وقتی تکرار آزاردهنده میشه...

در درسنامه‌ی قبلی (آرایه با طول ثابت)، برنامه‌ای نوشتیم که نمرات 3 درس (ریاضی، فارسی، تربیت بدنی) رو برای 4 دانش‌آموز دریافت می‌کرد، میانگین نمرات هر کدوم رو حساب می‌کرد، و در نهایت، همه‌ی میانگین‌ها رو چاپ می‌کرد.

کدی شبیه این:

```
var allGrades [4][3]float32
var averages [4]float32

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی آرین چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[0][0], &allGrades[0][1], &allGrades[0][2])

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی ندا چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[1][0], &allGrades[1][1], &allGrades[1][2])

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی امید چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[2][0], &allGrades[2][1], &allGrades[2][2])

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی نیما چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[3][0], &allGrades[3][1], &allGrades[3][2])

averages[0] = (allGrades[0][0] + allGrades[0][1] + allGrades[0][2]) / 3
averages[1] = (allGrades[1][0] + allGrades[1][1] + allGrades[1][2]) / 3
averages[2] = (allGrades[2][0] + allGrades[2][1] + allGrades[2][2]) / 3
averages[3] = (allGrades[3][0] + allGrades[3][1] + allGrades[3][2]) / 3

fmt.Println(
    "میانگین نمرات آرین، ندا، امید و نیما به ترتیب:",
    averages[0], averages[1], averages[2], averages[3],
)
```

برنامه کار می‌کنه... ولی یه مشکل جدی داره:

دستورهای تکراری!

به بخش دریافت نمرات نگاه کن:

```
fmt.Println("نمره ریاضی، فارسی و تربیت بدنی آرین چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[0][0], &allGrades[0][1], &allGrades[0][2])

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی ندا چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[1][0], &allGrades[1][1], &allGrades[1][2])

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی امید چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[2][0], &allGrades[2][1], &allGrades[2][2])

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی نیما چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&allGrades[3][0], &allGrades[3][1], &allGrades[3][2])
```

فقط اندیس‌ها دارن عوض می‌شن. یا توی محاسبه‌ی میانگین‌ها:

```
averages[0] = ...
averages[1] = ...
...
```

همه‌چیز داره تکرار میشه، ولی الگوش یکیه!

حالا فکر کن می‌شد یه بار این دستور رو بنویسیم و بگیم:

"این کار رو برای همه‌ی دانش‌آموزها انجام بده"

یه چیزی مثل:

```
fmt.Println("نمره‌ها رو وارد کن:")
fmt.Scanln(&allGrades[?][0], &allGrades[?][1], &allGrades[?][2])
```

البته این کد واقعی نیست. اون علامت سؤال فقط برای توضیحه.
اما ایده‌ی اصلی واضحه: می‌خوایم بتونیم اون بخش از برنامه که تکرار میشه رو **خودکار و هوشمند** انجام بدیم.

اینجا دقیقاً جاییه که با یک ابزار مهم در برنامه‌نویسی آشنا می‌شیم:

حلقه‌ها (Loops)

تو این درسنامه با مفاهیم پایه‌ای حلقه‌ها شروع می‌کنیم، و قدم‌به‌قدم جلو می‌ریم تا در نهایت، بتونیم همین برنامه رو به شکلی ساده‌تر، کوتاه‌تر و حرفه‌ای‌تر بازنویسی کنیم.

حلقه بی نهایت

تا حالا شده بخوای یه دستور یا چند دستور رو **بارها و بارها** اجرا کنی، اون‌قدر که اصلاً ندونی چند بار قراره این کار انجام بشه؟
اینجا دقیقاً جاییه که از ساده‌ترین نوع حلقه در Go استفاده می‌کنیم :**حلقه‌ی بی‌نهایت**.

ساختار کلی این حلقه تو زبان Go به شکل زیره:

```
for {
    statement
}
```

for

کلمه‌ی کلیدی (keyword) مخصوص ساخت حلقه در زبان Go هست.
هر حلقه‌ای تو Go باید با forشروع بشه — یعنی:

"برای تکرار این بخش، آماده شو!"

{}

این بخش رو بهش می‌گیم **بدنه‌ی حلقه** یا **بلوک کد**.
هر دستوری که بین این دو آکولاد قرار بگیره، اون دستور توی هر بار تکرار حلقه اجرا میشه.

statement

این همون دستور اجراییه که قراره بارها تکرار بشه.
میتونه فقط یه خط ساده باشه، یا چند خط کد پشت‌سرهم.

استفاده از حلقه‌ی بی‌نهایت خیلی وقت‌ها مفیده، اما باید بدونیم که این نوع حلقه می‌تونه **کمی خطرناک** هم باشه — مخصوصاً وقتی **راهی برای متوقف کردنش نذاریم!** چون این حلقه باعث میشه برنامه توی همون بخش بمونه و **هیچ‌وقت به خط‌های بعدی نرسه.**

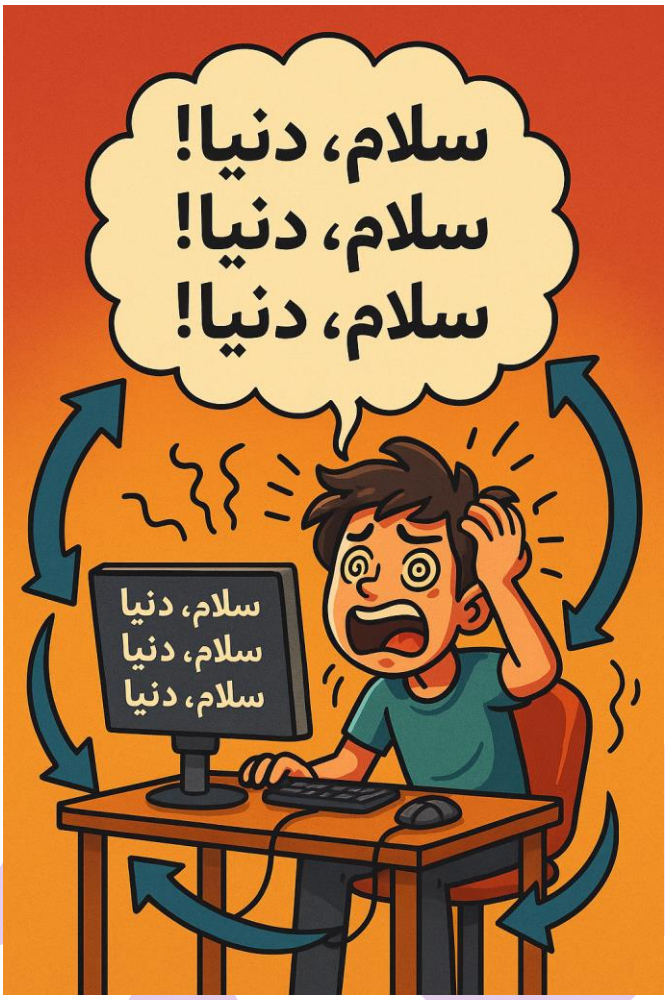
مثال:

```
package main


import “fmt”


func main() {
    for {
        fmt.Println(“سلام، دنیا!”)
    }
}
```

این برنامه به‌صورت مداوم (و بدون توقف) توی خروجی می‌نویسه:

```
سلام، دنیا!
سلام، دنیا!
سلام، دنیا!
...
```

و این ادامه داره... و ادامه داره... و ادامه داره...



مشکل چیه؟

تا وقتی که **برنامه رو دستی نبندی** (مثلاً با Ctrl + C یا بستن پنجره) این حلقه همچنان اجرا میشه و هیچ‌وقت تموم نمیشه.

اگه توی بدنه‌ی حلقه کدی باشه که:

- از رم زیاد استفاده کنه
- پردازنده رو درگیر کنه
- یا مدام روی خروجی بنویسه (مثل همین مثال)

ممکنه باعث بشه سیستم **کند بشه، داغ کنه یا حتی هنگ کنه**.

مثالی که دیدیم شاید خیلی کاربردی نباشه، ولی تونستی باهاش مفاهیم اولیه حلقه بی‌نهایت رو خوب بفهمی. بعضی وقت‌ها لازمه کاری رو بارها و بارها، حتی به تعداد نامحدود، انجام بدیم! اما یه فرق خیلی بزرگ داره: **معمولاً بین هر بار اجرا یه توقف یا وقفه طبیعی یا دستیه.**

مثلاً یادت هست دستور fmt.Scan() چطوری کار می‌کرد؟

وقتی برنامه به این دستور می‌رسید، همونجا وایمیستاد تا کاربر یه چیزی وارد کنه. وقتی کاربر جواب رو داد و اینتر زد، برنامه می‌رفت خط بعد و ادامه دستورات رو اجرا می‌کرد.

حالا فرض کن یه برنامه داشته باشیم که داخلش یه حلقه بی‌نهایت باشه و داخل این حلقه از fmt.Scan() استفاده کنیم!

تو این حالت، هر بار که حلقه اجرا می‌شه، برنامه منتظر می‌مونه تا کاربر یه ورودی بده، بعد دوباره دستورات داخل حلقه اجرا می‌شن. به این ترتیب، اون مشکلی که تو مثال قبل داشتیم دیگه پیش نمیاد و این یکی از کاربردهای خیلی خوب حلقه بی‌نهایت هست.

حالا یه سناریو رو تصور کن:

می‌خوایم برنامه‌ای داشته باشیم که ازمون ساعت رو بپرسه و به ما بگه الان وقت صبحانه‌ست، ناهار یا شام! نمی‌خوایم برنامه فقط یه بار این کارو انجام بده، بلکه می‌خوایم تا وقتی برنامه بازه، مرتب ازمون بپرسه:

ساعت چنده؟
ما جواب می‌دیم، برنامه نتیجه رو می‌گه، و دوباره همین سوال رو تکرار می‌کنه.

به برنامه زیر دقت کن:

```
package main


import "fmt"


func main() {
    var hour int


    for {
        fmt.Print("(عدد بین 0 تا 23 وارد کن)ساعت چند؟ : ")
        fmt.Scanln(&hour)


        if hour >= 6 && hour < 10 {
            fmt.Println("الان وقت صبحونه‌ست!")
        } else if hour >= 12 && hour < 15 {
            fmt.Println("الان وقت ناهاره!")
        } else if hour >= 19 && hour <= 21 {
            fmt.Println("الان وقت شامه!")
        } else if hour == 24 || hour < 0 || hour > 23 {
            fmt.Println("عدد وارد شده معتبر نیست!")
        } else {
            fmt.Println("الان هیچ وعده‌ای نیست... استراحت کن")
        }


        fmt.Println() // خط خالی برای زیبایی
    }
}
```

این برنامه یه حلقه بی‌نهایت داره که هر بار از کاربر می‌پرسه ساعت چند هست (بین 0 تا 23). بعد ورودی رو می‌گیره و بررسی می‌کنه:

- اگه ساعت بین 6 تا 9 بود، می‌گه: الان وقت صبحونه‌ست
- اگه بین 12 تا 14 بود، می‌گه: الان وقت ناهاره
- اگه بین 19 تا 21 بود، می‌گه: الان وقت شامه
- اگه عدد وارد شده خارج از بازه 0 تا 23 یا مساوی 24 بود، اخطار می‌ده که "عدد وارد شده معتبر نیست"
- در غیر این صورت می‌گه "الان هیچ وعده‌ای نیست... استراحت کن"

بعد یه خط خالی چاپ می‌کنه که خروجی مرتب‌تر باشه و دوباره سوال رو تکرار می‌کنه تا وقتی که برنامه رو ببندی.

نتیجه‌گیری

حلقه‌های بی‌نهایت ابزار قدرتمندین، ولی باید با **هوشیاری و کنترل درست** ازشون استفاده کنیم. معمولاً با دستوراتی مثل break (که به زودی بررسی میکنیم) یا شرط‌های داخل حلقه، راه خروج براشون می‌سازیم.

شرط خروج از حلقه بی‌نهایت

وقتی از حلقه بی‌نهایت استفاده می‌کنیم، معمولاً نمی‌خوایم واقعا تا بی‌نهایت ادامه پیدا کنه. برای همین باید یه **شرط خروج** مشخص کنیم که با برآورده شدنش، حلقه متوقف بشه و برنامه بره سراغ بقیه دستورات.

چند مثال ساده:

- در یک حلقه‌ی بی‌نهایت از کاربر ورودی بگیریم، اگه ورودی qبود، حلقه تموم بشه.
- مرتب عدد بگیریم، اگر عدد منفی بود، حلقه رو ترک کنیم.
- عملیات بانکی انجام بدیم، ولی اگه ساعت به بازه‌ی مشخصی رسید، متوقف کنیم.

برای این کار معمولا از دستور if به همراه کلیدواژه break استفاده می‌کنیم.

```
for {
    if condition {
        break
    }

    statement
}
```

مثال

```
for {
    var input string
    fmt.Print("یه چیزی بنویس (رو بزن q برای خروج): ")
    fmt.Scanln(&input)

    if input == "q" {
        break // از حلقه خارج شو
    }

    fmt.Println("تو نوشتی:", input)
}
```

در این برنامه تا وقتی که کاربر q نزنه، هی ازش ورودی می‌گیره و چاپ می‌کنه.
وقتی q وارد کرد، با break از حلقه خارج می‌شه.

حلقه بی نهایتی که در عمل محدود میشه

استفاده از حلقه‌ی بی‌نهایت با break فقط مخصوص وقتی نیست که بخوایم به تعداد نامشخصی از کاربر ورودی بگیریم. این ترکیب کاربردهای دیگه‌ای هم داره و می‌تونه خیلی جاها به دردمون بخوره.

می‌تونیم یه متغیر رو قبل از شروع حلقه تعریف کنیم، بعد داخل حلقه هر بار اون متغیر رو یه جوری آپدیت کنیم (مثلاً یکی یکی زیادش کنیم).

در کنارش هم یه شرط خروج بذاریم که وقتی اون متغیر به یه مقدار خاص رسید، break بزنه و از حلقه خارج شه.

یعنی با اینکه ظاهر حلقه‌مون بی‌نهایته، ولی در عمل بعد از چندبار اجرا خودش تموم می‌شه!
پس می‌تونیم بگیم حلقه‌مون بی‌نهایت نیست، فقط شکلش اینطوریه، و تعداد اجراش کاملاً مشخص و قابل محاسبه‌ست.

به مثال زیر توجه کن

```
var number uint = 1
var sum uint = 0

for {
    sum += number

    if number == 10 {
        break
    }

    number++
}
```

این برنامه چی کار می‌کنه؟

می‌خواد عددهای 1 تا 10 رو با هم جمع کنه.

یعنی اینو حساب کنه:

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

و در نهایت مقدار sum (که حاصل جمعه) رو نشون بده.

قدم به قدم بررسی کد:

```
var number uint = 1
var sum uint = 0
```

اینجا دوتا متغیر تعریف کردیم:

- number از 1 شروع می‌کنه، قراره عددهایی باشه که جمع می‌زنیم.
- sum از صفر شروع می‌کنه و قراره جمع عددها رو تو خودش نگه داره.

حالا حلقه بی‌نهایت:

```
for {
    sum += number
```

این خط یعنی: هر بار که این حلقه اجرا می‌شه، عدد فعلی (number) رو به sum اضافه کن.

مثلاً بار اول number = 1، پس sum می‌شه 1.
بار بعد number = 2، sum می‌شه 3، و همین‌طور ادامه داره...

```
if number == 10 {
    break
}
```

این یعنی: اگه عدد فعلی شد 10، دیگه جمع بسه!
با break از حلقه بی‌نهایت بپر بیرون.

```
number++
```

این خیلی مهمه! این خط باعث می‌شه مقدار number هر بار یکی زیاد بشه.

مثلاً از 1 بشه 2، بعد بشه 3، 4، ... تا برسه به 10.

اگه این خط نبود، number همیشه 1 می‌موند و هیچ‌وقت شرط number == 10 درست نمی‌شد، یعنی برنامه گیر می‌کرد تو یه حلقه بی‌نهایت!

وقتی حلقه تموم شد...

بعد از اینکه number رسید به 10، شرط برقرار می‌شه، break اجرا می‌شه و برنامه می‌پره بیرون از حلقه.

نکته:

تو حلقه‌ها، **ترتیب قرار گرفتن شرط خروج اهمیت زیادی داره**.

اگه شرط break رو **قبل از عملیات اصلی** بذاری، اون عملیاتی که مدنظرت هست ممکنه درست اجرا نشه (یعنی خطای منطقی اتفاق میوفته) یا **هیچ‌وقت اجرا نشه**.

اضافه کردن شرط به حلقه

تو مثال قبلی دیدیم که چطوری می‌شه با نوشتن یه شرط داخل حلقه، کاری کرد که وقتی اون شرط برقرار شد، برنامه با دستور break از حلقه بزنه بیرون.

اما تو زبان Go یه روش راحت‌تر هم برای این کار داریم!

می‌تونیم **مستقیم بعد از کلمه‌ی for شرطمون رو بنویسیم.**

فقط یه نکته مهم این وسط هست:

اون شرطی که جلوی for می‌نویسیم، با شرطی که داخل بدنه‌ی حلقه با if می‌نوشتیم فرق داره.

- شرط کنار for می‌گه: تا وقتی این شرط **برقراره**، حلقه رو ادامه بده
- ولی شرط داخل if می‌گه: اگه این شرط **برقرار شد**، از حلقه خارج شو

مثلاً اگه داخل حلقه نوشته بودی:

```
for {
    ...
    if x == 10 {
        break
    }
    ...
}
```

برای اینکه همین رفتار رو با شرط کنار for بنویسی، باید اون شرط رو تغییر بدی:

```
for x <= 10 {
    ...
}
```

چون اینجا دیگه خود حلقه مسئول چک کردن شرطه، نه یه if وسط کار.

بیا اول یه نگاهی بندازیم به ساختار جدید حلقه‌ای که تو Go خیلی به درد می‌خوره:

```
for condition {
    statement
}
```

- تا وقتی شرطی که جلوی for نوشتیم برقرار باشه، بدنه‌ی حلقه هی تکرار می‌شه.
- به محض اینکه اون شرط false بشه، حلقه تموم می‌شه و برنامه می‌ره سراغ خط بعدی.

همون برنامه‌ای که جمع اعداد 1 تا 10 رو حساب می‌کرد رو با این ساختار بازنویسی می‌کنیم:

```
var number uint = 1
var sum uint = 0

for number <= 10 {
    sum += number
    number++
}

fmt.Println("جمع اعداد 1 تا 10 می‌شود:", sum)
```

چه تغییری کرده؟

تو نسخه‌ی قبلی، از یه حلقه بی‌نهایت {} for استفاده کرده بودیم و شرط خروج رو این‌طوری می‌نوشتیم:

```
if number == 10 {
    break
}
```

ولی اینجا، همون شرط رو مستقیم جلوی for آوردیم:

```
for number <= 10 {
    ...
}
```

یعنی دیگه لازم نیست توی بدنه‌ی حلقه شرط بذاریم و break بزنیم.
خود حلقه می‌دونه کی باید تموم بشه. ساده‌تر، تمیزتر و خواناتر!

فقط یه نکته‌ی مهم:

اون شرطی که توی نسخه‌ی قبلی داخل حلقه نوشته بودیم (if number == 10) حالا شده (number <= 10) چون رفتار حلقه فرق کرده:

- داخل if می‌گفتیم: اگه به این نقطه رسیدی، قطع کن!
- کنار for می‌گیم: تا وقتی این شرط برقراره، ادامه بده!

یعنی دیگه نیازی به break نیست، چون تا وقتی number کوچکیتر یا مساوی 10 باشه، حلقه ادامه پیدا می‌کنه.

حالا کی می‌تونیم از این ساختار استفاده کنیم؟

وقتی شرایط زیر برقرار باشه:

- مقدار اولیه‌ی شرط رو **قبل از شروع حلقه** می‌تونی تعیین کنی.
- شرط خروجت ساده و قابل‌تبدیل به یه شرط بولی باشه.
- لازم نباشه بدنه‌ی حلقه **حتماً حداقل یه بار اجرا بشه**.

تو این شرایط، کاملاً منطقیه که به‌جای ترکیب {} for و break از for condition استفاده کنی.

اما همیشه نمی‌شه!

بعضی وقتا استفاده از حلقه‌ی بی‌نهایت ضروری‌تره. مثلاً:

- بدنه‌ی حلقه باید **حتماً یه بار اجرا بشه**، چون شرط خروج بعد از اولین اجرا معلوم می‌شه.
- شرط خروجت بستگی به چیزی داره که **داخل حلقه تولید می‌شه** (مثلاً یه ورودی از کاربر)
- اصلاً نمی‌دونی شرطت قبل از حلقه چیه یا از کجا قراره بیاد!

تو این حالت‌ها، بهتره از همون {} for و break استفاده کنی. نه فقط برای اینکه جواب بده، بلکه برای اینکه کدت واضح‌تر و قابل‌درک‌تر باشه.

مفهوم Variable Scope داخل حلقه

قبلاً تو درسنامه "مقایسه و شرط" با مفهوم **scope** آشنا شدیم. تو حلقه هم همین قضیه صدق می‌کنه! یعنی وقتی یه متغیر رو داخل **بدنه حلقه** تعریف کنی، اون متغیر فقط همونجا وجود داره و فقط داخل همون حلقه قابل دسترسه. بیرون از حلقه انگار اصلاً همچین متغیری وجود نداره.

مثلاً این کد رو نگاه کن:

```
for {
    var input string
    fmt.Print("یه چیزی بنویس (رو بزن q برای خروج): ")
    fmt.Scanln(&input)

    if input == "q" {
        break // از حلقه خارج شو
    }

    fmt.Println("تو نوشتی:", input)
}

fmt.Println("از حلقه خارج شد، چون آخرین چیزی که نوشتی بود: ", input) // اینجا خطا میده
```

چون متغیر input داخل حلقه تعریف شده، **خارج از حلقه نمی‌تونیم بهش دسترسی داشته باشیم** و برنامه کامپایل نمیشه چون به متغیری اشاره کردیم که تعریف نشده.

اگه نیاز داری **بعد از حلقه** هم به یه متغیر دسترسی داشته باشی، باید اون رو **قبل از حلقه** تعریف کنی، مثلاً:

```
var input string


for {
    fmt.Print("یه چیزی بنویس (رو بزن q برای خروج): ")
    fmt.Scanln(&input)

    if input == "q" {
        break // از حلقه خارج شو
    }

    fmt.Println("تو نوشتی:", input)
}

fmt.Println("از حلقه خارج شد، چون آخرین چیزی که نوشتی ", input, " بود") // اینجا دیگه مشکلی نداره
```

به طور کلی بهتره متغیرهایی که فقط داخل حلقه استفاده میشن، **داخل همون حلقه تعریف بشن** تا وقتی حلقه تموم شد، اون متغیرها هم از حافظه پاک بشن و برنامه سبک‌تر و بهینه‌تر باشه.

ولی اگر بخوای بعد از حلقه هم ازشون استفاده کنی، باید متغیر رو بیرون تعریف کنی.

حالا بریم سراغ برنامه جمع اعداد 1 تا 10 که قبل تر نوشتیم:

```
var number uint = 1
var sum uint = 0

for number <= 10 {
    sum += number
    number++
}

fmt.Println("جمع اعداد 1 تا 10 می شود: ", sum)
```

در این برنامه، متغیر number فقط داخل حلقه استفاده شده ولی بیرون از حلقه تعریف شده. پس به نظر میاد بهتر باشه که اون رو داخل حلقه تعریف کنیم، درست؟

اما اگه بخوایم این کار رو بکنیم و number رو داخل حلقه تعریف کنیم، دو مشکل خیلی جدی پیش میاد:

1. برنامه اصلاً کامپایل نمی‌شه چون شرط حلقه number <= 10 هست و تو اون خط هنوز هیچ متغیری به اسم number تعریف نشده.
2. حتی فرض کنیم برنامه کامپایل هم بشه! با هر بار اجرای حلقه، number دوباره تعریف و مقدار اولیه 1 می‌گیره. این باعث میشه حلقه هیچ وقت تموم نشه و در واقع تبدیل به حلقه بی‌نهایت بشه! یعنی خطای منطقی رخ میده چون ما قصد داشتیم حلقه بعد از چند بار اجرا تموم بشه.

پس نمی‌تونیم به این شکل number رو داخل حلقه تعریف کنیم.

بیایم یه بار دیگه با دقت نگاه کنیم به مسئله و محدودیت‌ها:

1. ما می‌خوایم number رو داخل حلقه تعریف کنیم چون فقط داخل حلقه بهش نیاز داریم.
2. اگر داخل حلقه تعریفش کنیم، شرط حلقه به مشکل می‌خوره چون شرط قبل از شروع حلقه اجرا میشه و متغیر هنوز تعریف نشده (خطای کامپایل).
3. حتی اگر برنامه کامپایل بشه، با تعریف داخل حلقه، هر بار مقدار number 1 میشه و حلقه بی‌نهایت میشه (خطای منطقی).

اینجاست که زبان Go یه ساختار کامل‌تر و هوشمندانه‌تر برای حلقه‌ها ارائه داده تا این مشکل رو حل کنه. ساختاری که می‌تونه متغیر شمارنده رو دقیقاً همونجا تعریف کنه و شرط حلقه رو هم کنترل کنه بدون این که دچار این مشکلات بشیم.

ساختار کاملتر حلقه

تا اینجا با حلقه‌های ساده مثل for condition {} آشنا شدیم. ولی حالا می‌خوایم یه مدل دیگه از حلقه‌ها رو ببینیم که بهش می‌گن **حلقه‌ی شمارشی** یا همون **for سه‌تایی** .این مدل خیلی پرکاربرده، مخصوصاً وقتی می‌خوایم یه کاری رو دقیقاً یه تعداد مشخص تکرار کنیم.

ساختارش این شکلیه:

```
for initialization; condition; post {
    statement
}
```

شروع (Initialization)

اولین قسمت حلقه، جاییه که متغیر شمارنده رو تعریف می‌کنی یا مقدار اولیه می‌دی.

شرط ادامه (Condition)

این شرطیه که تعیین می‌کنه حلقه ادامه پیدا کنه یا نه.

تغییر (Post / Update)

بعد از هر بار اجرای بدنه‌ی حلقه، این بخش اجرا میشه.
معمولاً برای افزایش یا کاهش متغیر شمارنده‌ست.

بدنه حلقه (Body)

کدی که می‌خوای بارها اجرا بشه. داخل { } قرار می‌گیره.

حالا نکته‌ی باحال این ساختار چیه؟

اینه که متغیری که اون اول تعریف می‌کنی (مثلاً i := 0) فقط و فقط داخل همون حلقه وجود داره. به‌محض اینکه حلقه تموم بشه، اون متغیر هم پاک میشه و بیرون حلقه دیگه بهش دسترسی نداری. این یعنی تمیز و مرتب؛ نه متغیر اضافی، نه سردرگمی.

این نوع حلقه رو کی استفاده می‌کنیم؟

هر وقت بخوای یه کاری رو **تعداد مشخصی** تکرار کنی، این مدل حلقه بهترین انتخابه.
درسته با for condition {} یا ترکیب for {} و break هم می‌تونی همین کارو بکنی، ولی این روش:

- کوتاه‌تره
- خوندنش راحت‌تره
- متغیر حلقه خودش پاک میشه، لازم نیست نگرانش باشی

برنامه جمع اعداد 1 تا 10 رو یکبار دیگه با این ساختار جدید بازنویسی میکنیم و بررسی میکنیم

```
var sum uint = 0


for number := 1; number <= 10; number++ {
    sum += number
}


fmt.Println("جمع اعداد 1 تا 10 می شود: ", sum)
```

برخلاف حالت for condition { ... } که فقط یه شرط داشت، این ساختار از **سه بخش** داره:

Initialization (مقداردهی اولیه)

فقط یک‌بار در ابتدای حلقه اجرا میشه.

در این مثال number := 1

متغیر number ایجاد و مقدار اولیه‌ی 1 بهش داده میشه.

Condition (شرط اجرای حلقه)

قبل از هر بار اجرای بدنه‌ی حلقه، این شرط بررسی میشه.

اگر true بود، حلقه اجرا میشه؛ اگه false بود، می‌پریم بیرون.

در این مثال number <= 10

یعنی تا زمانی که number برابر 10 نشده، ادامه بده.

Post (کد پایانی هر تکرار)

بعد از اجرای بدنه‌ی حلقه، این قسمت اجرا میشه. معمولاً برای به‌روزرسانی متغیر حلقه‌ست.

در این مثال number++

یعنی عدد رو یکی زیاد کن.

این حلقه از number = 1 شروع می‌کنه و تا number = 10 ادامه می‌ده.

چون وقتی number == 11 بشه، شرط number <= 10 دیگه false میشه.

نکته:

توی این نوع حلقه (همون for سه‌تایی)، وقتی می‌خوای متغیر حلقه رو تو بخش initialization تعریف کنی، **حتماً حتماً باید از := استفاده کنی** یعنی همون **short declaration**

مثلاً باید اینطوری بنویسی:

```
for i := 1; i <= 10; i++ {
    // ...
}
```

اگه بیای از var استفاده کنی مثل این:

```
for var i = 1; i <= 10; i++ {
    // ...
}
```

برنامه کلاً کامپایل نمیشه و بهت خطا میده!

بخش های اختیاری در حلقه for ; ; {}

توی ساختار حلقه‌ی for سه‌تایی یه نکته‌ی جالب وجود داره:
اون سه بخشی که بعد از for می‌نویسیم — یعنی initialization، condition و post —
همشون اختیاری‌ان! یعنی هر کدوم رو خواستی می‌تونی خالی بذاری. برنامه هیچ مشکلی نداره،
فقط باید بدونی داری چی‌کار می‌کنی.

فقط بخش initialization رو خالی بذاریم

اگه قبل از حلقه، متغیرت رو تعریف کرده باشی، دیگه لازم نیست توی خود for دوباره
بنویسیش:

```
var sum uint = 0
var number uint = 1


for ; number <= 10; number++ {
    sum += number
}


fmt.Println("جمع اعداد 1 تا 10 می شود: ", sum)
```

فقط condition رو خالی بذاریم

اگه شرط خاصی نداری یا می‌خوای با یه if + break خودت کنترلش کنی، می‌تونی بخش شرط
رو حذف کنی:

```
var sum uint = 0


for number := 1; ; number++ {
    sum += number
    if number == 10 {
        break
    }
}


fmt.Println("جمع اعداد 1 تا 10 می شود: ", sum)
```

فقط post رو خالی بذاریم

اگه ترجیح می‌دی آپدیت متغیرت رو وسط حلقه انجام بدی، post رو خالی بذار:

```
var sum uint = 0


for number := 1; number <= 10; {
    sum += number
    number++
}


fmt.Println("جمع اعداد 1 تا 10 می شود: ", sum)
```

هر سه بخش رو خالی بذاریم

وقتی هر سه بخش خالی باشه، حلقه تبدیل میشه به یه **حلقه بی‌نهایت**.
یعنی تا وقتی خودت نگفتی "کافیه!" و از break یا return استفاده نکردی، همین‌طور ادامه می‌ده.

از نظر اجرا؟ **هیچ فرقی ندارن**. هر دوتاشون حلقه‌ی بی‌نهایت هستن.

ولی از نظر ظاهر و خوانایی:

for {} خیلی ساده‌تر و تمیزتره

for ; ; {} همون کاری رو می‌کنه ولی بی‌خود شلوغش کرده

کاربرد continue در حلقه

فرقی نمی‌کنه داری از کدوم مدل حلقه استفاده می‌کنی، گاهی وقتا یه شرایط خاص پیش میاد که می‌خوای **یه دور از حلقه رو نادیده بگیری** و بری سراغ بعدی.

اینجا دقیقاً جاییه که continue به دادت می‌رسه.

فرض کن می‌خوای همون برنامه‌ی جمع عددهای 1 تا 10 رو بنویسی، اما این بار فقط می‌خوای **اعداد زوج** رو جمع بزنی.

خب دوتا راه داری

راه اول: حلقه‌ات رو فقط روی اعداد زوج بچرخونی

یعنی از اول فقط روی اعداد 2، 4، 6، 8، 10 حرکت کنیم.
پس:

```
var sum int = 0


for number := 2; number <= 10; number += 2 {
    sum += number
}


fmt.Println("جمع اعداد زوج 1 تا 10 می شود: ", sum)
```

راه دوم: از همون حلقه‌ی قبلی استفاده کنیم ولی شرط بگذاریم

این بار می‌خوای از number := 1 شروع کنی و number++ بری جلو، اما توی بدنه‌ی حلقه چک کنی که:

آیا این عدد زوجه؟ اگه نیست، بی‌خیالش... برو بعدی!

برای این کار، از continue استفاده می‌کنیم.

طبق تعریف، عددی **زوجه** که باقی‌مانده‌ی تقسیمش بر 2، صفر باشه.

if number % 2 != 0 عدد **فرده**، پس **برو بعدی**

```
var sum int = 0


for number := 1; number <= 10; number++ {
    if number % 2 != 0 {
        continue
    }
    sum += number
}


fmt.Println("جمع اعداد زوج 1 تا 10 می شود: ", sum)
```

اینجا continue باعث میشه وقتی عدد فرده، اون دور از حلقه رو ول کنیم و مستقیم بریم سراغ بعدی

- continue یه ابزار خیلی کاربردیه برای وقتایی که **نمی‌خوای همه‌ی کدهای داخل حلقه برای یه حالت خاص اجرا بشن**
- فقط با یه شرط ساده و یه continue، می‌تونی حلقه‌ت رو هوشمندتر کنی و از اضافه‌کاری خلاص شی

پیمایش آرایه با حلقه

همون‌طور که قبلاً گفتیم، آرایه یه ساختاره برای نگهداری چندتا مقدار از یه نوع خاصه؛ مثلاً چندتا عدد، چندتا نمره، یا چندتا نام. اما نکته‌ی جالبش اینه که این داده‌ها معمولاً از نظر معنایی هم به هم ربط دارن؛ مثلاً:

- نمره‌های یه دانش‌آموز
- قیمت‌های یه کالا تو چند روز مختلف

وقتی این مقادیر به هم ربط دارن، معمولاً می‌خوایم همه‌شون رو با هم بررسی کنیم. مثلاً:

- اگه بخوای **میانگین نمره‌ها** رو حساب کنی، باید همه رو جمع بزنی
- اگه بخوای **بالاترین نمره** رو پیدا کنی، باید همه رو با هم مقایسه کنی

خب حالا چطور این بررسی رو انجام بدیم؟
اینجا دقیقاً همون جاییه که **حلقه‌ها** به کارمون میان!

روش غیراصولی: محاسبه میانگین بدون حلقه

فرض کن قراره میانگین 3 تا نمره (مثلاً ریاضی، فارسی، تربیت‌بدنی) رو حساب کنیم.

```
var grades [3]float32
var average float32

fmt.Println("(نمره ریاضی، فارسی و تربیت بدنی دانش آموز چیه؟ (به ترتیب وارد کن")
fmt.Scanln(&grades[0], &grades[1], &grades[2])

average = (grades[0] + grades[1] + grades[2]) / len(grades)

fmt.Println("میانگین نمرات دانش آموز:", average)
```

فعلاً همه‌چی خوبه... اما صبر کن!

اگه بخوای تعداد نمره‌ها رو از 3 تا به 5 تا برسونی، نه تنها باید طول آرایه رو تغییر بدی، بلکه باید خود فرمول جمع کردن رو هم دستی آپدیت کنی!

```
var grades [5]float32
var average float32

fmt.Println("(نمره ریاضی، فارسی و تربیت بدنی و 2 نمره دیگر دانش آموز چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&grades[0], &grades[1], &grades[2], &grades[3], &grades[4])

average=(grades[0] + grades[1] + grades[2] + grades[3] + grades[4])) / len(grades)

fmt.Println("میانگین نمرات دانش آموز:", average)
```

اوضاع داره از کنترل خارج می‌شه.

حالا فکر کن بخوای میانگین **50 تا نمره** رو بگیری. باید 50 بار grades[i] رو بنویسی؟ معلومه که نه!

روش اصولی: استفاده از حلقه برای پیمایش آرایه

اینجا حلقه به کمکت میاد. می‌تونی با یه حلقه‌ی ساده، رو همه عناصر آرایه حرکت کنی، جمعشون کنی، و بعد خیلی راحت میانگین بگیری:

```
var grades [5]float32
var average float32

fmt.Println("نمره ریاضی، فارسی و تربیت بدنی دانش آموز چیه؟ (به ترتیب وارد کن)")
fmt.Scanln(&grades[0], &grades[1], &grades[2], &grades[3], &grades[4])

var sum float32 = 0
for i := 0; i < len(grades); i++ {
    sum = sum + grades[i]
}

average= sum / len(grades)

fmt.Println("میانگین نمرات دانش آموز:", average)
```

نسخه‌ی منعطف‌تر: استفاده از حلقه حتی برای ورودی گرفتن

بریم یه قدم جلوتر! حتی اون fmt.Scanln(...) طولانی هم می‌تونه تبدیل بشه به یه حلقه‌ی ساده برای گرفتن ورودی:

```
var grades [5]float32
var average float32

for i := 0; i < len(grades); i++ {
    fmt.Print("یه نمره وارد کن: ")
    fmt.Scan(&grades[i])
}

var sum float32 = 0
for i := 0; i < len(grades); i++ {
    sum = sum + grades[i]
}

average= sum / len(grades)
```

حالا اگه بخوای تعداد نمره‌ها رو از 5 به 10 برسونی، فقط طول آرایه رو از 5 به 10 تغییر می‌دی — هیچ چیز دیگه‌ای لازم نیست دست بزنی. این یعنی **کد تمیز، قابل نگهداری و منعطف!**

حلقه for range

وقتی می‌خوایم روی تمام عناصر یک آرایه یا لیست پیمایش انجام بدیم، یه روش ساده‌تر از حلقه‌ی کلاسیک for i := 0; i < len(arr); i++ هم هست که بهش می‌گیم **حلقه for range**

اول باید بفهمیم تو هر بار پیمایش دوست داریم به چه چیزی دسترسی داشته باشیم:

- مقدار اون عنصر
- جایگاه اون عنصر تو آرایه

تو حلقه‌ی معمولی که قبلاً دیدیم، با استفاده از شمارنده و ایندکس (i) هم می‌تونستیم مقدار هر خانه رو بخونیم، هم جایگاهش رو بفهمیم.

پس ساختار for range هم باید همین قابلیت رو بهمون بده.

ساختار کلی for range اینجوریه:

```
for index, value := range collection {
    statement
}
```

for

همون کلمه‌ای که تو Go برای همه‌ی حلقه‌ها استفاده می‌شه.

index, value :=

اینجا داریم دو تا متغیر تعریف می‌کنیم:
index شماره‌ی عنصر (از صفر شروع می‌شه)
value خودِ همون عنصر (مثلاً عدد یا رشته)

مثلاً اگه آرایه‌ای داشته باشیم مثل [10, 20, 30] تو اولین بار:

- index = 0
- value = 10

range

این کلمه کلیدیه که می‌گه: "بیا یکی‌یکی عناصر این مجموعه رو بده دستم"

collection

همون آرایه یا لیستی که می‌خوای روش حلقه بزنی. می‌تونه:

- آرایه ([5]int{...})
- اسلایس ([]string{...})
- مپ (map[string]int{...})
- رشته ("hello")
- حتی channel باشه

بدنه‌ی حلقه {...}

کدی که برای هر عنصر اجرا می‌شه و می‌تونی توش از index و value استفاده کنی.

مثال: محاسبه میانگین نمرات با حلقه for range

```
var grades [5]float32
var average float32

for i := range grades {
    fmt.Print("یه نمره وارد کن: ")
    fmt.Scan(&grades[i])
}

var sum float32 = 0
for _, grade := range grades {
    sum += grade
}

average = sum / float32(len(grades))

fmt.Println("میانگین نمرات دانش آموز:", average)
```

حلقه‌ی اول:

```
for i := range grades {
    fmt.Print("یه نمره وارد کن: ")
    fmt.Scan(&grades[i])
}
```

اینجا **فقط یک متغیر (i) گرفتیم** چون فقط به شماره خانه (ایندکس) نیاز داریم.

دلیلش اینه که ما می‌خوایم **مقادیر رو وارد کنیم و در آرایه ذخیره کنیم**.

پس باید به خود آرایه دسترسی داشته باشیم تا مقدار جدید رو جایگزین کنیم.

وقتی فقط یک متغیر بگیری، اون متغیر **ایندکس** هر خانه از آرایه یا اسلایس هست.

داخل حلقه به کمک grades[i] می‌تونیم مستقیماً مقدار اون خانه رو تغییر بدیم.

حلقه‌ی دوم:

```
for _, grade := range grades {
    sum += grade
}
```

اینجا **دو متغیر گرفتیم ولی اولی _ گذاشتیم که یعنی "ایندکس برام مهم نیست."**

grade خودش **مقدار هر عنصر** در آرایه است.

چون فقط می‌خوایم نمره‌ها رو بخونیم و جمع کنیم، نیازی به ایندکس نداریم.

وقتی دو متغیر می‌گیره، اولی ایندکس، دومی مقدار هست.

استفاده از _ یعنی می‌گیم "ایندکس رو نمی‌خوایم، فقط مقدار برامون مهمه."

حلقه تو در تو

همون‌طور که **آرایه تو در تو** داریم، می‌تونیم **حلقه تو در تو** هم داشته باشیم.
ولی خب دلیلش چیه؟
یادت هست که از حلقه استفاده می‌کردیم تا روی آرایه‌ها پیمایش کنیم، درسته؟
حالا اگه **آرایه‌مون خودش تو در تو باشه** (مثلاً آرایه‌ای که توش آرایه هست)، طبیعیه که برای پیمایشش نیاز داریم از حلقه‌ای استفاده کنیم که خودش **یه حلقه دیگه تو دلش داشته باشه.**

اما قبل از اینکه بریم سراغ کاربرد حلقه های تو در تو در آرایه‌های تو در تو، اول بیایم یه کم با خود حلقه تو در تو آشنا بشیم.

حلقه تو در تو یعنی چی؟

یعنی خیلی ساده، **یه حلقه‌ی کامل رو بذاری توی دل یه حلقه‌ی دیگه.**

به عنوان مثال:

- چند تا دانش‌آموز داریم.
  هر دانش‌آموز هم چندتا نمره داره.

حالا اگه بخوای میانگین نمره‌های هر دانش‌آموز رو حساب کنی، چی‌کار می‌کنی؟

- اول باید روی خود دانش‌آموزا پیمایش انجام بدی
- بعد برای هر دانش‌آموز، بری سراغ نمره‌هاش و اون‌ها رو بررسی کنی

یعنی یه حلقه واسه دانش‌آموزها و یه حلقه‌ی دیگه، تو دل اون، واسه نمره‌های هر دانش‌آموز.

یه مثال آشنا: جدول ضرب!

یادته جدول ضرب چطوری بود؟
می‌اومدیم:

- 1 ضربدر 1 تا 10
- 2 ضربدر 1 تا 10
- 3 ضربدر 1 تا 10
- و همین‌طور تا 10 ضربدر 1 تا 10

یعنی برای هر عدد از 1 تا 10، ده‌تا ضرب دیگه حساب می‌کردیم.

```
for i := 1; i <= 10; i++ {
    for j := 1; j <= 10; j++ {
        fmt.Println(i, "×", j, "=", i*j)
    }
}
```

تو این برنامه:

- حلقه بیرونی (i) داره اعداد 1 تا 10 رو پیمایش می‌کنه
- حلقه داخلی (j) هم هر بار از 1 تا 10 می‌ره و ضرب رو انجام می‌ده

به ازای هر چرخش از حلقه بیرونی، حلقه داخلی **کامل** اجرا می‌شه
یعنی در مجموع **100 بار** حلقه داخلی اجرا می‌شه

دسترسی متغیرهای حلقه داخلی و بیرونی به هم

- **حلقه داخلی میتونه به متغیرهای حلقه بیرونی دسترسی داشته باشه.**
  مثل نمونه بالا، داخل حلقه داخلی می‌تونیم از i استفاده کنیم چون در محدوده (scope) بالاتر تعریف شده.
- **حلقه بیرونی به متغیرهای حلقه داخلی دسترسی نداره.**
  چون متغیر j فقط داخل حلقه داخلی تعریف شده و وقتی حلقه داخلی تموم میشه، j دیگه وجود نداره، پس حلقه بیرونی اون رو نمی‌بینه.

نکته مربوط به ترتیب اجرای حلقه‌ها

- حلقه بیرونی هر بار که یک بار تکرار میشه، حلقه داخلی به صورت کامل اجرا میشه.
- یعنی مثلاً وقتی i=1 هست، j از 1 تا 10 تکرار میشه. بعد i=2 میشه و دوباره j از 1 تا 10 اجرا میشه و الی آخر.

اهمیت ترتیب متغیرهای شمارنده

- اگر دوتا حلقه شمارنده با یک اسم تعریف بشن، در حلقه داخلی باعث خطا یا رفتار غیرمنتظره میشه.
- بنابراین اسم‌ها باید متفاوت باشن و هر کدوم داخل محدوده خودشون تعریف بشن.

حالا که با کاربرد حلقه تو در تو آشنا شدیم میتونیم از این حلقه برای پیمایش رو آرایه دو بعدی نمرات چند دانش آموز استفاده کنیم. در این بخش ابتدا با استفاده از حلقه تو در تو for ; ; این برنامه رو مینویسم و سپس حلقه هارو با حلقه for range جایگزین میکنیم

```
const NumberOfStudnets = 4
const NumberOfGrades = 3

var allGrades [NumberOfStudnets][ NumberOfGrades]float32
var averages [NumberOfStudnets]float32

for i := 0; i < NumberOfStudnets; i++ {
    fmt.Println("نمرات ریاضی، فارسی و تربیت بدنی دانش آموز چیه؟ (به ترتیب وارد کن)")
    for j := 0; j < NumberOfGrades; j++ {
        fmt.Scan(&allGrades[i][j])
    }
}

for i := 0; i < NumberOfStudnets; i++ {
    var sum float32 = 0
    for j := 0; j < NumberOfGrades; j++ {
        sum = sum + allGrades[i][j]
    }
    averages[i] = sum / NumberOfGrades
}

fmt.Println("میانگین نمرات دانش آموزان به ترتیب:")

for i := 0; i < NumberOfStudents; i++ {
    fmt.Print(averages[i], " ")
}
```

تعریف تعداد دانش‌آموز و تعداد درس

```
const NumberOfStudents = 4
const NumberOfGrades = 3
```

به‌جای استفاده‌ی مستقیم از عدد در حلقه‌ها و آرایه‌ها، این اعداد به‌صورت ثابت (const) تعریف شده. این کار باعث می‌شه در آینده، تغییر در تعداد دانش‌آموزها یا دروس خیلی راحت انجام بشه.

تعریف آرایه‌ها

```
var allGrades [NumberOfStudents][NumberOfGrades]float32
var averages [NumberOfStudents]float32
```

allGrades یک آرایه‌ی دوبعدی برای نگهداری نمرات. هر ردیف مربوط به نمرات یک دانش‌آموزه.

averages برای نگهداری معدل هر دانش‌آموز.

گرفتن ورودی نمرات با دو حلقه تو در تو

```
for i := 0; i < NumberOfStudents; i++ {
    fmt.Println("نمرات ریاضی، فارسی و تربیت بدنی دانش‌آموز چیه؟ (به ترتیب وارد کن)")
    for j := 0; j < NumberOfGrades; j++ {
        fmt.Scan(&allGrades[i][j])
    }
}
```

حلقه بیرونی (i) برای پیمایش دانش‌آموزهاست.

حلقه داخلی (j) برای گرفتن نمره‌های هر درس.

محاسبه‌ی میانگین برای هر دانش‌آموز

```
for i := 0; i < NumberOfStudents; i++ {
    var sum float32 = 0
    for j := 0; j < NumberOfGrades; j++ {
        sum += allGrades[i][j]
    }
    averages[i] = sum / NumberOfGrades
}
```

برای هر دانش‌آموز، ابتدا sum صفر می‌شه.

سپس با حلقه‌ی داخلی، نمره‌های اون دانش‌آموز جمع می‌شن.

در نهایت معدل حساب شده و در averages[i] ذخیره می‌شه.

چاپ میانگین‌ها

```
for i := 0; i < NumberOfStudents; i++ {
    fmt.Print(averages[i])
}
```

معدل‌ها به همون ترتیبی که وارد شدن چاپ می‌شن.

ترتیب بر اساس ایندکس‌های آرایه‌ی averages هست.

حلقه‌های تو در تو و دستورات کنترلی continue و break

تا اینجا با دستورهای کنترلی break و continue آشنا شدیم. حالا وقتشه ببینیم وقتی حلقه‌هامون **تو در تو** (nested) هستن، این دو تا دستور چطوری رفتار می‌کنن.

یه نکته‌ی خیلی مهم

توی حلقه‌های تو در تو، **break و continue فقط روی همون حلقه‌ای اثر دارن که داخلش نوشته شدن.**

یعنی اگه continue رو توی حلقه داخلی بنویسی، فقط همون حلقه داخلی از ادامه اجرا می‌پره و می‌ره سر ایندکس بعد. حلقه بیرونی هیچ کاری بهش نداره و اجراش ادامه پیدا می‌کنه.

فرض کن داریم مثل قبل برنامه‌ای می‌نویسیم برای نمایش جدول ضرب.

حلقه‌ی بیرونی مقدار a رو می‌سازه (مولفه‌ی اول ضرب)، حلقه‌ی داخلی مقدار b رو (مولفه دوم).

حالا یه قانون عجیب می‌خوایم:

هر وقت a و b مساوی بودن، ضرب رو انجام نده!

یعنی مثلاً 2 * 2 یا 3 * 3 یا 7 * 7 نباید نمایش داده بشن.

یا به عبارت ساده تر. جدول ضرب زیرو در نظر بگیر. میخوایم ضرب هایی که با رنگ قرمز نشون داده شده محاسبه نشه

1 * 1 = 1	2 * 1 = 2	3 * 1 = 3	4 * 1 = 4	5 * 1 = 5	6 * 1 = 6	7 * 1 = 7	8 * 1 = 8	9 * 1 = 9	10 * 1 = 10
1 * 2 = 2	2 * 2 = 4	3 * 2 = 6	4 * 2 = 8	5 * 2 = 10	6 * 2 = 12	7 * 2 = 14	8 * 2 = 16	9 * 2 = 18	10 * 2 = 20
1 * 3 = 3	2 * 3 = 6	3 * 3 = 9	4 * 3 = 12	5 * 3 = 15	6 * 3 = 18	7 * 3 = 21	8 * 3 = 24	9 * 3 = 27	10 * 3 = 30
1 * 4 = 4	2 * 4 = 8	3 * 4 = 12	4 * 4 = 16	5 * 4 = 20	6 * 4 = 24	7 * 4 = 28	8 * 4 = 32	9 * 4 = 36	10 * 4 = 40
1 * 5 = 5	2 * 5 = 10	3 * 5 = 15	4 * 5 = 20	5 * 5 = 25	6 * 5 = 30	7 * 5 = 35	8 * 5 = 40	9 * 5 = 45	10 * 5 = 50
1 * 6 = 6	2 * 6 = 12	3 * 6 = 18	4 * 6 = 24	5 * 6 = 30	6 * 6 = 36	7 * 6 = 42	8 * 6 = 48	9 * 6 = 54	10 * 6 = 60
1 * 7 = 7	2 * 7 = 14	3 * 7 = 21	4 * 7 = 28	5 * 7 = 35	6 * 7 = 42	7 * 7 = 49	8 * 7 = 56	9 * 7 = 63	10 * 7 = 70
1 * 8 = 8	2 * 8 = 16	3 * 8 = 24	4 * 8 = 32	5 * 8 = 40	6 * 8 = 48	7 * 8 = 56	8 * 8 = 64	9 * 8 = 72	10 * 8 = 80
1 * 9 = 9	2 * 9 = 18	3 * 9 = 27	4 * 9 = 36	5 * 9 = 45	6 * 9 = 54	7 * 9 = 63	8 * 9 = 72	9 * 9 = 81	10 * 9 = 90
1 * 10 = 10	2 * 10 = 20	3 * 10 = 30	4 * 10 = 40	5 * 10 = 50	6 * 10 = 60	7 * 10 = 70	8 * 10 = 80	9 * 10 = 90	10 * 10 = 100

چطور این کارو بکنیم؟

خیلی ساده! فقط کافیه بگیم:

```
if a == b {
    continue
}
```

و این شرط رو توی **حلقه‌ی داخلی** بنویسیم. چون اونجا داره b تغییر می‌کنه و در هر گام یک ضرب انجام می‌شه.

پس چرا حلقه داخلی؟

چون ما داریم ضرب‌ها رو همون‌جا داخل حلقه داخلی چاپ می‌کنیم، پس اونجاست که باید بگیم:

"اگه a == b بود، بی‌خیال این ضرب شو، برو سراغ b بعدی".

برنامه زیر دقیقا همین کاری که خواسته بودیمو انجام میده

```
for i := 1; i <= 10; i++ {
    for j := 1; j <= 10; j++ {
        if a == b {
           continue
        }
        fmt.Println(i, "×", j, "=", i*j)
    }
}
```

حالا بریم سراغ یه قانون جالب‌تر:

اگه مولفه اول (a) عدد **زوج** بود، کل اون ردیف از جدول ضرب رو محاسبه نکن!

مثلاً 1 * 2 ، 2 * 2 ، 3 * 2 و ... اصلاً نباید نمایش داده بشن.

همین‌طور 1 * 4 ، 2 * 4 ، 3 * 4 و ...

یا به عبارت ساده تر. جدول ضرب زیرو در نظر بگیر. میخوایم ضرب هایی که با رنگ قرمز نشون داده شده محاسبه نشه

1 * 1 = 1	2 * 1 = 2	3 * 1 = 3	4 * 1 = 4	5 * 1 = 5	6 * 1 = 6	7 * 1 = 7	8 * 1 = 8	9 * 1 = 9	10 * 1 = 10
1 * 2 = 2	2 * 2 = 4	3 * 2 = 6	4 * 2 = 8	5 * 2 = 10	6 * 2 = 12	7 * 2 = 14	8 * 2 = 16	9 * 2 = 18	10 * 2 = 20
1 * 3 = 3	2 * 3 = 6	3 * 3 = 9	4 * 3 = 12	5 * 3 = 15	6 * 3 = 18	7 * 3 = 21	8 * 3 = 24	9 * 3 = 27	10 * 3 = 30
1 * 4 = 4	2 * 4 = 8	3 * 4 = 12	4 * 4 = 16	5 * 4 = 20	6 * 4 = 24	7 * 4 = 28	8 * 4 = 32	9 * 4 = 36	10 * 4 = 40
1 * 5 = 5	2 * 5 = 10	3 * 5 = 15	4 * 5 = 20	5 * 5 = 25	6 * 5 = 30	7 * 5 = 35	8 * 5 = 40	9 * 5 = 45	10 * 5 = 50
1 * 6 = 6	2 * 6 = 12	3 * 6 = 18	4 * 6 = 24	5 * 6 = 30	6 * 6 = 36	7 * 6 = 42	8 * 6 = 48	9 * 6 = 54	10 * 6 = 60
1 * 7 = 7	2 * 7 = 14	3 * 7 = 21	4 * 7 = 28	5 * 7 = 35	6 * 7 = 42	7 * 7 = 49	8 * 7 = 56	9 * 7 = 63	10 * 7 = 70
1 * 8 = 8	2 * 8 = 16	3 * 8 = 24	4 * 8 = 32	5 * 8 = 40	6 * 8 = 48	7 * 8 = 56	8 * 8 = 64	9 * 8 = 72	10 * 8 = 80
1 * 9 = 9	2 * 9 = 18	3 * 9 = 27	4 * 9 = 36	5 * 9 = 45	6 * 9 = 54	7 * 9 = 63	8 * 9 = 72	9 * 9 = 81	10 * 9 = 90
1 * 10 = 10	2 * 10 = 20	3 * 10 = 30	4 * 10 = 40	5 * 10 = 50	6 * 10 = 60	7 * 10 = 70	8 * 10 = 80	9 * 10 = 90	10 * 10 = 100

در واقع، جدول ضربمون فقط برای مقادیر **فردِ** a محاسبه می‌شه.

اینجا باید همون اول حلقه بیرونی، a رو چک کنیم.

اگه a زوج بود، همون‌جا با continue بریم سراغ عدد بعدی.

```
for i := 1; i <= 10; i++ {
    if a%2 == 0 {
        continue
    }
    for j := 1; j <= 10; j++ {
        fmt.Println(i, "×", j, "=", i*j)
    }
}
```

حلقه بیرونی داره a رو از 1 تا 10 تکرار می‌کنه.

همون اول حلقه، شرط گذاشتیم:

اگه a زوج بود (a%2 == 0) دیگه اصلاً وارد حلقه داخلی نشو، برو سراغ a بعدی.

پس فقط وقتی a فرد باشه، ضرب‌ها محاسبه می‌شن.

جمع‌بندی مهم: کنترل فقط در محدوده خودش!

تا اینجا یاد گرفتیم که چطوری با continue و break، جریان اجرای حلقه‌ها رو دستکاری کنیم.
اما یه نکته خیلی خیلی مهم هست که باید همیشه یادت بمونه:

دستورهای break و continue فقط روی همون حلقه‌ای اثر دارن که داخلش نوشته شده‌ان.

یعنی چی؟

یعنی:

- اگه توی حلقه داخلی continue بزنی، فقط همون حلقه داخلی یه بار از ادامه اجرا می‌پره.
- اگه توی حلقه بیرونی continue بزنی، کلاً از اون مرحله از حلقه بیرونی می‌پری و میری به مرحله بعد.
- همین ماجرا برای break هم هست: هر جا نوشته بشه، فقط همون حلقه رو متوقف می‌کنه، نه کل برنامه یا حلقه‌های دیگه رو.

چرا این مهمه؟

وقتی با حلقه‌های تو در تو کار می‌کنی، خیلی راحت ممکنه فکر کنی که break یا continue ممکنه هر دو حلقه رو متوقف کنن. ولی نه! فقط همون حلقه‌ای که داخلش هستن رو کنترل می‌کنن.

پس اگه خواستی اجرای حلقه داخلی رو قطع کنی، دستور رو باید اونجا بنویسی.
و اگه خواستی حلقه بیرونی رو کنترل کنی، باید اون دستور بره داخل حلقه بیرونی.

خروج کامل از حلقه تو در تو

تا اینجا فهمیدیم که اگه بخوایم از داخل یه حلقه تو در تو، به طور کامل از **هر دو حلقه** خارج بشیم، نمی‌تونیم فقط با نوشتن break در حلقه داخلی این کار رو بکنیم؛ چون break فقط همون حلقه‌ای که توش نوشته شده رو می‌شکنه.

پس راه حل چیه؟

دو راه کار منطقی و استاندارد برای این موضوع وجود داره

روش اول: استفاده از تکنیک برچسب‌گذاری (Labeling)

تو این روش، به حلقه یا بلوک مورد نظر یه اسم یا برچسب می‌دیم و بعد با دستورهای break یا continue مشخص می‌کنیم که کدوم حلقه رو می‌خوایم بشکنیم یا ادامه بدیم.

مثلاً فرض کن تو برنامه جدول ضرب می‌خوایم وقتی حاصل ضرب دو عدد مساوی یا بیشتر از 36 شد، از کل ساختار حلقه تو در تو خارج بشیم و بریم سراغ ادامه برنامه.

اینجاست که می‌تونیم به حلقه بیرونی یه اسم بدیم و از داخل حلقه داخلی بگیم:

"برو خارج شو از حلقه‌ای که اسمشو بهت دادم"

به برنامه زیر دقت کن:

```
first_loop:
for i := 1; i <= 10; i++ {
    for j := 1; j <= 10; j++ {
        if a * b >= 36 {
           break first_loop
        }
        fmt.Println(i, "×", j, "=", i*j)
    }
}
```

اینجا:

- first_loop: اسم حلقه بیرونی هست
- وقتی i * j بزرگ‌تر یا مساوی 36 شد، دستور break first_loop باعث میشه کلاً از حلقه بیرونی و داخلی خارج بشیم و به ادامه برنامه بریم.

روش دوم: استفاده از تکنیک Flag (علامت)

یه متغیر منطقی مثل shouldBreak := false تعریف می‌کنیم.

- وقتی شرط خاصی در حلقه داخلی اتفاق افتاد، این فلگ رو true می‌کنیم و با break از حلقه داخلی می‌زنیم بیرون.
- تو حلقه بیرونی قبل از شروع هر دور، چک می‌کنیم که آیا باید حلقه رو بشکنیم یا نه.

در ادامه همون برنامه ای که تو حالت اول (استفاده از تکنیک برچسب گذاری) نوشتیم به روش جدید بازنویسی کردیم

```
shouldBreak := false
for i := 1; i <= 10 && !shouldBreak; i++ {
    for j := 1; j <= 10; j++ {
        if a * b >= 36 {
           shouldBreak = true
           break
        }
        fmt.Println(i, "×", j, "=", i*j)
    }
}
```

- شرط حلقه بیرونی شده i <= 10 && !shouldBreak
- یعنی حلقه بیرونی فقط وقتی ادامه پیدا می‌کنه که هم i کوچکتر یا مساوی 10 باشه و هم فلگ shouldBreak برابر false باشه.
- وقتی تو حلقه داخلی شرط ضرب بزرگ‌تر یا مساوی 36 برقرار شد، فلگ رو true می‌کنیم و با break از حلقه داخلی می‌زنیم بیرون.
- این باعث میشه تو مرحله بعد، شرط حلقه بیرونی false بشه و حلقه بیرونی هم تموم بشه.

پیمایش رشته

رشته‌ها یکی از رایج‌ترین چیزایی هستن که تو برنامه‌نویسی باهاشون سر و کار داریم.
خیلی وقت‌ها لازمه یه رشته رو **کاراکتر به کاراکتر بررسی کنیم** یا روی حروفش حلقه بزنیم.

تو زبان Go چند روش برای این کار وجود داره.

روش اول: استفاده از for و ایندکس

اگه یادت باشه، حلقه for رو بلدیم. می‌تونیم از اون استفاده کنیم و به کمک شماره‌ی خونه‌های رشته (اندیس‌ها)، به تک‌تک حروف رشته دسترسی داشته باشیم.

```
text := "hello"
for i := 0; i < len(text); i++ {
    fmt.Println(text[i])
}
```

اما اینجا یه نکته هست:

- این روش در واقع داره **مقدار عددی هر حرف** رو نشون می‌ده (کد ASCII یا یونیکد) اگه بخوای خود حرف‌ها رو ببینی، می‌تونی از %c توی Printf استفاده کنی:

```
text := "hello"
for i := 0; i < len(text); i++ {
    fmt.Printf(“%c\n”, text[i])
}
```

اما یه نکته مهم درباره‌ی این روش:

روش for با ایندکس که بالا دیدیم، **همیشه هم خوب جواب نمی‌ده!**

این روش فقط وقتی درست کار می‌کنه که **رشته فقط شامل حروف انگلیسی ساده و اعداد** باشه (و چندتا کاراکتر خاص مثل نقطه و فاصله).

اما اگه توی رشته‌مون از **حروف فارسی، ایموجی، یا زبان‌های دیگه** استفاده کنیم، خروجی‌ای که می‌گیریم **درست و قابل خوندن نیست**.

چرا این اتفاق می‌افته؟

تابع len() طول رشته رو برحسب **تعداد بایت‌ها** برمی‌گردونه، و ما هم داریم با text[i] بایت به بایت رشته رو پیمایش می‌کنیم.

ولی همه‌ی کاراکترها یک بایتی نیستن!

- حروف انگلیسی معمولاً یه بایتی هستن
- ولی کاراکترهای فارسی، چینی، ایموجی‌ها و ... ممکنه **2، 3 یا حتی 4 بایت** باشن

برای اینکه این موضوع بهتر جا بیفته، بیایم یه رشته فارسی رو با همین روش پیمایش کنیم:

```
text := "ساعت چنده؟"
for i := 0; i < len(text); i++ {
    fmt.Printf(“%c\n”, text[i])
}
```

خروجی

```
Ø
³
Ø
§
Ø
¹
Ø
ª


Ú


Ù


Ø
```

همون‌طور که می‌بینی، این خروجی هیچ شباهتی به متن اصلی نداره!

پس چی کار کنیم؟

به جای این روش، **پیشنهاد ما استفاده از for range هست**
که تو روش بعدی باهاش آشنا می‌شی. این روش به جای بایت، **کاراکتر به کاراکتر** رشته رو پیمایش می‌کنه و برای زبان‌های مختلف کاملاً جواب می‌ده.

بنابراین بهتره از روش دو یعنی استفاده از for range برای پیمایش دو رشته استفاده کنیم

روش دوم: استفاده از for range

این روش هم خیلی ساده‌ست، هم قدرتمندتره، مخصوصاً وقتی با متن‌های فارسی یا ایموجی‌ها کار می‌کنی

مثلاً همون مثالی که تو روش اول زدیم (پیمایش رشته‌ی فارسی) رو این‌بار با for range می‌نویسیم:

```
text := "ساعت چنده؟"
for i, ch := range text {
    fmt.Println("Index:", i, "Character:", string(ch))
}
```

خروجی

```
Index: 0 Character: س
Index: 2 Character: ا
Index: 4 Character: ع
Index: 6 Character: ت
Index: 8 Character:
Index: 9 Character: چ
Index: 11 Character: ن
Index: 13 Character: د
Index: 15 Character: ه
Index: 17 Character: ؟
```

- متغیر i نشون‌دهنده‌ی جای کاراکتر تو رشته‌ست
- متغیر ch خود کاراکتره (از نوع rune)

برخلاف روش قبلی که بایت‌به‌بایت پیش می‌رفت و برای فارسی یا ایموجی خراب می‌شد، اینجا با for range، کاراکتر به کاراکتر پیش می‌ریم، پس هیچ چیز عجیب‌غریبی تو خروجی نمی‌بینیم.

یه نکته مهم درباره‌ی رشته‌ها:

درسته که رشته‌ها از بایت تشکیل شدن، اما **رفتارشون مثل یه آرایه‌ی معمولی نیست** .یعنی نمی‌تونی مستقیم بری و یه کاراکتر رو عوض کنی.

به مثال زیر توجه کن

```
text := “run”
text[1] = ‘a’
```

به ‘a’ دقت کن. میدونیم که رشته یک آرایه از بایت هاست با قرار دادن حرف a در '' مقدار عددی a که معادل عدد 97 هست جایگزین می‌شه

این برنامه خطای کامپایل میده و قابل اجرا نیست. Go اجازه نمی‌ده یه کاراکتر از یه رشته‌ی موجود رو تغییر بدی، چون رشته‌ها **immutable** هستن

برای تغییر یک کاراکتر از یک رشته مجبور هستیم رشته جدیدی ایجاد کنیم و در اون متغیر قرار بدیم

```
text := “run”
text = “ran”
```

جمع‌بندی:

- برای پیمایش رشته می‌تونی از for یا for range استفاده کنی.
- for range روش دقیق‌تریه، به‌خصوص برای کاراکترهای فارسی یا چندزبانه.
- رشته‌ها فقط قابل خوندن، اگه بخوای تغییرشون بدی باید یه رشته جدید بسازی.

حلقه‌هایی که 0 بار اجرا می‌شن – اشتباه یا طراحی عمدی؟

گاهی تو برنامه‌مون ممکنه حلقه‌ای بنویسیم که **اصلاً اجرا نشه**، یعنی حتی یک بار هم وارد بدنه‌ی حلقه نشیم. این اتفاق معمولاً تو یکی از دو حالت زیر می‌افته:

1- شرط حلقه رو اشتباه نوشتی

یعنی هدفت این بوده که حلقه اجرا بشه، ولی چون شرط رو اشتباه زدی، اصلاً واردش نمی‌شی.

به مثال زیر توجه کن:

```
for i := 20; i < 20; i-- {
    if i % 2 != 0 {
        fmt.Println(i)
    }
}
```

اینجا می‌خواستیم به صورت عقبگرد اعداد فرد بین 20 تا 0 رو چاپ کنیم. ولی چون شرط حلقه گفتیم $i < 20$ و مقدار اولیه‌مون هم 20 هست، این شرط از همون اول برقرار نیست، پس حلقه **هیچ‌وقت اجرا نمی‌شه**.

در نتیجه برنامه‌مون خطای منطقی داره و باید اصلاحش کنیم.

در اصل باید این‌جوری می‌نوشتیم:

```
for i := 20; i >= 0; i-- {
    if i % 2 != 0 {
        fmt.Println(i)
    }
}
```

2- حلقه درست نوشته شده، ولی اجرای اون ضرورتی نداره

گاهی هم حلقه‌ت کاملاً درسته و دقیق نوشتی، ولی شرایط برنامه طوریه که **بدنه‌ی حلقه نباید اجرا بشه**—و این کاملاً منطقیه!

به مثال زیر توجه کن

```
var grades [0]float32
for i := 0; i < len(grades); i++ {
    fmt.Println(grades[i])
}
```

تو این برنامه هدف اینه که نمرات دانش‌آموز رو چاپ کنیم. حلقه‌مون هم درست نوشته شده، ولی چون آرایه‌ی grades خالیه و طولش صفره، شرط i < len(grades) از همون اول برقرار نیست. پس برنامه درست کار می‌کنه و فقط چیزی برای چاپ وجود نداره.

جمع‌بندی:

گاهی وقت‌ها حلقه‌ای که اجرا نمی‌شه، **نشونه‌ی یه باگه** — مثلاً وقتی شرط رو اشتباه نوشتی. اما بعضی وقت‌ها هم این یه رفتار طبیعیه و نشونه‌ی اینه که برنامه‌ت داره **منطقی رفتار می‌کنه.**

یادت باشه:

"وجود حلقه به معنی اجرای اون نیست"!

تا وقتی شرط برقرار نباشه، بدنه‌ی حلقه اجرا نمی‌شه — چه عمدی، چه سهوی.

حلقه بدون بدنه

ما می‌دونیم که وقتی شرط حلقه برقرار باشه، بدنه‌ی اون حلقه بارها و بارها اجرا می‌شه.
اما حالا یه سؤال مهم :**اگه بدنه‌ی اون حلقه رو خالی بذاریم، چی میشه؟**

به مثال زیر دقت کن:

```
for time.Now().Second() != 30 {
}
fmt.Println("یادت باشه پروژه رو سیو کنی!")
```

در این مثال، تابع time.Now().Second() ثانیه‌ی فعلی سیستم رو به ما می‌ده.
حلقه، تا وقتی که ثانیه برابر با عدد 30 نشده، مدام اجرا می‌شه. ولی چون داخل حلقه هیچ دستوری نیست، عملاً برنامه هیچ کاری نمی‌کنه و فقط **منتظر می‌مونه**.

تا کی؟
تا لحظه‌ای که ثانیه به عدد 30 برسه.
به محض اینکه این شرط برقرار بشه، حلقه تموم می‌شه و پیام "یادت باشه پروژه رو سیو کنی!" چاپ می‌شه.

پس به بیان ساده‌تر، **برنامه تا وقتی که به شرایط خاص اتفاق نیفتاده، متوقف می‌مونه** — بدون اینکه کار خاصی انجام بده.

کاربرد اصلی این نوع حلقه‌ها چیه؟

منتظر نگه داشتن برنامه برای یک رویداد خاص.
کافیه اون رویدادی که منتظرشی، هنوز اتفاق نیفتاده باشه و ما شرط اون رو به صورت نقیض بنویسیم.
بعد هم بدنه حلقه رو خالی بذاریم.

درست مثل مثال بالا، که تا رسیدن زمان خاص منتظر موندیم.

یه حلقه‌ی بدون بدنه، وقتی به دردت می‌خوره که:

- فقط بخوای **یه شرط خاص رو دائماً چک کنی**
- تا وقتی اون شرط برقرار نشده، برنامه باید **منتظر بمونه**
- هیچ کار خاصی تو بدنه‌ی حلقه نداری که بخوای انجام بدی

این دقیقاً یکی از تمیزترین راه‌ها برای کنترل انتظار در برنامه‌ست — مخصوصاً توی برنامه‌های ساده یا وقتی هنوز با مبحث هم‌زمانی (Concurrency) آشنا نشدیم.

در برنامه‌های حرفه‌ای‌تر، معمولاً این نوع صبر کردن با ابزارهای بهتر مثل time.Sleep، select، یا channelها انجام می‌شه، که در آینده باهاشون آشنا می‌شیم.

تمرین 1: محاسبه فاکتوریل عدد

یک عدد صحیح از کاربر دریافت کن و فاکتوریل آن را محاسبه کن.

فاکتوریل یعنی ضرب همه اعداد صحیح از 1 تا n

به عنوان مثال فاکتوریل 5 برابر است 120 که به شکل زیر محاسبه شده

1 * 2 * 3 * 4 * 5

تمرین 2: دنباله فیبوناچی

دنباله فیبوناچی دنباله ای از اعداد است که هر عنصر طبق فرمول زیر محاسبه میشه

عنصر شماره n = (عنصر شماره n – 1) + (عنصر شماره n – 2)

عنصر اول و دوم دنباله فیبوناچی 0 1 هست

بنابراین عنصر سوم دنباله فیبوناچی مجموع عدد 0 و 1 است که می شود 1

به همین ترتیب عنصر چهارم دنباله فیبوناچی مجموع عنصر دوم و سوم یعنی مجموع اعداد 1 و 1 است که می شود 2

و به همین ترتیب ...

برنامه ای بنویس که تعداد جملات دنباله فیبوناچی رو از کاربر بگیر و دنباله رو چاپ کن

به عنوان مثال اگه ورودی عدد 5 بود یعنی 5 جمله اول دنباله فیبوناچی رو باید محاسبه و چاپ کنی

تمرین 3: الگوی ستاره ای ساده (Star Pattern)

الگوی ستاره ای یعنی چاپ یه تعداد کاراکتر * به شکلی که به صورت منظم شکل خاصی رو ایجاد کنن.

به عنوان مثال در زیر یک الگوی ستاره ای نشون داده شده

```
*
**
***
****
*****
******
```

در این الگو

در خط اول 1 ستاره

در خط دوم 2 ستاره

در خط سوم 3 ستاره

و به همین ترتیب در نهایت در خط ششم 6 ستاره چاپ شده است

برنامه ای بنویس که تعداد خطوط مورد نظرو از ورودی کاربر بگیره و این الگو رو چاپ کنه

تمرین 4: الگوی ستاره ای ساده ای پیچیده (Star Pattern)

الگوی ستاره ای یعنی چاپ یه تعداد کاراکتر * به شکلی که به صورت منظم شکل خاصی رو ایجاد کنن.

به عنوان مثال در زیر یک الگوی ستاره ای نشون داده شده

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

در این الگو

در خط اول 1 ستاره

در خط دوم 3 ستاره

در خط سوم 5 ستاره

در خط چهارم 7 ستاره

در خط پنجم مجدد 5 ستاره

در خط ششم مجدد 3 ستاره

در خط هفتم (آخر) مجدد 1 ستاره

چاپ شده است

برنامه ای بنویس که تعداد خطوط مورد نظرو از ورودی کاربر بگیره و این الگو رو چاپ کنه

نکته: تعداد خطوط مورد نظر حداقل میتونه عدد 5 باشه و حتما باید یک عدد فرد باشه

تمرین 5: بررسی و گزارش BMI افراد

یک آرایه از قبل داریم که شامل **BMI بیست نفر** هست.
برنامه‌ای بنویس که روی این لیست **پیمایش** کنه و برای هر مقدار BMI وضعیت وزنی فرد رو به صورت متنی چاپ کنه.

دسته‌بندی وضعیت بر اساس BMI

- اگر BMI < 18.5 وضعیت: "کم‌وزن"
- اگر 18.5 <= BMI < 25 وضعیت: "نرمال"
- اگر 25 <= BMI < 30 وضعیت: "اضافه‌وزن"
- اگر BMI >= 30 وضعیت: "چاق"

لیست BMI داده‌شده (از قبل در برنامه تعریف شده):

```
bmis := [20]float64{17.2, 22.5, 31.8, 24.9, 29.0, 18.3, 27.4, 20.5, 30.1, 19.6,
                    16.8, 25.5, 28.7, 23.3, 21.0, 33.0, 17.9, 26.1, 22.0, 31.0}
```

تمرین 6: اصلاح حروف در رشته

مازیار یه نامه‌ی رسمی به زبان انگلیسی برای مدیر شرکتش نوشته، ولی یه **عادت بد** داره!
اون همیشه به جای حرف o از عدد 0 استفاده می‌کنه. مثلاً به جای hello می‌نویسه hell0 یا به جای Monday می‌نویسه M0nday

خب مشخصه که تو یه نامه رسمی این کار درست نیست، پس باید کمکش کنیم.

وظیفه تو اینه که یه برنامه بنویسی که متن نامه رو بررسی کنه و هر جا عدد 0 دید، اونو با حرف o جایگزین کنه.

متن نامه:

```
Hell0 Mr. Smith,


I h0pe y0u are d0ing well. I am writing t0 request a meeting 0n M0nday.


Best regards,
Mazyar
```

تمرین 7: بازی Tic-Tac-Toe (XO)

این برنامه در واقع همون بازی معروف **دوز** هست — یه بازی دونفره که خیلی ساده و در عین حال سرگرم‌کننده‌ست. تو این بازی، دو بازیکن به نوبت علامت خودشون رو داخل خونه‌های جدول قرار می‌دن.

هر بازیکن یه علامت مخصوص داره:

- بازیکن اول X
- بازیکن دوم O

در هر نوبت، یکی از بازیکن‌ها باید یه خونه خالی رو انتخاب کنه و علامتش رو اونجا بذاره.
هدف اصلی اینه که یکی از بازیکن‌ها بتونه **3 تا علامت پشت سر هم** بچینه — فرقی نمی‌کنه که:

- توی یه **ردیف**
- یا یه **ستون**
- یا روی **قطر جدول** باشه

اگه هیچ‌کدوم از بازیکن‌ها موفق نشن سه‌تایی پشت سر هم قرار بدن، و همه‌ی خونه‌های جدول پر بشه، بازی **مساوی** میشه.

راهنمای نوشتن کد بازی

ساخت صفحه بازی

از یه آرایه دوبعدی با نوع [3][3]string استفاده کن

نمایش صفحه بازی

نیازی نیست که کل صفحه بازی رو رسم کنی یا شکل خاصی نشون بدی.
کافیه در هر نوبت، از بازیکن بپرسی که قصد داره علامتش رو توی کدوم «سطر» و «ستون» قرار بده.
قبل از قرار دادن علامت، چک کن که اون خونه خالیه یا نه:

- اگه خونه خالی بود، علامت بازیکن رو اونجا بذار و نوبت رو به بازیکن بعدی بده
- اگه خونه قبلاً پر شده بود، باید یه پیام خطا نشون بدی و از همون بازیکن بخوای که یه جای دیگه انتخاب کنه

گرفتن حرکت بازیکن

- از بازیکن بپرس که علامتش رو در کدوم سطر و ستون می‌خواد قرار بده
- مطمئن شو که اون خونه هنوز خالیه
- اگه خالی نیست، دوباره ازش بخواه یه خونه‌ی دیگه انتخاب کنه

بعد از هر حرکت

1. بررسی کن آیا بازیکن فعلی بازی رو برده یا نه
2. اگه کسی برنده نشده، بررسی کن آیا بازی مساوی شده یا نه (یعنی همه خونه‌ها پر شده باشن ولی کسی برنده نشده باشه)
3. اگه نه، نوبت بازیکن بعدی می‌رسه

دفعات اجرای بازی

بازی باید توی یه حلقه بی‌نهایت (یا نامعلوم) اجرا بشه
تا زمانی که یکی از این دو حالت پیش بیاد:

- یکی از بازیکن‌ها برنده بشه
- یا بازی مساوی بشه

به محض اینکه یکی از این دو حالت اتفاق افتاد، از حلقه خارج میشی و بازی تموم میشه.