مدیریت اطلاعات کارمندان

- سطح: آسان
- طراح: آرتین عضدی فر

ابتدا فایل پروژه اولیه را از این لینک دانلود کنید.

در یک شرکت در حال رشد، مدیر منابع انسانی تصمیم دارد گزارش دقیقی از وضعیت حقوق کارمندان در بخش‌های مختلف تهیه کند تا بتواند در تصمیم‌گیری‌های مدیریتی از آن استفاده کند. او از شما خواسته برنامه‌ای بنویسید که با دریافت فهرستی از کارمندان، برای هر دپارتمان کارمندی را که بالاترین حقوق را دارد شناسایی کرده و نتیجه را برگرداند.

معرفی کلاس‌ها:

کلاس Employee:

نشان دهنده اطلاعات یک کارمند است. کانستراکتور و `getter` های کلاس را کامل کنید.

معرفی فیلدها:

۱. `name` : نام کارمند از جنس `String`

۲. `department` : دپارتمانی که کارمند در آن مشغول به کار است.

۳. `salary` : حقوق دریافتی کارمند از جنس `int`

کلاس EmployeeAnalyzer:

مدیریت کارمندان به کمک متدهای این کلاس انجام می‌شود.

این کلاس فیلد خاصی ندارد و از کانستراکتور پیش‌فرض استفاده می‌کند.

معرفی متدها:

۱. `getUniqueDepartments` : یک لیست از کارمندان ورودی می‌گیرد و مجموعه‌ای بدون تکرار از دپارتمان‌هایی که این کارمندان در آنها مشغولند بر می‌گرداند.

توجه: بعضی از کارمندان که استخدامشان هنوز تکمیل نشده است، دپارتمان مشخصی ندارند. در صورت وجود این کارمندان در لیست، برای این متد و بقیه متدهایی که با دپارتمان‌ها مرتبطند، این کارمندان را در دسته `Unassigned` قرار دهید.

```
public Set<String> getUniqueDepartments(List<Employee> employees)
```

۲. `filterBySalary` : یک لیست از کارمندان ورودی می‌گیرد و لیستی از آن‌ها را بر می‌گرداند که حقوق دریافتی آنها بین min و max است (با در نظر گرفتن خود این مقدارها) بر می‌گرداند. در صورتی که هر کدام از طرفین این بازه -1 بود، از آن سمت محدودیت حقوقی وجود ندارد.

مثال: برای ورودی min=100 , max = -1 تمام کسانی که حقوقی بزرگتر مساوی ۱۰۰ دارند مورد تاییدند.

```
public List<Employee> filterBySalary(List<Employee> employees, int minSalary,
```

۳. `groupByDepartment` : لیستی از کارمندان ورودی می‌گیرد و به کمک یک Map مشخص می‌کند هر دپارتمان چند کارمند دارد.

```
public Map<String, Integer> groupByDepartment(List<Employee> employees)
```

۴. `highestPaidByDepartment` : به کمک یک Map مشخص می‌کند کدام کارمند بالاترین حقوق دریافتی در هر دپارتمان را دارد. (در صورتی که در یک دپارتمان دو نفر هم زمان بالاترین مقدار حقوق را داشتند، فردی که زودتر در لیست آمده انتخاب شود).

```
public Map<String, Employee> highestPaidByDepartment(List<Employee> employees
```

آنچه باید آپلود کنید:

یک فایل `zip` با ساختار زیر ارسال کنید

```
<zip_file_name.zip>
```

```
├── Employee.java
└── EmployeeAnalyzer.java
```

سیستم تحلیل ترافیک

- سطح: متوسط
- طراح: مهدی افشاری

پروژه اولیه را ازاینجا دانلود کنید

به **DigitalWeave**، سریع‌ترین رسانه اجتماعی در حال رشد جهان خوش آمدید! شما یک مهندس نرم‌افزار تازه‌کار هستید و امروز اولین روز کاری شماست. مدیر شما، دکتر وحیدی، از شما می‌خواهد که با انجام یک کار کوچک اما مهم، یعنی تحلیل گزارش‌های خام یک اپلیکشن وب، با پایگاه کد شرکت آشنا شوید.

این گزارش ها (logs) حاوی اطلاعات ارزشمندی هستند، اما برای مفید بودن باید پردازش شوند. هدف شما پیاده‌سازی مجموعه‌ای از متود ها برای استخراج معیارهای کلیدی از این داده‌ها است. شما باید برای هرکدام از متود ها مناسب ترین ساختار ذخیره سازی داده که یکی از اینترفیس های $List$, Set, Map را پیاده سازی میکند، استفاده کنید!

کلاس `TrafficEvent`

ابتدا، شما باید کلاسی برای نمایش هر گزارش ایجاد کنید.

فیلدها:

- `timestampHour (int)` : ساعتی که رویداد در آن رخ داده است (۰-۲۳).
- `userId (String)` : شناسه‌ی منحصر به فرد کاربر.
- `pageUrl (String)` : صفحه‌ای که به آن دسترسی پیدا شده است (مثلاً "/home").
- `httpStatusCode (int)` : نتیجه‌ی HTTP رویداد (مثلاً 200 برای موفقیت، 404 برای پیدا نشدن).

الزامات:

۱. کلاس باید یک سازنده (constructor) داشته باشد که تمام چهار فیلد را مقداردهی اولیه کند.

۲. کلاس باید getter های لازم را پیاده سازی کند.

کلاس `TrafficAnalyzer`

پنج متود شرح داده شده در زیر را پیاده‌سازی کنید.

متود `filterByHour` - فیلتر و مرتب‌سازی رویدادها

امضای متود:

```
public static List<TrafficEvent> filterByHour(List<TrafficEvent> log, int tar
```

این متود داده ها را برای یک ساعت معین فیلتر میکند. لیست رویدادهای بازگشتی باید طبق قواعد زیر **مرتب** شود:

- داده ها باید بر اساس **شناسه‌ی کاربری** مرتب شوند (Lexicographically)
- اگر دو داده دارای شناسه‌ی کاربری یکسان بود باید بر اساس **آدرس صفحه** مرتب شوند (Lexicographically)

متود `findUniqueUsers` - کاربران منحصر به فرد علاقه‌مند به محصولات

امضای متود:

```
public static Set<String> findUniqueUsers(List<TrafficEvent> log)
```

این متود یک `Set` از `userId` های **منحصر به فرد** را بازمی‌گرداند. این مجموعه باید فقط شامل کاربرانی باشد که **حداقل یک بار** از صفحه‌ای بازدید کرده‌اند که با پیشوند `"/products"` شروع می‌شود.

متود `calculatePageVisits` - محبوبیت صفحات با لیست استثنا

امضای متود:

```
public static Map<String, Integer> calculatePageVisits(List<TrafficEvent> log
```

این متود بازدیدهای صفحات را در یک `Map` جمع‌آوری می‌کند، به طوری که کلید آن، `pageUrl` و مقدارش تعداد بازدید آن می‌باشد. محاسبات شما باید صفحاتی را که در لیست استثنای زیر قرار دارند **نادیده بگیرد**: `"/login"` ، `"/error"` .

متود `groupEventsByUser` - بازسازی سفر کاربران (Map of Lists)

امضای متود:

```
public static Map<String, List<TrafficEvent>> groupEventsByUser(List<TrafficE
```

متودی ایجاد کنید که یک `Map` بازگرداند که در آن هر کلید یک `userId` و مقدار آن یک `List` از تمام `TrafficEvent` های تولید شده توسط آن کاربر باشد. لیست هر کاربر باید **فقط شامل رویدادهای موفق باشد** (جایی که `httpStatusCode` برابر 200 است).

متود `getHourlyErrorCount` - گزارش ساعتی خطاها

امضای متود:

```
public static Map<Integer, Integer> getHourlyErrorCount(List<TrafficEvent> lo
```

متودی ایجاد کنید که تمام رویدادهای خطای سرور را پیدا کند (جایی که `httpStatusCode` بین ۵۰۰ و

۵۹۹ است و شامل خود این اعداد نیز میشود!). این متود باید یک `Map` بازگرداند که کلید آن ساعت و مقدار آن تعداد خطاها در آن ساعت باشد. `Map` نهایی **باید بر اساس ساعت (کلید) مرتب شده باشد**.

داده‌هایی برای تست

برای تست متود های خود از نمونه‌ی زیر استفاده کنید.

ساعت	شناسه‌ی کاربر	آدرس صفحه	کد وضعیت
9	userA	/home	200
9	userB	/products	200
10	userA	/products/checkout	200
10	userC	/login	401
10	userB	/contact	200
10	userA	/products/cart	500
11	userD	/about	404
11	userA	/products	200
11	userB	/home	503
11	sys-admin	/error	200

آنچه باید آپلود کنید:

یک فایل `zip` با ساختار زیر ارسال کنید

```
<zip_file_name.zip>
    ├── TrafficEvent.java
    └── TrafficAnalyzer.java
```

کوییز

- سطح: متوسط
- طراح: سید محمد حسینی

حتما همه شما با بازي quiz of kings اشنا هستين (اگر هم اشنا نيستين اصلا نگران نباشين :)) ميتونين از اينجا دانلود كنين و بازي كنين) . بچه هاي كامپيوتر بهشتي با الهام گرفتن از اون بازي ميخوان برنامه اي درست كنن كه هرفرد بتونه تنهايي بازي كنه و اطلاعات عموميشو ارتقا بده. فايل ابتدايي برنامه را دانلود كنيد و موارد خواسته شده را پياده سازي كنيد . كلاينت اين بازي از پيش نوشته شده كه به همراه فايل خام سرور و فايل سوالات از اين لينک قابل بارگيری است.

شما باید تابع runserver را پیاده سازي كنيد.

سرور به محض وصل شدن يك كلاينت سوالات را به ترتيب از بالا به پايين براي كلاينت ميفرستد و منتظر جواب از سمت كلاينت ميماند اگر جواب درست بود ۵ امتياز اضافه ميشود و در غير اين صورت امتيازی اضافه نميشود.

اين كار را تا تمام شدن سوالات ادامه ميدهد و در اخر با فرستادن كلمه finish و سپس امتياز كلاينت, به برنامه پايان ميدهد .

- به كوچك و بزرگ بودن ورودي ها و خروجي ها دقت كنيد.
- پس از اينكه به هر سوال پاسخ داده شد بايد نتيجه (correct) يا (incorrect) مشخص شود.
- سوالات و پاسخ ها در فايل questions.txt هستند. دقت كنيد سوال و جواب با ? از هم جدا شده اند.
- فايل های client.java ,server.java را زيپ كرده و ارسال كنيد.

BidSocket

- سطح: سخت
- طراح: علیرضا متقی

پس از موفقیت در پروژه‌های پیام‌رسان، شرکت «توسعه‌گران شبکه» قصد دارد وارد بازار تجارت الکترونیک شود. هدف، ایجاد یک پلتفرم حراجی آنلاین بی‌درنگ (Real-time) است که در آن چندین کاربر بتوانند به صورت همزمان به سرور متصل شده، لیست کالاها را مشاهده کنند و پیشنهاد قیمت (Bid) خود را ارسال نمایند. شما به عنوان توسعه‌دهنده ارشد، وظیفه دارید هسته اصلی این سیستم را با معماری **Client-Server** پیاده‌سازی کنید. همچنین، مدیریت درخواست‌های همزمان برای ثبت قیمت روی یک کالای واحد باید به طوری باشد که داده‌ها دچار تداخل نشوند (Race Condition).

ساختار کلی برنامه

پروژه شامل دو بخش اصلی است:

۱. **سرور (AuctionServer):** وظیفه نگهداری لیست کالاها، مدیریت اتصال کلاینت‌ها و پردازش پیشنهادهای قیمت را بر عهده دارد.
۲. **کلاینت (AuctionClient):** رابط کاربری کنسولی برای اتصال به حراجی و ارسال دستورات.

پیاده‌سازی

کلاس `AuctionServer`

این کلاس سرور اصلی است که باید به صورت **Multi-threaded** پیاده‌سازی شود تا بتواند پاسخگوی چندین کلاینت به صورت همزمان باشد.

فیلدها:

- `int port` : پورت سرور (پیش‌فرض: ۱۲۳۴).
- `Map<String, Item> inventory` : یک `HashMap` که نام کالا را به شیء کالا (`Item`) نگاشت می‌کند. این لیست باید بین تمام کلاینت‌ها مشترک باشد.

- `boolean isRunning` : وضعیت اجرای سرور.

متد `start` : این متد باید:

۱. تعدادی کالای پیش‌فرض به `inventory` اضافه کند (مثلاً: "Laptop" با قیمت پایه 1000، "Phone" با قیمت پایه 500).
۲. `ServerSocket` را روی پورت مشخص شده باز کند.
۳. در یک حلقه بی‌نهایت، منتظر اتصال کلاینت بماند.
۴. به محض اتصال کلاینت، یک شیء از کلاس `ClientHandler` ساخته و آن را در یک `Thread` جدید اجرا کند.

```java
public void start() {
// 1. Initialize inventory
inventory.put("Laptop", new Item("Laptop", 1000));
inventory.put("Phone", new Item("Phone", 500));

// 2. Start ServerSocket
try (ServerSocket serverSocket = new ServerSocket(port)) {
    System.out.println("Auction Server started on port " + port);

    while (isRunning) {
        // TODO: Accept client connection
        // TODO: Create a new Thread for ClientHandler and start it
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
```

کلاس `ClientHandler`

این کلاس یک `Inner Class` درون `AuctionServer` است (یا یک کلاس جداگانه که به `inventory` دسترسی دارد) و وظیفه تعامل با یک کلاینت خاص را دارد. این کلاس باید رابط `Runnable` را پیاده‌سازی کند.

متد `run` : باید دستورات زیر را از کلاینت دریافت و پردازش کند:

۱. LIST : لیست تمام کالاها به همراه بالاترین قیمت فعلی و نام بالاترین پیشنهاد دهنده را ارسال کند.

۲. BID <ItemName> <Amount> : تلاش برای ثبت قیمت جدید.

- اگر کالا وجود نداشت: پیام Item not found ارسال شود.
- اگر قیمت پیشنهادی کمتر یا مساوی قیمت فعلی بود: پیام Bid too low ارسال شود.
- اگر پیشنهاد معتبر بود: قیمت کالا و نام پیشنهاد دهنده به‌روزرسانی شود و پیام Bid accepted ارسال گردد.
- **نکته مهم:** عملیات ثبت قیمت باید synchronized باشد تا دو کلاینت همزمان نتوانند قیمت را خراب کنند.

۳. QUIT : قطع ارتباط کلاینت.

```
private class ClientHandler implements Runnable {
private Socket socket;

public ClientHandler(Socket socket) {
    this.socket = socket;
}

@Override
public void run() {
    try (
        BufferedReader in = new BufferedReader(new InputStreamReader(so
        PrintWriter out = new PrintWriter(socket.getOutputStream(), tru
    ) {
        out.println("Welcome to BidSocket! Commands: LIST, BID <item> <

        String line;
        while ((line = in.readLine()) != null) {
            // TODO: Parse command (LIST, BID, QUIT)
            // TODO: Handle logic safely
        }
    } catch (IOException e) {
        System.out.println("Client disconnected.");
    }
}
}
```

کلاس Item

یک کلاس کمکی برای نگهداری اطلاعات کالا. **فیلدها:**

- `String name` : نام کالا.
- `double currentPrice` : قیمت فعلی.
- `String highestBidder` : نام کاربری کسی که بالاترین پیشنهاد را داده (در ابتدا "None").

نکته: برای جلوگیری از مشکلات همزمانی، بهتر است متد تغییر قیمت (`placeBid`) در این کلاس `synchronized` باشد.

کلاس AuctionClient

این کلاس مشابه کلاس کلاینت در تمرین‌های قبلی است.

۱. به سرور متصل می‌شود.
۲. یک حلقه برای خواندن دستور از کاربر (کنسول) دارد.
۳. دستور را به سرور می‌فرستد و پاسخ سرور را چاپ می‌کند.

نمونه ورودی و خروجی (Console):

```
Connected to Auction Server.
Server: Welcome to BidSocket! Commands: LIST, BID <item> <amount>, QUIT

Client Input: LIST
Server:
 - Laptop: $1000.0 (Holder: None)
 - Phone: $500.0 (Holder: None)

Client Input: BID Laptop 1005
Server: Bid accepted. New Price: 1005.0

Client Input: BID Laptop 900
Server: Bid too low. Current: 1005.0

Client Input: BID Car 2000
Server: Item not found.
```

```
Client Input: QUIT
Server: Connection closed.
```

آنچه باید آپلود کنید

ساختار فایل زیپ ارسالی باید به صورت زیر باشد:

```
<zip_file_name.zip>
    ├── AuctionServer.java
    ├── AucitionClient.java
    └── Item.java
```