مقدمه

تو پروژه‌ای که تو جلسه‌ی قبل با هم نوشتیم، همه توابع رو چپوندیم تو یه فایل. نتیجه چی شد؟ فایل main.go شد حدود 250 خط! خوندنش سخت شد، تغییر دادنش سخت‌تر. برای اینکه یه تابع خاص رو پیدا کنیم، باید کلی اسکرول کنیم بالا پایین تا برسیم به جایی که می‌خوایم.

اینجاست که زبان Go یه ابزار خیلی به‌دردبخور در اختیارمون می‌ذاره به اسم **پکیج** و یه چیز دیگه به اسم ماژول که بعداً درباره‌ش صحبت می‌کنیم.پکیج دقیقاً همون چیزیه که بهش نیاز داریم برای اینکه کدهامون رو مرتب و قابل مدیریت کنیم.

پکیج یعنی چی؟

هر پکیج مثل یه قفسه‌ست که توش کدهایی با یه موضوع خاص کنار هم چیده شدن.
مثلاً تو پروژه‌ی توییتر، می‌تونیم یه پکیج مخصوص چاپ توییت‌ها بسازیم، یکی برای فیلتر کردن و جستجو توی توییت‌ها، یکی برای مدیریت کاربران و...
اینجوری هر بخش از برنامه سر جای خودش قرار می‌گیره و لازم نیست همه‌چی تو یه فایل بزرگ قاطی پاتی بشه.

تعریف رسمی‌تر

پکیج یه واحد منطقی برای سازماندهی و گروه‌بندی کده.

هر فایلی که اولش نوشته باشه:

```
package something
```

داره می‌گه که "من عضوی از پکیج something هستم"

ساختن یه پکیج جدید

برای ساختن یه پکیج جدید، فقط کافیه:

1. یه فولدر جدید با اسم دلخواه (که تکراری نباشه) تو پروژه بسازی.
2. فایل‌های go. رو بندازی تو اون فولدر.
3. بالای هر فایل بنویسی:

```
package name
```

که name همون اسم پکیجت باشه. همه‌ی فایل‌های اون فولدر باید همین اسم پکیج رو داشته باشن، چون قراره همه با هم یه خانواده‌ی واحد رو تشکیل بدن.

ساخت پکیج phonebook

فرض کن ساختار پروژه‌مون اینه:

```
exercise/
├── main.go
```

حالا میایم یه فولدر جدید به اسم phone_book می‌سازیم و توش 4 تا فایل می‌ریزیم:

```
exercise/
├── main.go
└── phone_book/
    ├── repo.go
    ├── store.go
    ├── table.go
    └── report.go
```

توی همه‌ی این فایل‌ها، خط اول باید این باشه:

```
package phonebook
```

یعنی همه‌شون عضوی از یه پکیج به اسم phonebook هستن.

محتوای فایل‌ها:

محتوا هر فایل در زیر مشخص شده است

exercise/phone_book/repo.go

```
package phonebook

func New() map[string]string {
	return make(map[string]string)
}
```

exercise/phone_book/store.go

```
package phonebook

func Store(repo map[string]string, name string, phoneNumber string) bool {
	if len(phoneNumber) != 13 {
		return false
	}

	repo[name] = phoneNumber
	return true
}
```

exercise/phone_book/table.go

```
package phonebook

import "fmt"
import "strings"

func printRow(values []string, width int) {
	fmt.Print("|")
	for _, val := range values {
		fmt.Print(padCenter(val, width) + "|")
	}
	fmt.Println()
}

func printLine(columns int, width int) {
	line := "+"
	for i := 0; i < columns; i++ {
		line += strings.Repeat("-", width) + "+"
	}
	fmt.Println(line)
}

func padCenter(input string, width int) string {
	if len(input) >= width {
		return input
	}
	left := (width - len(input)) / 2
	right := width - len(input) - left
	return strings.Repeat(" ", left) + input + strings.Repeat(" ", right)
}
```

exercise/phone_book/report.go

```
package phonebook


func Report(repo map[string]string) {
	const colWidth = 17
	headers := []string{"NAME", "PHONE NUMBER"}

	printLine(len(headers), colWidth)
	printRow(headers, colWidth)
	printLine(len(headers), colWidth)

	for name, phone := range repo {
		printRow([]string{name, phone}, colWidth)
		printLine(len(headers), colWidth)
	}
}
```

خب حالا این 4 تا فایل چی کار می‌کنن؟

- repo.go یه دفترچه تلفن جدید از نوع map می‌سازه.
- store.go اسم و شماره تلفن رو داخل map ذخیره می‌کنه، فقط اگه شماره 13 رقم باشه.
- table.go توابع کمکی برای ساخت جدول‌های زیبا توی ترمینال داره (مثل کشیدن خط و ردیف و وسط‌چین کردن).
- report.go اطلاعات دفترچه تلفن رو به شکل جدول چاپ می‌کنه (و از توابع table.go کمک می‌گیره).

چرا پکیج ساختیم؟

می‌شد همه‌ی این کدها رو تو main.go بنویسیم.

ولی اون موقع:

- main.go خیلی شلوغ می‌شد.
- پیدا کردن یه تابع خاص سخت می‌شد.
- همکاری روی پروژه سخت‌تر می‌شد.
- تست‌نویسی و استفاده‌ی مجدد از کدها تو پروژه‌های دیگه تقریباً غیرممکن می‌شد.

اما وقتی کدها تو پکیج‌های مختلف تقسیم بشن، کلی مزیت داره:

سازمان‌دهی بهتر :هر چیزی سر جای خودش قرار می‌گیره.
قابلیت استفاده مجدد :می‌تونی یه پکیج رو تو پروژه‌های دیگه هم استفاده کنی.
مخفی‌سازی جزئیات داخلی :مثلاً بعضی توابع رو فقط برای داخل پکیج بنویسی (با حروف کوچیک)، و نذاری بقیه استفاده کنن.

پکیج main، درِ ورودی برنامه

تو زبان Go یه پکیج خاص داریم به اسم main این پکیج اون‌قدر خاصه که اجرای برنامه از همین‌جا شروع می‌شه.

فرقی نمی‌کنه این پکیج کجا تعریف شده باشه، فقط کافیه **حداقل یک فایل** تو پروژه‌مون باشه که با این خط شروع شده باشه:

```
package main
```

اما یه نکته‌ی مهم این وسط هست:
نباید تو همون مسیری که فایل main.go رو ساختی، فایل دیگه‌ای باشه که پکیج متفاوتی داشته باشه.

مثال از یه اشتباه رایج:

اگه تو روت پروژه‌ات یه فایل main.go داشته باشی که پکیجش main باشه، و یه فایل دیگه به اسم utils.go که پکیجش utils باشه، برنامه کامپایل نمی‌شه

چرا؟ چون تو زبان Go **هر فولدر فقط می‌تونه فایل‌هایی با یه پکیج مشترک داشته باشه.**
یعنی نمی‌تونی تو یه فولدر چندتا پکیج مختلف تعریف کنی.

روت پروژه هم یه فولدر حساب میشه، پس اونم از این قانون مستثنا نیست.

چرا پکیج main خاصه؟

چون نقطه‌ی شروع برنامه همینه.
وقتی Go می‌خواد برنامه رو اجرا کنه:

1. دنبال پکیجی به اسم main می‌گرده.
2. بعدش دنبال تابعی به اسم main داخل اون پکیج می‌گرده.
3. و از همون‌جا اجرا شروع می‌شه.

یه‌جورایی این پکیج مثل درِ ورودی یه سالن کنسرت می‌مونه.
اگه از در وارد نشی، هیچ‌وقت اجرای بی نظیرو گروه موسیقی رو نمیشنوی!

قوانین نام‌گذاری پکیج‌ها

شاید تو مثال دفترچه تلفن که قبلاً زدیم برات سوال شده باشه:
آیا اسم فولدر و اسم پکیج باید خاص باشن؟
پاسخ: آره، چندتا قانون و نکته هست که بد نیست بدونی:

پکیج:

- اسم پکیج بهتره با حروف کوچیک نوشته بشه.
- از کاراکترهای خاص مثل فاصله، نقطه یا علامت سؤال استفاده نکن.

فولدر:

- اسم فولدر می‌تونه هر چیزی باشه، ولی بهتره با اسم پکیج هماهنگ باشه.
- از کاراکترهای خاص مثل فاصله، نقطه یا علامت سؤال استفاده نکن.
- در صورتی که اسم چند کلمه ای بود از _ برای جدا کردن کلمات استفاده کن.

فایل:

- اسم فایل‌ها مهم نیست چی باشن، فقط باید پکیج در خط اولشون مشخص شده باشه.
- از کاراکترهای خاص مثل فاصله، نقطه یا علامت سؤال استفاده نکن.
- در صورتی که اسم چند کلمه ای بود از _ برای جدا کردن کلمات استفاده کن.

استفاده از پکیج

همون‌طور که گفتیم، اجرای برنامه از پکیج main شروع میشه. حالا فرض کن بخوایم از پکیج phonebook استفاده کنیم. از اونجایی که برنامه از پکیج main شروع میشه پس باید از پکیج phonebook تو پکیج main استفاده کنیم.

قدم اول import :کردن پکیج

باید تو main.go درست بعد از package main این خط رو بنویسیم:

```
import “exercise/phone_book”
```

اینجا exercise اسم فولدر اصلی پروژه‌مونه، و phone_book هم اسم فولدریه که پکیجمون توشه. Go می‌فهمه که این پکیج کجاست، و کدهاشو میاره.

نکته: مسیر پکیج باید داخل " " باشه و از ریشه‌ی پروژه نوشته بشه.

قدم دوم: استفاده از توابع پکیج ایمپورت شده

بعد از import حالا می‌تونیم تو هر تابعی از پکیج main توابع پکیج phonebook رو اینطوری صدا بزنیم:

exercise/main.go

```
package main


import "exercise/phone_book"


func main() {
	phoneBookRepo := phonebook.New()
	phonebook.Store(phoneBookRepo, "Arian", "+989051020304")
	phonebook.Store(phoneBookRepo, "Neda", "+989051010202")
	phonebook.Report(phoneBookRepo)
}
```

توابع رو با .phonebook صدا می‌زنیم. هیچ تفاوتی با توابع عادی ندارن؛ می‌تونیم ورودی بدیم، خروجی بگیریم، و بریزیمش تو متغیر.

سطح دسترسی ها در پکیج

حالا بیا یه سناریوی جالب بررسی کنیم:

ما تو پکیج phonebook یه فایل داشتیم به اسم table.go که توابعی مثل printLine و printRow توش بودن.

اگه بخوایم تو main.go از این توابع استفاده کنیم، چی میشه؟

به مثال صفحه بعد توجه کن

exercise/main.go

```
package main

import "exercise/phone_book"

func main() {
	phoneBookRepo := phonebook.New()
	phonebook.Store(phoneBookRepo, "Arian", "+989051020304")
	phonebook.Store(phoneBookRepo, "Neda", "+989051010202")
	phonebook.Report(phoneBookRepo)

	colorInterpreter()
}

func colorInterpreter() {
	var repo = map[string]string{
		"Red":    "Passion",
		"Blue":   "Calm",
	}

	const colWidth = 15
	headers := []string{"Color", "Vibe"}

	phonebook.printLine(len(headers), colWidth)
	phonebook.printRow(headers, colWidth)
	phonebook.printLine(len(headers), colWidth)

	for color, vibe := range repo {
		phonebook.printRow([]string{color, vibe}, colWidth)
		phonebook.printLine(len(headers), colWidth)
	}
}
```

این برنامه کامپایل نمیشه و خطای هنگام کامپایل دریافت میکنیم.

```
# exercise
.\main.go:32:12: undefined: phonebook.printLine
.\main.go:33:12: undefined: phonebook.printRow
.\main.go:34:12: undefined: phonebook.printLine
.\main.go:37:13: undefined: phonebook.printRow
.\main.go:38:13: undefined: phonebook.printLine
```

اما دلیل این موضوع چیه؟

یادت میاد تو دلایلی که برای استفاده از پکیج گفتیم، مورد سوم چی بود؟

"**مخفی‌سازی جزئیات داخلی یا همون encapsulation**"

حالا وقتشه این مفهومو عمیق‌تر بررسی کنیم!

کنترل سطح دسترسی

وقتی داری داخل یه پکیج تابع یا متغیر یا حتی یه ثابت تعریف می‌کنی، می‌تونی مشخص کنی که آیا اون عنصر از **بیرون پکیج هم قابل دسترسی باشه یا نه**.

نام گذاری	قابل دسترسی از بیرون از پکیج
PrintLine	بله (چون با حرف بزرگ شروع شده)
printLine	نه (چون با حرف کوچیکه)

اگه اسم تابع، متغیر یا ثابت با **حرف بزرگ** شروع بشه، از بیرون پکیج هم می‌تونی بهش دسترسی داشته باشی. اگه با **حرف کوچیک** شروع بشه، فقط **داخل همون پکیج** قابل استفادست.

مفهوم Encapsulation

یعنی تو می‌تونی تصمیم بگیری که کدهایی که نوشتی:

- فقط داخل خود پکیج استفاده بشن (خصوصی)
- یا اینکه از بیرون هم در دسترس باشن (عمومی)

این باعث می‌شه یه‌جور **مرز حفاظتی** برای کدت بسازی. چیزایی که داخلی‌ان و نباید کسی از بیرون بهشون دست بزنه، قایم می‌کنی. به این می‌گن **encapsulation** یا همون مخفی‌سازی

مزیت‌های Encapsulation

وقتی درست از این قابلیت استفاده کنی، کلی سود می‌بری:

1. **کدها تمیزتر و مرتب‌تر می‌شن**

   چون فقط چیزهایی که واقعاً لازمه در دسترس بیرون قرار می‌گیرن.

2. **انعطاف در طراحی می‌گیری**

   می‌تونی پیاده‌سازی داخلی رو بدون اینکه برنامه‌های دیگه خراب شن، تغییر بدی.

3. **اشتباهات احتمالی کمتر می‌شن**

   چون توابع و متغیرهایی که نباید دستکاری بشن، اصلاً در دسترس نیستن.

4. **امنیت کد بالاتر می‌ره**

   چون یه‌جورایی «در پشتی» براش ساختی که بقیه نمی‌تونن ازش بیان تو!

حالا تو مثال phonebook چی شد؟

وقتی داشتم پکیج phonebook رو طراحی می‌کردم، تصمیم گرفتم توابع مربوط به چاپ جدول مثل printLine و printRow رو **با حروف کوچیک** بنویسم.

چرا؟

چون دلم نمی‌خواست هیچ‌کدوم از بیرون پکیج بتونه مستقیماً بهشون دست بزنه.

این تصمیم کاملاً **طراحی‌ای و سلیقه‌ای** بود. یعنی طبق منطق ذهنی خودم، به این نتیجه رسیدم که بهتره این توابع فقط داخل phonebook قابل استفاده باشن.

کی باید تابع یا متغیر رو encapsulate کنیم؟

اگه بخوای بدونی که کِی باید یه عنصر رو عمومی بذاری و کِی مخفی، این دستورالعمل‌ها رو همیشه دم دست داشته باش:

سوال	اگه جوابش «بله» بود...
آیا این تابع یا متغیر فقط برای استفاده داخلی همین پکیجه؟	پس **حرف کوچیک** باشه
آیا لازمه پکیج‌های دیگه هم بهش دسترسی داشته باشن؟	پس **حرف بزرگ** باشه
اگه فردا خواستم پیاده‌سازی داخلی رو تغییر بدم، نباید بقیه برنامه‌ها خراب بشن؟	پس پنهونش کن

تعریف پکیج جدید برای Table

خب حالا که فهمیدیم ممکنه از توابع printRow، printLine و padCenter هم تو پکیج main استفاده کنیم و هم شاید جاهای دیگه، منطقیه که بیایم براشون یه پکیج مستقل بسازیم به اسم table و این توابع رو منتقل کنیم اونجا.

اما یه سوال!

چرا همین توابعی که تو پکیج phonebook تعریف کردیم رو با حروف بزرگ ننویسیم که از بیرونم بهشون دسترسی داشته باشیم؟!

یعنی مثلاً PrintRow، PrintLine و...

اینکه خیلی ساده‌تر به نظر می‌رسه، نه؟

اگه اینجوری فکر کردی، در نگاه اول شاید حق با تو باشه، ولی بیا با هم بررسی کنیم که چرا این کار رو **نمی‌کنیم**:

چرا بهتره این توابع برن تو یه پکیج جدا؟

1. این توابع ماهیت عمومی‌تری دارن و به دفترچه‌تلفن (phonebook) ربطی ندارن. کارشون صرفاً رسم جدول و چیدمان متنه. پس بهتره تو یه جای عمومی‌تر تعریف بشن
2. اگه روزی بخوایم از این توابع تو پروژه دیگه‌ای یا تو پکیج دیگه‌ای هم استفاده کنیم، نمی‌خوایم وابسته به phonebook باشن. با این کار قابلیت **استفاده مجدد** (reusability) بالا میره.
3. پکیج phonebook فقط باید مسئول چیزایی باشه که مستقیماً به دفترچه تلفن مربوطه.
4. کد ساختارمندتر و خواناتر میشه. چون توابع مشابه کنار هم قرار می‌گیرن.

تعریف پکیج

1. یه فولدر جدید توی روت پروژه درست می‌کنیم به اسم table
2. داخلش یه فایل می‌سازیم به نام draw.go و اون 3 تا تابعی که قبلاً داشتیم رو بهش منتقل می‌کنیم.

اما یه نکته‌ی مهم!

این بار تابع‌هایی که می‌خوایم از بیرون در دسترس باشن رو با حرف بزرگ می‌نویسیم:
PrintRow، PrintLine

و اون تابعی که فقط قرار هست داخلی استفاده بشه رو با حرف کوچیک نگه می‌داریم:
padCenter

چرا padCenter رو همچنان مخفی نگه داشتیم؟

چون طبق منطق برنامه، بیرون از table نیازی بهش نداریم. این همون **encapsulation**ـه که قبلاً با هم یاد گرفتیم.

ساختار پروژه الان این شکلی شده

```
exercise
├── main.go
├── phone_book
│   ├── repo.go
│   ├── store.go
│   └── report.go
└── table
    └── draw.go
```

و فایل table.go که قبلاً تو phone_book بود رو پاک کردیم چون دیگه بهش نیازی نیست.

exercise/table/draw.go

```
package table

import "fmt"
import "strings"

func PrintRow(values []string, width int) {
	fmt.Print("|")
	for _, val := range values {
		fmt.Print(padCenter(val, width) + "|")
	}
	fmt.Println()
}

func PrintLine(columns int, width int) {
	line := "+"
	for i := 0; i < columns; i++ {
		line += strings.Repeat("-", width) + "+"
	}
	fmt.Println(line)
}

func padCenter(input string, width int) string {
	if len(input) >= width {
		return input
	}
	left := (width - len(input)) / 2
	right := width - len(input) - left
	return strings.Repeat(" ", left) + input + strings.Repeat(" ", right)
}
```

بروزرسانی پکیج phonebook

حالا بریم ببینیم تو phonebook کجا از این توابع استفاده می‌کردیم. تنها جایی که توابع جدول رو صدا زده بودیم، فایل report.go بود. پس اونجا توابع قدیمی رو با توابع table جایگزین می‌کنیم:

exercise/phone_book/report.go

```
package phonebook


import "exercise/table"


func Report(repo map[string]string) {
	const colWidth = 17
	headers := []string{"NAME", "PHONE NUMBER"}

	table.PrintLine(len(headers), colWidth)
	table.PrintRow(headers, colWidth)
	table.PrintLine(len(headers), colWidth)

	for name, phone := range repo {
		table.PrintRow([]string{name, phone}, colWidth)
		table.PrintLine(len(headers), colWidth)
	}
}
```

اینجا اومدیم پکیج table رو **صراحتاً** ایمپورت کردیم و از توابع PrintRow و PrintLine استفاده کردیم.

حالا می‌ریم سراغ همون کاری که از اول می‌خواستیم انجام بدیم

ما یه پکیج جدا برای رسم جدول ساختیم، پس تو main.go هم می‌تونیم خیلی راحت ازش استفاده کنیم:

exercise/main.go

```
func colorInterpreter() {
	var repo = map[string]string{
		"Red":  "Passion",
		"Blue": "Calm",
	}

	const colWidth = 15
	headers := []string{"Color", "Vibe"}

	table.PrintLine(len(headers), colWidth)
	table.PrintRow(headers, colWidth)
	table.PrintLine(len(headers), colWidth)

	for color, vibe := range repo {
		table.PrintRow([]string{color, vibe}, colWidth)
		table.PrintLine(len(headers), colWidth)
	}
}
```

import ضمنی نداریم!

شاید تو ذهنت بیاد که:
خب phonebook داره از table استفاده می‌کنه و main هم داره phonebook رو ایمپورت می‌کنه... پس یعنی table هم به طور ضمنی وارد mainمیشه دیگه؟!

نه به هیچ وجه!
تو Go هیچ چیزی به اسم **import ضمنی** نداریم. اگه پکیجی بخواد از یه پکیج دیگه استفاده کنه، **باید صراحتاً ایمپورتش کنه**. پس تو main.go حتی اگه phonebook داره از table استفاده می‌کنه، باز هم اگه خود main می‌خواد به table دسترسی داشته باشه، باید خودش مستقیماً اون رو ایمپورت کنه.

متغیر و ثابت در سطح پکیج

بیا یه نگاهی به روش استفاده از پکیج phonebook بنداز

```
phoneBookRepo := phonebook.New()
phonebook.Store(phoneBookRepo, "Arian", "+989051020304")
phonebook.Store(phoneBookRepo, "Neda", "+989051010202")
phonebook.Report(phoneBookRepo)
```

تو این مثال، همه‌ی توابعی که کاری با دفترچه تلفن دارن، نیاز دارن که repo رو به عنوان ورودی بگیرن. حتی اگه در آینده یه تابع جدید هم اضافه کنیم، باز هم باید اون map[string]string رو بهش بدیم.

حالا فرض کن اگه می‌شد یه جوری این repo رو یک‌بار تعریف کنیم و بعد تو همه توابع ازش استفاده کنیم بدون اینکه مجبور باشیم هی به صورت پارامتر پاسش بدیم! عالی می‌شد، نه؟

خبر خوب اینه که دقیقاً همچین چیزی تو Go ممکنه!

تو زبان Go اگه یه متغیر یا ثابت رو **بیرون از تابع** تعریف کنیم (یعنی تو سطح فایل و نه داخل تابع) اون تبدیل میشه به **متغیر یا ثابت سطح پکیج**.

ویژگی‌های مهم متغیر یا ثابت سطح پکیج

1. **تو همه‌ی فایل‌های پکیج قابل دسترسیه**
   فرقی نمی‌کنه اسمش با حرف بزرگ شروع شده باشه یا کوچیک، تا وقتی داخل همون پکیج هستیم، می‌تونیم ازش استفاده کنیم.
2. **فقط یک‌بار مقداردهی میشه (initialize)**
   یعنی اولین باری که یه پکیج import میشه، متغیر سطح پکیج مقداردهی میشه و بعدش همون مقدار حفظ میشه و برای همه‌ی جاهایی که ایمپورتش می‌کنن مشترکه.

حالا بیایم repo رو سطح پکیج تعریف کنیم

exercise/phone_book/repo.go

```
package phonebook


var repo = make(map[string]string)
```

چون قرار نیست پکیج‌های دیگه مستقیماً به repo دسترسی داشته باشن، با حرف کوچیک شروعش کردیم که private بمونه. دیگه نیازی به تابع New() هم نداریم، چون همه‌ی توابع پکیج phonebook مستقیم به repo دسترسی دارن.

exercise/phone_book/store.go

```
package phonebook


func Store(name string, phoneNumber string) bool {
	if len(phoneNumber) != 13 {
		return false
	}

	repo[name] = phoneNumber
	return true
}
```

توجه کردی؟ پارامتر ورودی repo رو حذف کردیم. الان repo همون متغیریه که تو فایل repo.go تعریفش کردیم.

exercise/phone_book/report.go

```
package phonebook

import "exercise/table"

func Report() {
	const colWidth = 17
	headers := []string{"NAME", "PHONE NUMBER"}

	table.PrintLine(len(headers), colWidth)
	table.PrintRow(headers, colWidth)
	table.PrintLine(len(headers), colWidth)

	for name, phone := range repo {
		table.PrintRow([]string{name, phone}, colWidth)
		table.PrintLine(len(headers), colWidth)
	}
}
```

به همین ترتیب... پارامتر ورودی repo رو حذف کردیم. الان repo همون متغیریه که تو فایل repo.go تعریفش کردیم.

exercise/main.go

```
package main

import "exercise/phone_book"
import "exercise/table"

func main() {
	phonebook.Store("Arian", "+989051020304")
	phonebook.Store("Neda", "+989051010202")
	phonebook.Report()
}
```

خیلی خوشگل‌تر و ساده‌تر شد، نه؟
دیگه لازم نیست برای هر تابع repo پاس بدیم. نیازی هم به New() نیست که repo بسازه.
همه چیز پشت پرده تو پکیج phonebook کنترل میشه.

یه نکته‌ی مهم درباره‌ی امنیت و طراحی

ما با این کار در واقع اصل **encapsulation** رو رعایت کردیم. یعنی متغیر repo فقط **داخل خود پکیج** قابل دسترسیه. پکیج‌های دیگه نمی‌تونن مستقیماً تغییری توش بدن. فقط از طریق توابعی که خود پکیج بهمون داده (مثل Store و Report) می‌تونیم باهاش تعامل کنیم.
این یعنی **کنترل کامل روی داده‌ها** و جلوگیری از تغییرات ناخواسته از بیرون.

مفهوم Scope و Visibility

scope یه موجودیت (مثلاً یه متغیر) مشخص می‌کنه که اون موجودیت از کجاها قابل دسترسیه. مثلاً:

- متغیر تعریف‌شده توی یه تابع، فقط همون تابع بهش دسترسی داره.
- متغیر تعریف‌شده در سطح پکیج، تمام فایل‌های داخل اون پکیج می‌تونن بهش دسترسی داشته باشن.

اما visibility به این معنیه که آیا فایل‌های پکیج‌های دیگه هم می‌تونن بهش دسترسی داشته باشن یا نه؟ اینو با **حرف اول بزرگ یا کوچیک** مشخص می‌کنی.

جدول مقایسه Scope و Visibility

ویژگی	Scope (دامنه)	Visibility (قابلیت دیده شدن)
تعریف ساده	جایی که یک متغیر یا تابع در اون قابل دسترسیه	اینکه آیا میشه از پکیج‌های دیگه بهش دسترسی داشت یا نه
سطح تحلیل	**داخلی به پکیج و فایل**	**بین پکیج‌ها** (public/private بودن)
محل تصمیم‌گیری	با توجه به موقعیت تعریف (داخل تابع، فایل، پکیج)	با توجه به **اولین حرف اسم (بزرگ یا کوچک)**
نحوه تعیین	محل قرارگیری در کد	حروف بزرگ = Exported حروف کوچک = Unexported
دایره‌ی اثر	داخل تابع، فایل یا پکیج	از پکیج جاری یا از سایر پکیج‌ها
پیش‌فرض	Local - متغیر تو تابع فقط همونجا معتبره	Private - تا وقتی حرف اول بزرگ نیست از بیرون قابل دیدن نیست
مثال	اگه تو تابع تعریف بشه، فقط اونجاست؛ اگه بیرون باشه، تو کل پکیج	اگه اسمش X باشه از بقیه پکیج‌ها هم دیده میشه
استفاده اشتباه رایج	انتظار داریم توابع یا متغیرهای لوکال تو جاهای دیگه هم دیده بشن	فراموش می‌کنیم که برای export باید با حرف بزرگ شروع کنیم

پکیج اعتبار سنجی

وقتشه یه قدم دیگه برداریم و یه پکیج جدید به پروژه‌مون اضافه کنیم. اسم این پکیج رو می‌ذاریم validator و وظیفه‌ش هم خیلی مشخصه: بررسی اعتبار مقادیری که تو برنامه استفاده می‌کنیم. چون تو این پروژه با **نام** و **شماره تلفن** سروکار داریم، این پکیج رو اولش فقط برای اعتبارسنجی شماره تلفن می‌سازیم.

می‌خوایم مطمئن بشیم که:

- طول شماره دقیقاً 13 کاراکتره
- با "+98" شروع میشه
- بقیه‌ی کاراکتراش فقط عددن (0 تا 9)

چرا پکیج جدید؟

الان یه اعتبارسنجی ساده تو تابع Store از پکیج phonebook داریم، ولی بیایم منطقی نگاه کنیم:

آیا واقعاً وظیفه‌ی phonebook اینه که بدونه شماره تلفن معتبر چیه یا نه؟

نه واقعاً! وظیفه‌ی اون ذخیره کردنه، نه تحلیل اعتبار شماره. برای همین هم تصمیم گرفتیم یه پکیج مستقل بسازیم به اسم validator که فقط مسئول این‌جور چیزاست. حالا اگه بعداً خواستیم یه پکیج دیگه هم از همون تابع استفاده کنه، راحتیم!

وضعیت الان چطوره؟

تابع Store تو phonebook اینطوریه:

```
func Store(name string, phoneNumber string) bool {
	if len(phoneNumber) != 13 {
		return false
	}

	repo[name] = phoneNumber
	return true
}
```

اما با اضافه شدن پکیج validator، می‌خوایم به این شکل درش بیاریم:

```
func Store(name string, phoneNumber string) bool {
	if !validator.IsPhoneValid(phoneNumber) {
		return false
	}

	repo[name] = phoneNumber
	return true
}
```

چه اتفاقی افتاد؟ حالا دیگه تابع Store کاری به جزئیات نداره که شماره چطوری باید باشه. فقط منتظر یه جواب true یا false از تابع IsPhoneValid پکیج validator می‌مونه. این یعنی تفکیک مسئولیت به بهترین شکل.

ساخت پکیج validator

بیا یه فولدر جدید درست کنیم به اسم validator تو ریشه‌ی پروژه و داخلش یه فایل به اسم phone.go بسازیم. حالا ساختار پروژه‌مون این شکلی میشه:

```
exercise
├── main.go
├── phone_book
│   ├── repo.go
│   ├── store.go
│   └── report.go
└── table
│   └── draw.go
└── validator
    └── phone.go
```

exercise/validator/phone.go

```
package validator

import "exercise/phone_book"

func IsPhoneValid(phone string) bool {
	if len(phone) != 13 || phone[0:3] != "+98" {
		return false
	}

	numericSection := phone[3:]

	for _, char := range numericSection {
		if char < '0' || char > '9' {
			return false
		}
	}

	return true
}
```

اگه این اعتبارسنجی رو تو خود Store می‌نوشتیم چی میشد؟

فکرشو بکن همون کدی که این بالا نوشتیم رو میاوردیم تو تابع Store! یهو یه عالمه شرط و حلقه و بررسی می‌اومد وسط یه تابعی که فقط قراره یه مقدار رو توی map ذخیره کنه!

در آینده اگه خواستیم تغییراتی تو پروژه بدیم یا کدها رو بررسی کنیم، مجبور بودیم هم کارای ذخیره‌سازی رو بخونیم هم اعتبارسنجی رو! یعنی کلی کد بی‌ربط تو یه تابع. این دقیقاً همون چیزیه که سادگی و خوانایی کد رو خراب می‌کنه.

حالا با تعریف پکیج Validator

- هم ساختار پروژه مرتب‌تر شد
- هم مسئولیت‌ها درست تقسیم شدن
- هم یه ابزار قابل استفاده‌ی مجدد ساختیم که می‌تونیم هر جا لازم شد ازش استفاده کنیم

مشکل import cycle

بیا یه سوال ساده ازت بپرسم:
به نظرت منطقیه یه شماره تلفن مشخص دوبار تو دفترچه تلفن ذخیره بشه؟
قطعاً نه! پس بیایم یه راهی پیدا کنیم که جلو این اتفاق رو بگیریم.

قراره تو پکیج validator یه تابع جدید بنویسیم به اسم IsPhoneAlreadyAdded چون یه جورایی داریم شماره تلفن رو اعتبارسنجی می‌کنیم از نظر اینکه آیا قبلاً وارد شده یا نه.

تابع Store با اعتبارسنجی جدید:

```
func Store(name string, phoneNumber string) bool {
	if !validator.IsPhoneValid(phoneNumber) {
		return false
	}

	if validator.IsPhoneAlreadyAdded(phoneNumber) {
		return false
	}

	repo[name] = phoneNumber
	return true
}
```

قبل از اینکه تابع IsPhoneAlreadyAdded رو تو پکیج validator بنویسیم باید به یه موضوع مهم دقت کنیم

چالش اصلی

تابع IsPhoneAlreadyAdded باید به لیست همه شماره‌هایی که **قبلاً ذخیره شدن** دسترسی داشته باشه.
ولی اون شماره‌ها تو یه map[string]string به اسم repo تو پکیج phonebook ذخیره شدن و اونم **متغیر سطح پکیجه.** یعنی فقط خود phonebook بهش دسترسی داره، نه پکیج‌های دیگه مثل validator.

خب حالا چی کار کنیم؟

میایم یه تابع تو پکیج phonebook می‌نویسیم که همه شماره‌ها رو به شکل []string برگردونه.

برای این کار یه فایل جدید به اسم get.go تو مسیر exercise/phone_book ایجاد میکنیم.

exercise/phone_book/get.go

```
package phonebook


func GetAllPhoneNumbers() []string {
	var allPhoneNumbers []string


	for _, phoneNumber := range repo {
		allPhoneNumbers = append(allPhoneNumbers, phoneNumber)
	}


	return allPhoneNumbers
}
```

الان تابع GetAllPhoneNumbers به ما لیست شماره‌ها رو میده. حالا می‌تونیم بریم سراغ نوشتن تابع توی پکیج validator.

exercise/validator/phone.go

```
func IsPhoneAlreadyAdded(phone string) bool {
	allPhoneNumber := phonebook.GetAllPhoneNumbers()

	for _, phoneNumber := range allPhoneNumber {
		if phone == phoneNumber {

			return true
		}
	}

	return false
}
```

بوم! خطای import cycle

الان یه مشکلی پیش میاد!

- phonebook داره از validator استفاده می‌کنه
- validator هم برگشته از phonebook استفاده کرده

یعنی چی؟ یعنی **هر دو پکیج دارن همدیگه رو ایمپورت می‌کنن** !این یه **حلقه‌ی وارد کردن** یا همون **import cycle** به حساب میاد.

نتیجه کامپایل برنامه در زیر آورده شده که دقیقا به همین مشکل اشاره داره.

```
package exercise
	imports exercise/phone_book from main.go
	imports exercise/validator from store.go
	imports exercise/phone_book from phone.go: import cycle not allowed
```

Go خیلی رک و پوست‌کنده میگه "این چرخه‌ی ایمپورت مجازه نیست!"

چرا Go با import cycle مشکل داره؟

دلایل منطقی داره، مثل:

1. **سخت شدن ترتیب کامپایل:**
   Go نمی‌تونه بفهمه کدوم پکیج باید اول کامپایل شه.
2. **افزایش پیچیدگی و خطاهای پنهان:**
   کدهات در هم گره می‌خورن. هر تغییری، همه‌جا رو ممکنه خراب کنه!
3. **نقض تفکیک مسئولیت‌ها:**
   یعنی کدهات درست دسته‌بندی نشده و پکیج‌هات بیش از حد به هم وابستن.

راه‌حل چیه؟

تابع IsPhoneAlreadyAdded فقط نیاز به یه لیست از شماره‌ها داره. پس چرا بیایم کل پکیج phonebook رو وارد کنیم؟ فقط کافیه اون لیست رو به صورت **پارامتر** به تابع بدیم!

exercise/validator/phone.go

```
func IsPhoneAlreadyAdded(phoneNumbers []string, phone string) bool {
	for _, phoneNumber := range phoneNumbers {
		if phone == phoneNumber {

			return false
		}
	}

	return true
}
```

exercise/phone_book/store.go

```
func Store(name string, phoneNumber string) bool {
	if !validator.IsPhoneValid(phoneNumber) {
		return false
	}

	allPhoneNumber := GetAllPhoneNumbers()

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

	repo[name] = phoneNumber
	return true
}
```

دیگه خبری از import cycle نیست! حالا برنامه راحت اجرا میشه و از ذخیره‌شدن شماره‌های تکراری هم جلوگیری می‌کنه.

بریم که تستش کنیم!

exercise/main.go

```
func main() {
	phonebook.Store("Arian", "+989051020304")
	phonebook.Store("Neda", "+989051010202")
	phonebook.Store("Narges", "+989051010202")
	phonebook.Report()
}
```

در اینجا شماره‌ی نرگس ذخیره نمیشه، چون همون شماره قبلاً برای ندا ذخیره شده بود.

تابع init

تا حالا به این فکر کردی که تو هر دفترچه تلفنی، وجود شماره‌های اورژانسی مثل پلیس، آتش‌نشانی و آمبولانس ضروریه؟ خب ما هم می‌خوایم تو پروژه‌مون کاری کنیم که **به صورت خودکار این شماره‌ها از قبل توی دفترچه تلفن ذخیره شده باشن**.

اما چطوری؟

بیا یه نگاه بندازیم به فایل repo.go از پکیج phonebook

exercise/phone_book/repo.go

```
package phonebook


var repo = make(map[string]string)
```

تا اینجا اومدیم یه متغیر سراسری (package-level) ساختیم که همه اطلاعات مخاطب‌ها رو توش نگه می‌داریم. حالا اگه بخوایم شماره‌های اورژانسی رو هم همین‌جا بهش اضافه کنیم، شاید وسوسه بشی بنویسی:

```
package phonebook


var repo = make(map[string]string)


repo["Police"] = "110"
repo["FireDepartment"] = "125"
repo["Emergency"] = "115"
```

ولی خب... خبر بد اینه که Go باهات راه نمیاد و یه خطای معروف می‌گیری:

```
syntax error: non-declaration statement outside function body
```

چرا این خطا رو می‌گیریم؟

چون تو زبان Go فقط **اعلان‌ها** (مثل تعریف متغیر یا ثابت) و **توابع** اجازه دارن مستقیم تو سطح پکیج قرار بگیرن. هر نوع دستور دیگه‌ای (مثل مقداردهی به map) باید **داخل یه تابع باشه**.

راهکار چیه؟

میشه یه تابع New() تعریف کنیم که این مقداردهی رو انجام بده. ولی این‌جوری ما مجبور می‌شیم تو main() اون تابع رو صدا بزنیم فقط برای اینکه شماره‌های اورژانسی وارد بشن! انگار فقط داریم برای مقداردهی دستی، یک تابع زورکی تعریف می‌کنیم!

اما خوشبختانه Go یه قابلیت عالی داره: تابعی به اسم init

تابع init

یه تابع خاصه که **نه ورودی می‌گیره و نه خروجی می‌ده** و **خودش به صورت خودکار اجرا می‌شه**، اونم درست قبل از اینکه main() اجرا بشه.

ساختار تابع init

```
init() {
}
```

ویژگی‌های مهم تابع init

- تو هر فایل می‌تونی **یک یا چند init()** داشته باشی.
- در **یک فایل خاص**، اگه چند تا init() بنویسی، **به ترتیب ظاهر شدن در فایل اجرا می‌شن**.
- اگه init() ها تو فایل‌های مختلف یک پکیج باشن، ترتیب اجراشون **غیرقابل پیش‌بینیه** (ولی همشون قبل از main() اجرا می‌شن).
- هر init() فقط **یه بار اجرا می‌شه، اونم فقط وقتی پکیج برای اولین بار import می‌شه**.

استفاده از init() برای مقداردهی اولیه

خب حالا بریم به سراغ پروژه‌ی خودمون و یه init() بنویسیم تا شماره‌های اورژانسی رو بریزه داخل repo

exercise/phone_book/repo.go

```
package phonebook

var repo = make(map[string]string)

func init() {
    repo["Police"] = "110"
    repo["FireDepartment"] = "125"
    repo["Emergency"] = "115"
}
```

تست کنیم ببینیم چی میشه

وقتی برنامه رو اجرا کنیم، بدون اینکه هیچ کاری بکنیم، این شماره‌ها خودشون تو دفترچه تلفن هستن:

```
+-----------------+-----------------+
|      NAME       |  PHONE NUMBER   |
+-----------------+-----------------+
|     Police      |       110       |
+-----------------+-----------------+
| FireDepartment  |       125       |
+-----------------+-----------------+
|    Emergency    |       115       |
+-----------------+-----------------+
|      Arian      | +989051020304   |
+-----------------+-----------------+
|      Neda       | +989051010202   |
+-----------------+-----------------+
```

چند تا init() تو یک فایل

اگه بخوای چند init() جداگانه بنویسی Go مشکلی نداره!

exercise/phone_book/repo.go

```
package phonebook

var repo = make(map[string]string)

func init() {
    fmt.Println(“Emergency phone number has been added to the phonebook”)
    repo["Emergency"] = "115"
}

func init() {
    fmt.Println(“Police phone number has been added to the phonebook”)
    repo["Police"] = "110"
}

func init() {
    fmt.Println(“Fire Department phone number has been added to the phonebook”)
    repo["FireDepartment"] = "125"
}
```

خروجی

```
Emergency phone number has been added to the phonebook
Police phone number has been added to the phonebook
Fire Department phone number has been added to the phonebook
+-----------------+-----------------+
|      NAME       |  PHONE NUMBER   |
+-----------------+-----------------+
|      Arian      | +989051020304   |
+-----------------+-----------------+
|      Neda       | +989051010202   |
+-----------------+-----------------+
|    Emergency    |       115       |
+-----------------+-----------------+
|     Police      |       110       |
+-----------------+-----------------+
| FireDepartment  |       125       |
+-----------------+-----------------+
```

نکته

همون‌طور که قبلاً تو درسنامه "ساختار داده map" گفتیم، **عناصر داخل map ترتیب ثابتی ندارد** .پس این ترتیب نمایش در جدول **ربطی به ترتیب اجرای init() ها نداره**. ممکنه Emergency اول ذخیره بشه ولی آخر نشون داده بشه.

جمع بندی

توابع init() ابزار خفن و تمیزی هستن برای زمانی که بخوای **یه سری مقدار اولیه رو بدون دخالت مستقیم کاربر، قبل از اجرای برنامه مقداردهی کنی** .مخصوصاً وقتی داری روی پکیج‌های جدا کار می‌کنی و نمی‌خوای زورکی از بیرون بهش مقدار بدی.

کاربردهای تابع init

توابع ()init در ظاهر ساده‌ن، ولی تو دل پروژه‌های واقعی کلی کار مهم ازشون برمیاد. اینا چندتا از **کاربردهای رایج ()init** هستن:

1. **مقداردهی اولیه به متغیرهای سطح پکیج**
   - مثل کاری که ما با repo کردیم؛ شماره‌های اورژانسی رو قبل از هر چیز دیگه وارد کردیم.
2. **لود کردن تنظیمات اولیه**
   - مثلاً وقتی می‌خوای از یه فایل config بخونی و تنظیمات رو آماده کنی.
3. **برقراری اتصال اولیه**
   - مثلاً اتصال به دیتابیس، باز کردن فایل، یا ساختن connection به یک سرویس خارجی.
4. **ثبت ساختارها در رجیستری**
   - گاهی در برنامه‌های پیچیده‌تر مثل web framework ها یا پلاگین سیستم‌ها، init برای auto-register کردن چیزها استفاده میشه.
5. **نمایش پیغام یا لاگ برای آماده‌سازی سیستم**
   - مثل همون کاری که ما کردیم و گفتیم "Police phone number has been added to the phonebook"

مقایسه تابع init با main

هر دو تابع ویژه هستن، ولی نقش و زمان اجراشون با هم فرق داره. بذار برات یه مقایسه تمیز بیارم:

ویژگی	init()	main()
محل تعریف	در هر فایل و هر پکیج	فقط در فایل main از پکیج main
تعداد مجاز	چندتا تو یک فایل میشه داشت	فقط یکی در کل برنامه
ترتیب اجرا	قبل از main()	آخر از همه، بعد از همه ()init ها
ورودی و خروجی	نداره	میتونه از args ورودی بخونه
کاربرد اصلی	آماده‌سازی اولیه پکیج	نقطه شروع برنامه
اجرا در import	بله ()init اجرا میشه	نه ()main فقط موقع اجرای برنامه

ترتیب اجرای توابع init

ترتیب اجرا شدن ()init ها تو Go خیلی مهمه و باید درست متوجهش باشیم. بیایم از سه منظر بررسیش کنیم:

1-توابع ()init در یک فایل از یک پکیج

اینا دقیقاً **به ترتیبی که توی فایل نوشتیمشون اجرا میشن** .اینو قبلاً تو مثال شماره‌های اورژانسی دیدیم.

2- توابع ()init در فایل‌های مختلف از یک پکیج

اینجا دیگه دست ما نیست! Go میاد **به صورت internal خودش ترتیب فایل‌ها رو مشخص می‌کنه** (بر اساس وابستگی‌های داخلی)، ولی قول نمیده که ترتیبش بر اساس اسم فایل یا ترتیب تعریف ما باشه.

پس نمی‌تونیم دقیقاً پیش‌بینی کنیم اول ()init توی repo.go اجرا میشه یا store.go

3-توابع ()init در پکیج‌های مختلف

برای اینکه اینو خوب بفهمیم، باید با یه چیز مهم آشنا بشیم :**گراف وابستگی**

گراف وابستگی

گراف وابستگی یه نقشه‌ست از اینکه هر پکیج به چه پکیج‌هایی **وابسته‌ست**. این کمک می‌کنه Go بفهمه باید **کدوم پکیج‌ها رو اول کامپایل و اجرا کنه**، و ترتیب اجرای توابع init() رو هم از رو همین نقشه درمیاره.

تحلیل گراف وابستگی در پروژه دفترچه تلفن

ما تو پروژه‌مون این پکیج‌ها رو داریم:

- main
- phonebook
- validator
- table

وابستگی‌ها

- main به phonebook
- phonebook به validator و table
- validator به هیچ‌کس
- table به هیچ‌کس

رسم گراف وابستگی

ترتیب اجرای پکیج‌ها (بر اساس گراف)

1. validator و table (چون مستقل هستن)
2. phonebook (چون به اون دوتا وابسته‌ست)
3. main در نهایت، بعد از همه

حالا توابع init() هم **به ترتیب اجرای همین پکیج‌ها اجرا می‌شن**

1. اول init() های داخل validator
2. بعد init() های داخل table
3. بعد init() های داخل phonebook
4. و در نهایت init() های main

روش های مختلف import پکیج

ما معمولاً وقتی می‌خوایم از یه پکیج تو فایل خودمون استفاده کنیم، از کلمه‌ی کلیدی import استفاده می‌کنیم. ولی جالب اینه که Go چند مدل مختلف برای import کردن داره که هرکدوم کاربرد خاص خودش رو دارن.

1- Import معمولی (با اسم خودش)

این همون روش کلاسیکه که همیشه تو مثال‌ها دیدیم:

```
import "exercise/phone_book"
```

تو این مدل، اگه بخوای به توابع یا متغیرهای داخل این پکیج دسترسی پیدا کنی، باید از اسم پکیج استفاده کنی.

```
phonebook.Store("Arian", "+989051020304")
```

2- Import با alias (اسم مستعار)

اگه خواستی یه اسم دلخواه یا کوتاه‌تر به پکیج بدی، می‌تونی با alias یا همون مستعار بیاریش داخل:

```
import pb "exercise/phone_book"
```

دیگه به‌جای phonebook.Store می‌تونی بنویسی:

```
pb.Store("Arian", "+989051020304")
```

کی به درد می‌خوره؟

وقتی مثلاً دوتا پکیج با اسم یکسان داری، اما تو مسیرهای مختلفن. مثل این سناریو:

```
import "exercise/personal_phone_book"
import OPhoneBook "exercise/organizational_phone_book"
```

با این کار از تداخل (conflict) جلوگیری می‌کنی.

3- Import با شناسه‌ی خالی (_)

این مدل مخصوص وقتیه که فقط می‌خوای **side-effect** های پکیج (مثل اجرای تابع‌های init) فعال شن، ولی نیازی نداری مستقیماً چیزی از اون پکیج استفاده کنی:

```
import _ "exercise/phone_book"
```

یعنی توابع ()init اجرا می‌شن، ولی تو نمی‌تونی مثلاً phonebook.Store رو صدا بزنی چون هیچ اسمی برای پکیج ندادی.

4- Importبا نقطه (.)

یه روش نسبتاً خطرناک ولی تو بعضی شرایط خاص مفید:

```
import . "exercise/phone_book"
```

تو این حالت می‌تونی مستقیم بنویسی:

```
Store("Arian", "+989051020304")
```

انگار این تابع داخل فایل خودت تعریف شده

مشکلش چیه؟

دیگه نمی‌فهمی یه تابع یا متغیر از خودته یا از یه پکیج دیگه! پس فقط تو پروژه‌های کوچیک یا دم‌دستی استفاده‌ش کن.

نکته مهم:

می‌تونی چندتا پکیج رو با هم توی یه بلاک import بیاری تا کدت تمیزتر باشه:

```
import (
	"fmt"
	"strings"
)
```

هم خواناتر میشه، هم شیک‌تر.

جمع بندی

نوع import	شکل نوشتن	کاربرد اصلی
معمولی	import "fmt"	دسترسی عادی با اسم پکیج
باalias	import f "fmt"	وقتی اسم دلخواه یا جلوگیری از تداخل می‌خوای
با شناسه‌ی خالی _	import _ " fmt "	فقط اجرای init() بدون استفاده مستقیم
با نقطه .	import . "fmt"	استفاده مستقیم از توابع بدون نوشتن اسم پکیج

تفاوت دستور go build و go run

دستور go run برای **تست سریع** کده. میاد همه چیزو موقتی کامپایل می‌کنه و فوراً اجرا می‌کنه. بعدشم فایل اجرایی موقت پاک میشه. اما go build دقیقا کارش **ساخت فایل اجرایی** دائمیه. یعنی کدتو می‌گیره، کامپایل می‌کنه، و یه فایل باینری بهت تحویل میده که می‌تونی بعدها هر وقت خواستی اجراش کنی.

جدول مقایسه

ویژگی	go run	go build
اجرای سریع بدون فایل نهایی	✓	✗
تولید فایل اجرایی	✗	✓
مناسب برای تست	✓	✗
مناسب برای تولید نهایی	✗	✓

نحوه انتخاب نام خروجی در دستور go build

در دستور go build به دو روش می‌تونیم مسیر پکیج main رو بهش بدیم. اینکه اسم فایل اجرایی چی بشه، بستگی به نحوه‌ای داره که مسیر رو معرفی کردی.

حالت اول: اشاره به پوشه‌ای که پکیج main توشه

فرض کن ساختار پروژه‌مون اینه:

```
exercise
├── main.go
├── phone_book
│   ├── get.go
│   ├── repo.go
│   ├── store.go
│   └── report.go
└── table
│   └── draw.go
└── validator
    └── phone.go
```

اگه تو کنسول توی فولدر exercise باشی و بزنی:

```
go build .
```

چون با . به همین فولدری که توش هستی (یعنی فولدر حاوی main.go) اشاره کردی، اسم فایل خروجی میشه:

```
exercise
├── exercise
```

یعنی هم‌نام با فولدری که main.go توشه.

حالا فرض کن main.go رو می‌بری توی یه پوشه جدید به اسم application

```
exercise
├── application
│   └── main.go
├── phone_book
│   ├── get.go
│   ├── repo.go
│   ├── store.go
│   └── report.go
└── table
│   └── draw.go
└── validator
    └── phone.go
```

و حالا بزنی:

```
go build ./application
```

تو این حالت اسم فایل اجرایی میشه:

```
exercise
├── application
│   └── main.go
├── application
```

حالت دوم: اشاره مستقیم به فایل حاوی تابع main از پکیج main

این بار به جای فولدر، مستقیماً میای به خود فایل main.go اشاره می‌کنی:

```
go build ./application/main.go
```

```
exercise
├── main
```

مشخص کردن نام فایل اجرایی با o-

اگه دوست داری اسم فایل اجرایی رو خودت تعیین کنی، مهم نیست از چه روشی برای مسیردهی استفاده می‌کنی. فقط کافیه از فلگ o- استفاده کنی:

```
go build -o app ./application
```

میشه.

```
exercise
├── application
│   └── main.go
├── app
```

خروجی برای سیستم‌عامل و پردازنده‌های مختلف (Cross Compilation)

فرض کن یه برنامه با زبان C++ یا Java یا Rust یا حتی Python نوشتی. حالا می‌خوای این برنامه رو بدی به یکی که از ویندوز استفاده می‌کنه، در حالی که خودت داری روی مک یا لینوکس کار می‌کنی.

اینجاست که مشکل شروع میشه:

تو زبان‌های کلاسیکی مثل C:

باید **کامپایلر مخصوص سیستم‌عامل مقصد** رو نصب کنی. مثلاً اگه از لینوکسی، باید یه cross-compiler مخصوص ویندوز نصب کنی (مثلاً x86_64-w64-mingw32-gcc) تازه کلی کانفیگ هم باید بکنی که اون کامپایلر از کدتو درست بفهمه.

تو Java یا Python:

برنامه‌ت وابسته‌ست به ماشین مجازی (JVM یا Python Interpreter) یعنی کاربر مقصد باید **نرم‌افزار خاصی نصب داشته باشه** تا اصلاً برنامه‌ت بتونه اجرا شه. خود برنامه‌ی تو یه فایل اجرایی native نیست!

تازه تو این زبان‌ها اگه یه کد ساده هم بخوای برای چند سیستم‌عامل بسازی، باید بری روی اون سیستم اجرا و تستش کنی یا از VM استفاده کنی.

خلاصه؟ **دردسر، پیچیدگی، اعصاب‌خوردی، زمان تلف‌شده...**

اما Go این کابوسو تبدیل کرده به یه رؤیای شیرین!

تو Go فقط با تنظیم دو تا متغیر محیطی ساده، می‌تونی برای هر سیستم‌عاملی که دلت بخواد خروجی بسازی — بدون نیاز به هیچ ابزار اضافه‌ای!

```
GOOS=...
GOARCH=...
```

این دو تا متغیر به Go میگن:

- **GOOS** مشخص میکنه سیستم‌عامل مقصد چیه؟
- **GOARCH** مشخص میکنه معماری CPU مقصد چیه؟

و بعد فقط کافیه بزنی:

```
go build
```

در نتیجه اگه بخوای برنامتو جوری خروجی بگیری که بتونی به شخصی که ویندوز داره بدی و بتونه به سادگی استفاده کنه کافیه دستور زیرو برای کامپایل اجرا کنی

```
GOOS=windows GOARCH=amd64 go build .
```

هیچ نیازی به نصب چیز خاصی نداری، خبر خوب اینه که Go از همون اول همه این کامپایلرهای مختلف رو **تو خودش جاسازی کرده**!

لیست سیستم‌عامل‌های قابل پشتیبانی

مقدار	توضیح
linux	لینوکس انواع توزیع‌ها مثل (Ubuntu, Debian, Arch, ...)
windows	ویندوز
darwin	macOS اپل
android	اندروید
ios	سیستم‌عامل آیفون
freebsd	یکی از سیستم‌عامل‌های یونیکسی
openbsd	مشابه FreeBSD ولی امنیت‌محورتر
netbsd	یکی دیگه از خانواده BSD
dragonfly	سیستم‌عامل مبتنی بر BSD
plan9	سیستم‌عاملی از آزمایشگاه Bell
aix	سیستم‌عامل IBM
js	اجرای داخل WebAssembly برای مرورگرها

لیست معماری‌های قابل پشتیبانی

مقدار	توضیح
amd64	معماری 64 بیتی برای کامپیوترهای معمولی (اینتل وAMD)
386	معماری 32 بیتی قدیمی
arm	معماری برای پردازنده‌های ARM مثل Raspberry Pi
arm64	معماری ARM 64 بیتی (تلفن‌های هوشمند جدید، اپل M1 و M2)
wasm	WebAssembly برای اجرای داخل مرورگر
ppc64	معماری 64 بیتی PowerPC
ppc64le	PowerPC با little-endian
mips, mipsle, mips64	معماری‌های MIPS قدیمی‌تر
s390x	معماری IBM برای سرورهای بزرگ

مستندسازی پکیج‌ها با کامنت‌های استاندارد

تا الان هرچی تابع و متغیر و پکیج ساختیم، همه‌ش کد بود. ولی حالا وقتشه یه پله بریم بالاتر و حرفه‌ای‌تر بنویسیم. یکی از چیزای خفن زبان Go اینه که خودش ابزار داره برای تولید **مستندات خودکار** از روی کدت. فقط کافیه بالای توابع، متغیرها یا حتی خود پکیج یه کامنت درست‌درمون بذاری.

فرمتش خیلی سادست. مثلاً اگه تابعی به اسم PrintRow داری، اینجوری براش کامنت می‌ذاری:

```
// PrintRow prints a row of values centered in columns with a fixed width.
func PrintRow(values []string, width int) {
	//...
}
```

حتماً حواست باشه جمله با نام تابع شروع بشه. چون ابزار Go اینو می‌فهمه و توی مستندات نشونش می‌ده.

استفاده از go doc

حالا این کامنتا به چه دردی می‌خورن؟ اینجاست که ابزارهای go doc و godoc میان وسط.

go doc

اگه بخوای سریع از تو ترمینال ببینی یه تابع چیکار می‌کنه، این ابزار به دردت می‌خوره.
مثلاً فرض کن یه پکیج table داری و می‌خوای ببینی PrintRow چیه:

```
go doc exercise/table.PrintROw
```

خروجیش همون توضیحیه که بالای تابع نوشتی!

نوشتن کامنت برای توابع Export شده

یادت هست گفتیم اگه اسم یه تابع با حرف بزرگ شروع بشه، بقیه پکیج‌ها هم می‌تونن ازش استفاده کنن؟ خب اگه قراره بقیه ازش استفاده کنن، پس حتماً باید بدونن دقیقاً چه کاری انجام می‌ده، چه ورودی‌هایی می‌گیره، و چه خروجی‌ای می‌ده.

اگه کامنت ننویسی، اونی که پکیجتو ایمپورت کرده مجبوره بیاد تو سورس‌کدت بگرده ببینه هر موجودیت چی‌کار می‌کنه. ولی اگه همون بالا یه کامنت استاندارد گذاشته باشی، با یه go doc ساده یا حتی IDE ای که استفاده میکنه بهش کامنت اون موجودیت (متغیر، ثابت و تابع) رو نشون میده و همه‌چی براش روشنه.

یعنی این کامنتا فقط یه لطف نیست، یه **وظیفه‌ی حرفه‌ایه**

جمع‌بندی

هر تابع export شده = نیاز به کامنت حرفه‌ای

هر پکیج درست = مستندسازی درست

هر برنامه‌نویس واقعی = کسی که فقط کد نمی‌زنه، بلکه **قابل فهم** و **مستند** می‌نویسه

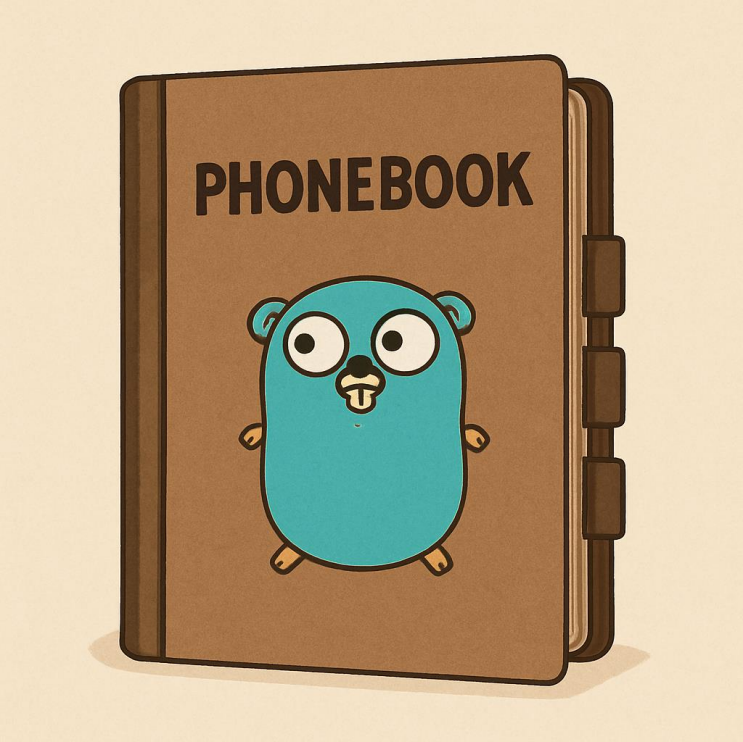
تو Go هیچوقت لازم نیست بری مستند جدا بنویسی، فقط همون کامنت بالا کافیه.

Go خودش بقیه‌شو هندل می‌کنه

تمرین 1: تکمیل پکیج دفترچه تلفن

تا اینجا با هم یه پکیج به اسم phonebook ساختیم که کارش ذخیره و نمایش شماره تلفن مخاطب‌ها بود. توابع Store و Report رو با هم نوشتیم و دیدیم چطور با استفاده از map و یه سری ابزار از پکیج‌های دیگه مثل table و validator، یه سیستم ساده اما منظم ساختیم.

حالا نوبت توئه که این دفترچه تلفن رو حرفه‌ای‌تر کنی!



وظیفه‌ی تو اینه که قابلیت‌های زیر رو به پکیج phonebook اضافه کنی:

1-حذف مخاطب

یه تابع به اسم Delete(name string) bool اضافه کن که:

- اگر نام داده شده توی repo وجود داشت، حذفش کن و true برگردون.
- اگر نامی با اون مشخصات پیدا نشد، false برگردون.

دقت کن شماره‌های اضطراری (Emergency ، Police و FireDepartment) نمیتونن حذف بشن

2- جستجوی مخاطب با نام

یه تابع دیگه به اسم Find(name string) string بساز که:

- اگه مخاطبی با اون اسم پیدا شد، شماره تلفنش رو برگردونه.
- اگر پیدا نشد، رشته "not found" رو برگردونه.

3-یه تابع به اسم DeleteAll() بنویس که کل repo رو خالی کنه. فقط حواست باشه شماره‌های اضطراری (Emergency ، Police و FireDepartment) حذف نشن

4- اعتبارسنجی اسم مخاطب

فعلاً فقط شماره تلفن رو با validator.IsPhoneValid بررسی می‌کردیم. الان باید یه تابع جدید به اسم IsNameValid(name string) توی پکیج validator بنویسی که:

- اطمینان حاصل کنه اسم مخاطب کمتر از 3 کاراکتر نباشه.

بعد بیا توی تابع Store از این تابع استفاده کن تا فقط اسم‌های معتبر ذخیره بشن.

5-تمرین برای تست نهایی

تو فایل main.go یا هر فایل تست دیگه‌ای، این موارد رو پیاده‌سازی کن:

- سعی کن یه مخاطب جدید با اسم خیلی کوتاه (مثلاً "A" ذخیره کنی و بررسی کن که ذخیره میشه یا نه (نباید ذخیره بشه).
- چند مخاطب جدید با اسم درست و شماره درست اضافه کن.
- یکی از اون‌ها رو با تابع Find پیدا کن و چاپش کن.
- یکی دیگه از مخاطب‌ها رو حذف کن.
- یه بار هم خروجی Report() رو چک کن که ببینی همه چیز درسته یا نه.

تمرین 2: پکیج ورود و خروج کارمندان

قراره یه پکیج به اسم attendance بسازیم که ورود و خروج پرسنل سازمان رو مدیریت کنه.

ساختار کلی:

یه متغیر خصوصی به اسم repo داریم که شناسه‌ی پرسنل‌هایی که الان داخل سازمان هستن رو نگه می‌داره.

دو تابع اصلی داریم:

```
func CheckIn(id string) bool
func CheckOut(id string) bool
```

انتظار:

- CheckIn: اگر پرسنل قبلاً وارد نشده باشه، ورودش رو ثبت کنه و true برگردونه. اگر قبلاً وارد شده باشه، false برگردونه.
- CheckOut: اگر پرسنل داخل سازمان باشه، خروجش رو ثبت کنه و true برگردونه. اگر نباشه، false برگردونه.

یعنی برای اینکه بفهمیم کسی الان داخل سازمان هست یا نه، کافیه CheckIn رو براش صدا بزنیم. اگر false برگشت، یعنی قبلاً وارد شده و هنوز خارج نشده.

ظرفیت سازمان:

- یه متغیر خصوصی به اسم capacity داریم که ظرفیت کل سازمان رو مشخص می‌کنه.
- مقدار اولیه‌ی این متغیر با تابع New(cap int) تنظیم می‌شه. این تابع فقط عددهای مثبت رو قبول می‌کنه.
- اگر کسی New رو فراموش کرد، تابع init به صورت پیش‌فرض ظرفیت رو روی 100 تنظیم می‌کنه.

گزارش غیبت:

تابعی بنویس به اسم AbsentCount() که تعداد افراد غایب رو بر اساس ظرفیت کل و تعداد افراد حاضر حساب کنه:

```
func AbsentCount() int {
    // capacity - تعداد افراد حاضر
}
```

تمرین 3: پکیج امتیاز دهی به راننده تاکسی اینترنتی

در این تمرین قراره یک پکیج ساده اما کاربردی برای مدیریت امتیاز راننده‌های تاکسی اینترنتی طراحی کنیم. این تمرین هم مفاهیم پایه‌ای مثل map و اعتبارسنجی رو پوشش می‌ده، هم به دنیای واقعی نزدیکه و قابل توسعه‌ست.

هدف پکیج

پکیجی به نام rating بساز که امتیاز هر راننده را بر اساس شناسه‌ی یکتای او نگه‌داری کند.
امتیازها به صورت عدد اعشاری بین 1 تا 5 ثبت می‌شوند.

ساختار داده

در سطح پکیج، یک map خصوصی تعریف کن که شناسه‌ی راننده را به امتیاز او نگه‌داری کند:

```
var repo = map[string]float64{}
```

توابع مورد نیاز

1-ثبت امتیاز راننده

```
func Rate(driverID string, score float64) bool
```

- اگر امتیاز معتبر باشد (بین 1 تا 5)، در map ذخیره شود و true برگردد.
- اگر امتیاز خارج از محدوده باشد، false برگردد و هیچ تغییری در داده‌ها ایجاد نشود.

2- دریافت امتیاز راننده

```
func GetRating(driverID string) float64
```

- اگر راننده قبلاً امتیاز گرفته باشد، مقدار آن برگردد.
- اگر راننده وجود نداشته باشد، مقدار 0 برگردد.

نکات تکمیلی

- تابع Rate فقط آخرین امتیاز را نگه‌داری می‌کند. یعنی اگر چند بار امتیاز داده شود، مقدار قبلی جایگزین می‌شود.