DataSource

در این تمرین قصد داریم تا یک برنامه مدیریت فروشگاه رو پیاده‌سازی کنیم.

فروشنده‌ای قصد داره برای مدیریت فروشگاه خود برنامه‌ای رو سفارش بده. با توجه به اینکه هزینه ساخت برنامه زیاده و منابع فروشنده کم، تلاش بر اینه که این برنامه به ساده‌ترین شکل ممکن پیاده بشه. به همین خاطر تنها موجودیت‌ها تو این برنامه **محصول** و **فروش** هست. ویژگی‌های یه محصول به صورت زیر هست:

- شناسه محصول : int
- نام محصول : String
- قیمت محصول : int
- تعداد محصول : int

ویژگی‌های یه فروش به صورت زیر است:

- شناسه محصول فروخته شده : int
- تعداد فروخته شده : int
- تاریخ فروش محصول : String

برای اینکه بخوایم اطلاعات مغازه رو ذخیره کنیم باید اول از ورودی استاندارد اطلاعات یک **محصول** و یا **فروش** رو بگیریم و در فایل ذخیره کنیم. بعد از این فروشنده باید بتونه لیست **محصولات** و **فروش** رو ببینه در نتیجه نیاز داریم تا اطلاعات رو از فایل بخونیم و در خروجی استاندارد چاپ کنیم. پس به طور کلی به عملیات‌های زیر نیاز داریم:

- نوشتن اطلاعات **فروش** در فایل
- نوشتن اطلاعات **محصول** در فایل
- خواندن اطلاعات **فروش** از فایل
- خواندن اطلاعات **محصول** از فایل
- نوشتن اطلاعات **فروش** در کنسول
- نوشتن اطلاعات **محصول** در کنسول
- خواندن اطلاعات **فروش** از کنسول
- خواندن اطلاعات **محصول** از کنسول

انجام هر یک از این عملیات‌ها به سادگی قابل انجام است و قبلا نمونه آن را انجام داده‌ایم اما فروشنده از ما خواسته است تا کد را به صورت **قابل توسعه** انجام دهیم تا هر وقت وضعش خوب شد برای بقیه قسمت‌های کسب‌وکارش هم برنامه رو توسعه بده. یکی از معیارهای قابل توسعه بودن برنامه **بخش‌پذیر** بودن آن است. منظور از بخش‌پذیر بودن این است که برنامه به بخش‌های کوچکی تقسیم شود که هر کدام کار **مشخصی** رو انجام میدن. به عنوان مثال ما می‌تونیم برنامه بزنیم که تمام عملیات‌های بالا رو در تابع main انجام بده اما این کار باعث میشه که کد به شدت **ناخوانا** بشه به این معنی که کسی جز خودمون نفهمه چی نوشتیم و یا حتی اگه خودمون بعد از مدتی کد رو بررسی کنیم نتونیم توسعه‌اش بدیم.

توی این تمرین قصد داریم تمام عملیات‌های بالا رو بخش‌بندی کنیم و بین بخش‌های مختلف ارتباط مناسب ایجاد کنیم تا یک **کد تمیز** داشته باشیم.

خب با توجه به نوع عملیات‌ها به کلاس‌های زیر نیاز داریم:

کلاس SaleFileDataSource

- نوشتن اطلاعات **فروش** در فایل
- خواندن اطلاعات **فروش** از فایل

کلاس SaleInputDataSource

- نوشتن اطلاعات **فروش** در کنسول
- خواندن اطلاعات **فروش** از کنسول

کلاس ProductFileDataSource

- نوشتن اطلاعات **محصول** در فایل
- خواندن اطلاعات **محصول** از فایل

کلاس ProductInputDataSource

- نوشتن اطلاعات **محصول** در کنسول
- خواندن اطلاعات **محصول** از کنسول

نمودار دو کلاس مربوط به Product به صورت زیره:

نمودار دو کلاس مربوط به Sale به صورت زیره:

قبلا از اینکه درمورد توابع توضیح بدم باید توجه کنید که هر دو کلاسی که در یک دیاگرام اومدن خیلی به هم شبیه هستند در واقع هر چهار تابع اونها یکی هست و تنها تفاوتش در **نحوه پیاده‌سازی** هست به عنوان مثال هر دو کلاس ProductFileDataSource و ProductInputDataSource دارای تابع getProducts هستند با این تفاوت که اولی اطلاعات محصولات رو از فایل و دومی از کنسول می‌گیره. این شباهت در **ساختار کلی** و تفاوت در **نحوه پیاده‌سازی** مارو یاد **چندریختی** میندازه به این معنی که این دو شکل‌های متفاوت از یک چیز هستند. به نمودار زیر دقت کنید:

همونطور که مشاهده می‌کنید دو کلاس ProductFileDataSource و ProductInputDataSource، اینترفیس ProductDataSource رو پیاده‌سازی کردن. همین اتفاق برای Sale می‌افته:

به کل فرآیندی که تو بالا گفته شد **generalize** کردن گفته میشه که از چند ماهیت جزئی به یک ماهیت کلی می‌رسیم که ویژگی مشترک تمام ماهیت‌های جزئی رو داراست.

این رو هم اضافه کنم که دو کلاس SaleDataSource و ProductDataSource رو هم میشه به کلاسی مثل DataSource تعمیم داد اما برای این کار نیاز به مفهوم **generic type** داریم به همین خاطر از گفتن این بخش می‌گذرم و به عنوان مطالعه آزاد به خودتون می‌سپرم.

رابط ProductDataSource

توابع این کلاس به صورت زیر هستند:

```
List<Product> getProducts(int n);

Product getProduct();

void saveProducts(List<Product> products);

void saveProduct(Product product);
```

تابع getProducts به این صورت عمل می‌کنه که اطلاعات n محصول آخر رو از source مربوطه جمع‌آوری می‌کنه و خروجی میده. تابع getProduct دقیقا مثل getProducts عمل می‌کنه با این تفاوت که آخرین محصول رو برمی‌گردونه. تابع saveProducts لیست محصولاتی که به عنوان ورودی می‌گیره در source مربوطه ذخیره می‌کنه مثلا اگه این source فایل باشه اونها رو در file ذخیره می‌کنه.

رابط SaleDataSource

توابع این کلاس به صورت زیر هستند:

```
List<Sale> getSales(int n);

Sale getSale();

void saveSales(List<Sale> sales);

void saveSale(Sale sale);
```

توابع این رابط دقیقا مانند ProductDataSource است با این تفاوت که اطلاعات مربوط به Sale را مدیریت می‌کند.

تا اینجای کار باید توابع و رابط‌های زیر رو پیاده‌سازی کرده باشیم:

- رابط ProductDataSource
- کلاس ProductInputDataSource
- کلاس ProductFileDataSource
- رابط SaleDataSource
- کلاس SaleInputDataSource
- کلاس SaleFileDataSource

به همراه دو کلاس زیر که برای ذخیره‌سازی داده هستند:

- کلاس Product
- کلاس Sale

کلاس Repository

وقتی که برنامه Datasource های متفاوتی داره نیاز به یک کلاس داریم که اونها رو اجماع کنه و استفاده از داده رو راحت کنه. مثلا تو بعضی از برنامه‌های بزرگ بعضی از داده‌ها از اینترنت میاد و بعضی از حافظه داخلی سیستم تو این حالت دو source برای کار با داده وجود داره, یکی RemoteDatasource که برای مدیریت داده از طریق اینترنت هست و یکی LocalDataSource که مربوط به مدیریت داده از طریق حافظه داخلی سیستم هست. تو این شرایط یک کلاس Repository بین RemoteDatasource و LocalDataSource ارتباط برقرار می‌کنه و استفاده از داده رو آسون می‌کنه. وقتی که کلاس Repository داریم سایر بخش‌های برنامه برای گرفتن داده **تنها** به Repository مراجعه می‌کنن و کاری به Datasource ها ندارن.

حالا ما تو برنامه چهارتا DataSource داریم که برای مدیریت اونها کلاس Repository رو می‌سازیم.

```
public class Repository {

    private ProductDataSource productInputDataSource;
    private ProductDataSource productFileDataSource;
    private SaleDataSource saleInputDataSource;
    private SaleDataSource saleFileDataSource;

    public Repository() {
        //TODO
    }

    public void saveProduct() {
        //TODO
    }

    public void saveProducts(int n) {
        //TODO
    }

    public void saveSale() {
        //TODO
    }

    public void saveSales(int n) {
        //TODO
    }

    public void printProduct() {
        //TODO
    }

    public void printProducts(int n) {
        //TODO
    }

    public void printSale() {
        //TODO
    }

    public void printSales(int n) {
        //TODO
    }
}
```

شما باید کلاس بالا رو طوری کامل کنید که توابع اون به صورت زیر کار کنند:

- تابع saveProduct اطلاعات یک محصول رو از کنسول می‌گیرد و نتیجه آن را در فایل ذخیره می‌کند.
- تابع printProduct اطلاعات آخرین محصول را از فایل می‌خواند و در خروجی چاپ می‌کند.
- تابع saveSale اطلاعات یک فروش رو از کنسول می‌گیرد و نتیجه آن را در فایل ذخیره می‌کند.
- تابع printSale اطلاعات آخرین فروش را از فایل می‌خواند و در خروجی چاپ می‌کند.

چهار تابع دیگر نیز دقیقا به صورت بالا هستند با این تفاوت که برای n محصول و فروش این کار را انجام می‌دهند.

> تمام توابع فوق حداکثر به دو خط کد نیاز دارند!

کلاس Main

در این کلاس تابع main را قرار می‌دهیم و منطق برنامه را پیاده‌سازی می‌کنیم. برنامه اصلی در یک **لوپ بینهایت** جریان خواهد داشت و هر بار پیام زیر به کاربر نمایش داده خواهد شد:

```
1. Print Products
2. Print last Product
```

```
3. Print Sales
4. Print last Sale
5. Save a new Product
6. Save new Products
7. Save a new Sale
8. Save new Sales
9. Exit
```

کاربر یکی از موارد فوق را انتخاب خواهد کرد. اگر کاربر گزینه ۲ را انتخاب کند اطلاعات آخرین محصول با فرمت زیر نمایش داده خواهد شد:

```
<id> <name> <price> <number>
```

به عنوان مثال مورد زیر را در نظر بگیرید:

```
1 pen 1000 23
```

اگر کاربر گزینه ۱ را انتخاب کند پیام زیر نمایش داده خواهد شد:

```
Please enter number of Products to show :
```

و بعد از گرفتن تعداد، اطلاعات محصولات در سطرهای جدا نمایش داده خواهد شد. توجه کنید که اگر تعداد ورودی بیشتر از تعداد موجود در فایل بود، همان تعداد که در فایل است نمایش داده شود مثلا اگر ۱۰ وارد شد و فقط اطلاعات ۳ محصول در فایل ذخیره شده بود اطلاعات همان سه مورد نمایش داده شود.

گزینه ۳ و ۴ دقیقا مانند ۱ و ۲ است با این تفاوت که برای اطلاعات **فروش** است. کاربرد چهار گزینه بعد نیز مشخص است تنها توجه کنید اگر کاربر می‌خواست بیش از یک مورد داده وارد کند ابتدا باید تعداد آن پرسیده شود.

با انتخاب گزینه ۹ **حلقه بینهایت** شکسته می‌شود و برنامه به اتمام می‌رسد.

- پس از انجام هر یک از عملیات‌های بالا لیست انتخاب‌ها دوباره نمایش داده می‌شود.
- توجه کنید تنها راه کار کردن با فایل و ورودی استاندارد در کلاس Main با استفاده از کلاس Repository انجام شود.

نحوه آپلود کردن

فایل‌های خود را در پکیج datasource قرار دهید و به صورت فشرده آپلود کنید.

```
.
└── datasource
    ├── Product.java
    ├── Sale.java
    ├── ProductDatasource.java
    ├── ProductInputDatasource.java
    ├── ProductFileDatasource.java
    ├── SaleDatasource.java
    ├── SaleFileDatasource.java
    ├── SaleInputDatasource.java
    └── Repository.java
```

Matrix

همه با ماتریس و بردار در جبرخطی آشنا هستیم. تو این تمرین قصد داریم این دو شیئ ریاضی رو مدل‌سازی کنیم و براشون کلاس بسازیم.

توجه کنید که تمام اجزاء این دو کلاس نمایش داده **نشده**!

کلاس Matrix

همونطور که در نمودار بالا مشاهده می‌کنید در سازنده این کلاس یک آرایه دوبعدی می‌گیریم که به عنوان داده‌های ماتریس ازش نگه‌داری می‌کنیم. در ادامه می‌خوایم عملیات‌هایی که روی ماتریس انجام می‌شن رو با استفاده از توابع پیاده‌سازی کنیم:

```
public int det() {
    //TODO
}

private int det(int[][] matrix) {
    //TODO
}

public Matrix multiply(Matrix secondMatrix) {
    //TODO
}

public Matrix add(Matrix secondMatrix) {
    //TODO
}

public boolean isInvertible() {
    //TODO
}

public int getElement(int i, int j) {
    //TODO
}
```

- تابع det دترمینان ماتریس را محاسبه می‌کند و برمی‌گرداند. پیاده‌سازی این تابع **باید** به صورت **بازگشتی** انجام شود.
  - در صورتی که ماتریس مربعی نباشد دترمینان ندارد در این حالت خطای IllegalStateException پرت کنید.
- تابع multiply یک Matrix به عنوان ورودی می‌گیرد و ضرب ماتریسی انجام می‌دهد. توجه کنید که ماتریس ورودی در سمت راست ضرب قرار دارد.
  - در صورت که بعد دوم ماتریس اول با بعد اول ماتریس دوم برابر نباشد ضرب ماتریسی امکان‌پذیر نیست در این حالت خطای IllegalArgumentException پرت کنید.
- تابع add یک Matrix به عنوان ورودی می‌گیرد و جمع ماتریسی انجام می‌دهد.
  - در صورتی که بعد اول و دوم ماتریس اول با بعد اول و دوم ماتریس دوم برابر نباشد جمع ماتریسی امکان‌پذیر نیست در این حالت خطای IllegalArgumentException پرت کنید.
- تابع isInvertible مشخص می‌کند که آیا ماتریس وارون‌پذیر است یا خیر. یک تابع در صورتی وارون‌پذیر است که دترمینان آن مخالف صفر باشد.
- تابع getElement عنصر سطر i و ستون j را برمی‌گرداند.

کلاس Vector

کلاس Vector فرزند کلاس Matrix است و درنتیجه توابع آن را به ارث می‌برد. علاوه بر توابع کلاس پدر به تعدادی تابع دیگر نیز در این کلاس احتیاج داریم:

```
public int dotProduct(Vector secondVector) {
    //TODO
}

public int getElement(int i) {
    //TODO
}
```

- تابع dotProduct بین دو بردار ضرب داخلی انجام می‌دهد. + در صورتی که بعد دو بردار برابر نباشد ضرب داخلی امکان‌پذیر نیست در این صورت باید خطای IllegalArgumentException را چاپ کنید.
- تابع getElement عنصر iام بردار را برمی‌گرداند.

نحوه آپلود

- توجه کنید که کد شما با **تست کیس** تصحیح خواهد شد در نتیجه در انتخاب اسم کلاس و تابع دقت کنید همچنین ورودی توابع هم باید مثل توضیحات باشد و همچنین اسم آنها.
- کد خود را به صورت زیر پکیج‌بندی کنید و به صورت فشرده ارسال کنید:

```
.
└── matrix
    ├── Matrix.java
    └── Vector.java
```

تعداد سن

در این برنامه قصد داریم تا کار با **Set** و **Map** رو تمرین کنیم.

برنامه در کنسول اجرا می‌شود و در یک **لوپ بینهایت** از کاربر سوالات زیر پرسیده می‌شود:

```
1. Enter new record.
2. Print list of ages.
3. Exit.
```

وقتی کاربر گزینه یک رو انتخاب می‌کنه دو تا ورودی نام و سن رو ازش می‌گیریم، اول نام:

```
Please Enter Name :
```

و بعد سن:

```
Please Enter Age :
```

بعد باید این داده رو ذخیره کنیم.

وقتی گزینه دوم انتخاب میشه باید اول یک سن مشخص گزارش بشه بعد نشون داده بشه از این سن چندتا داده داشتیم. فرض کنید داده‌های ما به صورت زیر باشه:

```
Mamad 22
Ali 40
Roz 12
Sina 12
```

وقتی گزینه ۲ انتخاب بشه باید لیست زیر رو نمایش بدیم:

```
22 --> 1
40 --> 1
12 --> 2
```

با انتخاب گزینه ۳ برنامه تموم میشه.

الگوریتم

- برای اینکه زوجِ نام و سن رو ذخیره کنیم از یه Map از String به Integer استفاده کنید.
- وقتی می‌خواید لیست سن و تعداد رو نشون بدید کارای زیر رو انجام بدید:
  - اول یه Set بسازید و سن‌های متفاوتی که تو Map وجود داره رو بهش اضافه کنید اینطوری سن‌های تکراری حذف میشه.
  - یه Map از Integer به Integer بسازید تا بتونید هر سن و تعداد تکرارشو ذخیره کنیم.
  - بعد بازای هر کدوم از سن‌های توی Set کارای زیر رو انجام بدید:
    - داخل Map مربوط به اسم و سن رو بگردید و تعداد افرادی که این سن رو دارن مشخص کنید.
    - سن و تعدادش رو به Map مربوطه اضافه کنید.
  - آخر سر با استفاده از Map مربوط به سن و تعداد لیست رو نمایش بدید.

نحوه ارسال

کد خودتون رو داخل تابع main پیاده‌سازی کنید و ارسال کنید.

- مطمئن باید که اسم پکیج بالای فایلتون درج نشده باشه!