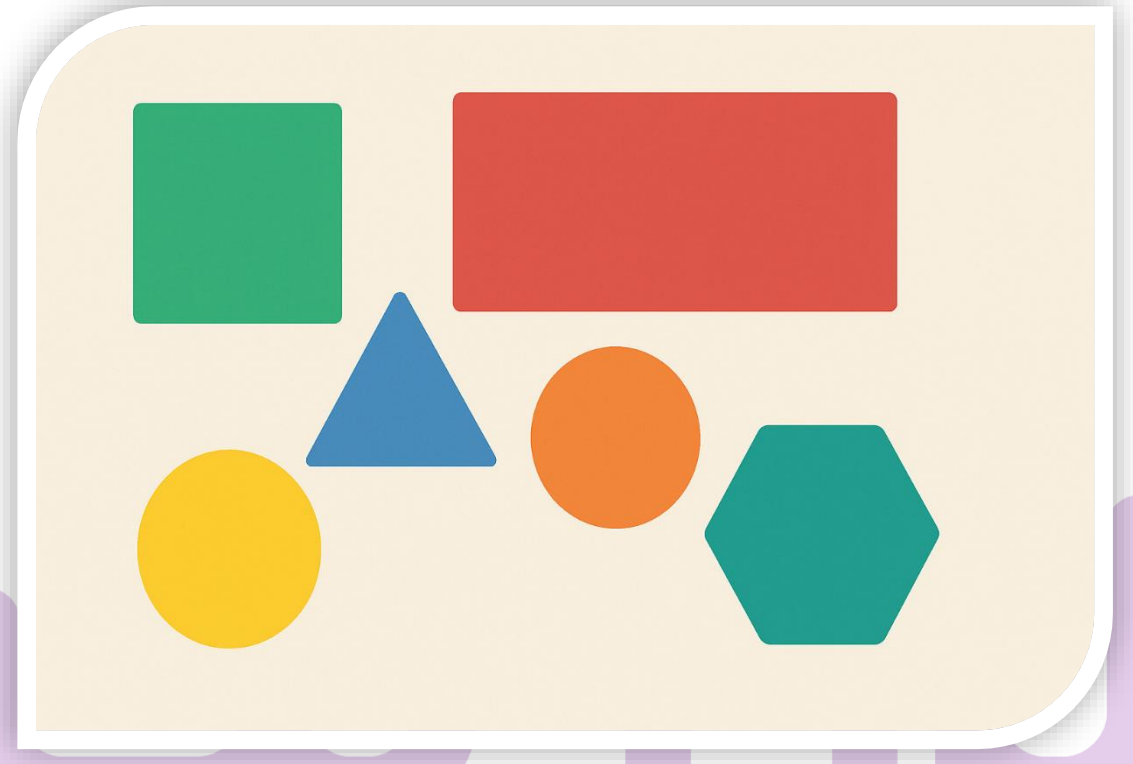
مفهوم واسط



وقتی یه type جدید تعریف می‌کنیم، تو بیشتر مواقع هدفمون اینه که براش یه سری متد هم بسازیم. این متدها در واقع دارن یه‌جور عملیات روی همون type که ساختیم انجام میدن.

مثلاً فرض کن یه type داریم برای مربع که فقط طول ضلع رو نگه میداره. خب ما می‌دونیم که مساحت و محیط مربع قراره تو آینده به کارمون بیاد، پس دو تا متد براش تعریف می‌کنیم: یکی برای محاسبه محیط، یکی برای محاسبه مساحت.

```
type Square struct{
    SideLength float64
}

func (s Square) Perimeter() float64 {
    return 4 * s.SideLength
}

func (s Square) Area() {
    return s.SideLength * s.SideLength
}
```

این دو متد دارن با استفاده از فیلد SideLength، محیط و مساحت مربع رو برامون حساب می‌کنن.

حالا فرض کن تو برنامه یه تابع داریم که با گرفتن یه مربع، می‌تونه تصمیم بگیره این مربع به درد تبدیل شدن به جعبه می‌خوره یا نه!

```
func isSuitableForBox(s Square) bool {
    if s.Perimeter() >= 20.0 && s.Area() >= 25.0 {
        return true
    }

    return false
}
```

تا اینجا همه‌چی خوبه. ولی اگه بیایم یه type برای مستطیل هم تعریف کنیم چی؟

```
type Rectangle struct{
    Width float64
    Height float64
}

func (r Rectangle) Perimeter() float64 {
    return 2 * (r.Width + r.Height)
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

الان دیگه این تابع isSuitableForBox با مستطیل کار نمی‌کنه! چون ورودی تابع نوعش Square هست.

خب راه‌حل ساده اینه که یه تابع جدید هم برای مستطیل بسازیم:

```
func isSuitableForRectangleBox(r Rectangle) bool {
    if r.Perimeter() >= 20.0 && r.Area() >= 25.0 {
        return true
    }

    return false
}
```

ولی مشکل اینه که تابع isSuitableForRectangleBox تنها فرقی که با تابع isSuitableForBox داره، نوع ورودی هست! حالا اگه یه روز تصمیم بگیریم محدودیت اندازه محیط رو از 20 به 30 تغییر بدیم، باید تو هر دو تا تابع تغییر بدیم.

حالا تصور کن همین تابع رو برای اشکال دیگه هم نوشته باشیم: مثلث، ذوزنقه، دایره، شش‌ضلعی و...

هر بار بخوایم یه تغییر کوچیک بدیم، باید کلی تابع رو یکی‌یکی ویرایش کنیم. این یعنی فاجعه!

پس وقتشه به یه راه بهتر فکر کنیم.

چی میشه اگه به جای اینکه این تابع رو به یه نوع خاص وابسته کنیم، بیایم کاری کنیم که تابع فقط به **وجود متدهای Perimeter و Area** وابسته باشه؟
فرقی هم نکنه اون type چیه. اینطوری فقط یه تابع خواهیم داشت و هر نوع داده‌ای که این دو متد رو پیاده‌سازی کرده باشه (مربع، مستطیل، مثلث، دایره و...) می‌تونه بهش پاس داده بشه.

اینجاست که **واسط‌ها (interface)** وارد میشن. تو Go می‌تونیم هر وقت خواستیم کدمون رو از وابستگی به نوع خاص آزاد کنیم و به جای نوع، به **مجموعه رفتارها** (یعنی مجموعه‌ای از متدها) وابسته کنیم، از interface استفاده کنیم.

تعریف واسط

برای تعریف یه واسط (interface) تنها چیزی که لازم داریم اینه که **لیستی از متدهایی** که برامون مهم هست رو مشخص کنیم.

توی مثال قبلی دیدیم که برای انعطاف‌پذیر کردن تابع isSuitableForBox، به یه واسطی احتیاج داشتیم که فقط دو متد Perimeter و Area رو داشته باشه.

ساختار کلی تعریف یه interface توی Go به این شکله:

```
type InterfaceName interface {
    Method1(params) returnType
    Method2(params) returnType
    // ...
}
```

توضیح بخش‌ها:

- **type**: کلمه کلیدی برای تعریف یه نوع جدید
- **InterfaceName**: اسم اینترفیس، که هر چی بخوای می‌ذاری (ترجیحاً توصیفی باشه)
- **interface**: کلمه کلیدی برای تعریف واسط
- **{}**: لیست متدهایی که باید توسط نوع‌های مختلف پیاده‌سازی بشن (بدون کلمه func)

استفاده از واسط

فرض کن می‌خوایم با اشکال هندسی کار کنیم. این اشکال باید بتونن محیط و مساحت خودشون رو حساب کنن. خب، پس یه واسط به اسم Shape می‌سازیم:

```
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

حالا چطور از این واسط استفاده کنیم؟

هر جا که به جای وابسته شدن به **یک نوع مشخص** (مثل Square یا Rectangle) می‌خوایم با **هر نوعی که این متدها رو داره** کار کنیم، از interface استفاده می‌کنیم.

مثلاً تابع اولیه isSuitableForBox وابسته به Square بود. ولی الان می‌تونیم بازنویسیش کنیم که به Shape وابسته باشه:

```
func isSuitableForBox(shape Shape) bool {
    if shape.Perimeter() >= 20.0 && shape.Area() >= 25.0 {
        return true
    }

    return false
}
```

نکته جالب اینه که از نظر معنی هم کد شفاف‌تر شده:

- این تابع یه شکل می‌گیره (هر شکلی که باشه، مهم نیست)
- شرطش فقط اینه که اون شکل **دو متد Perimeter و Area رو داشته باشه**

بریم که تستش کنیم:

```
var square = Square{
    SideLength: 10,
}

if isSuitableForBox(square) {
    fmt.Printf(
        "Square with SideLength=%.2f OK for box!\n",
        square.SideLength,
    )
}

var rectangle = Rectangle{
    Width:  4,
    Height: 7,
}

if isSuitableForBox(rectangle) {
    fmt.Printf(
        "Rectangle with Width=%.2f and Height=%.2f OK for box!\n",
        rectangle.Width,
        rectangle.Height,
    )
}
```

خروجی

```
Square with SideLength=10.00 OK for box!
Rectangle with Width=4.00 and Height=7.00 OK for box!
```

پس با این کار، تابع isSuitableForBox رو جوری نوشتیم که هم با Square کار کنه، هم با Rectangle، و حتی هر شکل دیگه‌ای که توی آینده اضافه بشه، بدون اینکه نیاز باشه کد تابع رو تغییر بدیم.

وقتی یک نوع داده، ناقص یا اشتباه یک واسط رو پیاده‌سازی کنه

پیاده‌سازی ناقص

فرض کن یه نوع داده داریم که **همه‌ی متدهایی که یک interface انتظار داره رو پیاده نکرده**.

سؤال اینه: آیا می‌تونیم هر جا که اون interface نیازه از این نوع داده استفاده کنیم؟

جواب منطقی: نه

ولی بذار بریم با یه مثال تستش کنیم تا ببینیم دقیقاً چه خطایی می‌گیریم.

فرض کن برای دایره یه struct تعریف کنیم و فقط متد Perimeter رو براش بنویسیم:

```
type Circle struct {
	Radius  float64
}

func (c Circle) Perimeter() float64 {
	return 2 * math.Pi * c.Radius
}
```

اینجا Circle فقط یک متد Perimeter داره و متد Area رو پیاده نکرده.

حالا بیایم اینو به تابع isSuitableForBox پاس بدیم:

```
var circle = Circle{
    Radius: 4.0,
}


isCircleSuitable := isSuitableForBox(circle)
if isCircleSuitable {
    fmt.Printf(
        "Circle with Radius=%.2f OK for box!\n",
        circle.Radius,
    )
}
```

خروجی

```
cannot use circle (variable of struct type Circle) as Shape value in argument to
isSuitableForBox: Circle does not implement Shape (missing method Area)
```

کامپایلر داره میگه که این Circle که تعریف کردی کامل نیست، متد Area رو نداره. پس نمی‌تونم به‌عنوان یک Shape بپذیرمش.

پیاده سازی اشتباه

حالا فرض Area رو هم اضافه می‌کنیم، ولی اشتباهی نوع خروجیش رو float32 نوشتیم:

```
func (c Circle) Area() float32 {
      return math.Pi * float32(c.Radius) * float32(c.Radius)
}
```

بریم دوباره تستش کنیم

```
cannot use circle (variable of struct type Circle) as Shape value in argument to
isSuitableForBox: Circle does not implement Shape (wrong type for method Area)
                have Area() float32
                want Area() float64
```

این بار کامپایلر بهمون میگه با وجود اینکه متد Area رو برای نوع Circle پیاده کردی، ولی نوع خروجی‌اش(float32) با اون چیزی که Shape خواسته (float64) فرق داره ، باید دقیقاً همون باشه!

اصلاح نهایی

پس بیایم درستش کنیم و نوع خروجی رو همون float64 بذاریم:

```
func (c Circle) Area() float64 {
	return math.Pi * c.Radius * c.Radius
}
```

خروجی

```
Circle with Radius=4.00 OK for box!
```

برنامه به خوبی کامپایل شد و همون نتیجه ای که مورد انتظار بود چاپ شد.

جمع‌بندی

وقتی یه نوع داده بخواد یک interface خاص رو پیاده‌سازی کنه باید:

1. **تمام متدهایی** که توی interface تعریف شده رو پیاده کنه.
2. امضای متدها (اسم، ورودی‌ها، خروجی‌ها) باید **دقیقاً** با interface یکی باشه.

پیاده سازی ضمنی interface (واسط)

یکی از خارق‌العاده‌ترین نکات درباره‌ی **interface**ها تو زبان Go اینه که برای اینکه یک **نوع داده** یک interface خاص رو پیاده‌سازی کنه، اصلاً لازم نیست از وجود اون interface خبر داشته باشه! کافیه فقط متدهایی رو که اون interface انتظار داره، پیاده کرده باشه.

بیا این سناریو رو باهم پیش بریم:

فرض کن به‌جای اینکه خودمون برای هر شکل هندسی مثل Square، Rectangle، Circle و ... Struct تعریف کنیم، یک پکیج آماده نصب کردیم که تمام این اشکال رو تعریف کرده و برای هر کدوم هم متدهای:

- Boundary() برای محیط
- Area() برای مساحت

رو از قبل پیاده کرده.

حالا ما تو کد خودمون تابعی نوشتیم به اسم isSuitableForBox و می‌خوایم **انعطاف‌پذیر** باشه و با هر شکل هندسی که از اون پکیج میاد کار کنه.

کاری که باید بکنیم اینه که یک واسط (interface) به اسم Shape تعریف کنیم و انتظار داشته باشه دو متد Boundary و Area وجود داشته باشه.

حالا چون اون پکیج این دو متد رو **از قبل** پیاده کرده، به‌صورت پیش‌فرض، داره interface ما رو **بدون هیچ تغییر و خبر داشتن** پیاده می‌کنه! یعنی حتی بدون اینکه دست به کد اون پکیج بزنیم، کد ما و پکیج هماهنگ میشن.

نکته جالب اینه که تو زبان‌هایی مثل PHP یا Java ،برای اینکه یک کلاس یک interface رو پیاده کنه، باید **صراحتاً** داخل کلاس بنویسیم که این کلاس فلان interface رو implement می‌کنه. و این یعنی فاجعه!

چون خیلی وقت‌ها ما اجازه نداریم کد پکیج‌های دیگران رو دستکاری کنیم. خوشبختانه Go این مشکل رو با همین روش **پیاده‌سازی ضمنی (implicit)** حل کرده.

تشخیص نوع واقعی پشت Interface (Type Assertion)

وقتی با یک interface کار می‌کنیم، بعضی وقت‌ها پیش میاد که باید بفهمیم **نوع واقعی** که پشت این interface قایم شده، چی بوده؟

مثلاً توی تابع isSuitableForBox، تنها چیزی که ما داریم، پارامتر shape هست که یک interface از نوع Shape هستش.

ما می‌دونیم این shape یک سری متد مثل Boundary و Area رو داره، ولی نمی‌دونیم **واقعا** Square هست یا Rectangle یا حتی Circle.

اینجاست که **type assertion** میاد وسط.

ساختار کلی type assertion

```
value, ok := interfaceValue.(TargetType)
```

interfaceValue: مقداری که از نوع interface هست (مثل shape در isSuitableForBox)

TargetType: نوع داده‌ای که می‌خوایم چک کنیم (مثل Square، Rectangle یا Circle)

اگر مقدار داخل interfaceValue واقعاً از نوع TargetType باشه:

- value: همون مقدار با نوع TargetType میشه
- ok: مقدارش true میشه

اگر نباشه:

- value: zero value اون نوع میشه
- ok: مقدارش false میشه

روش type assertion

بیا یه تابع اضافه کنیم که دو تا ورودی بگیره:

1. پارامتر اول: از نوع Shape به اسم shape
2. پارامتر دوم: از نوع bool به اسم isSuitableForBox

این تابع قراره بررسی کنه:

اگر isSuitableForBox مقدارش true بود (با توجه به نوع واقعی shape که با type assertion می‌فهمیم) پیام مناسب رو چاپ کنه.

```
func printSuitabilityReport(shape Shape, isSuitableForBox bool) {
    if isSuitableForBox {
        square, isSquare := shape.(Square)
        if isSquare {
            fmt.Printf(
                "Square with SideLength=%.2f OK for box!\n",
                square.SideLength,
            )
        }

        rectangle, isRectangle := shape.(Rectangle)
        if isRectangle {
            fmt.Printf(
                "Rectangle with Width=%.2f and Height=%.2f OK for box!\n",
                rectangle.Width,
                rectangle.Height,
            )
        }

        circle, isCircle := shape.(Circle)
        if isCircle {
            fmt.Printf(
                "Circle with Radius=%.2f OK for box!\n",
                circle.Radius,
            )
        }
    }
}
```

بریم تستش کنیم

```
var square = Square{
    SideLength: 10,
}

isSquareSuitable := isSuitableForBox(square)
printSuitabilityReport(square, isSquareSuitable)

var rectangle = Rectangle{
    Width:  4,
    Height: 7,
}

isRectangleSuitable := isSuitableForBox(rectangle)
printSuitabilityReport(rectangle, isRectangleSuitable)

var circle = Circle{
    Radius: 4.0,
}

isCircleSuitable := isSuitableForBox(circle)
printSuitabilityReport(circle, isCircleSuitable)
```

خروجی

```
Square with SideLength=10.00 OK for box!
Rectangle with Width=4.00 and Height=7.00 OK for box!
Circle with Radius=4.00 OK for box!
```

استفاده از ساختار switch case برای type assertion

یکی از راه‌های type assertion استفاده از **type assertion** با if-else هست، ولی اگه تعداد نوع‌هایی که می‌خوای بررسی کنی زیاد باشه، اون روش ممکنه کد رو شلوغ و سخت‌خوان کنه.

اینجاست که **type switch** وارد میشه و قضیه رو خیلی تمیزتر و راحت‌تر حل می‌کنه. با استفاده از switch case می‌تونی انواع مختلف رو بررسی کنی و برای هر نوع، رفتار مخصوص خودش رو داشته باشی.

اینم یه نمونه ساده از استفاده‌ی type assertion با switch:

```
func checkType(i interface{}) {
    switch v := i.(type) {
    case int:
        fmt.Println("int:", v)
    case string:
        fmt.Println("string:", v)
    case bool:
        fmt.Println("bool:", v)
    default:
        fmt.Println("unknown type")
    }
}
```

i.(type) فقط داخل switch مجازه و برای بررسی نوع واقعی متغیر i استفاده می‌شه.

v := i.(type) باعث می‌شه مقدار i با نوع مشخص‌شده در هر case به v تبدیل بشه.

در هر case، می‌تونی با v مثل اون نوع رفتار کنی.

حالا میتونیم تابع printSuitibilityReport رو با ساختار switch case بازنویسی کنیم:

```
func printSuitabilityReport(shape Shape, isSuitableForBox bool) {
    if isSuitableForBox {
        switch s := shape.(type) {
            case Square:
                fmt.Printf(
                    "Square with SideLength=%.2f OK for box!\n",
                    s.SideLength,
                )
            case Rectangle:
              fmt.Printf(
                    "Rectangle with Width=%.2f and Height=%.2f OK for box!\n",
                    s.Width,
                    s.Height,
                )
            case Circle:
                fmt.Printf(
                    "Circle with Radius=%.2f OK for box!\n",
                    s.Radius,
                )
            default:
                fmt.Println("unknown type")
        }
    }
}
```

ما یه مقدار از نوع Shape داشتیم که خودش می‌تونه هر نوعی باشه (Square، Rectangle یا Circle) با استفاده از **type switch**، نوع واقعی اون رو در زمان اجرا تشخیص دادیم و بر اساسش پیام مناسب چاپ کردیم. این کار با switch s := shape.(type) انجام شد، که هم نوع رو بررسی می‌کنه و هم مقدار رو به متغیر s با نوع درست تبدیل می‌کنه.

مزایای استفاده از روش switch case

- **کد مرتب‌تر و خواناتر:** به‌جای چندین if پشت‌سرهم، همه‌ی بررسی‌ها توی یه ساختار منظم انجام می‌شن.
- **ایمن‌تر:** نیازی نیست دستی type assertion انجام بدی و نگران panic باشی.
- **دسترسی مستقیم به مقدار با نوع درست:** توی هر case ، متغیر s خودش از نوع مورد نظره، پس راحت می‌تونی باهاش کار کنی.

مفهوم interface{} یا any

فرض کن یک interface داریم به اسم EmptyInterface که هیچ متدی داخلش تعریف نشده:

```
type EmptyInterface interface{
}
```

خب حالا سوال:

چه نوع‌هایی این interface رو پیاده‌سازی کرده‌اند؟

جوابش ساده‌ست:

چون این interface **هیچ متدی** رو لیست نکرده، پس **هر نوع داده‌ای** در Go به‌صورت پیش‌فرض این interface رو پیاده‌سازی کرده!

یک تابع همه‌فن‌حریف

```
func getType(input EmptyInterface) string {
	return fmt.Sprintf("%T", input)
}
```

این یعنی تابع ما می‌تونه **هر نوع داده‌ای** رو بپذیره و نوعش رو برگردونه.

بیا تستش کنیم:

```
var circle = Circle{
    Radius: 4.0,
}

var numberOfShapes uint8 = 3

fmt.Println(getType(numberOfShapes))
fmt.Println(getType(circle))
```

خروجی

```
uint8
main.Circle
```

دقیقاً همونطور که فکر می‌کردیم، شد!

تابع getType با هر نوعی کار کرد چون همه نوع‌ها EmptyInterface رو پیاده‌سازی می‌کنن و برای این کار مجبور نیستن هیچ متد جدیدی تعریف کنن.

حذف تعریف اضافی type

وقتی می‌خوایم پارامتر ورودی یک تابع رو interface خالی بذاریم، لزومی نداره اول یک type تعریف کنیم و بعد ازش استفاده کنیم.

میتونیم مستقیم اینطوری بنویسیم:

```
func getType(input interface{}) string {
	return fmt.Sprintf("%T", input)
}
```

اینجا پارامتر input از نوع interface{} هست، یعنی همون interface خالی که گفتیم: "هیچ متدی نداره و همه‌چیزو قبول می‌کنه".

نتیجه؟ ساده‌تر، کوتاه‌تر و خواناتر!

هر وقت می‌خوای تابعی بنویسی که بتونه با هر نوع داده‌ای کار کنه، فقط کافیه نوع پارامترش رو interface{} بذاری.

بررسی دقیق تر توابع چاپ کتابخانه fmt

تا حالا به این فکر کردی که توابع چاپ کتابخونه‌ی fmt چطور می‌تونن **هر چیزی** رو چاپ کنن؟ حتی نوع‌هایی که بعداً تعریف می‌کنیم (مثل Circle)، حتی اگه براشون متد String() ننوشته باشیم؟

بیایم نگاهی به امضای تابع Println بندازیم:

```
func Println(a ...any) (n int, err error)
```

دو نکته‌ی مهم:

1. **ورودی variadic** یعنی میشه هر تعداد پارامتر خواستیم بدیم و داخل تابع به شکل slice دریافت میشه.
2. **نوع any** که در واقع همون interface{} هست!

از کجا بدونیم any همون interface{} هست؟

اگر بریم تو فایل builtin.go از کتابخونه‌های هسته‌ای Go اینو می‌بینیم:

```
// any is an alias for interface{} and is equivalent to interface{} in all ways.
type any = interface{}
```

یعنی any فقط یک **اسم مستعار** برای interface{} هست. پس هر جا any دیدی، مطمئن باش که همون interface خالی دوست‌داشتنی خودمونه.

خلاصه:

- interface{} یعنی: من هیچ متدی نمی‌خوام، هر چی داری بده.
- همه‌ی نوع‌ها به‌صورت پیش‌فرض این interface رو پیاده‌سازی می‌کنن.
- any فقط یه اسم دیگه برای همین مفهومه.

نمایشی کوچکی از قدرت پنهان واسط ها

یادت هست قبلاً گفتیم میشه برای هر نوع داده‌ای که تعریف می‌کنیم، یک متد String() نوشت و بعد با خیال راحت اون متغیر رو مستقیماً به توابع چاپ fmt داد و بدون فراخوانی هیچ متد خاصی، اطلاعاتش رو چاپ کرد؟

اما خب، تا حالا نگفتیم این جادو **چطور** اتفاق میفته. حالا که با مفهوم واسط‌ها (interface) آشنا شدیم، وقتشه پرده از این راز برداریم!

1-متد String() و اینترفیس fmt.Stringer

توی بسته‌ی استاندارد fmt یک interface مهم به اسم Stringer تعریف شده:

```
type Stringer interface {
    String() string
}
```

هر نوع داده‌ای که این متد String() رو پیاده کنه، یعنی **به طور ضمنی** (implicit) واسط fmt.Stringer رو هم پیاده کرده.

2-وقتی از fmt.Print و هم‌خانواده‌ها استفاده می‌کنیم چه میشه؟

توابعی مثل fmt.Print، fmt.Println یا fmt.Printf، پارامترهاشون رو به شکل عمومی (any یا interface{}) می‌گیرن.

وقتی یک مقدار بهشون میدی، اول میرن بررسی می‌کنن که آیا این مقدار اینترفیس fmt.Stringer رو پیاده کرده یا نه.

اگر بله :متد String() اون مقدار رو صدا می‌زنن و خروجیش رو چاپ می‌کنن.

اگر نه :مقدار رو به صورت پیش‌فرض (با %v یا بر اساس نوعش) چاپ می‌کنن.

3-این بررسی چطور انجام میشه؟

داخل کد fmt، ماجرا به شکل ساده‌شده اینطوریه:

```
if stringer, ok := value.(fmt.Stringer); ok {
    // اگر مقدار پیاده کننده‌ی Stringer بود، متد String صدا زده می‌شود
    output := stringer.String()
    // خروجی چاپ می‌شود
} else {
    // چاپ پیش فرض
}
```

این value.(fmt.Stringer) یک **type assertion** هست.

یعنی میگه: چک کن، این value آیا واقعاً از نوع fmt.Stringer هست یا نه.

4-چرا فقط با نوشتن ()String این اتفاق میفته؟

چون وقتی توی struct خودت متد ()String رو با همون امضا بنویسی، به طور ضمنی واسط (بدون اینکه صراحتاً اعلام کنی) fmt.Stringer رو پیاده کردی.

مثلاً:

```
func (s Square) String() string {
    return fmt.Printf(
        "Square with SideLength=%.2f OK for box!\n",
        square.SideLength,
    )
}
```

از این به بعد، وقتی بنویسی:

```
fmt.Println(square)
```

کتابخونه‌ی fmt میره نگاه می‌کنه که آیا square پیاده‌کننده‌ی Stringer هست یا نه.
اگه جواب بله باشه، پس متد ()String رو صدا میزنه و همون رو چاپ میکنه.

پیاده‌سازی Stringer برای چند نوع داده

حالا برای مثال، واسط Stringer رو برای Square، Rectangle و Circle پیاده می‌کنیم:

```
func (s Square) String() string {
    return fmt.Sprintf(
        "Square with SideLength=%.2f OK for box!",
        s.SideLength,
    )
}

func (r Rectangle) String() string {
    return fmt.Sprintf(
        "Rectangle with Width=%.2f and Height=%.2f OK for box!",
        r.Width,
        r.Height,
    )
}
func (c Circle) String() string {
    return fmt.Sprintf(
        "Circle with Radius=%.2f OK for box!",
        c.Radius,
    )
}
```

بریم تستش کنیم

```
var square = Square{
    SideLength: 10,
}

isSquareSuitable := isSuitableForBox(square)
if isSquareSuitable {
    fmt.Println(square)
}

var rectangle = Rectangle{
    Width:  4,
    Height: 7,
}

isRectangleSuitable := isSuitableForBox(rectangle)
if isRectangleSuitable {
    fmt.Println(rectangle)
}

var circle = Circle{
    Radius: 4.0,
}

isCircleSuitable := isSuitableForBox(circle)
if isCircleSuitable {
    fmt.Println(circle)
}
```

خروجی

```
Square with SideLength=10.00 OK for box!
Rectangle with Width=4.00 and Height=7.00 OK for box!
Circle with Radius=4.00 OK for box!
```

همونطور که دیدیم، فقط با ارسال یک متغیر به fmt.Println، بدون هیچ کار اضافه‌ای، خروجی خاص و شخصی‌سازی‌شده چاپ شد.

همه‌ی این کار به لطف **واسط‌ها** و **type assertion** پشت پرده اتفاق میفته.

روش های مختلف پیاده سازی interface

وقتی داریم یه interface تعریف می‌کنیم، تنها چیزی که توش می‌نویسیم **لیست متدهایی هست که برامون اهمیت داره**. برای هر متد فقط **اسم، ورودی و خروجی** رو مشخص می‌کنیم. اینجا اصلاً کاری نداریم که این متد قراره با **value receiver** پیاده‌سازی بشه یا با **pointer receiver**.

اگر یه نوع بیاد این متدها رو با value receiver پیاده کنه و یه نوع دیگه بیاد همون متدها رو با pointer receiver پیاده کنه، چی میشه؟

پیاده‌سازی متدهای یک interface به شکل value receiver

این همون حالتیه که تا الان تو مثال‌هامون داشتیم:

انواع **Square**، **Rectangle** و **Circle** رو تعریف کردیم و برای هر کدوم متدهای Perimeter و Area رو با **value receiver** نوشتیم.

رفتارش رو هم کامل دیدیم و فهمیدیم که بدون مشکل با interface ما سازگارن.

پیاده‌سازی متدهای یک interface به شکل pointer receiver

حالا بیایم برای تست، متدهای Perimeter و Area رو در نوع **Square** به صورت **pointer receiver** پیاده کنیم.

```
func (s *Square) Perimeter() float64 {
	return 4 * s.SideLength
}

func (s *Square) Area() float64 {
	return s.SideLength * s.SideLength
}
```

بریم که تستش کنیم

```
var square = Square{
    SideLength: 10,
}

isSquareSuitable := isSuitableForBox(square)
if isSquareSuitable {
    fmt.Println(square)
}
```

خروجی

```
cannot use square (variable of struct type Square) as Shape value in argument to
isSuitableForBox: Square does not implement Shape (method Area has pointer
receiver)
```

کامپایلر اخطار داد و اجازه نداد برنامه اجرا بشه. از نظر کامپایلر، ما یه متغیر Square دادیم، ولی این متغیر متدهای مورد نیاز واسط Shape رو **پیاده نکرده**!

اینجا معمولاً آدم گیج میشه و میگه:

مگه من همین الان این متدها رو ننوشتم؟!

بله نوشتی... ولی به شکل **pointer receiver**.

و اینجاست که می‌رسیم به یک مفهوم خیلی مهم تو Go به اسم **Method Set** که دلیل این رفتار رو کامل برات توضیح میده.

Method Set

Method set یعنی: مجموعه‌ی متدهایی که یه نوع (type) داره.
به عبارت دیگه، وقتی یه نوع T یا *T داریم، method set اون نوع مشخص می‌کنه چه متدهایی می‌تونه صدا زده بشه.

نکته مهم :این موضوع خیلی به این مربوط میشه که وقتی یه نوع می‌خواد یه **interface** رو پیاده‌سازی کنه، Go بررسی می‌کنه که method set اون نوع شامل همه متدهای interface هست یا نه.

روش‌های تعریف متد

دو روش کلی برای تعریف متد رو هر نوع دلخواه داریم.

به عنوان مثال برای نوع Square:

1-روش value receiver

```
func (s Square) ScaleUp(factor float64) {
    s.SideLength *= factor
}
```

2-روش pointer receiver

```
func (s *Square) ScaleDown(factor float64) {
    s.SideLength /= factor
}
```

Method Set برای انواع

الف) برای T (مقدار)

شامل تمام متدهایی که **با value receiver** تعریف شدن هست.

متدهای pointer receiver **تو method set نوع مقدار نمیاد**.

در مثال بالا در حالتی که متغیر از نوع Square باشه، Method set شامل متد ScaleUp میشه چون تنها این متد هست که به روش value receiver تعریف شده.

ب) برای *T (اشاره‌گر)

شامل تمام متدهای value receiver **و** pointer receiver هست.

یعنی pointer می‌تونه هم متدهایی که با value تعریف شدن رو صدا بزنه و هم متدهایی که با pointer تعریف شدن.

در مثال بالا در حالتی که متغیر از نوع *Square باشه، Method set شامل متدهای ScaleUp (چون به شکل value reciver تعریف شده) و ScaleDown (چون به شکل pointer receiver تعریف شده) میشه.

جدول راهنمای Method Set

نوع گیرنده متد	Method Set برای T (value)	Method Set برای *T (pointer)
value receiver	✓	✓
pointer receiver	✗	✓

حالا برگردیم به مثال قبلی...

دلیل اینکه کامپایلر خطا داد و نتونست متدهای Perimeter و Area رو شناسایی کنه این بود که **متغیر ما از نوع T(مقدار) بود**.
یادت باشه: وقتی متغیر از نوع مقدار باشه، **Method set فقط شامل متدهایی میشه که با value receiver تعریف شدن**.

حالا از اونجایی که هر دو متدی که واسط Shape انتظار داره، یعنی Perimeter و Area، **با pointer receiver** تعریف شدن، Method set تو این حالت برای Square خالیه.
و دقیقاً به همین دلیل کامپایلر اخطار داد.

راه حل

چون متدها با pointer receiver تعریف شدن، کافیه **متغیر از نوع T* (اشاره‌گر)** باشه تا این متدها تو Method set قرار بگیرن.

پس فقط کافیه آدرس square رو به تابع isSuitableForBox بدیم، اونوقت همه چی درست کار می‌کنه و خطایی نخواهیم داشت.

```
var square = Square{
    SideLength: 10,
}

isSquareSuitable := isSuitableForBox(&square)
if isSquareSuitable {
    fmt.Println(square)
}
```

خروجی

```
Square with SideLength=10.00 OK for box!
```

ترکیب interface ها

تعریف

تا اینجا یاد گرفتیم که چطور یه interface ساده مثل Shape داشته باشیم که میگه: هر نوعی که منو پیاده می‌کنه، باید بتونه محیط (Perimeter) و مساحت (Area) رو بده.

```
type Shape interface {
    Perimeter() float64
    Area() float64
}
```

حالا میایم یه interface دیگه تعریف میکنیم به اسم Scalable که میگه: هر نوعی که منو پیاده می‌کنه، باید دو متد ScaleUp و ScaleDown داشته باشه.

```
type Scalable interface {
    ScaleUp(float64)
    ScaleDown(float64)
}
```

حالا فرض کن تو جایی از برنامه شکلی نیاز داریم که **هم** شکل باشه (محیط و مساحت بده) **هم** قابلیت مقیاس‌پذیری داشته باشه.

راه‌حلش تعریف یه interface جدید که ترکیب هر دو باشه:

```
type ScalableShape interface {
    Scalable
    Shape
}
```

یعنی به عبارت دیگه، هر نوعی برای پیاده سازی ScalableShape ، باید متدهای واسط‌های Shape و Scalable رو پیاده کنه.

پیاده‌سازی

قبلاً برای Square متدهای Perimeter و Area رو تعریف کرده بودیم. حالا برای اینکه Square بتونه واسط ScalableShape رو پیاده کنه کافیه دو متد ScaleUp و ScaleDown رو بهش اضافه کنیم.

```
func (s *Square) ScaleUp(factor float64) {
    s.SideLength *= factor
}

func (s *Square) ScaleDown(factor float64) {
    s.SideLength /= factor
}
```

دلیل اینکه متدها رو به روش Pointer receiver تعریف کردیم این بود که تغییراتی که تو SideLength انجام میشه باید رو نسخه اصلی تاثیر بذاره.

استفاده

پیش تر یه تابع داشتیم به اسم isSuitableForBox که چک میکرد که آیا، این شکل با ابعاد فعلیش به درد جعبه شدن می‌خوره یا نه.

حالا میخوایم یه تابع بنویسیم که اگه شکل مناسب جعبه شدن نبود، امتحان کنه با بزرگ کردنش، آیا مناسب میشه یا نه:

```
func tryMakeItBox(sh ScalableShape, maxTries int, attempts int) (bool, int) {
    if isSuitableForBox(sh) {
        return true, attempts
    }

    if maxTries <= 0 {
        return false, attempts
    }

    sh.ScaleUp(2)
    return tryMakeItBox(sh, maxTries-1, attempts+1)
}
```

بریم که تستش کنیم

```
var square = Square{
    SideLength: 1,
}

if isSuitableForBox(&square) {
    fmt.Println("The square is already OK for box.")
} else {
    clonedSquare := square

    success, scaleCount := tryMakeItBox(&clonedSquare, 3, 0)

    if success {
        fmt.Printf("Scaled ×2 %d times — now it's box-ready!\n", scaleCount)
    } else {
        fmt.Printf("Scaled ×2 %d times — still not box-ready.\n", scaleCount)
    }
}
```

خروجی

```
Scaled ×2 3 times — now it's box-ready!
```

مزایای ترکیب Interface ها

خوانایی بالا: به جای اینکه یه interface غول‌پیکر بسازیم که همه‌چی رو توش بچپونیم، چندتا interface کوچیک می‌سازیم و هر جا لازم شد، ترکیبشون می‌کنیم.

انعطاف‌پذیری: می‌تونی یه type رو فقط با بخش‌هایی که لازم داره پیاده‌سازی کنی.

تطبیق با اصل Interface Segregation: یعنیinterface ها رو تا حد ممکن کوچک و تخصصی نگه داریم، ولی اگه جایی نیاز شد، ترکیبشون کنیم و یه قابلیت جامع‌تر بسازیم.

توصیه: همیشه interface هاتو تا جایی که میشه کوچیک و تک‌وظیفه‌ای طراحی کن. اینطوری در آینده، خیلی راحت می‌تونی مثل لِگو، چندتا رو کنار هم بذاری و ترکیب‌های قدرتمند بسازی.

استفاده از متغیر از نوع interface

گاهی وقت‌ها توی برنامه‌نویسی، از قبل نمی‌دونیم که نوع دقیق داده‌ای که قراره باهاش کار کنیم چی خواهد بود. مثلا فرض کن می‌خوایم برنامه‌ای بنویسیم که از کاربر بپرسه:
چه شکلی رو میخوای رسم کنم؟ مربع؟ دایره؟ یا شاید یه مستطیل؟

چون تصمیم‌گیری به زمان اجرا (runtime) موکول شده و ما موقع نوشتن کد نوع دقیق رو نمی‌دونیم، بهترین راه اینه که یک **متغیر از نوع interface** بسازیم. این متغیر می‌تونه هر نوعی رو در خودش نگه داره، به شرط اینکه اون نوع، متدهای موردنیاز این interface رو پیاده‌سازی کرده باشه.

به این نوع‌هایی که interface رو پیاده‌سازی می‌کنن، اصطلاحاً **Concrete Type** می‌گیم.

به روزرسانی واسط

قبلاً یه interface داشتیم به اسم Shape که فقط محیط (Perimeter) و مساحت (Area) رو محاسبه می‌کرد.
حالا می‌خوایم قابلیت رسم شکل رو هم بهش اضافه کنیم، پس متد جدیدی به اسم Draw رو بهش اضافه می‌کنیم:

```
type Shape interface {
    Perimeter() float64
    Area() float64
    Draw()
}
```

از این به بعد، هر نوعی که بخواد Shape باشه، باید حتماً متد Draw رو هم داشته باشه.

پیاده‌سازی متد Draw

برای مربع (Square)

```
func (s Square) Draw() {
    for i := 0; i < int(s.SideLength); i++ {
        for j := 0; j < int(s.SideLength); j++ {
            fmt.Print("*")
        }
        fmt.Println()
    }
}
```

برای دایره (Circle)

```
func (c Circle) Draw() {
    r := int(c.Radius)

    for y := -r; y <= r; y++ {
        for x := -r; x <= r; x++ {
            if float64(x*x+y*y) <= c.Radius*c.Radius {
                fmt.Print("*")
            } else {
                fmt.Print(" ")
            }
        }
        fmt.Println()
    }
}
```

بریم که تستش کنیم

```
func getFloat(prompt string) float64 {
    fmt.Print(prompt)
    var input string
    fmt.Scan(&input)
    val, err := strconv.ParseFloat(input, 64)
    if err != nil {
        fmt.Println("Invalid number!")
        os.Exit(1)
    }
    return val
}
```

```
var shape Shape

fmt.Print("Choose shape (1: Square, 2: Circle): ")
var choice string
fmt.Scan(&choice)

switch choice {
case "1":
    side := getFloat("Enter side length: ")
    shape = Square{SideLength: side}
case "2":
    radius := getFloat("Enter radius: ")
    shape = Circle{Radius: radius}
default:
    fmt.Println("Invalid choice!")
    os.Exit(1)
}

fmt.Println("\nDrawing shape:\n")
shape.Draw()
```

مزیت این روش

با این کار، فقط با **یک متغیر (shape)** می‌تونیم هر شکل هندسی رو مدیریت کنیم. این یعنی کد ما **انعطاف‌پذیرتر** و **قابل‌گسترش‌تر** میشه. هر وقت بخوایم شکل جدیدی اضافه کنیم، فقط کافیه اون شکل جدید متدهای Shape رو پیاده کنه، بدون اینکه نیاز باشه منطق ورودی‌گیری یا رسم رو تغییر بدیم.

نمای درونی یک Interface

تا اینجا ما interface رو از بیرون دیدیم:

یه موجود جادویی که فقط می‌گه "هر نوعی که این متدهای مشخص رو داشته باشه، من می‌تونم بهش اشاره کنم".

اما بیاید یک بار درپوشش رو برداریم و ببینیم توی دل interface واقعاً چه خبره.

دو جزء اصلی درون یک interface

وقتی شما یک متغیر از نوع interface دارید، پشت صحنه Go این متغیر رو به شکل یک ساختار دو بخشی ذخیره می‌کنه:

نوع (Type)

این بخش مشخص می‌کنه که interface الان داره به چه نوع واقعی (Concrete Type) اشاره می‌کنه.

مثلا اگه شما یک Shape (که interface هست) رو به یک Rectangle نسبت بدید، این قسمت میگه: "نوع واقعی من Rectangle هست"

مقدار یا آدرس (Value)

این بخش یا خود مقدار متغیر رو نگه می‌داره (برای انواع مقداری مثل int، struct کوچک و...) یا یک **آدرس (Pointer)** به اون مقدار رو ذخیره می‌کنه (برای struct های بزرگ یا وقتی که اشاره‌گر داده‌اید).

به زبان ساده:

یک interface = **[نوع واقعی] + [مقدار یا آدرسش]**

چرا این مهمه؟

چون باعث میشه که:

Go بتونه در زمان اجرا بفهمه کدوم متدها رو باید صدا بزنه (بر اساس نوع واقعی).

یا اگه نیاز باشه بر اساس نوعی که interface داره بهش اشاره می‌کنه برنامه‌ای بنویسی که رفتار متفاوتی داشته باشه. میتونی از قابلیت‌هایی مثل **type assertion** یا **type switch** استفاده کنی تا نوع واقعی رو دوباره بیرون بکشید.

مثال

```
var s Shape
r := Rectangle{Width: 5, Height: 3}
s = r


fmt.Printf("Interface holds: Type=%T, Value=%v\n", s, s)
```

خروجی

```
Interface holds: Type=main.Rectangle, Value={5, 3}
```

این دقیقاً نشون میده که interface الان:

- **نوعش:** main.Rectangle
- **مقدارش:** {5 3}

nil interface

تا الان یاد گرفتیم که هر interface در Go در واقع پشت‌صحنه از **دو بخش** ساخته شده:

1. **نوع واقعی (Concrete Type)**
2. **مقدار یا آدرس اون نوع**

اگه یکی یا هر دوی این بخش‌ها nil باشن چی میشه؟

دو حالت اصلی وجود داره

Interface-1 کاملاً nil (nil interface)

یعنی **هم نوعش nil باشه و هم مقدارش nil.**
به زبان ساده interface هنوز مقداردهی نشده و به هیچ چیزی اشاره نمی‌کنه.

مثال

```
var s interface{}
fmt.Println(s == nil)
```

خروجی

```
true
```

اینجا Go می‌دونه که s یک interface هست، ولی هنوز هیچ نوع واقعی و مقداری توش قرار نگرفته، پس s == nil درست (true) هست.

Interface-2 با نوع مشخص ولی مقدار nil

اینجا داستان جالب میشه!

ممکنه interface به یک **نوع خاص** اشاره کنه، اما **مقدار اون نوع nil باشه**.

مثال:

```
var p *int = nil
var s interface{} = p

fmt.Println(s == nil)
fmt.Printf("Type: %T, Value: %v\n", s, s)
```

خروجی:

```
false
Type: *int, Value: <nil>
```

چرا false؟

چون بخش "نوع" پر شده (*int) و Go میگه: "این interface به یه نوع مشخص اشاره می‌کنه، حتی اگه مقدار اون nil باشه".

پس، این دیگه nil کامل محسوب نمیشه.

نکته مهم

نوع nil باشه و **مقدار nil باشه** میشه نتیجه گرفت **متغیر واسط مقدارش nil هست**

نوع nil نباشه ولی **مقدار nil باشه** واسط داره به **یک نوع خاص اشاره می‌کنه** (با وجود اینکه مقدار اون نوع nil هست) در نتیجه **متغیر واسط مقدارش nil نیست**

یک قیاس ساده:

فرض کنید interface مثل یک جعبه است:

اگه جعبه خالی باشه (هیچ برچسب نوع نداره و چیزی توش نیست) این همون nil interface هست.

اگه جعبه برچسب داشته باشه ("این جعبه برای پیچ‌گوشتیه") ولی توش هیچی نباشه، این دیگه nil کامل نیست، چون هویت جعبه مشخصه.

تمرین 1: اعتبارسنجی کلمه عبور

مرحله 1: تعریف واسط

یه **واسط (interface)** به اسم PasswordValidator بساز که فقط یه متد داشته باشه:

```
Validate(password string) bool
```

هدف اینه که هر نوع اعتبارسنجی رمز با این واسط سازگار باشه.

مرحله 2: دو نوع پیاده‌سازی

1-سخت‌گیر (StrongValidator)

شرایطش اینه که رمز عبور **واقعاً قوی** باشه:

- حداقل 10 کاراکتر
- حداقل یک حرف انگلیسی
- حداقل یک عدد
- حداقل یکی از کاراکترهای ویژه: @ # _

2-آسان‌تر (EasyValidator)

یه نسخه ساده‌تر که شرایطش کمتره:

- حداقل 8 کاراکتر
- حداقل یک حرف انگلیسی
- حداقل یک عدد

مرحله 3: تابع اعتبارسنجی عمومی

یه تابع بنویس به اسم CheckPassword که:

یه رمز عبور (string) می‌گیره

یه پارامتر از نوع واسط (PasswordValidator) می‌گیره

چک می‌کنه **واسط nil نباشه**

بعد با استفاده از متد Validate رمز عبور رو بررسی کنه و نتیجه (true/false) بده

مرحله 4: تست برنامه

رمز عبورهای زیر رو با هر دو validator تستشون کن.

"Pass123"

"StrongPass1@"

"weakpass"

"12345678"

"MyPass_12"

"VeryStrong#1234"

اون رمزهایی که با هیچ‌کدوم از validator ها pass نمیشن رو تو خروجی چاپ کن.

مرحله 5: روش جایگزین بدون واسط

یه تابع جدید بساز با این مشخصات:

اسم تابع پیشنهادی: validateDirect

ورودی: **رمز عبور و نوع اعتبارسنجی (hard یا easy)**

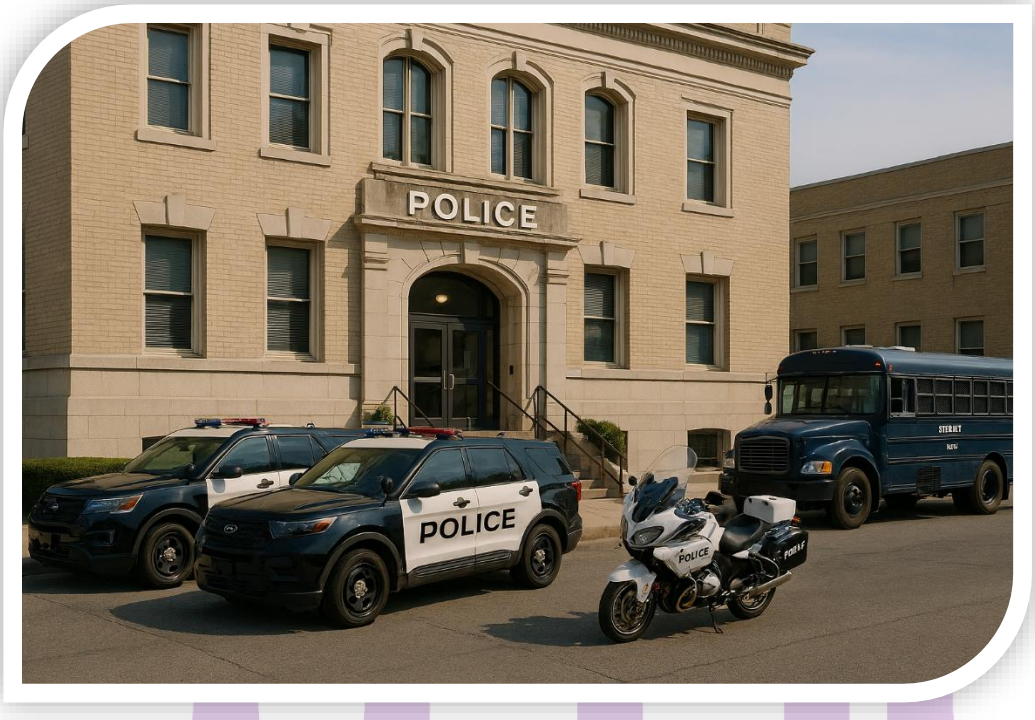
خروجی: معتبر بودن یا نبودن رمز عبور

مرحله 6: مقایسه دو روش

مقایسه کن که روش **واسطی** و روش **مستقیم با پارامتر** چه تفاوتی دارن:

- انعطاف‌پذیری
- قابلیت گسترش
- سادگی پیاده‌سازی
- خوانایی کد

تمرین 2: انتخاب وسیله نقلیه مناسب پلیس



مرحله 1: تعریف واسط

یه **interface** به اسم Vehicle بساز.

متدها:

متد MaxSpeed() int برای دریافت بیشترین سرعت وسیله نقلیه(km/h)

متد NumWheels() int برای دریافت تعداد چرخ‌ها

مرحله 2: انواع وسیله نقلیه پایه

سه نوع پایه از وسیله نقلیه بساز: Motorcycle، Car و Bus

هر کدوم باید متدهای واسط Vehicle رو پیاده کنه

مرحله 3: مدل‌های سفارشی باcomposition

برای هر مدل خاص از وسیله نقلیه، یه type جدید بساز که struct پایه رو **داشته باشه**.

مثال برای موتور سیکلت: YamahaR1Motorcycle

مثال برای خودرو: BMWX3Car

مثال برای اتوبوس: Volvo9700Bus

مرحله 4: تابع بررسی مناسب بودن برای پلیس

اسم تابع: IsSuitableForPolice

ورودی: یک Vehicle

خروجی: آیا وسیله برای پلیس مناسب است یا نه

شرایط:

- اگر وسیله دو چرخ دارد و سرعت آن حداقل 200 km/h باشد: مناسب هست.
- اگر وسیله چهار چرخ دارد و سرعت آن حداقل 220 km/h باشد: مناسب هست.
- اگر وسیله شش چرخ دارد و سرعت آن حداقل 150 km/h باشد: مناسب هست.
- در غیر این صورت: مناسب نیست.

مرحله 5: تست برنامه

یه slice از وسایل نقلیه بساز و مدل‌های مختلف رو داخلش قرار بده

دونه دونه بفرست به تابع IsSuitableForPolice

فقط اون‌هایی که مناسب پلیس هستن رو چاپ کن

مرحله 6: روش جایگزین بدون واسط

یه تابع جدید بساز با این مشخصات:

اسم تابع پیشنهادی: IsSuitableDirect

ورودی :**تعداد چرخ و حداکثر سرعت**

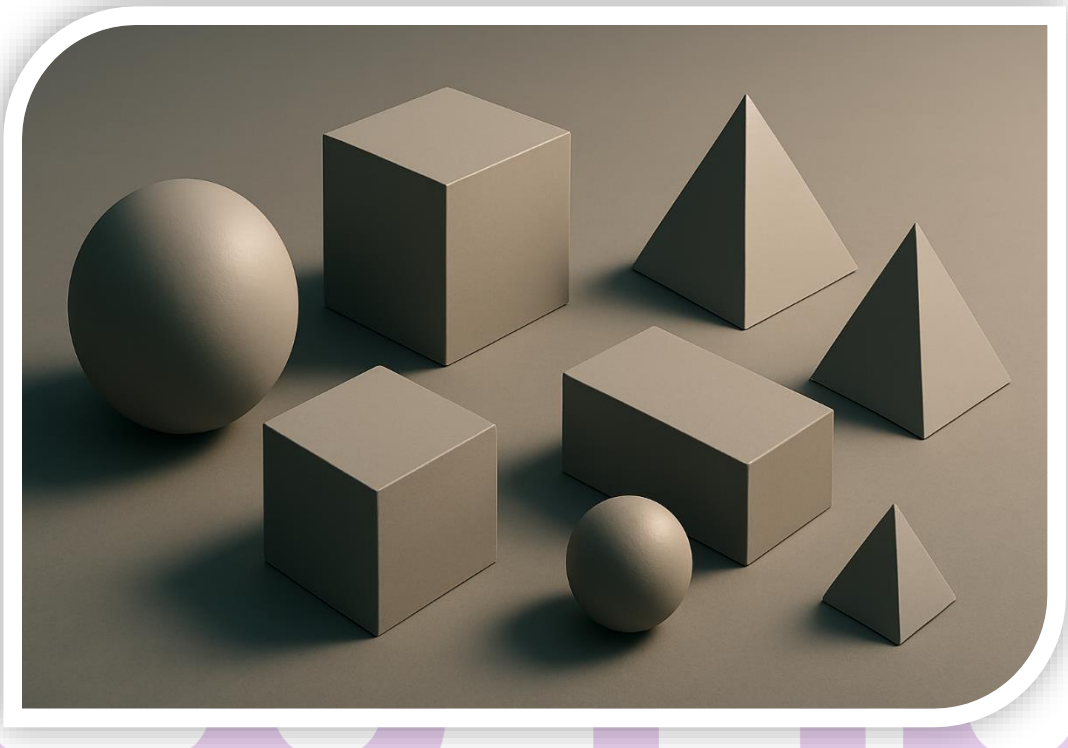
خروجی: مناسب بودن یا نبودن برای وسیله نقلیه پلیس.

مرحله 7: مقایسه دو روش

مقایسه کن که روش **واسطی** و روش **مستقیم با پارامتر** چه تفاوتی دارن:

- انعطاف‌پذیری
- قابلیت گسترش
- سادگی پیاده‌سازی
- خوانایی کد

تمرین 3: اشکال هندسی 3 بعدی



مرحله 1: تعریف واسط

یه **interface**به اسم Shape3D بساز.

متدها:

متد float64 ()Volume برای محاسبه حجم شکل

متد float64 ()SurfaceArea برای محاسبه مساحت سطح شکل

مرحله 2: تعریف اشکال

Cube برای مکعب

Sphere برای کُره

هر کدوم باید متدهای واسط Shape3D رو پیاده کنه.

مرحله 3: تابع محاسبه ویژگی‌ها

اسم تابع پیشنهادی: PrintShapeDetails

ورودی: یک Shape3D

خروجی: چاپ حجم و مساحت سطح

نکته: قبل از استفاده حتما بررسی کن که ورودی nil نباشه

مرحله 4: تست برنامه

یه slice از اشکال بساز و مدل‌های مختلف رو داخلش قرار بده

دونه دونه بفرست به تابع PrintShapeDetails

نتایج حجم و مساحت هر شکل چاپ بشه