونن **به‌صورت خودکار** به انواع عددی سازگار (مثل float64 یا Kelvin) تبدیل بشن. اما این فقط در مورد **ثابت‌ها** (literal)ها صدق می‌کنه. متغیرهای float64 معمولی این اجازه رو ندارن و باید دستی به نوع موردنظر cast بشن.

نتیجه‌گیری:

با تعریف type های سفارشی مثل Celsius, Fahrenheit, Kelvin، نه‌تنها کدها معنای بیشتری پیدا می‌کنن، بلکه:

- احتمال اشتباه رو پایین می‌ارن
- فهم و نگهداری کد رو آسون‌تر می‌کنن
- باعث می‌شن Go ازت بخواد همیشه دقیق باشی و ورودی درست بدی

یعنی به‌جای اینکه کدت یه سری محاسبه بی‌معنا باشه، می‌تونه واقعاً بازتابی از دنیای واقعی باشه — دقیق، امن، و قابل فهم.

اسم مستعار برای Type (Type Alias)

گاهی وقتا ممکنه بخوایم برای یه نوع از قبل تعریف‌شده، یه **اسم جدید** بذاریم که خوندنش راحت‌تر یا معنی‌دارتر باشه. اینجاست که می‌تونیم از **اسم مستعار یا همون type alias** استفاده کنیم.

چطوری تعریف می‌کنیم؟

برای تعریف اسم مستعار، از این ساختار استفاده می‌کنیم:

```
type NewName = ExistingType
```

type همون کیورد همیشگی برای تعریف نوعه.

NewName اسم جدیده که خودمون انتخابش می‌کنیم. این اسم باید تو سطح پکیج **منحصربه‌فرد** باشه (یعنی تکراری نباشه).

= تفاوت اصلی تعریف alias با تعریف نوع جدید، **همین علامت مساویه**.

ExistingType نوعی هست که قبلاً وجود داشته (مثل int, float64 یا حتی نوع‌هایی که خودمون تعریف کردیم).

یه مثال معروف byte:

اگه یه نگاه به مستندات Go بندازی، می‌بینی که byte در واقع همونه uint8ـه:

```
// byte is an alias for uint8 and is equivalent to uint8 in all ways. It is
// used, by convention, to distinguish byte values from 8-bit unsigned
// integer values.
type byte = uint8
```

یعنی byte فقط یه **اسم دیگه برای uint8** حساب می‌شه، بدون اینکه هیچ تفاوت فنی‌ای داشته باشه.

و خب چرا این کار رو کردن؟

چون وقتی کسی اینو ببینه:

```
var character byte = 65
```

ذهنش می‌ره سمت **داده‌ی خام**، یا **کاراکترASCII** ، یا مثلاً یه تیکه از یه فایل یا پیام شبکه.

ولی وقتی می‌نویسی:

```
var character uint8 = 65
```

بیشتر حس می‌ده که با **عدد صحیح بدون علامت** سر و کار داری، نه الزاماً داده‌ی باینری یا کاراکتر.

چرا باید از اسم مستعار استفاده کنیم؟

1. **ساده‌تر کردن و قابل فهم‌تر کردن کد**

وقتی یه نوع طولانی یا سخت‌فهم داری، یه اسم جدید خوش‌معنی می‌تونه کدت رو قشنگ‌تر کنه

2. **هماهنگی با مفاهیم دنیای واقعی**

مثلاً byte برای همه‌ی برنامه‌نویس‌ها قابل درکه. کسی که ببینه byte فوراً می‌فهمه منظور چیه، ولی uint8 ممکنه یه لحظه ذهنو منحرف کنه.

3. **سازگاری با کدها یا کتابخونه‌های قدیمی**

گاهی لازمه برای هماهنگ شدن با کدهای قدیمی یه اسم مستعار تعریف کنی، بدون اینکه کل ساختار برنامه رو به هم بزنی

4. **فرار از اسم‌گذاری تکراری**

فرض کن یه پکیج خارجی یه type تعریف کرده به اسم User، تو هم می‌خوای از همون استفاده کنی ولی نمی‌خوای باهاش تداخل پیدا کنی:

تعریف اسم مستعار برای Celsius

حتماً شنیدی که به دمای **Celsius** می‌گن **سانتی‌گراد** (Centigrade) در واقع اینا **دو اسم برای یه چیز واحد** هستن.

- Celsius: اسم رسمی استاندارد دما در سیستم SI
- Centigrade: اسم قدیمی‌تر ولی هم‌معنی

اگه بخوای یه اسم مستعار برای Celsius تعریف کنی که به‌جای اون از Centigrade استفاده بشه، کافیه اینطوری بنویسی:

```
type Centigrade = Celsius
```

حالا می‌تونی خیلی راحت بنویسی:

```
var freezingPointOfBlood Centigrade = -0.56
freezingPointOfBloodInFahrenheit := CelsiusToFahrenheit(freezingPointOfBlood)
fmt.Println(
    “The freezing point of human blood is: ”,
    freezingPointOfBloodInFahrenheit,
)
```

چون Centigrade فقط یه alias برای Celsiusـه، کاملاً قابل جایگزینی تو همه جاهایی هست که Celsius انتظار می‌ره.

مقایسه Type Definition با Type Alias

ویژگی	Type Definition (تعریف نوع جدید)	Type Alias (نام مستعار نوع موجود)
نحوه تعریف	type Celsius float64	type Celsius = float64
ماهیت	تعریف یک **نوع جدید و مستقل**	فقط یک **اسم جدید** برای یک نوع موجود
تفاوت در کامپایل	نوع جدید **متفاوت** از نوع زیرساخت است	دقیقاً **همان نوع** است
نیاز به تبدیل(Cast)	بله، باید به‌صراحت تبدیل شود	خیر، تبدیل لازم نیست
قابلیت تعریف متد	بله، می‌توان متد اختصاصی تعریف کرد	بله، ولی روی نوع زیرساختی تعریف می‌شود
استفاده در ایمنی نوع(Type Safety)	بالا — جلوگیری از اشتباهات منطقی	نه — مثل نوع اصلی رفتار می‌کند
کاربرد معمول	ساختاردهی بهتر کد، مستندسازی، محدودسازی	خوانایی بیشتر، سازگاری با کدهای دیگر

ثابت با نوع سفارشی

همون‌طور که می‌تونیم یه متغیر از نوع سفارشی داشته باشیم، می‌تونیم **ثابت** هم از نوع سفارشی تعریف کنیم.

فرض کن می‌خوای نقطه‌ی انجماد و جوش آب، الکل و خون رو تو برنامه داشته باشی. چون این مقادیر ممکنه زیاد استفاده بشن، بهترین کار اینه که به صورت **ثابت** تعریفشون کنیم:

```
const FreezingPointOfWater Celsius = 0.0
const BoilingPointOfWater Celsius = 100.0

const FreezingPointOfAlcohol Celsius = -114.1
const BoilingPointOfAlcohol Celsius = 78.37

const FreezingPointOfHumanBlood Celsius = -0.56
const BoilingPointOfHumanBlood Celsius = 100.0
```

استفاده از این ثوابت هم دقیقاً مثل ثوابت معمولیه، هیچ فرقی نداره:

```
var temperatureOfPool Celsius = 110.0
if temperatureOfPool >= BoilingPointOfWater {
    fmt.Println("Oh no! This pool is so hot, you could cook eggs in it!")
}
```

تعریف ثابت ها به صورت گروهی

گاهی لازم میشه چندتا ثابت که از نظر معنا به هم ربط دارن، با هم تعریف کنیم.
به جای اینکه تک‌تک بنویسیم:

```
const name1 = value1
const name2 = value2
```

میتونیم خیلی تمیزتر و جمع‌وجورتر بنویسیم:

```
const (
    name1 = value1
    name2 = value2
)
```

مثلاً اگه بخوای دمای 0 تا 10 درجه سلسیوس رو به صورت ثابت داشته باشی:

```
const (
       C0  Celsius = 0
       C1  Celsius = 1
       C2  Celsius = 2
       C3  Celsius = 3
       C4  Celsius = 4
       C5  Celsius = 5
       C6  Celsius = 6
       C7  Celsius = 7
       C8  Celsius = 8
       C9  Celsius = 9
       C10 Celsius = 10
)
```

هم تو نوشتن const صرفه‌جویی شد، هم برنامه خواناتر شد، و هم مشخص شد اینا از نظر معنایی به هم نزدیکن.

استفاده از iota

وقتی قراره چندتا ثابت داشته باشیم که مقادیرشون پشت‌سرهمه (مثلاً 0، 1، 2، ...)، استفاده از iota کارمون رو خیلی راحت‌تر می‌کنه:

```
const (
	C0 Celsius = iota
	C1
	C2
	C3
	C4
	C5
	C6
	C7
	C8
	C9
	C10
)
```

نکته:

وقتی از iota استفاده می‌کنی، فقط کافیه تو خط اول مقدار iota رو بدی Go خودش بقیه رو به ترتیب مقدار می‌ده. iota همیشه مقدارش از صفر شروع می‌شه. پس تو مثال بالا مقدار C0 میشه 0، C1میشه 1، و همین‌طور تا C10

شروع از عدد دلخواه با iota

حالا اگه نخوای C0 داشته باشی چی؟ مثلاً بخوای از C1 شروع کنی. چون iota همیشه از 0 شروع میشه، فقط کافیه یه 1+ بهش اضافه کنیم:

```
const (
	C1 Celsius = iota + 1
	C2
	C3
	C4
	C5
	C6
	C7
	C8
	C9
	C10
)
```

Enum چیست؟

یه لیست بسته از انتخاب‌ها که فقط همون گزینه‌ها مجازن، نه چیز دیگه!

فرض کن توی برنامه می‌خوای حالت‌های دما رو به‌عنوان یه موجودیت داشته باشی.

از اونجایی که دما سه حالت بیشتر نداره:

- سرد
- معتدل
- گرم

و این حالت‌ها همیشه مشخص و ثابت هستن و تغییر نمی‌کنن، پس تعریف کردنشون به‌صورت **ثابت**، بهترین انتخابه.

اما حالا سؤالی که پیش میاد اینه که **مقدار این ثابت‌ها رو چی بذاریم؟**

قبل از اینکه مقدار بدیم، بیا اول یه **نوع جدید** تعریف کنیم تا برنامه‌مون معنایی‌تر و قابل‌فهم‌تر بشه. مثلاً یه نوع بسازیم به اسم TempLevel(سطح دما) چون دقیقاً داریم در مورد سطوح مختلف دما حرف می‌زنیم.

```
type TempLevel int
```

حالا شاید بپرسی **خب چرا int ؟ یعنی چرا نوع پایه رو int گذاشتیم؟**

جواب ساده‌ست:

چون می‌خوایم برای تعریف مقدارهای ثابت (Cold, Moderate, Hot) از iota استفاده کنیم. و iota همیشه مقدار عددی از نوع عدد صحیح (int) تولید می‌کنه. پس منطقیه که نوع TempLevel رو هم بر پایه‌ی int تعریف کنیم تا همه‌چیز با هم جور دربیاد.

حالا می‌تونیم Enum خودمون رو با استفاده از iota خیلی تمیز تعریف کنیم:

```
type TempLevel int


const (
	Cold TempLevel = iota
	Moderate
	Hot
)
```

حالا دیگه هر جا تو برنامه بخوایم یه متغیر از نوع TempLevel بسازیم و مقدار بدیم، خیلی راحت می‌تونیم از این ثابت‌ها استفاده کنیم:

```
var todayTemperature Celsius = 3
var todayTemperatureLevel TempLevel

if todayTemperature < 0 {
    todayTemperatureLevel = Cold
} else if todayTemperature <= 20 {
    todayTemperatureLevel = Moderate
} else {
    todayTemperatureLevel = Hot
}

tempLevelNames := map[TempLevel]string{
    Cold:     "Cold",
    Moderate: "Moderate",
    Hot:      "Hot",
}

fmt.Printf("Today's temperature: %05.2f°C\n", todayTemperature)
fmt.Printf("Temperature level: %s\n", tempLevelNames[todayTemperatureLevel])
```

مزایای استفاده از Enum

- جلوی اشتباه رو می‌گیره (مثلاً کسی نتونه بنویسه todayTemperatureLevel = 42)
- کد خواناتر و واضح‌تر میشه
- قابل استفاده در شرط‌ها، switch، یا map به شکل معنای
- باعث میشه مفاهیم ثابت دنیای واقعی (مثل سطح دما) رو به شکل درستی مدل کنیم
- تغییرش تو کل برنامه خیلی ساده‌تره (چون یه جا تعریف شده)

لرن پث

تعریف تابع برای type تعریف شده

تا اینجای کار یاد گرفتیم که چجوری یه نوع داده‌ی جدید (type) تعریف کنیم تا کدهامون معنی‌دارتر و خوندنی‌تر بشن. اما خب، فقط تعریف کردن یه type کافی نیست! قراره یه قدم حرفه‌ای‌تر برداریم: تعریف تابع‌هایی که **فقط و فقط مخصوص اون type خاص باشن**.

این کار باعث می‌شه که هم ساختار کدمون مرتب‌تر بشه، هم اشتباهات احتمالی کمتر، و هم کدها خیلی طبیعی‌تر خونده بشن.

به این تابع‌هایی که فقط روی یه type خاص تعریف می‌شن، تو Go می‌گن **متد (Method)**.

چجوری یه متد تعریف کنیم؟

```
func (receiver ReceiverType) MethodName() {
    // statement
}
```

ساختار کلی تعریف متد این شکلیه:

func-1

همون کیورد آشنای تعریف توابعه. بدون func، نه تابعی داریم نه متدی.

(receiver ReceiverType)-2

اینجا نقطه‌ی تفاوت متد با تابع مشخص می‌شه Receiver یعنی اون چیزی که این متد قراره روش اجرا بشه. مثلاً اگه بنویسی (c Celsius) یعنی این متد مخصوص نوع Celsius تعریف شده و روی اون قابل اجراست.

- نام c مثل پارامتر تابع عمل می‌کنه. هرچی بذاری قبوله، سعی کن کوتاه و با معنی باشه.
- این بخش باعث می‌شه تابع به نوع Celsius یا هر نوع دیگه‌ای که بخوای *وصل* بشه.

MethodName-3

اسم متده که بعد از پرانتز گیرنده میاد. مثلاً ToFahrenheit یا Describe یا IsHot
اگه بخوای از بیرون پکیج بهش دسترسی داشته باشی، باید با حرف بزرگ شروع شه.

4-پرانتز ورودی (...)

می‌تونی مثل توابع عادی به متدت هم ورودی بدی. اینجا تعریفشون می‌کنی.

5-نوع خروجی (اختیاری)

می‌تونی یه مقدار (یا چند تا) از متدت برگردونی، یا اصلاً هیچی برنگردونی. مثل int، bool، یا حتی یه نوع دلخواهی که خودت ساختی.

6-بدنه‌ی متد { ... }

اینجا همون‌جاییه که منطق اصلی متدت رو پیاده‌سازی می‌کنی. تمام دستورات متد اینجا نوشته می‌شن.

لرن پث

یه مثال آشنا

قبلاً سه نوع داده تعریف کردیم Celsius، Fahrenheit و Kelvin

برای هر کدوم هم دوتا تابع داشتیم که مقدارشونو به نوع دیگه‌ای تبدیل کنن.

مثلاً برای Celsius دوتا تابع نوشته بودیم:

- CelsiusToFahrenheit
- CelsiusToKelvin

یکی از این توابعو ببین:

```
func CelsiusToFahrenheit(c Celsius) Fahrenheit {
    return (c * 1.8) + 32
}
```

خب، چرا می‌گیم این تابع "مربوط" به نوع Celsiusه؟
چون ورودی c از نوع Celsius هست و داره عملیات تبدیل رو روی همین نوع انجام می‌ده.

تبدیل تابع به متد

حالا اگه بخوای این تابعو به یه **متد** تبدیل کنی، کافیه پارامتر c Celsius رو بیاری به جای receiver:

```
func (c Celsius) CelsiusToFahrenheit() Fahrenheit {
    return (c * 1.8) + 32
}
```

دیگه نیازی به پارامتر ورودی نیست، چون از طریق c که همون receiver هستش، به مقدار دما دسترسی داری.

استفاده از متد

حالا اینطوری از متد استفاده می‌کنی:

```
var freezingPointOfBlood Celsius = -0.56
freezingPointOfBloodInFahrenheit := freezingPointOfBlood.CelsiusToFahrenheit()
fmt.Println(
    “The freezing point of human blood is: ”,
    freezingPointOfBloodInFahrenheit,
)
```

به‌جای اینکه freezingPointOfBlood رو به عنوان ورودی به تابع بدی، مستقیماً متد رو روی خودش صدا زدی. چون متغیر از نوع Celsius تعریف شده و این متد هم مخصوص Celsius هست، همه‌چی به خوبی و خوشی اجرا می‌شه.

یه نکته مهم درباره اسم‌گذاری متد

حالا که این متد روی نوع Celsius تعریف شده، نوشتن Celsius اول اسم متد یه جورایی اضافه‌کاریه! خب واضحه که داره روی Celsius اجرا می‌شه دیگه. بهتره اسم متد رو کوتاه‌تر و تمیزتر بنویسیم:

```
func (c Celsius) ToFahrenheit() Fahrenheit {
    return (c * 1.8) + 32
}
```

حالا می‌خوایم بقیه توابع تبدیل دما رو هم با توجه به نوعشون، به شکل متد بازنویسی کنیم.

```
func (c Celsius) ToFahrenheit() Fahrenheit {
    return Fahrenheit(float64(c)*1.8 + 32)
}

func (c Celsius) ToKelvin() Kelvin {
    return Kelvin(float64(c) + 273.15)
}

func (f Fahrenheit) ToCelsius() Celsius {
    return Celsius((float64(f) - 32) * 5.0 / 9.0)
}

func (k Kelvin) ToCelsius() Celsius {
    return Celsius(float64(k) - 273.15)
}

func (f Fahrenheit) ToKelvin() Kelvin {
    return f.ToCelsius().ToKelvin()
}

func (k Kelvin) ToFahrenheit() Fahrenheit {
    return k.ToCelsius().ToFahrenheit()
}
```

Chain کردن فراخوانی متدها

بیا یه نگاهی به متدهای ToKelvin و ToFahrenheit بندازیم. چی شد که یه همچین ساختاری توشون نوشتیم؟!

مثلاً توی این متد:

```
func (f Fahrenheit) ToKelvin() Kelvin {
    return f.ToCelsius().ToKelvin()
}
```

اول از همه receiver این متد، یعنی f از نوع Fahrenheit هست. خب چون روی Fahrenheit یه متد به اسم ToCelsius تعریف کردیم، خیلی راحت می‌تونیم بگیم:

```
f.ToCelsius()
```

که خروجیش از نوع Celsius میشه. حالا که یه Celsius داریم، می‌تونیم متد ToKelvin رو هم روی اون صدا بزنیم، چون متد ToKelvin برای Celsius تعریف شده.

این یعنی بدون اینکه نیاز باشه هی متغیر تعریف کنیم و وسط راه مقدار رو ذخیره کنیم، زنجیروار (chain) متدها رو صدا می‌زنیم و نتیجه نهایی رو برمی‌گردونیم. این کار علاوه بر اینکه کدت رو خیلی تمیز و جمع‌وجور می‌کنه، خوندنش رو هم آسون‌تر می‌کنه. این دقیقاً همون جادوی متده

اگه از تکنیک chain استفاده نمیکردیم:

```
func (f Fahrenheit) ToKelvin() Kelvin {
    c := f.ToCelsius()
    k := c.ToKelvin()
    return k
}
```

متدهایی با اسم‌های تکراری؟ مشکله؟

یه چیز خیلی جالب دیگه هم هست... تا حالا متوجه شدی که چندتا متد با اسم یکسان داریم؟

مثلاً:

- دو تا ToCelsius
- دو تا ToFahrenheit
- دو تا ToKelvin

اگه یادت باشه تو درسنامه‌ی «تابع» گفتیم که تو Go نمی‌تونی چندتا تابع با اسم تکراری تو یه پکیج داشته باشی. ولی الان اینجا همچین کاری کردیم و هیچ مشکلی هم پیش نیومده! چرا؟

چون اینا تابع نیستن! بلکه **متدن**.

توی متدها، محدودیت تکرار اسم فقط در صورتی اعمال میشه که **روی یک type مشخص** دو متد با اسم یکسان تعریف کرده باشی. ولی وقتی متدهایی با نام مشابه رو روی type های مختلف تعریف می‌کنی، هیچ مشکلی پیش نمیاد. مثلاً:

- ToKelvin() روی Celsius
- ToKelvin() روی Fahrenheit

پس Go خیلی راحت تشخیص میده که بسته به نوع متغیر، کدوم متد رو صدا بزنه

تعریف متد GetTempLevel فقط برای Celsius

حالا بیایم یه قدم جلوتر بریم. فرض کن می‌خوایم بر اساس مقدار دما تشخیص بدیم که دما تو کدوم سطحه؟ (سرد، معتدل یا گرم). برای این کار یه متد به اسم GetTempLevel فقط روی Celsius تعریف می‌کنیم:

```
func (c Celsius) GetTempLevel() TempLevel {
	if c < 10 {
		return Cold
	} else if c >= 10 && c <= 25 {
		return Moderate
	} else {
		return Hot
	}
}
```

با این متد می‌تونیم خیلی راحت بفهمیم که یه دمای خاص تو کدوم دسته‌بندیه

تست و بررسی متدها

بریم چندتا تست ساده بزنیم:

```
var berlinTemperature Fahrenheit = 51.5
var newYorkTemperature Celsius = 21
var londonTemperature Kelvin = 290.15

fmt.Printf("Berlin's temperature: %05.2fK\n", berlinTemperature.ToKelvin())
fmt.Printf("NewYork's temperature: %05.2f°F\n", newYorkTemperature.ToFahrenheit())
fmt.Printf(
    "The temperature in London is %05.2f°C, which is considered %s.\n",
    londonTemperature.ToCelsius(),
    londonTemperature.GetTempLevel(),
)
```

ولی برنامه کامپایل نمی‌شه و با خطای زیر مواجه می‌شیم:

```
londonTemperature.GetTempLevel undefined (type Kelvin has no field or method
GetTempLevel)
```

چرا؟ چون ما متد GetTempLevel رو فقط روی نوع Celsius تعریف کردیم، نه Kelvin!

و اینجا داریم سعی می‌کنیم اون متد رو روی متغیری از نوع Kelvinصدا بزنیم، که خب معلومه به مشکل می‌خوریم

همیشه باید حواسمون باشه که متدی که صدا می‌زنیم، فقط روی اون نوع خاصی کار می‌کنه که براش تعریف شده.

برای اینکه این برنامه به درستی کار کنه، باید اول دما رو از Kelvin به Celsius تبدیل کنیم و بعد متد GetTempLevel رو صدا بزنیم:

```
fmt.Printf(
    "The temperature in London is %05.2f°C, which is considered %s.\n",
    londonTemperature.ToCelsius(),
    londonTemperature.ToCelsius().GetTempLevel(),
)
```

خروجی

```
Berlin's temperature: 283.98K
NewYork's temperature: 69.80°F
The temperature in London is 17.00°C, which is considered %!s(main.TempLevel=1).
```

با اینکه londonTemperature از نوع Kelvin بود، اول با ()ToCelsius به Celsius تبدیلش کردیم و بعد روی خروجی، متد ()GetTempLevel رو صدا زدیم. و اینطوری همه چی به درستی کار کرد!

قوانین call by value و call by reference در متدها

همون قوانینی که تو درسنامه‌ی «تابع» درباره‌ی تفاوت بین **Call by Value** و **Call by Reference** گفتیم، دقیقاً برای متدها هم صدق می‌کنه. مثلاً یادمون هست که **slice**ها به‌صورت ضمنی **مثل Call by Reference** رفتار می‌کنن؛ یعنی اگه تو یه تابع یا متد چیزی رو توی یه slice تغییر بدی، اون تغییر بیرون از تابع هم دیده میشه.

بیا یه مثال واقعی ببینیم:

تعریف Type برای مجموعه دماها

```
type CelsiusCollection []Celsius
type FahrenheitCollection []Fahrenheit
type KelvinCollection []Kelvin
```

اینجا اومدیم سه نوع جدید تعریف کردیم برای نگهداری لیستی از دماها:

- CelsiusCollection: مجموعه‌ای از دما به سلسیوس
- FahrenheitCollection: مجموعه‌ای از دما به فارنهایت
- KelvinCollection: مجموعه‌ای از دما به کلوین

حالا دو متد ساده برای تبدیل CelsiusCollection به FahrenheitCollection و KelvinCollection می‌نویسیم:

```
func (cc CelsiusCollection) ToFahrenheit() FahrenheitCollection {
	output := make(FahrenheitCollection, len(cc))
	for i := 0; i < len(cc); i++ {
		output[i] = cc[i].ToFahrenheit()
	}
	return output
}

func (cc CelsiusCollection) ToKelvin() KelvinCollection {
	output := make(KelvinCollection, len(cc))
	for i := 0; i < len(cc); i++ {
		output[i] = cc[i].ToKelvin()
	}
	return output
}
```

تست متد ToKelvin

```
var berlinTemperature Celsius = 10
var newYorkTemperature Celsius = 21
var londonTemperature Celsius = 17

var cityTemperatures CelsiusCollection = CelsiusCollection{
    berlinTemperature,
    newYorkTemperature,
    londonTemperature,
}

cityTemperaturesInKelvin := cityTemperatures.ToKelvin()

for i, k := range cityTemperaturesInKelvin {
    fmt.Printf("City_%d temperature: %05.2fK\n", i+1, k)
}
```

خروجی

```
City_0 temperature: 283.15K
City_1 temperature: 294.15K
City_2 temperature: 290.15K
```

همه چی دقیقا طبق انتظار پیش رفت — دماها به‌خوبی به کلوین تبدیل شدن.

حالا یه سناریو جالب:

آیا تغییر در متد روی داده اصلی هم اثر می‌ذاره؟

بیایم تو همون متد ToKelvin بعد از انجام تبدیل، دمای اصلی رو صفر کنیم! اینطوری می‌تونیم بفهمیم که آیا اون تغییر توی slice اصلی هم اعمال میشه یا نه؟

```
func (cc CelsiusCollection) ToKelvin() KelvinCollection {
	output := make(KelvinCollection, len(cc))
	for i := 0; i < len(cc); i++ {
		output[i] = cc[i].ToKelvin()
		cc[i] = 0
	}
	return output
}
```

```
var berlinTemperature Celsius = 10
var newYorkTemperature Celsius = 21
var londonTemperature Celsius = 17

var cityTemperatures CelsiusCollection = CelsiusCollection{
    berlinTemperature,
    newYorkTemperature,
    londonTemperature,
}

cityTemperaturesInKelvin := cityTemperatures.ToKelvin()

for i, k := range cityTemperaturesInKelvin {
    fmt.Printf("City_%d temperature: %05.2fK\n", i+1, k)
}

for i, c := range cityTemperatures {
    fmt.Printf("City_%d temperature: %05.2f°C\n", i+1, c)
}
```

خروجی

```
City_1 temperature: 283.15K
City_2 temperature: 294.15K
City_3 temperature: 290.15K
City_1 temperature: 00.00°C
City_2 temperature: 00.00°C
City_3 temperature: 00.00°C
```

دقیقاً همون چیزی که انتظارش رو داشتیم اتفاق افتاد! تغییراتی که توی متد دادیم، مستقیماً روی cityTemperatures اثر گذاشت، چون CelsiusCollection یه slice‌ـه و رفتارش مثل Call by Reference هست.

یعنی تغییر تو متد، تغییر واقعی روی داده اصلی ایجاد می‌کنه.

نکته خیلی مهم

تو Go **فقط وقتی می‌تونی برای یه type متد بنویسی که داخل همون پکیجی باشی که اون type تعریف شده**. پس نمی‌تونی برایtype های از پیش تعریف شده‌ی Go (مثل int, float64, string) متد جدید بنویسی.

مفهوم Type Safety در Type های تعریف شده

فرض کن توی یه سوله‌ی صنعتی، دوتا حسگر دما داریم:

- یکی **داخل سوله** نصبه و دما رو به **سلسیوس (Celsius)** گزارش می‌ده
- اون یکی **بیرون سوله** نصبه و دما رو به **فارنهایت (Fahrenheit)** ثبت می‌کنه

توی یکی از بخش‌های برنامه‌مون قراره بررسی کنیم:

"اگه دمای داخل سوله از بیرون بیشتر بود، هشدار بده".

اگه بدون استفاده ازtype های اختصاصی جلو بریم، کدمون چیزی شبیه این می‌شه:

```
func celsiusToFahrenheit(c float64) float64 {
	return (c * 1.8) + 32
}

func shouldTriggerAlarm(insideTemp float64, outdoorTemp float64) bool {
	if celsiusToFahrenheit(insideTemp) > outdoorTemp {
		return true
	}

	return false
}
```

تابع celsiusToFahrenheit رو که از قبل می‌شناسیم؛ و تابع shouldTriggerAlarm قراره دمای داخل و بیرون سوله رو با هم مقایسه کنه. اگه دمای داخل بیشتر از بیرون باشه، احتمال آتش‌سوزی هست و باید آلارم روشن بشه (یعنی true برگرده)

تا اینجا همه‌چی اوکیه. حالا فرض کن دماها رو از طریق سنسورهای حرارتی خوندیم و می‌خوایم نتیجه رو چاپ کنیم:

```
// Temperature readings inside and outdoor the North warehouse (in Fahrenheit)
var insideTemp float64 = 38.0 // Celsius
var outdoorTemp float64 = 104.0 // Fahrenheit

// Name of the warehouse being monitored
var warehouseName string = "North"

// Determine whether the alarm should be triggered based on temperature conditions
shouldAlarmTrigger := shouldTriggerAlarm(insideTemp, outdoorTemp)

if shouldAlarmTrigger {
    fmt.Printf(
        "⚠  Warning: Temperature in the %s warehouse is critical!\n",
        warehouseName,
    )
} else {
    fmt.Printf(
        "✅ All clear: Temperature in the %s warehouse is within safe limits.\n",
        warehouseName,
    )
}
```

خروجی

```
✅ All clear: Temperature in the North warehouse is within safe limits.
```

خب، منطقیه. چون 38 سلسیوس از 104 فارنهایت کمتره، پس خطری نیست.

حالا تصور کن یه اشتباه ساده کنیم!

مثلاً موقع فراخوانی تابع shouldTriggerAlarm اشتباهی جای دوتا دما رو عوض کنیم:

```
shouldTriggerAlarm(outdoorTemp, insideTemp)
```

خروجی

```
⚠ Warning: Temperature in the North warehouse is critical!
```

یه اشتباه ساده، اما نتیجه فاجعه‌بار شد!

در حالی که هیچ خطری وجود نداشت، برنامه ما اعلام خطر کرد. حالا تصور کن این هشدار باعث بشه آژیر کارخانه به صدا دربیاد، همه کارکنان کار رو متوقف کنن، دستگاه‌ها خاموش بشن و کلی ضرر مالی وارد بشه... فقط چون دوتا متغیر اشتباهی جابه‌جا شدن!

چطور می‌تونیم جلوی این اشتباه رو بگیریم؟

با تعریف type های مخصوص به خودمون — کاری که اسمش هست **type safety**!

بازنویسی برنامه با type های اختصاصی:

```
type Warehouse string


func (warehouse Warehouse) ShouldTriggerAlarm(insideTemp Celsius, outdoorTemp
Fahrenheit) bool {
	if insideTemp.ToFahrenheit() > outdoorTemp {
		return true
	}

	return false
}
```

و حالا تستش کنیم:

```
// Temperature readings inside and outdoor the North warehouse (in Fahrenheit)
var insideTemp Celsius = 38.0
var outdoorTemp Fahrenheit = 104.0

// Identifier for the warehouse being monitored
var warehouse Warehouse = "North"

// Determine whether the alarm should be triggered based on temperature conditions
shouldAlarmTrigger := warehouse.ShouldTriggerAlarm(insideTemp, outdoorTemp)

if shouldAlarmTrigger {
    fmt.Printf(
        "⚠  Warning: Temperature in the %s warehouse is critical!\n",
        warehouse,
    )
} else {
    fmt.Printf(
        "✅ All clear: Temperature in the %s warehouse is within safe limits.\n",
        warehouse,
    )
}
```

خروجی

```
✅ All clear: Temperature in the North warehouse is within safe limits.
```

حالا اگه همون اشتباه رو تکرار کنیم چی؟

مثلاً دوباره جای دوتا دما رو اشتباه بدیم:

```
shouldTriggerAlarm(outdoorTemp, insideTemp)
```

این بار برنامه حتی اجرا هم نمی‌شه!

خطای کامپایل دریافت می‌کنی:

```
.\main.go:132:53: cannot use outdoorTemp (variable of float64 type Fahrenheit) as
Celsius value in argument to warehouse.ShouldTriggerAlarm

.\main.go:132:66: cannot use insideTemp (variable of float64 type Celsius) as
Fahrenheit value in argument to warehouse.ShouldTriggerAlarm
```

و این یعنی **کامپایلر خودش هواتو داره**! نمی‌ذاره یه اشتباه ساده، باعث یه فاجعه بشه.

نتیجه‌گیری:

با تعریف type های جدید، داری به کامپایلر کمک می‌کنی که ازت محافظت کنه. این کار:

- جلوی اشتباهات تایپی و منطقی رو می‌گیره
- باعث می‌شه کدت خودمستند و واضح باشه
- حتی دیگه نیاز نیست برای هر متغیر کامنت بنویسی! وقتی می‌گی Celsius، خودش حرف می‌زنه
- توی تیم‌های بزرگ، باعث کاهش باگ‌های ناخواسته می‌شه

نکته:

هر وقت دیدی یه متغیر معنا یا نقش خاصی داره، براش یه type مخصوص تعریف کن! حتی اگه زیرساختش ساده‌ست. اینطوری هم خوانایی بالا میره، هم ایمنی کدت چند پله قوی‌تر میشه. مثل اینه که به جای لباس یک‌شکل برای همه، لباس فرم مخصوص برای هر نقش طراحی کنی.

مزایای استفاده از type سفارشی

1. **معنادار شدن داده‌ها**

- به‌جای اینکه فقط یه float64 یا string داشته باشی، می‌تونی بگی این عدد «دمای سلسیوس» یا این رشته «کد ملی» کاربره.
- باعث می‌شه کدت **مثل مستندات خودش باشه!**

2. **افزایش ایمنی (Type Safety)**

- Go جلوی اشتباهاتی مثل دادن Fahrenheit به جای Celsius رو می‌گیره.
- یعنی نمی‌تونی به‌اشتباه یه مقدار از نوع دیگه رو وارد تابع یا متغیر کنی مگر اینکه **صریحاً تبدیلش کنی.**

3. **امکان تعریف متد روی نوع سفارشی**

- حتی روی یه نوع ساده مثل int یا float64 می‌تونی متد بنویسی!

4. **ساده‌سازی کد و افزایش خوانایی**

- وقتی یه type تعریف می‌کنی، دیگه لازم نیست توی همه‌جا map[string]string یا float64 ببینی.
- به‌جاش از PhoneBook یا Celsius استفاده می‌کنی، که فهمش سریع‌تره.

5. **قابلیت تغییر متمرکز در آینده**

- اگه بعداً خواستی رفتار یا نوع داده‌ی پشت Celsius رو عوض کنی، فقط کافیه تو تعریف نوع تغییر بدی، نه کل پروژه.

6. **تست‌نویسی راحت‌تر**

- تو تست‌ها می‌تونی نوع‌های سفارشی رو جدا بررسی کنی یا حتی با استفاده از اونا mock بسازی.

7. **مستقل‌سازی لایه‌های برنامه**

- توی پروژه‌های بزرگ می‌تونی بین لایه‌ها نوع‌های خاص تعریف کنی (مثل UserID, ProductID) تا لایه‌ها به‌طور محکم ولی جدا از هم کار کنن.

8. **جلوگیری از سوءتفاهم‌های منطقی**

- مثلا دیگه کدی نمینویسی که یه int رو اشتباهی به‌جای UserID به تابع eleteProduct(id int) بدی!

مقایسه استفاده کردن و استفاده نکردن از تعریف نوع سفارشی

ویژگی	بدون type سفارشی	با type سفارشی
تعریف نوع	map[string]string	PhoneBook
خوانایی	پایین	بالا
قابلیت اشتباه تایپی	زیاد	کمتر
امکان تعریف متد	فقط روی struct می‌شه	روی هر نوع سفارشی
کمک به تیم	سخت‌تر برای فهمیدن نقش داده	آسون‌تر برای مشارکت دیگران
تست‌پذیری	سخت‌تر	راحت‌تر

تمرین 1: ساخت پکیج برای مجموعه دماها

تو درسنامه با چند تا type آشنا شدیم که برای نگهداری مجموعه دماهای مختلف بودن:

```
type CelsiusCollection []Celsius
type FahrenheitCollection []Fahrenheit
type KelvinCollection []Kelvin
```

ما فقط برای CelsiusCollection دوتا متد تبدیل نوشته بودیم ToKelvin() و ToFahrenheit()

کارهایی که باید انجام بدی:

1. یه پکیج جدید بساز به اسم temp و کدهای مربوط به انواع دما (Celsius, Fahrenheit, Kelvin) رو بذار توش.
2. یه پکیج دیگه بساز به اسم tempcollection و کدهای مجموعه دماها رو بذار اونجا.
3. برای FahrenheitCollection دوتا متد بنویس : ToKelvin() و ToCelsius()
4. برای KelvinCollection هم دوتا متد بنویس : ToFahrenheit() و ToCelsius()
5. حالا برای CelsiusCollection, FahrenheitCollection, KelvinCollection سه متد بنویس:
   - GetHigh() بیشترین دما رو پیدا کنه
   - GetLow() کم‌ترین دما رو پیدا کنه
   - CalculateAverage() میانگین دما رو تا 2 رقم اعشار حساب کنه

تمرین 2: ساخت پکیج محاسبه امتیاز

می‌خوایم یه پکیج طراحی کنیم برای مدیریت امتیاز کاربر تو یه بازی ساده

مراحل:

1. یه پکیج بساز به اسم gamescore
2. یه تایپ جدید تعریف کن به اسم Score که نوعش همون int باشه
3. یه تابع بساز به اسم New که یه عدد int از ورودی تابع بگیره و یه مقدار از نوع Score برگردونه
   - اگه عدد بیشتر از 100 بود، عدد 100 رو بذاره.
   - اگه عدد کمتر از 0 بود، مقدار 0 رو بذاره.
4. برای نوع Score دو تا متد بنویس:
   - Increment(step int) با یه مقدار مشخص امتیاز رو زیاد می‌کنه
   - Decrement(step int) با یه مقدار مشخص کمش می‌کنه
5. یه متد دیگه بنویس به اسم Report() که یه توصیف متنی از امتیاز ارائه بده.
   - اگه امتیاز بیشتر از 85 بود "Excellent"
   - اگه بین 45 تا 85 بود "Good job"
   - اگه کمتر از 45 بود "Needs improvement"
6. حالا یه برنامه بنویس که:
   - از پکیج gamescore استفاده کنه
   - با فراخوانی تابع New از این پکیج یه Score دریافت کن
   - با فراخوانی متد های Increment و Decrement مقدارشو زیاد یا کم کن
   - در نهایت نتیجه رو با فراخوانی متد Report() چاپ کن
7. حالا اسم متد Report() رو بذار String()

تو برنامه‌ی اصلی به جای fmt.Println(score.Report()) بنویس fmt.Println(score) و ببین چه جادویی رخ می‌ده! تحقیق کن چرا این اتفاق افتاد؟

تمرین 3: ساخت پکیج قیمت و تخفیف

می‌خوایم با قیمت و تخفیف و محاسبه قیمت نهایی سر و کله بزنیم.

دوتا پکیج نیاز داریم

- price
- discount

در پکیج price:

- یه type تعریف کن به اسم Price از جنس int
- یه تابع بساز به اسم New(price int) که:
  - اگه عدد ورودی کمتر از 1000 بود، 1000بذاره.
  - مقدار نهایی رو به صورت Price برگردونه.

در پکیج discount:

- یه type تعریف کن به اسم Discount از جنس int
- دوتا متغیر min و max در سطح پکیج و به صورت private تعریف کن
- توی تابع init() این دو مقدار رو مقداردهی کن:
  - min = 10
  - max = 70
- یه تابع بساز به اسم New(discount int) که:
  - اگه کمتر از min بود، مقدارش بشه min
  - اگه بیشتر از max بود، مقدارش بشه max
- یه متد به اسم ApplyDiscount(price Price) Price بنویس که:
  - مقدار تخفیف رو به اعشار تبدیل کن (مثلاً 45 میشه 0.45)
  - مقدار تخفیف رو با حاصلضرب مقدار اعشاری تخفیف در قیمت محاسبه کن
  - مقدار تخفیف رو از قیمت اولیه کم کن و نتیجه نهایی رو به صورت عدد صحیح و رو به پایین گرد کن
  - در نهایت یه Price جدید برگردون که در واقع قیمت جدید بعد از تخفیف هست

برنامه اصلی:

- یه slice بساز از نوع Price با 100 قیمت رندوم
- یه slice دیگه از Discount با 5 مقدار تصادفی
- تخفیف‌ها رو به صورت تصادفی روی قیمت‌ها اعمال کن
- قیمت‌هایی که کمتر از 500,000 شدن رو بذار توی یه slice جدید به اسم affordableProducts
- در نهایت این لیست رو چاپ کن