کتابخانه هوشمند

- سطح: آسان
- طراح: آیسودا فضلی خانی

مسئول کتابخانه دانشکده سال هاست که با شیوه دستی و کاغذی کتابخانه را مدیریت می‌کند؛ دفترهای پر از نوشته و یادداشت‌هایی که هربار گم و دوباره پیدا می‌شوند. اما با گذر زمان تعداد کتاب‌ها و دانشجویان بیشتر شده و هرترم با ورود و خروج ده‌ها کتاب جدید، کنترل این همه تغییر برای او به یک کابوس تبدیل شده است! یک روز که از خستگی در دفترش قهوه می‌نوشید و در اینترنت دنبال سیستم مدیریت کتابخانه می‌گشت، ناگهان چشمش به version control افتاد! کنترل تغییرات بدون نیاز به نوشتن همه چیز از اول، برایش خبر خوشحال کننده‌ای بود.

او از شما خواسته با استفاده از گیت، سیستمی طراحی کنید که کتاب های رشته‌های مختلف را به‌صورت جداگانه مدیریت کند و تاریخچه کامل ورود هرکتاب به کتابخانه را ثبت کند و اعضای کتابخانه را مدیریت کند.

خواسته مسئله

۱. برای شروع کار یک پوشه با نام **library** بسازید و داخل آن repository ایجاد کنید.
۲. نام خود را بعنوان مدیر تنظیم کنید.
۳. با استفاده از دستور **touch** فایل‌های اولیه که شامل `computer_engineering.txt` و `electrical_engineering.txt` و `members.txt` هستند را ایجاد کنید و آن‌ها را با پیغام **"basic structure"** کامیت کنید.
۴. اسامی کتاب‌های مهندسی کامپیوتر که شامل "**computer network**"، "**Introduction to algorithm**" و "**Database system concepts**" هستند را به `computer_engineering.txt` اضافه کنید و آن را به staging area اضافه کنید.
۵. اسامی کتاب های مهندسی برق که شامل "**Machines**"، "**Circuits**" و "**Power Electronics**" هستند را به `electrical_engineering.txt` اضافه کنید و آن را به staging area اضافه کنید.
۶. فایل `computer_engineering.txt` را کامیت کنید و وضعیت فایل‌ها را مشاهده کنید.

۷. فایل `electrical_engineering.txt` را کامیت کنید.

۸. قبل از اضافه کردن کتاب‌های **mathematics** یک tag با نام `v1.0-stable` ایجاد کنید تا نسخه فعلی را ذخیره کند.

۹. یک **branch جدید** با نام **mathematics** ایجاد کنید و سپس لیست تمام branchهای موجود را مشاهده کنید.

۱۰. در برنچ **mathematics** یک فایل جدید به نام `mathematics.txt` ایجاد کنید و اسامی کتاب‌های "**Linear Algebra**", "**Calculus**" و "**Statistics**" را به ترتیب به آن اضافه کنید و سپس آن را با پیغام "**Add mathematics books**" کامیت کنید.

۱۱. به شاخه master برگردید و اسامی کتاب‌های "**Numerical methods**" و "**Advanced Calculus**" را به `mathematics.txt` اضافه کنید.

۱۲. تغییرات را با پیغام "**Add mathematics books in main branch**" کامیت کنید و در صورت رخ دادن conflict، آن را طوری برطرف کنید که اسامی هر ۵کتاب گفته شده داخل فایل `mathematics.txt` وجود داشته باشد.

۱۳. متوجه شده‌ایم که کتاب‌های **mathematics** نباید داخل این کتابخانه قرار بگیرند و اشتباه اضافه شده‌اند. به تگ `v1.0-stable` که قبل از اضافه کردن کتاب‌های ریاضی ایجاد کردید برگردید و سپس با بررسی وضعیت پروژه، مطمئن شوید کتاب‌های ریاضی حذف شده‌اند.

چیزی که باید تحویل دهید

همه عملیات‌های گفته شده را در **VS Code** و یا **intelliJ IDEA** انجام دهید. در نهایت باید یک فایل zip حاوی **ریپازیتوری پروژه** در کنار یک فایل pdf شامل تصویر نمودار **Gitgraph** و تمام **دستورات لازم** به ترتیب گفته شده آپلود کنید.

رستوران بهنام

- سطح: متوسط
- طراح: روزبه سلطانی

مدیر رستوران بهنام قصد دارد سیستم ساده‌ای برای ثبت و محاسبه‌ی صورتحساب مشتریان طراحی کند.
در این سیستم، چند غذا وجود دارد و هر مشتری (عادی یا VIP) می‌تواند تعدادی از آن‌ها را سفارش دهد.
شما باید با استفاده از مفاهیم برنامه‌نویسی شیءگرا (OOP) این سیستم را طراحی کنید.

پروژه اولیه را از این لینک دانلود کنید.

توضیحات کلاس‌ها

کلاس `FoodItem`

نمایانگر یک غذا در منو است.

ویژگی‌ها (private):

- `String name` : نام غذا
- `int price` : قیمت غذا

سازنده:

```
1 | public FoodItem(String name, int price)
```

متدها:

```
1 | public String getName()
2 | public int getPrice()
```

متد `getName()` ویژگی `name` را برمیگرداند.

متد `getPrice()` ویژگی `price` را برمیگرداند.

کلاس Customer

نمایانگر یک مشتری است.

ویژگی‌ها (private):

- `String name` : نام مشتری
- `boolean isVip` : نشان دهنده VIP بودن یا نبودن مشتری
- `int balance` : موجودی مشتری
- `int x, y` : مختصات محل مشتری (برای محاسبه فاصله تا رستوران)

سازنده:

```
1 public Customer(String name, boolean isVip, int balance, int x, int y)
```

متدها:

```
1 public String getName()
2 public boolean isVip()
3 public int getBalance()
4 public int getX()
5 public int getY()
6 public boolean canAfford(int amount)
7 public void pay(int amount)
```

متد `getName()` ویژگی `name` را برمیگرداند.

متد `isVip()` ویژگی `isVip` را برمیگرداند.

متد `getBalance()` ویژگی `balance` را برمیگرداند.

متد `getX()` ویژگی `x` را برمیگرداند.

متد `getY()` ویژگی `y` را برمیگرداند.

متد `canAfford(int amount)` بررسی میکند که مشتری آیا موجودی بزرگتر یا مساوی `amount` دارد یا

خیر.

متد `pay(int amount)` مبلغ را از موجودی کم می‌کند، تنها در صورتی که موجودی کافی داشته باشد.

کلاس `Order`

نمایانگر یک سفارش از طرف مشتری است.

ویژگی‌ها (private):

- `Customer customer` : مشتری
- `FoodItem[] items` : آرایه ای برای نگهداری غذاهای سفارش داده شده.
- `int itemCount` : تعداد غذاهای سفارش داده شده.

توجه: تضمین میشود اندازه 50 برای آرایه items کافی باشد.

سازنده:

```
public Order(Customer customer)
```

متدها:

```
public void addItem(FoodItem foodItem)
public int calculateTotal()
public int countFood(String foodName)
public int getItemCount()
public Customer getCustomer()
```

متد `addItem(FoodItem foodItem)` غذای مورد نظر را به آرایه items اضافه میکند.

متد `calculateTotal()` مبلغ نهایی سفارش را به صورت زیر برمیگرداند:

۱. **جمع پایه:** مجموع قیمت تمام غذاها

۲. اگر تعداد آیتم‌ها کمتر از 5 باشد، هزینه سرویس `20` واحد به مبلغ اضافه شود.

۳. اگر تعداد آیتم‌ها 5 یا بیشتر باشد، هزینه سرویس حذف شود.

۴. اگر مشتری VIP باشد، 15٪ تخفیف روی جمع اعمال شود.

۵. در نهایت 9% مالیات به مبلغ اضافه میشود.

۶. خروجی باید **عدد صحیح (int)** باشد و قسمت اعشاری حذف شود.

متد `countFood(String foodName)` تعداد غذاهای سفارش داده شده که اسمشان `foodName` است را برمیگرداند.

متد `getItemCount()` ویژگی `itemCount` را برمیگرداند.

متد `getCustomer()` ویژگی `Customer` را برمیگرداند.

کلاس `DeliveryPerson`

نمایانگر پیک تحویل‌دهنده‌ی سفارش است.

ویژگی‌ها (private):

- `String name` : نام غذا
- `int deliveredCount` : تعداد سفارش‌های تحویل داده‌شده
- `int earnedMoney` : مجموع درآمد پیک از پاداش‌ها

سازنده:

```
public DeliveryPerson(String name)
```

متدها:

```
public String getName()
public int getDeliveredCount()
public int getEarnedMoney()
public void deliverOrder(Order order, int estimatedTime, int actualTime)
```

متد `getName()` ویژگی `name` را برمیگرداند.

متد `getDeliveredCount()` ویژگی `deliveredCount` را برمیگرداند.

متد `getEarnedMoney()` ویژگی `earnedMoney` را برمیگرداند.

متد `deliverOrder(Order order, int estimatedTime, int actualTime` سفارش را به مقصد میرساند و مبلغی را به عنوان پاداش به پیک اضافه میکند.

۱. اگر سفارش **زودتر یا سر وقت** تحویل داده شود، پاداش 20٪ از مبلغ کل سفارش است.
۲. اگر سفارش **دیرتر از زمان تخمینی** تحویل داده شود، پاداش 5٪ از مبلغ سفارش است.
۳. توجه داشته باشید مبالغ باید به **عدد صحیح (int)** تبدیل شوند و قسمت اعشاری حذف شود و به موجودی پیک اضافه شود.
۴. با هر تحویل، شمارنده‌ی `deliveredCount` افزایش می‌یابد.
۵. این متد باید فقط در متد `registerOrder` در کلاس `Restaurant` که پایین تر توضیح داده شده استفاده شود.

کلاس `Restaurant`

نمایانگر رستوران است که سفارش‌ها را نگهداری می‌کند و اطلاعات مربوط به آن‌ها را مدیریت می‌کند.

ویژگی‌ها (private):

- `Order[] orders` : آرایه‌ای برای ذخیره‌ی سفارش‌های ثبت‌شده.
- `int orderCount` : تعداد سفارش‌های ثبت‌شده.
- `int x, y` : مختصات موقعیت رستوران

توجه: تضمین میشود اندازه 50 برای آرایه orders کافی باشد.

سازنده:

```
public Restaurant(int x, int y)
```

متدها:

```
public double calculateDistance(Customer customer)
public int calculateEstimatedTime(Customer customer)
public void registerOrder(Order order, DeliveryPerson deliveryPerson, int act
```

```
public int getTotalPaidByCustomer(String customerName)
public int countFoodOrdered(String foodName)
public int getAverageOrderCost()
public int getMaxOrderCost()
public int getOrderCount()
public int getX()
public int getY()
```

- `calculateDistance(Customer c)` فاصله مشتری تا رستوران را محاسبه می‌کند. (میتوانید از متد `Math.sqrt()` استفاده کنید.)
- `calculateEstimatedTime(Customer c)` مدت زمان ارسال غذا را پیش بینی میکند. طوری که زمان ارسال برابر است با **دو برابر فاصله** و به صورت **int** برگردانده میشود.
- `registerOrder(order, deliveryPerson, actualDeliveryMinutes)` : این متد برای ثبت و ارسال سفارش به مشتری استفاده میشود.

۱. ابتدا اگر مشتری **موجودی کافی نداشت**، سفارش **ثبت نمی‌شود** و از متد خارج میشویم.
۲. در غیر این صورت، **مجموع مبلغ** سفارش محاسبه شده، از **موجودی مشتری کسر** می‌شود، سفارش در **آرایه** ذخیره می‌گردد، و بعد از محاسبه زمان ارسال پیش بینی شده, به `deliveryPerson.deliverOrder()` ارجاع داده می‌شود تا سفارش تحویل داده شود.

- متد `getTotalPaidByCustomer(String customerName)` مجموع هزینه‌هایی که یک مشتری با نام `customerName` در تمام سفارش‌هایش پرداخت کرده است را برمی‌گرداند.
- متد `countFoodOrdered(String foodName)` تعداد کل دفعاتی که یک غذای خاص با نام `foodName` در تمام سفارش‌ها ثبت شده است را برمی‌گرداند. (برای مثال اگر در یک سفارش 2 عدد پیتزا داشته باشیم باید هردو حساب شوند)
- متد `getAverageOrderCost()` میانگین مبلغ تمام سفارش‌های ثبت‌شده در رستوران را برمی‌گرداند. اگر هیچ سفارشی ثبت نشده باشد، مقدار `0` برمی‌گرداند.
- متد `getMaxOrderCost()` بیشترین مبلغ میان تمام سفارش‌های ثبت‌شده را برمی‌گرداند. اگر هیچ سفارشی ثبت نشده باشد، مقدار `0` برمی‌گرداند.
- متد `getOrderCount()` تعداد سفارش‌های ثبت‌شده در رستوران را برمی‌گرداند.
- متد `getX()` ویژگی `x` را برمیگرداند.
- متد `getY()` ویژگی `y` را برمیگرداند.

نکته :داخل پروژه اولیه یک فایل `Main.java` قرار داده شده که با اجرای آن صحت عملکرد برنامه خود را بسنجید. خروجی مورد نظر باید به صورت زیر باشد :

```
--- Final Status Check ---
Reza Balance: 126
Samad Balance: 150
Karim Balance: 810
Mina Balance: 268
Ali Delivered Count: 3
Ali Earned Money: 83
Sara Delivered Count: 1
Sara Earned Money: 6
Restaurant Order Count: 4
Pizza Count: 3
Salad Count: 5
Water Count: 3
Steak Count: 1
Avg Cost: 149
Max Cost: 203
Total Paid by Reza: 374
Total Paid by Karim: 190
Total Paid by Samad: 0
Total Paid by Mina: 32
```

فایل زیپ آپلودی وقتی باز میشود باید فقط شامل فایل های زیر باشد:

```
├── FoodItem.java
├── Customer.java
├── Restaurant.java
├── DeliveryPerson.java
└── Order.java
```

بانک تک‌سیتی

- سطح: متوسط
- طراح: محسن نوروزی

فایل اولیه پروژه را ازدریافت این لینک کنید.

در یک شهر پیشرفته به نام **تک‌سیتی**، همه چیز روی تکنولوژی می‌چرخد. یکی از مهم‌ترین بخش‌های این شهر، بانک دیجیتالی بزرگی به نام **Ideal Bank** است که نه شعبه فیزیکی دارد و نه کارمند انسانی — همه کارها توسط سیستم‌های هوشمند و ATMهای فوق‌مدرن انجام می‌شود.

مدیرعامل بانک، دکتر **الگوریتمی**، تصمیم گرفته یک نرم‌افزار جدید ATM طراحی کند که بتواند:

۱. **مدیریت انواع حساب‌ها** (حساب بانکی ساده، حساب پس‌انداز، و حساب پس‌انداز ویژه با پاداش سالانه)
۲. **امنیت بالا با استفاده از کد PIN** برای هر حساب (همان رمز حساب است که فقط برای جابه جایی پول استفاده می‌شود)
۳. **قوانین حفظ حداقل موجودی** برای حساب‌های پس‌انداز
۴. **امکان انتقال وجه بین حساب‌ها**
۵. **افزودن پاداش سالانه به حساب‌های ویژه**
۶. **قابلیت بستن حساب در صورت نیاز**

اما یک مشکل پیش آمده...

در روز افتتاح سیستم جدید، برق شهر در اثر طوفان قطع و وصل شد و داده‌های چند حساب به هم ریخت. حالا بانک از شما به عنوان **برنامه‌نویس ارشد** خواسته است این سیستم را به طور کامل پیاده‌سازی کنید تا همه حساب‌ها به شکل ایمن دوباره فعال شوند.

شما باید کلاس‌های لازم را با استفاده از اصول **OOP در جاوا** کامل کنید:

نکته بسیار مهم:پیاده سازی getter ها و setter و همچنین constructor ها به عهده دانشجو است !

۱. کلاس `BankAccount` برای حساب پایه .

به شکل زیر است:

```
public class BankAccount{
    private String accountNumber;
    protected double balance;
    private boolean isActive;
    private String pinCode;
}
```

در فایل های اولیه constructor و متدهای get و set ساخته شده‌اند ، وظیفه شما کامل کردن متد های close account ، deposit withdraw و resetPIN است.

۱. از **deposit** برای واریز پول به حساب استفاده می‌شود.

۲. از **withdraw** برای برداشت پول از حساب استفاده می‌شود.

۳. از **close account** برای بستن اکانت استفاده می‌شود.

۴. از **resetPIN** برای تغییر PIN استفاده میشود (بررسی کنید که PIN قبلی صحیح است یا نه).

۵. از **authenticate** برای بررسی اینکه پین حساب با پین ورودی ما یکی هست یا نه استفاده می‌شود.

نکته:در هر کدام از این متدها باید بررسی بشود که حساب active هست یا نه و اینکه مبلغ برداشت یا واریز مثبت هست یا نه.

۲. کلاس `SavingAccount` برای حساب پس‌انداز. به شکل زیر است:

```
private double minimumBalance;
```

حساب پس‌انداز یا همان SavingAccount این قابلیت رو داره که اگر موجودی از یک حدی کمتر بشه اجازه برداشت رو به کاربر نده. و در اینجا شما تنها باید متد withdraw و canWithdraw را کامل کنید.

۳. از **canWithdraw** برای بررسی فراهم بودن امکان برداشت استفاده می‌شود.

۴. از **withdraw** برای انجام عملیات برداشت استفاده می‌شود.

نکته:در هر کدام از این متدها باید بررسی بشود که حساب active هست یا نه و اینکه مبلغ برداشت یا واریز مثبت هست یا نه.

۳. کلاس `SpecialSavingAccount` برای حساب‌های ویژه با پاداش سالانه. به شکل زیر است:

```
private double bonusRate;
```

این نوع حساب به کاربر سود سالانه اضافه می‌کند و دو متد دارد:

previewAnnualBonus که مقدار این سود را حساب میکند و applyAnnualBonus که این مقدار را اعمال میکند.

نکته:برای محاسبه AnnualBonus باید نرخ سود را در موجودی حساب ضرب کنید و تقسیم بر 100 کنید.

۴. کلاس `ATMDevice` برای عملیات عمومی ATM .

در این بخش شما باید سه متد زیر را پیاده سازی کنید:

۱. از **transfer** برای جابه جایی پول بین دو اکانت استفاده میشود (در آن باید نکات مربوط به PIN و درست بودن آن و مبلغ و فعال بودن اکانت چک شود).
۲. از**checkBalance** برای دیدن مقدار پول موجود استفاده می‌شود.
۳. از **quickWithdraw** برای برداشت پول استفاده می‌شود (در آن باید نکات مربوط به PIN و درست بودن آن و مبلغ و فعال بودن اکانت چک شود).

`Main` برای شبیه‌سازی کل ماجرا (به شما کمک میکند که از صحیح بودن کد خود مطمئن شوید).

ورودی نمونه

```
public class Main {
    public static void main(String[] args) {

        BankAccount acc1 = new BankAccount("ACC1001", 500.0, "1234");
        SavingAccount acc2 = new SavingAccount("ACC2002", 1000.0, "5678", 200
        SpecialSavingAccount acc3 = new SpecialSavingAccount("ACC3003", 1500.
        ATMDevice atm = new ATMDevice();

        System.out.println("=== Accounts Information ===");
        System.out.println(acc1.getAccountInfo());
        System.out.println(acc2.getAccountInfo());
        System.out.println(acc3.getAccountInfo());

        System.out.println("\n=== Deposit / Withdraw Tests ===");
        System.out.println("Deposit success? " + acc1.deposit(300));
```

```
        System.out.println("Withdraw success? " + acc1.withdraw(200));
        System.out.println("Saving withdraw(900): " + acc2.withdraw(900));  /
        System.out.println("Saving canWithdraw(300)? " + acc2.canWithdraw(300

        System.out.println("\n=== Special Saving Account Tests ===");
        System.out.println("Bonus Rate: " + acc3.getBonusRate() + "%");
        System.out.println("Preview Bonus: " + acc3.previewAnnualBonus());
        System.out.println("Bonus Applied (amount): " + acc3.applyAnnualBonus
        System.out.println("New Balance after Bonus: " + acc3.getBalance());

        System.out.println("\n=== ATM Operations ===");
        boolean t1 = ATMDevice.transfer(acc1, "1234", acc2, 150);
        System.out.println("Transfer 150 from acc1 → acc2 : " + t1);
        boolean t2 = ATMDevice.transfer(acc2, "5678", acc3, 100);
        System.out.println("Transfer 100 from acc2 → acc3 : " + t2);

        acc1.closeAccount();
        boolean t3 = ATMDevice.transfer(acc1, "1234", acc3, 50);
        System.out.println("Transfer 50 from closed acc1 → acc3 : " + t3);

        System.out.println("\n=== Quick Withdraw Demo ===");
        boolean q1 = atm.quickWithdraw(acc3, "2468", 50);
        System.out.println("Quick Withdraw(50) from acc3: " + q1);
        System.out.println("Balance acc3: " + atm.checkBalance(acc3));

        System.out.println("\n=== Reset PIN Tests ===");
        System.out.println("Reset PIN valid (old=1234,new=9999): " + acc1.res
        System.out.println("Reset PIN invalid (old=wrong,new=0000): " + acc1.

    }
}
```

خروجی نمونه

```
=== Accounts Information ===
Account: ACC1001 | Balance: 500.0 | Status: Active
Account: ACC2002 | Balance: 1000.0 | Status: Active
Account: ACC3003 | Balance: 1500.0 | Status: Active

=== Deposit / Withdraw Tests ===
Deposit success? true

```

```
Withdraw success? true
Saving withdraw(900): false
Saving canWithdraw(300)? true

=== Special Saving Account Tests ===
Bonus Rate: 5.0%
Preview Bonus: 75.0
Bonus Applied (amount): 75.0
New Balance after Bonus: 1575.0

=== ATM Operations ===
Transfer 150 from acc1 → acc2 : true
Transfer 100 from acc2 → acc3 : true
Transfer 50 from closed acc1 → acc3 : false

=== Quick Withdraw Demo ===
Quick Withdraw(50) from acc3: true
Balance acc3: 1625.0

=== Reset PIN Tests ===
Reset PIN valid (old=1234,new=9999): true
Reset PIN invalid (old=wrong,new=0000): false
```

نمایندگی

- سطح: سخت
- طراح: سید محمد حسینی

آقا اسکندر که تازه از شغل قبلی خود بیرون آمده بود تصمیم گرفت تا نمایندگی فروش لپتاپ و گوشی موبایل بزند و از این راه کسب درآمد کند. به همین دلیل او به یک سیستم هوشمند نیاز دارد تا مدیریت این نمایندگی را با آن سیستم جلو ببرد. به همین دلیل او به شما نیاز دارد تا اینکار را برای او انجام دهید و برنامه مدیریت فروشگاه را برای او بنویسید(پیشاپیش از طرف اسکندر بابت این لطف شما سپاس عرض می‌کنم).

توجه: برای تمام کلاس‌ها getter,setter های مرتبط با هر فیلد را تعریف کنید.

ابتدا پروژه اولیه را از این لینک دانلود نمایید.

کلاس Phone:

فیلدها

این کلاس دارای فیلدهای زیر است:

```
1 private int price;
2 private ModelOfPhone model;
```

- `price` : قیمت آن گوشی موبایل.
- `model` : مدل آن گوشی که یکی از گزینه‌های `enum ModelOfPhone` است.

کانستراکتور

کانستراکتور این کلاس به شکل زیر است:

```
1 public Phone(int price, ModelOfPhone model)
```

در این سازنده قیمت موبایل و مدل آن تعیین می‌شود. دقت کنید که این کلاس صرفا همین سازنده را دارد.

در سازنده کلاس موبایل از یک **enum** به‌نام `ModelOfPhone` استفاده شده‌است که به شکل زیر تعریف می‌شود:

```
public enum ModelOfPhone {
    Samsung, IPhone, Xiaomi
}
```

در اینجا مدل‌های مختلف گوشی موبایل که در دسترس هستند نشان داده شده‌است که در سازنده یکی از این مقادیر را انتخاب می‌کنیم.

کلاس Laptop:

فیلدها

این کلاس دارای فیلدهای زیر است:

```
private int price;
private ModelOfLaptop model;
```

- `price` : قیمت آن شی لپتاپ.
- `model` : مدل آن لپتاپ که یکی از گزینه‌های `enum ModelOfLaptop` است.

کانستراکتور

کانستراکتور این کلاس به شکل زیر است:

```
public Laptop(int price,ModelOfLaptop model)
```

در این سازنده قیمت لپتاپ و مدل آن تعیین می‌شود. دقت کنید که این کلاس صرفا همین سازنده را دارد.

در سازنده کلاس لپتاپ از یک **enum** به‌نام `ModelOfLaptop` استفاده شده‌است که به شکل زیر تعریف

می‌شود:

```
public enum ModelOfLaptop {
    Asus, Lenovo, MacBook
}
```

در اینجا مدل‌های مختلف لپتاپ که در دسترس هستند نشان داده شده‌است که در سازنده یکی از این مقادیر را انتخاب می‌کنیم.

کلاس Employee:

فیلدها

این کلاس دارای فیلدهای زیر است:

```
private String name;
private Degree degree;
private int netWorth = 0;
private boolean isShift = false;
private boolean isFire = false;
```

- `name` : نام کارمند استخدام شده (یکتا است و کارمندی با نام تکراری استخدام نمی‌شود).
- `degree` : درجه آن کارمند که یک نوع `enum` است و جلوتر آن‌را خواهیم دید .
- `netWorth` : دارایی آن کارمند است که کارمند می‌تواند از این مقدار عددی را بردارد و آن را خرج کند یا در حالت دیگر مقداری پول (حاصل درآمد یا موارد دیگر) به آن اضافه کند. مانند موجودی حساب کارمند می‌ماند.
- `isShift` : آیا کارمند در حال حاضر شیفت است یا خیر.
- `isFire` : وضعیت کارمند که اخراج شده‌است یا خیر.

کانستراکتور

کانستراکتور این کلاس به شکل زیر است:

```
public Employee(String name, Degree degree)
```

در این سازنده صرفا نام آن کارمند به همراه درجه آن کارمند که از `enum Degree` می‌آید، مشخص می‌شود. این `enum` را جلوتر بررسی خواهیم کرد.

در سازنده کلاس کارمند از یک **enum** به‌نام `Degree` استفاده شده‌است که به شکل زیر تعریف می‌شود:

```
public enum Degree {
   Fire ,Worker ,Senior ,Leader
}
```

در اینجا سلسله مراتب این نمایندگی نشان داده شده‌است. همانطور که مشاهده می‌کنید، کارمندان می‌توانند اخراج شوند یا داخل خود شرکت یکی از درجات Worker, Senior,Leader داشته باشند.

متدها

متدهای این کلاس به صورت زیر است:

```
public void depositSalary(int value)
```

این متد برای این است که کارمند بتواند به حساب خود پولی واریز کند. اگر این مقدار مثبت باشند آنگاه مقدار آن با مقدار فعلی حساب جمع شده و مقدار جدید به عنوان موجودی حال حاضر آن کارمند شناخته می‌شود.

```
public boolean withdraw(int value)
```

این متد برای این است که کارمند بتواند از حساب خود پولی برداشت کند. اگر این مقدار بیشتر از موجودی کارمند باشد یا مقدارش منفی باشد نباید کاری انجام شود و صرفا این متد باید مقدار `false` را برگرداند. در غیر اینصورت این مقدار از حساب کارمند کسر شده و مقدار `true` برگردانده می‌شود.

```
public boolean isShift()
```

این متد وضعیت شیفت بودن یا نبودن یک کارمند را بر می‌گرداند. اگر آن کارمند شیفت بود مقدار `true` و در غیر اینصورت مقدار `false` را بر می‌گرداند.

```
public boolean isFire()
```

این متد چک می‌کند که آیا یک کارمند اخراج شده‌است یا خیر. اگر اخراج شده‌بود مقدار `true` و در غیر اینصورت مقدار `false` را بر می‌گرداند.

کلاس Product:

فیلدها

این کلاس دارای فیلدهای زیر است:

```
private final String productId;
private final ProductType type;
private final Phone phone;
private final Laptop laptop;
private int stockQuantity;
```

- `productId` : هر محصول (چه گوشی موبایل چه لپتاپ) برای اینکه در انبار فروشگاه قابل شناسایی باشد، یک آیدی منحصر به فرد دارد.
- `type` : نوع این کالا را نشان می‌دهد که از جنس یک `enum` است و جلوتر این `enum` را معرفی خواهیم کرد.
- `phone` : اگر این محصول موبایل باشد آنگاه یک شی به این فیلد اختصاص داده خواهد شد. در غیر اینصورت مقدار `null` می‌گیرد.
- `laptop` : اگر این محصول لپتاپ باشد آنگاه یک شی به این فیلد اختصاص داده خواهد شد. در غیر اینصورت مقدار `null` می‌گیرد.
- `stockQuantity` : مقدار این محصول در فروشگاه را نشان می‌دهد. در واقع این فیلد نشان می‌دهد که از یک محصول (یک مدل موبایل) چه مقدار داخل فروشگاه موجود است.

کانستراکتور

این کلاس دو سازنده دارد. یکی از این سازنده‌ها برای این است که محصول موبایل باشد و دیگری برای این است که محصول مورد نظر لپتاپ باشد.

```
public Product(String productId, Phone phone, int stockQuantity)
```

در این سازنده آیدی آن محصول، یک شی موبایل و تعداد آن در فروشگاه مقداردهی می‌شود. از این سازنده برای زمانی استفاده می‌شود که محصول جدید ما یک گوشی موبایل باشد.

توجه: دقت کنید که مقدار `type` در این سازنده برابر `ProductType.PHONE` قرار می‌گیرد.

نکته: الان فیلد `phone` مقدار دهی شده‌است و یک شی به آن نسبت داده شده‌است. در حال حاضر مقدار فیلد `laptop` چه باید باشد؟ (در توضیحات قبلی گفته شده که مقداردهی این فیلدها به چه صورت باید انجام بگیرد).

```
public Product(String productId, Laptop laptop, int stockQuantity)
```

این سازنده، سازنده دوم این کلاس است و برای زمانی است که محصول انتخابی ما یک لپتاپ باشد.

نکته: الان فیلد `laptop` مقدار دهی شده‌است و یک شی به آن نسبت داده شده‌است. در حال حاضر مقدار فیلد `phone` چه باید باشد؟ (در توضیحات قبلی گفته شده که مقداردهی این فیلدها به چه صورت باید انجام بگیرد).

در سازنده کلاس محصولات از یک **enum** به‌نام `ProductType` استفاده شده‌است که به شکل زیر تعریف می‌شود:

```
public enum ProductType {
   LAPTOP, PHONE
}
```

در اینجا به دلیل اینکه محصولات ما می‌توانند دو نوع باشند(موبایل یا لپتاپ) بنابراین این `enum` را تعریف کردیم تا در سازنده این کلاس مشخص شود از کدام نوع محصول وارد فروشگاه کرده‌ایم.

متدها

متدهای این کلاس به صورت زیر است:

```
public void addStock(int quantity)
```

این متد برای این است که تعداد محصولات را اضافه کنیم.

```
public boolean sellProduct(int quantity)
```

این متد برای این است که یک محصول را بفروشیم. اگر تعداد درخواستی از تعداد موجود برای محصول مورد نظر بیشتر باشد این متد مقدار `false` را برمی‌گرداند. در غیر اینصورت این تعداد را از تعداد موجود در فروشگاه کم کرده و سپس مقدار `true` را بر می‌گرداند.

نکته: علیرغم اینکه این کلاس فیلد قیمت (`price`) را ندارد اما متد `getPrice` را دارد و باید آنرا هم پیاده‌سازی کنید.

کلاس Customer:

فیلدها

این کلاس دارای فیلدهای زیر است:

```
private String name;
private Product[] savedProduct = new Product[10];
private int numOfSaveProduct = 0;
private Product[] purchasedProduct = new Product[10];
private int numOfPurchaseProduct = 0;
private int amountOfMoney = 0;
```

- `name` : هر مشتری یک اسم دارد.
- `savedProduct` : هر مشتری می‌تواند یک‌سری محصولات را ذخیره کند که بعدا آنها را بخرد(ویژگی که شما داخل فروشگاه‌های آنلاین هم دیده‌اید و می‌توانید محصولاتی را دخیره کنید که بعدا راحت‌تر

به آن محصول دسترسی داشته باشید.)

- `numOfSaveProduct` : تعداد محصولاتی که در حال حاضر یک مشتری ذخیره کرده‌است.
- `purchasedProduct` : آرایه‌ای است که لیست محصولاتی که مشتری خریده‌است را ذخیره می‌کند.
- `numOfPurchasedProduct` : تعداد محصولاتی که کاربر خریده‌است.
- `amountOfMoney` : مقدار پولی که مشتری با خود به همراه دارد تا با آن محصولاتی را بخرد.

توجه: اندازه آرایه‌هایی که فیلد این متد هستند به صورت پیش‌فرض مقدار 10 داده شده‌اند و در روند پیاده‌سازی این کلاس اگر این آرایه‌ها پر شدند، میتوانید اندازه آرایه را به روش دلخواه خودتان افزایش دهید.

نکته: مشتری نمی‌تواند برای همیشه محصولات را در لیست ذخیره شده خود نگه دارد و محدودیت دارد. اگر 5 چرخه کار انجام شود و محصولات ذخیره شده را نخرد یا خود مشتری آنرا از لیست ذخیره شده خود حذف نکند، به صورت خودکار آن محصولی که در طول 5 چرخه در لیست ذخیره شده بود، از این لیست حذف خواهد شد. توضیحات بیشتر درباره چرخه جلوتر بیان خواهد شد.

کانستراکتور

سازنده این کلاس به صورت زیر است:

```
1 public Customer(String name, int amountOfMoney)
```

در این سازنده اسم مشتری و مقدار پولی که آن مشتری به همراه خود دارد را مقداردهی می‌کنیم.

متدها

متدهای این کلاس به صورت زیر است:

```
1 public boolean saveProduct(Product product)
```

این متد برای این است که محصولی به لیست محصولات ذخیره‌شده مشتری اضافه شود. اگر این محصول قبلا جز محصولات ذخیره‌شده در لیست محصولات ذخیره‌شده مشتری بوده‌است باید این متد مقدار `false` را برگرداند. در غیر اینصورت آن محصول جز محصولات ذخیره‌شده مشتری ذخیره می‌شود و مقدار

`true` را برمی‌گرداند.

```
1 | public boolean unSaveProduct(Product product)
```

این متد برای این است که یک محصولی که ذخیره شده‌است را از لیست ذخیره‌شده‌ها حذف کنیم. اگر این محصول داخل آرایه موردنظر بود، آنرا از لیست حذف کرده و مقدار `true` را برمی‌گرداند. در غیر اینصورت مقدار `false` را بر می‌گرداند.

نکته: توجه کنید این حذف محصول روی تعداد محصولات ذخیره شده هم اثر دارد. محصول موردنظر هم باید به طور کامل از آرایه مورد نظر حذف شود اما شیوه حذف کردن محصول به عهده خودتان است.

```
1 | public void printSavedProducts()
```

این متد برای نمایش محصولات ذخیره شده‌است. نحوه پیاده‌سازی این متد را از طریق اجرا گرفتن مثال نمونه متوجه خواهید شد.

```
1 | public void printPurchasedProduct()
```

این متد برای نمایش محصولاتی است که مشتری خریده است. نحوه پیاده‌سازی این متد را از طریق اجرا گرفتن مثال نمونه متوجه خواهید شد.

```
1 | public boolean purchaseProduct(Product product)
```

این متد برای زمانی است که مشتری می‌خواهد محصولی را بخرد. اگر این محصول را قبلا خریده باشد این متد مقدار `false` را برمی‌گرداند. اگر قیمت این محصول بیشتر از مقدار پولی باشد که مشتری همراه خود دارد باز هم اتفاقی نمی‌افتد و مقدار `false` برگردانده می‌شود. اگر تعداد این محصول کمتر از 1 باشد کاربر نمی‌تواند آنرا بخرد چون فروشگاه این محصول را موجود ندارد بنابراین در این حالت هم مقدار `false` را برمی‌گرداند. اگر این محصول که کاربر می‌خواهد بخرد داخل لیست محصولات ذخیره شده باشد آنگاه هنگام خرید از این لیست حذف می‌شود.

توجه: برای خرید هر محصول باید مبلغی پرداخت شود !!

```
1 | public boolean isProductSaved(Product product)
```

این متد برای این است که چک بشود آیا کالایی را یک مشتری ذخیره کرده است یا خیر. اگر آن کالا را ذخیره کرده بود مقدار `true` و اگر کالا را مشتری ذخیره نکرده بود مقدار `false` را برگرداند.

```
1 | public boolean isPurchased(Product product)
```

این متد چک می‌کند که آیا یک کالا را مشتری خریده‌است یا خیر. اگر این کالا از دسته کالاهایی بود که مشتری خریداری کرده‌بود مقدار `true` و در غیر اینصورت مقدار `false` را بر می‌گرداند.

کلاس ElectronicsStore:

فیلدها

این کلاس دارای فیلدهای زیر است:

```
1 | private Employee[] employees = new Employee[6];
2 | private Customer[] customers = new Customer[20];
3 | private Product[] products = new Product[50];
4 | private int numOfEmployee=0;
5 | private int numOfCustomers =0;
6 | private int numOfProducts = 0;
```

- `employees` : لیست کارمندان این فروشگاه.
- `customers` : لیست مشتریان این فروشگاه.
- `products` : محصولاتی که در این فروشگاه موجود می‌باشد.
- `numOfEmployee` : تعداد کارمندان.
- `numOfCustomers` : تعداد مشتریان.
- `numOfProducts` : تعداد کالاهای فروشگاه.

توجه: دقت کنید که هر فروشگاه تعداد ثابتی مشتری و کارمند می‌تواند داشته باشد و به تعداد دلخواه این امر میسر نمی‌باشد (حداکثر تعداد مجاز کارمندان 6 عدد، مشتریان 20 نفر و حداکثر تعداد کالا 50 عدد می‌باشد).

کانستراکتور

این کلاس سازنده‌ای ندارد و هنگام استفاده از این کلاس برای نمونه‌سازی سازنده پیش‌فرض آن فراخوانی می‌شود.

متدها

متدهای این کلاس به صورت زیر است:

```
1 | public boolean addEmployee(Employee employee)
```

این متد برای این است که کارمندی را استخدام کند. اگر مقدار داده شده `null` باشد یا تعداد کارمندان به حد نصاب رسیده باشد این تابع مقدار `false` بر می‌گرداند اما در غیر اینصورت کارمند را اضافه کرده و مقدار `true` را بر می‌گرداند.

```
1 | public boolean removeEmployee(Employee employee)
```

این متد برای حذف کارمند از لیست کارمندان فروشگاه است. اگر کارمند از لیست کارمندان فروشگاه پیدا نشود مقدار `false` را بر می‌گرداند در غیر اینصورت کارمند را از لیست کارمندان شرکت حذف می‌کند و مقدار `true` را برمی‌گرداند.

```
1 | public boolean addCustomer(Customer customer)
```

این متد برای افزودن مشتری به فروشگاه استفاده می‌شود. شروطی که برای اضافه کردن کارمند وجود داشت اینجا هم صادق است.

```
public boolean removeCustomer(Customer customer)
```

این متد برای حذف مشتری از فروشگاه است و قواعدی که برای کارمندان فروشگاه بیان شد اینجا هم صادق است.

```
public boolean addProduct(Product product)
```

این متد برای اضافه کردن محصول به فروشگاه است.

توجه: اضافه کردن محصول به فروشگاه یک چرخه حساب می‌شود و چه محصول با موفقیت اضافه شود چه نتوانیم محصول را به فروشگاه اضافه کنیم یک چرخه مصرف می‌شود. به این معنی که محصولاتی که در حال حاضر در لیست محصولات ذخیره‌شده مشتریان این فروشگاه هستند یک چرخه به چرخه فعلیشان اضافه می‌شود و اگر تعداد چرخه برای یک محصول ذخیره شده برای یک مشتری به عدد 5 برسد آن محصول به طور خودکار از لیست محصولات ذخیره شده مشتری حذف می‌شود.

اگر تعداد محصولات به حداکثر رسیده باشد این متد `false` برمی‌گرداند و اگر محصول با موفقیت به لیست محصولات اضافه شد این متد مقدار `true` را برمی‌گرداند.

```
public boolean removeProduct(Product product)
```

این متد برای حذف کردن محصول از فروشگاه است.

توجه: اگر محصول وجود داشته باشد که بتوانیم آنرا حذف کنیم بعد از حذف این متد مقدار `true` را برمی‌گرداند در غیر اینصورت این متد اگر نتواند محصول موردنظر را حذف کند مقدار `false` را بر می‌گرداند.

```
public boolean assignEmployeeToShift(String employeeName)
```

این متد برای این است که وضعیت شیفت کارمندان را تغییر دهد. اگر کارمندی شیفت نباشد آنرا به سر شیفت می‌آورد و اگر کارمند شیفت باشد آنگاه آنرا از شیفت به حالت استراحت تبدیل می‌کند.

```java
public boolean customerBuysProduct(String customerName, String productID)
```

این متد برای این است که یک مشتری یک محصول را می‌خرد.

در این متد یک چرخه مصرف می‌شود و تمام محصولاتی که در سبد ذخیره‌شده مشتریان این فروشگاه هستند یک چرخه به تعداد چرخه آنها اضافه می‌شود تا در صورت نیاز محصولاتی از سبد ذخیره شده مشتریان حذف شوند.

نکته: اگر محصول یا مشتری یافت نشد این متد مقدار `false` را بر می‌گرداند. در غیر اینصورت محصول را برای مشتری خریداری می‌کند و از مقدار موجود آن محصول یکی کم می‌کند و در آخر هم این متد مقدار `true` را بر می‌گرداند.

```java
public void handleProductTheft(String productID)
```

این متد برای زمانی است که محصولی از داخل فروشگاه دزدیده می‌شود. اگر محصولی از فروشگاه دزدیده شود تمام کارمندانی که شیفت بوده‌اند تنبیه خواهند شد به این صورت که مقدار دو برابر محصول دزدیده شده از موجودی حساب آنها کم خواهد شد. اگر این مقدار پول در حساب آنها نباشد آنها تنزل مقام پیدا خواهند کرد و اگر کارمندی در درجه `Worker` باشد اخراج خواهد شد.

توجه: این دزدیده شدن محصول باعث می‌شود تا یک عدد از تعداد محصولات دزدیده شده کم شود. همچنین آن محصول از لیست محصولات فروشگاه حذف خواهد شد چون دزدیده شده‌است. به علاوه اگر هر یک از مشتریان این محصول را در لیست ذخیره شده خود داشته باشند از این لیست هم حذف خواهد شد چون فروشگاه دیگر این محصول را ندارد. در نهایت هم این فرآیند دزدیده شدن محصول یک چرخه برای کالای ذخیره شده توسط مشتریان است.

با اجزای متد `main` زیر:

```java
    public static void main(String[] args) {
        ElectronicsStore store = new ElectronicsStore();

        Employee e1 = new Employee("Ali", Degree.Worker);
        Employee e2 = new Employee("Sara", Degree.Leader);

```

```
Customer c1 = new Customer("Reza", 3000);
Customer c2 = new Customer("Neda", 1500);

Phone p1Phone = new Phone(1000, ModelOfPhone.Samsung);
Phone p2Phone = new Phone(800, ModelOfPhone.Xiaomi);
Laptop l1Laptop = new Laptop(2000, ModelOfLaptop.Asus);

Product p1 = new Product("P100", p1Phone, 3);
Product p2 = new Product("P200", p2Phone, 5);
Product p3 = new Product("L300", l1Laptop, 2);

store.addEmployee(e1);
store.addEmployee(e2);

store.addCustomer(c1);
store.addCustomer(c2);


store.addProduct(p1);
store.addProduct(p2);
store.addProduct(p3);


store.assignEmployeeToShift("Ali");
store.assignEmployeeToShift("Sara");


c1.saveProduct(p1);
c1.saveProduct(p2);
c2.saveProduct(p3);

c1.printSavedProducts();
c2.printSavedProducts();


store.customerBuysProduct("Reza", "P100");
store.customerBuysProduct("Neda", "L300");

c1.printPurchasedProduct();
c2.printPurchasedProduct();


c1.unSaveProduct(p2);
```

```
        c1.printSavedProducts();

        store.handleProductTheft("P200");

        store.removeEmployee(e1);

        store.removeCustomer(c2);

        store.removeProduct(p3);

    }
```

باید خروجی زیر تولید شود:

```
Saved products:
1- :P100 1000
2- :P200 800
Saved products:
1- :L300 2000
Purchased products:
1- :P100 1000
No purchased product.
No saved products.
```

آنچه باید آپلود کنید

ساختار فایل زیپ ارسالی باید به صورت زیر باشد:

```
<zip_file_name.zip>
    ├── Laptop.java
    ├── Phone.java
    ├── ModelOfPhone.java
    ├── ModelOfLaptop.java
    ├── Employee.java
```

```
├── Degree.java
├── Customer.java
├── Product.java
├── ProductType.java
└── ElectronicsStore.java
```

Empire Duel(امتیازی)

- سطح: سخت
- طراح: روزبه سلطانی

در این پروژه باید یک بازی دو نفره‌ی استراتژیک با برنامه‌نویسی شیءگرا (OOP) طراحی کنید. دو بازیکن با جمع‌آوری طلا (**gold**) و تربیت سرباز (**Soldier**) برای تخریب ساختمان‌های دشمن با هم رقابت می‌کنند. بازی نوبتی است و در هر نوبت بازیکن فقط می‌تواند یک عمل انجام دهد، ولی ساخت سرباز یا ساختمان نوبت مصرف نمی‌کند.

در این برنامه تضمین میشود اندازه **100** برای آرایه ها کافی باشد.

پروژه اولیه را از این لینک دانلود بفرمایید.

قوانین بازی:

۱. دو بازیکن **(Player 1 و Player 2)** حضور دارند.
۲. بازی توسط کلاس **GameEngine** کنترل می‌شود که نوبت‌ها و شرایط پایان بازی را مدیریت می‌کند.
۳. هر بازیکن یک نوع منبع اصلی دارد: **gold**.
۴. در هر نوبت، بازیکن می‌تواند یکی از اقدامات زیر را انجام دهد:
   - جمع‌آوری طلا: `gatherGold()` یا
   - حمله به ساختمان‌های حریف: `attack()`
۵. ساخت سرباز (**trainSoldier**) و ساخت ساختمان (**addBuilding**) نوبت مصرف نمی‌کند.
۶. **VIP Player** طلا بیشتری جمع‌آوری می‌کند: معمولی = 50 واحد، VIP = 70 واحد.
۷. بازیکن آرایه‌ای از ساختمان‌ها دارد. اگر تمام ساختمان‌ها نابود شوند => بازنده است.

سیستم نبرد:

۱. هر سرباز (**Soldier**) فقط یک ویژگی عددی دارد: **power** (سلامتی و قدرت حمله).
۲. سربازها به ترتیب آرایه **Army** و از آخر به آرایه ساختمان‌های دشمن از آخر دمیج وارد می‌کنند.
۳. دمیج سرباز روی ساختمان‌های دشمن اعمال می‌شود. اگر باقی‌مانده دمیج وجود داشته باشد به

ساختمان بعدی منتقل می‌شود.

۴. نابودی کامل ساختمان‌های دشمن => بازیکن مهاجم برنده.

ساختار کلاس‌ها:

1. کلاس Soldier

نمایانگر **یک سرباز** در بازی است.

```
class Soldier {
    private int power;

    public Soldier()
    public boolean isAlive()
    public void reducePower(int amount)
    public int getPower()
}
```

ویژگی‌ها:

- ویژگی **power**: عددی که مشخص می‌کند سرباز چقدر می‌تواند دمیج وارد کند و چقدر می‌تواند زنده بماند.
- مقدار پیش فرض آن **30** است و در سازنده آن مقدار دهی میشود.
- وقتی سرباز به ساختمان دشمن حمله می‌کند، این مقدار **مستقیماً به عنوان دمیج اعمال می‌شود** و مقدار دمیج وارد شده از پاور آن سرباز کم میشود برای مثال : فرض کنیم سرباز `power = 30` دارد و دشمن دو ساختمان دارد:

ساختمان A: 25 سلامتی

ساختمان B: 20 سلامتی

حمله انجام می‌شود:

۱. سرباز به ساختمان A حمله می‌کند -> 25 دمیج به A وارد می‌شود -> A نابود می‌شود (25 واحد از پاور سرباز کم میشود)

۲. 5 دمیج باقی‌مانده -> به ساختمان B اعمال می‌شود -> B = 15 باقی می‌ماند

۳. سرباز حذف می‌شود.

متدها:

- متد `isAlive` بررسی میکند آیا ویژگی `power` بزرگ تر از 0 است یا خیر.
- متد `reducePower(int amount)` مقدار مشخص شده را از `power` کم میکند و اگر منفی شد آن را 0 میکند.

2. کلاس `Building`

نمایانگر یک **ساختمان** متعلق به بازیکن است که سربازها باید آن را نابود کنند.

```
class Building {
    private int health;

    public Building()
    public void takeDamage(int damage)
    public boolean isDestroyed()
    public int getHealth()
}
```

ویژگی‌ها:

- ویژگی `health` : سلامتی ساختمان را نشان میدهد که مقدار اولیه آن **200** میباشد.

متدها:

- متد `void takeDamage(int damage)`
  - مقدار `damage` را از `health` کم می‌کند.
  - اگر `health` منفی شود، باید آن را صفر کرد.
- متد `boolean isDestroyed()`
  - بررسی می‌کند آیا `health == 0` است؟
  - اگر بله باید true برگرداند (ساختمان نابود شده)
- متد `int getHealth()`

- برای گرفتن مقدار فعلی سلامتی

3. کلاس `Army` (مدیریت سربازان هر بازیکن)

مدیریت **آرایه‌ای از سربازها (Soldier)** برای یک بازیکن. این کلاس مسئول اضافه کردن سرباز جدید و ارائه‌ی اطلاعات مربوط به سربازهاست.

```
class Army {
    private Soldier[] soldiers;
    private int soldierCount;

    public Army()
    public void addSoldier(Soldier s)
    public Soldier[] getSoldiers()
    public int getSoldierCount()
    public void setSoldierCount(int soldierCount)
}
```

ویژگی‌ها:

- ویژگی `soldiers` : آرایه‌ای با اندازه ثابت **100** که تمام سربازهای یک بازیکن را نگه می‌دارد.
- ویژگی `soldierCount` : تعداد واقعی سربازهایی که در آرایه موجود هستند.

متدها:

- متد `addSoldier` سرباز را به آرایه سربازها اضافه میکند و `soldierCount` یک واحد زیاد میکند.
- `Soldier[] getSoldiers()` آرایه سربازهای فعلی را برمی‌گرداند.
- `int getSoldierCount()` تعداد واقعی سربازها را برمی‌گرداند.

4. کلاس `ResourceManager` (مدیریت منابع بازیکن)

وظیفه آن مدیریت **طلا بازیکن** (**gold**) و جمع‌آوری یا مصرف آن است. این کلاس مشخص می‌کند که بازیکن چقدر طلا دارد، چه مقدار می‌تواند جمع‌آوری کند و آیا می‌تواند سرباز بسازد یا مصرف کند.

```java
class ResourceManager {
    private int gold;
    private boolean isVIP;

    public ResourceManager(boolean isVIP)
    public void gatherGold()
    public boolean canUseResources(int goldCost)
    public void useResources(int goldCost)
    public int getGold()
}
```

ویژگی‌ها:

- مقدار اولیه `gold` برابر با **0** میباشد.

متدها:

- متد `gatherGold()` : مقدار **gold** را افزایش پیدا می‌دهد: اگر پلیر Vip باشد 70 واحد و در غیر این صورت 50 واحد افزایش میدهد.
- متد `canUseResources(int goldCost)` بررسی می‌کند که آیا بازیکن **طلا کافی** برای ساخت سرباز یا دیگر عملیات دارد یا خیر.
- متد `useResources(int goldCost)` اگر طلا کافی باشد، مقدار داده شده را از `gold` کم میکند.

5. کلاس `Player`

وظیفه آن نمایش یک بازیکن در بازی و مدیریت **منابع (ResourceManager)**، **ارتش (Army)** و **ساختمان‌ها (Buildings)** است. کلاس Player متدهای اصلی برای جمع‌آوری منابع و ساخت سرباز دارد.

```java
class Player {
    private ResourceManager resourceManager;
    private Army army;
    private Building[] buildings;
    private int buildingCount;

    public Player(boolean isVIP)
    public void gatherGold()
```

```
    public void trainSoldier(int count)
    public void attack(Player opponent)
    public boolean hasLost()
    public void addBuilding(int count)
    public Army getArmy()
    public Building[] getBuildings()
    public int getBuildingCount()
    public ResourceManager getResourceManager()
    public void decrementBuildingCount()
    public void decrementSoldierCount()

}
```

ویژگی‌ها:

- ویژگی **resourceManager**: تنها طلا را نگه می‌دارد و عملیات جمع‌آوری و مصرف طلا را مدیریت می‌کند.
- ویژگی **army**: ارتش بازیکن، شامل آرایه سربازهاست.
- ویژگی **buildings**: آرایه ساختمان‌ها، تعداد ثابت **100**.
- ویژگی **buildingCount**: تعداد واقعی ساختمان‌های باقی‌مانده.

متدها:

- سازنده `Player` :

۱. مقدار اولیه `ResourceManager` ساخته می‌شود.

۲. `Army` ایجاد می‌شود.

۳. آرایه ساختمان‌ها ساخته می‌شود (اندازه ثابت 100) و اولین ساختمان به آن اضافه میشود(هر بازیکن در ابتدا به صورت پیشفرض یک ساختمان دارد).

- `void gatherGold()`
  - صدا زدن متد `resourceManager.gatherGold()`
  - منابع طلا بازیکن افزایش می‌یابد.
  - نوبت توسط `GameEngine` که پایین تر توضیح داده شده کنترل می‌شود و نیازی به بررسی نوبت در این متد نیست.

- `void trainSoldier(int count)`
  - بازیکن تعداد مشخصی سرباز می‌سازد.
  - قدرت هر سرباز **30** است .
  - متد بررسی می‌کند که **طلا کافی برای ساخت تعداد مورد نظر سرباز وجود دارد یا خیر**.
    - هزینه ساخت هر سرباز **20** طلا است.
    - اگر منابع کافی باشد، سربازها به ارتش (army) اضافه می‌شوند و طلا مصرف می‌شود.
    - اگر منابع کافی نباشد، عملیات رد می‌شود.
    -
    - توجه داشته باشید ساخت سرباز **نوبت** بازیکن را مصرف **نمیکند**.

- متد `attack(Player opponent)`
  - نمایانگر **حمله یک بازیکن به بازیکن دیگر** است و دمیج سربازها را به ساختمان‌های دشمن اعمال می‌کند.

رفتار و قوانین:

۱. **شروع حمله:**
   - بازیکن مهاجم (`this`) آرایه **سربازهای خودش** را بررسی می‌کند.
   - دشمن (`opponent`) آرایه **ساختمان‌هایش** را بررسی می‌کند.
   - حمله از **آخرین سرباز آرایه** به **آخرین ساختمان آرایه دشمن** شروع می‌شود.

۲. **دمیج سرباز به ساختمان:**
   - هر سرباز قدرت (`power`) مشخصی دارد.
   - سرباز به ساختمان آخر دشمن حمله می‌کند.
   - اگر `power >= building.health` :
     - تمام سلامت ساختمان کم می‌شود و ساختمان نابود می‌شود (`health = 0`).
     - باقی‌مانده دمیج سرباز (`power -= building.health`) به ساختمان بعدی منتقل می‌شود.
   - اگر `power < building.health` :
     - سلامت ساختمان کاهش می‌یابد (`health -= power`).
     - سرباز تمام قدرتش را از دست می‌دهد (`power = 0`) و از ارتش حذف می‌شود.

۳. **حذف سربازان و ساختمان‌های نابود شده:**

- هر سرباز که قدرتش صفر میشود از ارتش حذف می‌شود و (`soldierCount`) یک واحد کاهش می یابد.
- هر ساختمان که سلامتش صفر میشود از ساختمان ها حذف می‌شود و (`buildingCount`) یک واحد کاهش می یابد.

۴. **حمله به ساختمان بعدی:**

- اگر دمیج سرباز هنوز باقی مانده باشد، به ساختمان قبلی در آرایه (ساختمان بعدی در جهت حمله) منتقل می‌شود.
- این روند تا وقتی که:
  - هیچ سربازی از سمت حمله کننده زنده نماند. یا
  - هیچ ساختمان زنده‌ای باقی نماند، ادامه پیدا می‌کند.

۵. **نتیجه حمله:**

- اگر تمام ساختمان‌های حریف نابود شود => بازیکن مهاجم برنده است.
- اگر هیچ ساختمان زنده‌ای باقی نماند => بازیکن حریف باخته است.
- این بخش توسط `GameEngine` مدیریت میشود.

- `boolean hasLost()`
  - بررسی می‌کند که آیا **تمام ساختمان‌ها نابود شده‌اند یا خیر**.
  - در `hasLost` باید با حلقه تمام ساختمان‌ها را پیمایش کرد و اگر همه‌ی آن‌ها `isDestroyed()` بودند مقدار true برگرداند. بررسی صرفاً بر اساس `buildingCount` کافی نیست!
- `void addBuilding()`
- ۱. بازیکن تعداد مشخصی ساختمان می‌سازد.

۲. هزینه ساخت هر ساختمان **70 طلا** است.

۳. متد بررسی می‌کند که آیا **بازیکن طلا کافی برای ساخت همه ساختمان‌ها دارد یا خیر**.

- اگر منابع کافی نباشد، **هیچ ساختمانی ساخته نمی‌شود**.
- اگر منابع کافی باشد، **تمام ساختمان‌ها ساخته شده** و طلا به مقدار لازم از منابع بازیکن کسر می‌شود.

۴. ساخت ساختمان **نوبت بازیکن را مصرف نمی‌کند**.

- `void decrementBuildingCount()`
  - تعداد ساختمان‌های بازیکن را **یک واحد کاهش می‌دهد**، زمانی که یک ساختمان نابود می‌شود.
- `void decrementSoldierCount()`
  - تعداد سربازان ارتش را **یک واحد کاهش می‌دهد**، زمانی که سربازی قدرتش به صفر می‌رسد و می‌میرد .

6. کلاس `GameEngine`

نمایانگر **موتور اصلی بازی** است که جریان بازی، نوبت‌ها و پایان مسابقه را کنترل می‌کند.

این کلاس وظیفه دارد نوبت بازیکن‌ها را مدیریت کند، اعمال مجاز را اجرا کند و بررسی کند آیا بازی به پایان رسیده است یا خیر.

```
class GameEngine {
    private Player player1, player2;
    private int currentTurn; // 0 = player1, 1 = player2

    public GameEngine(Player player1, Player player2)
    public void startGame()
    private void nextTurn()
    public void performAction(String actionType, Player actor, Player opponen
    public boolean isGameOver()
    public int getCurrentTurn()
}
```

ویژگی‌ها:

- **player1, player2** — دو بازیکن شرکت‌کننده در بازی.
- **currentTurn** — عددی برای تعیین نوبت بازیکن جاری:
  - `0` برای player1
  - `1` برای player2
  - `-1` به‌معنی شروع‌نشده بودن بازی

- قبل از startGame مقدار currentTurn باید -1 باشد، پس از فراخوانی آن، مقدار 0 می‌شود (player1 شروع می‌کند).

متدها:

- `public void startGame()`
  - بازی را آغاز می‌کند و نوبت را روی **player1 (مقدار 0)** تنظیم می‌کند.
- `private void nextTurn()`
  - پس از هر حرکت (اکشن)، نوبت را بین دو بازیکن جابه‌جا می‌کند.

`public void performAction(String actionType, Player actor, Player opponent)`

  - اجرای یک **عمل (action)** از سمت بازیکن فعلی (`actor`) را مدیریت می‌کند.
  - `actionType` می‌تواند یکی از موارد زیر باشد:
    - `"gathergold"` → بازیکن طلا جمع می‌کند. (`actor.gatherGold()`)
    - `"attack"` → بازیکن به حریف حمله می‌کند. (`actor.attack(opponent)`)
  - بررسی می‌کند که:

    ۱. بازی تمام نشده باشد (`isGameOver() == false`)

    ۲. نوبت بازیکن درست باشد (فقط بازیکنی که نوبتش است اجازه‌ی حرکت دارد).
  - در صورت اجرای موفق، نوبت به بازیکن بعدی منتقل می‌شود.
- `public boolean isGameOver()`
  - بررسی می‌کند آیا یکی از بازیکنان **تمام ساختمان‌هایش را از دست داده است یا خیر**.
  - اگر بله → بازی تمام شده و مقدار `true` برمی‌گرداند.
- `public int getCurrentTurn()`
  - مقدار فعلی `currentTurn` را برمی‌گرداند تا مشخص شود نوبت کدام بازیکن است.

نکات:

- تنها اکشن‌هایی که از طریق `performAction()` انجام می‌شوند، **نوبت مصرف می‌کنند** (مثل `gatherGold` و `attack`) و برای بازی اصلی تضمین میشود پلیر فقط از طریق `performAction()` مجاز است این دو عمل را اجرا کند.
- ساخت سرباز یا ساختمان در کلاس `Player` انجام می‌شود و **نوبت مصرف نمی‌کند**.
- پس از هر اکشن، در صورتی که بازی هنوز تمام نشده باشد، متد `nextTurn()` اجرا می‌شود تا نوبت

عوض شود.

نکات مهم:

- استفاده از **آرایه ثابت** برای سربازها و ساختمان‌ها الزامی است.
- بازی ادامه پیدا می‌کند تا یکی از بازیکنان همه ساختمان‌هایش نابود شود.
- نوبت بازیکنان به صورت دقیق توسط **GameEngine** کنترل شود.

نکته :داخل پروژه اولیه یک فایل `Main.java` قرار داده شده که با اجرای آن صحت عملکرد برنامه خود را بسنجید. خروجی مورد نظر باید به صورت زیر باشد :

```
=== SIMPLE VALIDATION START ===
[OK] P1 gold after T1 gather should be 50
[OK] Turn should pass to P2
[OK] P2 gold after T2 gather should be 70
[OK] Turn should pass to P1
[OK] P1 soldierCount should be 2
[OK] P1 gold should be 10
[OK] P2 soldierCount should be 3
[OK] P2 gold should be 10
[OK] P1 attack should deal 60 damage
[OK] P1 soldiers consumed => 0 remain
[OK] Turn should pass to P2
[OK] P2 attack should deal 90 damage
[OK] P2 soldiers consumed => 0 remain
[OK] Turn should pass to P1
[OK] P1 gold should be 60
[OK] Turn should pass to P2
[OK] P2 gold should be 80
[OK] Turn should pass to P1
[OK] P1 gold should be 110
[OK] Turn should pass to P2
[OK] P2 gold should be 150
[OK] Turn should pass to P1
[OK] P1 gold should be 160
[OK] Turn should pass to P2
[OK] P1 soldierCount should be 7
[OK] P1 gold should be 20
[OK] Turn should pass to P1
```

```
[OK] P2 main should be destroyed
[OK] P2 buildingCount should be 0
[OK] Game should be over

=== FINAL STATUS ===
P1 Gold=20, Buildings=1, Soldiers=3, MainHP=110
P2 Gold=220, Buildings=0, Soldiers=0, MainHP=0
Winner: P1
=== SIMPLE VALIDATION END ===
```

فایل زیپ آپلودی وقتی باز میشود باید فقط شامل فایل های زیر باشد:

```
├── Army.java
├── Player.java
├── Building.java
├── ResourceManager.java
├── GameEngine.java
└── Soldier.java
```