مدیریت سوخت

- سطح : ساده
- طراح: آناهیتا بیانی

فایل اولیه پروژه را از این‌لینک دریافت کنید.

مدیر یک پارکینگ قصد دارد سیستمی برای مدیریت سوخت و انرژی خودروها طراحی کند. در این سیستم سه نوع منبع انرژی برای خودروها وجود دارد: خودروهای بنزینی، دیزلی و برقی. هر خودرو دارای یک منبع انرژی است که میزان سوخت/شارژ موجود، ظرفیت، و نوع انرژی آن مشخص است. همچنین خودروها می‌توانند سوخت بگیرند یا انرژی مصرف کنند. شما باید با استفاده از مفاهیم برنامه‌نویسی شیءگرا این سیستم را پیاده‌سازی کنید.

توضیحات:

enum Fuel:

```
1 | public enum Fuel { GASOLINE, DIESEL, ELECTRICITY }
```

یک enum نمایانگر نوع انرژی خودرو است.

اینترفیس EnergySource:

هر کلاس منبع انرژی باید آن را پیاده‌سازی کند و شامل متدهای زیر است:

```
1 | public interface EnergySource {
2 |  Fuel getFuelType();
3 |  double getLevel();
4 |  void accept(Fuel fuel, double amount);
5 |  void consumeFor(double kmRequested); }
```

توضیحات متدها:

- متد `getFuelType()` : نوع سوخت (GASOLINE، DIESEL یا ELECTRICITY) را برمی‌گرداند.
- متد `getLevel()` : سطح فعلی سوخت یا شارژ را برمی‌گرداند.
- در تمام خروجی‌های برنامه، مقادیر عددی (مثل سطح سوخت، میزان کمبود یا اضافه‌ی سوخت) باید با دقت دو رقم اعشار چاپ شوند.

کلاس GasTank:

نمایانگر منبع انرژی خودروهای بنزینی است.

پراپرتی‌ها (private):

```
double level; // مقدار فعلی بنزین
double capacity; // ظرفیت باک
```

سازنده:

```
public GasTank(double initialLevel, double capacity)
```

متدها:

- متد `accept(Fuel fuel, double amount)` : متدی برای پذیرش سوخت است. ابتدا بررسی می‌کند که نوع سوخت درست باشد، اگر نوع سوخت اشتباه باشد، پیام زیر چاپ شود:

`Refueling failed: wrong fuel type for gasoline tank.`

اگر مقدار بیش از ظرفیت باشد، تا حد ظرفیت پر شود و مقدار سوخت اضافه در پیام زیر چاپ شود:

`{x} L is more than needed; tank filled to capacity.`

در غیر این صورت پیام زیر چاپ شود:

`Gasoline refueled successfully.`

- متد `consumeFor(double kmRequested)` : برای مسافت درخواستی مقدار سوخت لازم را محاسبه می‌کند. اگر کافی باشد، از سطح کم کرده و میزان سوخت فعلی باک را در پیام زیر چاپ کند:

```
Trip complete: enough gasoline. {new level} level is reached.
```

اگر کافی نباشد، میزان کمبود سوخت در پیام زیر چاپ شود:

```
Not enough gasoline for the trip. Need {x} L more.
```

ویژگی‌ها:

- نوع سوخت: GASOLINE
- مصرف سوخت: 0.06 لیتر در هر کیلومتر.

کلاس DieselTank:

نمایانگر منبع انرژی خودروهای دیزلی است. پراپرتی‌ها مشابه کلاس قبل.

سازنده:

```
public DieselTank(double initialLevel, double capacity)
```

متدها:

- متد `accept(Fuel fuel, double amount)` : متدی برای پذیرش سوخت است. ابتدا بررسی می‌کند که نوع سوخت درست باشد، اگر نوع سوخت اشتباه باشد، پیام زیر چاپ شود:

```
Refueling failed: wrong fuel type for diesel tank.
```

اگر مقدار بیش از ظرفیت باشد، تا حد ظرفیت پر شود و مقدار سوخت اضافه در پیام زیر چاپ شود:

```
{x} L is more than needed; tank filled to capacity.
```

در غیر این صورت پیام زیر چاپ شود:

```
Diesel refueled successfully.
```

- متد `consumeFor(double kmRequested)` : برای مسافت درخواستی مقدار سوخت لازم را محاسبه می‌کند. اگر کافی باشد، از سطح کم کرده و میزان سوخت فعلی باک را در پیام زیر چاپ کند:

```
Trip complete: enough diesel. {new level} level is reached.
```

اگر کافی نباشد، میزان کمبود سوخت در پیام زیر چاپ شود:

```
Not enough diesel for the trip. Need {x} L more.
```

ویژگی‌ها:

- نوع سوخت: DIESEL
- مصرف سوخت: 0.05 لیتر در هر کیلومتر.

کلاس BatteryPack:

نمایانگر منبع انرژی خودروهای برقی است. پراپرتی‌ها مشابه کلاس قبل.

سازنده:

```
public BatteryPack(double initialLevel, double capacity)
```

متدها:

- متد `accept(Fuel fuel, double amount)` : متدی برای پذیرش انرژی است. ابتدا بررسی می‌کند که نوع انرژی درست باشد، اگر نوع انرژی اشتباه باشد، پیام زیر چاپ شود:

```
Charging failed: wrong energy type for battery pack.
```

اگر مقدار بیش از ظرفیت باشد، تا حد ظرفیت شارژ شود و مقدار انرژی اضافه در پیام زیر چاپ شود(kWh):

```
{x} kWh is more than needed; battery charged to full capacity.
```

در غیر این صورت پیام زیر چاپ شود:

```
Charging successful.
```

- متد `consumeFor(double kmRequested)` : برای مسافت درخواستی مقدار انرژی لازم را محاسبه می‌کند. اگر کافی باشد، از سطح انرژی باتری کم کرده و میزان شارژ فعلی را در پیام زیر چاپ کند:

```
Trip complete: battery sufficient. {new level} level is reached.
```

اگر کافی نباشد، میزان کمبود انرژی در پیام زیر چاپ شود:

```
Battery low for the trip. Need {x} kWh more.
```

ویژگی‌ها:

- نوع سوخت: ELECTRICITY
- مصرف انرژی: 0.15 kWh در هر کیلومتر.

آنچه باید آپلود کنید

ساختار فایل `zip` ارسالی باید به صورت زیر باشد:

```
<zip_file_name.zip>
    ├── GasTank.java
    ├── DieselTank.java
    ├── BatteryPack.java
    ├── Fuel.java
    └── EnergySource.java
```

عملیات محرمانه

- سطح: متوسط
- طراح: آرتین عضدی فر

ابتدا فایل اولیه پروژه را از این لینک دانلود کنید. در یک سازمان اطلاعاتی فوق‌محرمانه، گروهی از مأموران ویژه برای انجام مأموریت‌های پیچیده در سراسر جهان فعالیت می‌کنند. هر مأمور تخصص متفاوتی دارد و بسته به نوع مأموریت، عملکرد خاصی از خود نشان می‌دهد.

سازمان شما را استخدام کرده تا سیستمی طراحی کنید که رفتار این مأموران را در موقعیت‌های مختلف شبیه‌سازی کند.

معرفی فایل ها

کلاس Mission

کلاس پایه‌ی انتزاعی برای مدیریت انواع ماموریت‌ها

فیلدها:

```
protected String codeName;
protected int enemyCount;
protected int securityLevel;
protected boolean isHacked;
protected boolean isCompleted;
```

توضیحات فیلد ها به ترتیب:

۱.اسم رمز ماموریت.

۲.تعداد دشمنان در ماموریت را مشخص می‌کند.

۳.سطح امنیت ماموریت را مشخص می‌کند.

۴.هر ماموریت نیاز به انجام عملیات سایبری دارد، این فیلد مشخص می‌کند که آیا سیستم‌ها با موفقیت هک شده‌اند یا نه.

۵. نشان می‌دهد ماموریت به اتمام رسیده یا خیر.

کانستراکتور:

```
public Mission(String codeName, int enemyCount, int securityLevel)
```

در این سازنده اسم رمز ماموریت، تعداد دشمنان، سطح امنیتی مشخص میشوند. ماموریت‌ها در شروع هک نشده‌اند و به پایان نرسیده‌اند.

متدها:

```
public abstract void hackSystem(int hackerLevel);
public abstract void reduceEnemies(int count);
public abstract void finish();
public abstract String getType();
```

پیاده سازی این متدها در کلاس‌های فرزند توضیح داده خواهد شد.

کلاس MissionStealth

از کلاس Mission ارثبری می‌کند. این نوع ماموریت ها انواع آسان‌تری هستند و راحتتر به پایان می‌رسند.

سازنده:

```
public MissionStealth(String codeName, int enemyCount, int securityLevel)
```

پیاده سازی متدها:

متد `hackSystem(int hackerLevel)` : به دلیل ساده بودن بدون محدودیتی سیستم‌ها هک می‌شوند و **فیلد مربوطه آپدیت میشود** .

متد `reduceEnemies(int count)` : از این متد برای کشتن دشمنان ماموریت استفاده می‌شود. از

آنجایی که ماموریت نوع ساده است به تعداد *دو برابر ورودی* دشمن کشته می‌شود. **توجه:** تعداد دشمنان نمیتواند منفی باشد.

متد `finish()` : در صورتی که سیستم‌ها هک شده باشند و تعداد دشمنان باقی مانده کمتر یا حداکثر برابر با ۳ باشد، ماموریت به اتمام می‌رسد.

متد `getType()` : متن زیر برگردانده می‌شود:

```
stealth mission
```

کلاس MissionHighSecurity

سازنده:

```
1 | public MissionHighSecurity(String codeName, int enemyCount, int securityLevel
```

پیاده سازی متدها:

متد `hackSystem(int hackerLevel)` : در صورتی که سطح توانایی هکر از سطح امنیتی ماموریت بالاتر یا برابر بود سیستم هک می‌شود و وضعیت آپدیت می‌شود.

متد `reduceEnemies(int count)` : به تعداد ورودی دشمن کشته می‌شود. **توجه:** تعداد دشمنان نمیتواند منفی باشد.

متد `finish()` : در صورتی که سیستم‌ها هک شده باشند و دشمنی باقی نمانده باشد، ماموریت به اتمام می‌رسد.

متد `getType()` : متن زیر برگردانده می‌شود:

```
high security mission
```

کلاس Agent

کلاس پایه انتزاعی برای کنترل انواع مامورها

فیلدها:

```
private String codename;
private int energy;
```

معرفی فیلدها به ترتیب:

۱.اسم رمز مامور

۲.میزان انرژی مامور، هر فرد برای انجام عملیات باید انرژی صرف کند و تنها در صورت کافی بودن انرژی میتواند عملیاتش را انجام دهد.

سازنده:

```
public Agent(String codename, int energy)
```

هر دو فیلد کاراکتر را مقداردهی می‌کند.

متدها:

```
public abstract void actOnMission(Mission mission);
public void consumeEnergy(int amount)
public boolean canAct(int required)
@Override
public String toString()
```

پیاده سازی متدها:

متد انتزاعی `actOnMission` در زیرکلاس‌ها توضیح داده خواهد شد.

متد `consumeEnergy(int amount)` : از انرژی کاراکتر به اندازه *amount* کم می‌شود. **توجه:** در صورت منفی شدن، مقدار انرژی صفر می‌شود.

متد `canAct(int required)` : اگر مقدار انرژی کاراکتر از مقدار required کمتر نبود، true و در غیر این صورت false بر میگرداند.

بازنویسی متد `toString` : رشته زیر برگردانده می‌شود:

```
[codename] | Energy: [energy]
```

کلاس های بعدی همگی از کلاس Agent ارثبری می‌کنند.

کلاس HackerAgent

فیلدها:

```
1 | private int level;
```

سطح هکرها باهمدیگر متفاوت است و این فیلد سطح هکر را مشخص می‌کند.

سازنده:

```
1 | public HackerAgent(String codename, int energy, int level)
```

علاوه بر مقداردهی فیلدهایی که در کلاس والد وجود داشت، سطح هکر هم مقداردهی می‌شود.

پیاده سازی متدها:

متد `actOnMission` : کلاس هکر با استفاده از این متد، در صورت امکان ماموریت داده شده را هک می‌کند. شرایط:

هکرها برای انجام عملیات به 25 واحد انرژی نیاز دارند؛ در صورت کافی نبودن انرژی، هیچ تغییری در وضعیت مأموریت یا انرژی ایجاد نمی‌شود. در صورت وجود انرژی کافی، دستور هک ماموریت، با اطلاع دادن سطح هکر، اجرا می‌شود و ۲۵ واحد انرژی مصرف می‌شود.

بازنویسی متد `toString` : رشته زیر برگردانده می‌شود:

```
Hacker | [codename] | Energy: [energy] | Level: [level]
```

کلاس SniperAgent

سازنده:

```
1 | public SniperAgent(String codename, int energy)
```

پیاده سازی متدها:

متد actOnMission : انرژی مورد نیاز برای انجام عملیات 40 واحد است، در صورت کافی نبودن انرژی اتفاقی نمی‌افتد. در غیر این صورت: تلاش می‌شود 1 دشمن از ماموریت کشته شود و سپس ۴۰ واحد انرژی مصرف می‌شود.

بازنویسی متد toString : رشته زیر برگردانده می‌شود:

```
Sniper | [codename] | Energy: [energy]
```

کلاس SpyAgent

سازنده:

```
1 | public SpyAgent(String codename, int energy)
```

پیاده سازی متدها:

متد actOnMission : جاسوس برای انجام عملیات 30 واحد انرژی نیاز دارد، علاوه بر آن سیستم‌های ماموریت هدف باید هک شده باشند و جاسوس نمیتواند در ماموریتی که هنوز هک صورت نگرفته حمله ای انجام دهد. در صورت امکان 3 دشمن از ماموریت مورد هدف قرار می‌گیرند و ۳۰ واحد انرژی مصرف می‌شود.

بازنویسی متد toString :

رشته زیر برگردانده می‌شود:

```
Spy | [codename] | Energy: [energy]
```

آنچه باید آپلود کنید:

پروژه اولیه را کامل کنید و به صورت فایل زیپ آپلود کنید

```
<zip_file_name.zip>
    ├── Mission.java
    ├── MissionStealth.java
    ├── MissionHighSecurity.java
    ├── Agent.java
    ├── HackerAgent.java
    ├── SniperAgent.java
    └── SpyAgent.java
```

سیستم مدیریت وسایل نقلیه

- سطح: متوسط
- طراح: محسن نوروزی

فایل اولیه پروژه را از این لینک دریافت کنید.

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

در یک شهر آینده‌نگر به نام **NeoTransportia**، مهندسان نرم‌افزار تصمیم گرفتند برای کنترل وسایل نقلیه‌ی هوشمند یک سیستم شیءگرا بنویسند که بتواند رفتار انواع خودروها را در قالب کد مدل کند.

در این دنیا، همه‌ی وسایل نقلیه باید قابلیت‌هایی مشترک داشته باشند: بتوانند **روشن شوند، حرکت کنند، توقف کنند، ترمز کنند** و وضعیت کار کردنشان را بدانیم.

اینجاست که مهندسان تصمیم گرفتند پایه‌ای‌ترین قرارداد را تعریف کنند:

اینترفیس `Controllable`

این اینترفیس مثل قراردادی بین وسایل نقلیه است — یعنی هر وسیله‌ای که قرار است قابل کنترل باشد، باید این متدها را داشته باشد.

۱. متد `start` : وسیله را روشن می‌کند.

```
void start();
```

۲. متد `stop` : وسیله را خاموش می‌کند.

```
void stop();
```

۳. متد `accelerate` : سرعت خودرو را به اندازه‌ی مقدار داده‌شده افزایش می‌دهد.

```
void accelerate(int amount);
```

۴. متد `decelerate` : سرعت خودرو را به اندازه‌ی مقدار داده‌شده کاهش می‌دهد(در پیاده سازی این متد در کلاس ها حواستان باشد که سرعت منفی نشود).

```
void decelerate(int amount);
```

۵. متد `isRunning` : وضعیت موتور را برمی‌گرداند.

```
boolean isRunning();
```

۶. متد `emergencyBrake` : ترمز ناگهانی. سرعت = صفر.

```
void emergencyBrake();
```

همچنین در NeoTransportia، خودروها یا **سوختی**، یا **برقی**، یا **ترکیبی از هر دو** هستند. برای اینکه سیستم بتواند انواع منبع انرژی را مدل کند، دو اینترفیس جداگانه طراحی شد.

اینترفیس `FuelBased`

این قرارداد مربوط به وسایل نقلیه‌ای است که با بنزین یا گاز کار می‌کنند. کارکرد متد ها :

۱. متد `refuel` : به مقدار سوخت اضافه می‌کند.

```
void refuel(double liters);
```

۲. متد `getFuelLevel` : مقدار فعلی سوخت را برمی‌گرداند.

```
double getFuelLevel();
```

۳. متد `getEfficiency` : بازده انرژی را بازمیگرداند(بر حسب کیلومتر بر لیتر).

```
double getEfficiency();
```

۴. متد `consumeFuel` :سوخت مصرف‌شده بر اساس مسافت را محاسبه می‌کند و از سوخت فعلی کم می‌کند و مقدار مصرفشده را باز میگرداند .

```
double consumeFuel(int distance);
```

اینترفیس ElectricPowered

نسخه‌ی برقی همان مفاهیم را دارد، اما با واحد **کیلووات‌ساعت (kWh)**. کارکرد متد ها :

۱. متد `recharge` : به مقدار شارژ باتری اضافه می‌کند.

```
void recharge(int kWh);
```

۲. متد `getBatteryLevel` : مقدار فعلی شارژ باتری را برمی‌گرداند.

```
int getBatteryLevel();
```

۳. متد `getEfficiency` : بازده انرژی را بازمیگرداند.

```
double getEfficiency();
```

۴. متد `consumeBattery` :شارژ مصرف‌شده بر اساس مسافت را محاسبه می‌کند و از مقدار فعلی کم میکند و مقدار شارژ مصرف شده را باز میگرداند.

```
double consumeBattery(int distance);
```

کلاس Vehicle

یک کلاس **انتزاعی (abstract)** است و از `Controllable` پیروی می‌کند. همه‌ی وسایل نقلیه از اینجا شروع می‌شوند.

ویژگی‌های اساسی:

- `model` : نام خودرو.
- `speed` : سرعت فعلی.
- `running` : وضعیت موتور.

متدهایش پایه رفتار مشترک را تعریف می‌کنند (شروع، توقف، ترمز، شتاب...). پیاده سازی های مربوط به این متد هارا انجام دهید.

دقت کنید که در ابتدا وسیله نقلیه ساکن میباشد.

پیاده سازی متد های `calculateRange` و `getEfficiency` بسته به نوع ماشین بازنویسی می‌شوند.

پیاده سازی اولیه متد ها باید در همین کلاس انجام شود و در صورت نیاز در کلاس های دیگر override شود .

۱. متد `accelerate` : سرعت خودرو را به اندازه‌ی مقدار داده‌شده افزایش می‌دهد(شرط روشن بودن ماشین بررسی شود).

```
1 public void accelerate(int amount)
```

۲. متد `decelerate` : سرعت خودرو را به اندازه‌ی مقدار داده‌شده کاهش می‌دهد (شرط روشن بودن ماشین بررسی شود و دقت کنید که سرعت ماشین نباید منفی شود).

```
1 public void decelerate(int amount)
```

۳. متد `emergencyBrake` : ماشین در لحظه متوقف می‌شود(شرط روشن بودن ماشین بررسی شود).

```
1 public void emergencyBrake()
```

۳. متد `stop` : ماشین علاوه بر متوقف شدن خاموش هم می‌شود.

```
1 public void stop()
```

کلاس `GasCar`

از **Vehicle** و **FuelBased** ارث‌بری می‌کند.یعنی هم یک وسیله‌ی قابل کنترل است، هم یک خودرو سوختی.

ویژگی‌های اساسی:

- `fuelLevel` : میزان سوخت.
- `fuelEfficiency` : بازده (مثلاً 15 کیلومتر بر لیتر).

۱. متد `refuel` : به مقدار سوخت اضافه می‌کند(نباید منفی باشد).

```
public void refuel(double liters)
```

۲. متد `consumeFuel` : سوخت مصرف‌شده بر اساس مسافت طی شده را محاسبه می‌کند و از سوخت فعلی کم می‌کند(سوخت مصرف‌شده برابر است با مسافت طی‌شده تقسیم بر بازده خودرو.)

```
public double consumeFuel(int distance)
```

۳. متد `accelerate` : افزایش سرعت و کاهش سوخت را انجام می‌دهد(برای محاسبه‌ی مسافت طی‌شده، سرعت افزایش‌یافته را تقسیم بر ۵ کنید).

```
public void accelerate(int amount)
```

۴. متد `calculateRange` :مسافتی که ماشین توانایی پیمایش آن را دارد می‌دهد.

```
public double calculateRange()
```

کلاس ElectricCar

کلاس `ElectricCar` هم از `Vehicle` و هم از `ElectricPowered` استفاده می‌کند.

ویژگی‌های اساسی:

- `batteryLevel` : سطح فعلی باتری.
- `electricEfficiency` : بازده مصرف انرژی.

- `ecoMode` : فعال بودن حالت رانندگی کم‌مصرف.

توجه: با فعال بودن `ecoMode` ماشین فکر میکند که بازده آن 1.5 برابر شده ولی در واقعیت این مقدار ثابت مانده(این مورد را در محاسبه باتری مصرف شده لحاظ کنید).

متدها:

۱. متد `recharge` : به مقدار شارژ اضافه می‌کند(نباید منفی باشد)..

```
public void recharge(double units)
```

۲. متد `consumeBattery` :شارژ مصرف‌شده بر اساس مسافت را محاسبه می‌کند(مقدار شارژ باتری مصرف‌شده برابر است با مسافت طی‌شده تقسیم بر بازده خودرو).

```
public double consumeBattery(int distance)
```

۳. متد `accelerate` : افزایش سرعت و کاهش سوخت ر انجام می‌دهد.این تابع مصرف متفاوتی در حالت معمول و Eco دارد (جهت محاسبه مسافت طی شده در حالت eco سرعت افزایش یافته را تقسیم بر 5 کنید ، همچنین اگر حالت eco فعال بود سرعت افزایش یافته را ضرب در 0.85 کنید و سپس به سرعت قبل اضافه کنید.).

```
public void accelerate(int amount)
```

۴. متد `calculateRange` :مسافتی که ماشین توانایی پیمایش آن را دارد می‌دهد.

```
public double calculateRange()
```

کلاس HybridCar

کلاس `HybridCar` از هر دو Interface (`FuelBased` , `ElectricPowered`) و از `Vehicle` ارث می‌برد.

ویژگی‌ها:

- `ecoMode`
- `fuelEfficiency`
- `electricEfficiency`
- `batteryLevel`

متدها: همه متد ها تعاریفی مانند متد های کلاس های دیگر دارند به جز متد های زیر:

۱. درمتد `accelerate` ، اگر باتری کافی داشته باشد با برق حرکت می‌کندو از برق کم می‌شود، وگرنه با سوخت حرکت می‌کند و از آن کم می‌شود.(اگر لول باتری بالاتر از دو بود همواره از باتری استفاده کنید و در غیر این صورت اگر سوخت موجود بود از سوخت استفاده کند همچنین اثر فعال بودن eco در هردو حالت هم بررسی کنید) .

```
1 public void accelerate(int amount)
```

۲. متد ویژه‌ی `hybridBalance` تعادل بین سوخت و باتری را برقرار می‌کند تا منبع‌ها همیشه منصفانه مصرف شوند.(این متد بررسی می‌کند که اگر مقدار انرژی موجود در باتری بیشتر از سوخت باشد، نیمی از اختلاف انرژی باتری از سوخت را به سوخت اضافه میکند و برعکس).

```
1 public void hybridBalance()
```

۳. `calculateRange` : مجموع برد را با دو منبع جمع می‌زند.

```
1 public double calculateRange()
```

۴. `getEfficiency` : میانگین بازده حالت باتری و سوختی را می‌دهد.

```
1 public double getEfficiency()
```

ورودی نمونه

```
1 public class Main {
2     public static void main(String[] args) {
3
```

```
GasCar gasCar = new GasCar("Sedan", 8);
System.out.println(" GasCar: " + gasCar.getModel());
System.out.println("Efficiency: " + String.format("%.2f", gasCar.getE

gasCar.start();
gasCar.refuel(20);
System.out.println("Fuel after refuel: " + String.format("%.2f", gasC
gasCar.accelerate(30);
System.out.println("Speed after accelerating: " + String.format("%.2f
System.out.println("Fuel left: " + String.format("%.2f", gasCar.getFu
System.out.println("Estimated range: " + String.format("%.2f", gasCar

gasCar.decelerate(10);
System.out.println("Speed after decelerate: " + String.format("%.2f",

gasCar.emergencyBrake();
gasCar.stop();
System.out.println("Car stopped. Running: " + gasCar.isRunning());
System.out.println("----------------------------------\n");


ElectricCar electricCar = new ElectricCar("Tesla", 9);
System.out.println(" ElectricCar: " + electricCar.getModel());
System.out.println("Efficiency: " + String.format("%.2f", electricCar

electricCar.start();
electricCar.recharge(40);
System.out.println("Battery after recharge: " + String.format("%.2f",
electricCar.switchToEcoMode();
System.out.println("Eco mode activated: " + electricCar.isEcoModeActi

electricCar.accelerate(50);
System.out.println("Speed after accelerate: " + String.format("%.2f",
System.out.println("Battery left: " + String.format("%.2f", electricC
System.out.println("Estimated range: " + String.format("%.2f", electr

electricCar.emergencyBrake();
electricCar.stop();
System.out.println("Electric car stopped. Running: " + electricCar.is
System.out.println("----------------------------------\n");

HybridCar hybridCar = new HybridCar("Prius", 7, 9);
System.out.println(" HybridCar: " + hybridCar.getModel());
```

```
        System.out.println("Average Efficiency: " + String.format("%.2f", hyb

        hybridCar.start();
        hybridCar.refuel(30);
        hybridCar.recharge(20);
        System.out.println("Fuel after refuel: " + String.format("%.2f", hybr
        System.out.println("Battery after recharge: " + String.format("%.2f",
        System.out.println("Total range before eco mode: " + String.format("%

        hybridCar.activateEcoMode();
        System.out.println("Eco mode activated: " + hybridCar.isEcoModeActive

        hybridCar.accelerate(60);
        System.out.println("Speed after accelerate: " + String.format("%.2f",
        System.out.println("Fuel left: " + String.format("%.2f", hybridCar.ge
        System.out.println("Battery left: " + String.format("%.2f", hybridCar

        System.out.println("Balancing power...");
        hybridCar.hybridBalance();
        System.out.println("Fuel after balance: " + String.format("%.2f", hyb
        System.out.println("Battery after balance: " + String.format("%.2f",
        System.out.println("New estimated range: " + String.format("%.2f", hy

        hybridCar.emergencyBrake();
        hybridCar.stop();
        System.out.println("Hybrid car stopped. Running: " + hybridCar.isRunn

    }
}
```

خروجی نمونه

```
GasCar: Sedan
Efficiency: 8.00
Fuel after refuel: 70.00
Speed after accelerating: 30.00
Fuel left: 69.25
Estimated range: 554.00
Speed after decelerate: 20.00
Car stopped. Running: false
----------------------------------

```

```
ElectricCar: Tesla
Efficiency: 9.00
Battery after recharge: 90.00
Eco mode activated: true
Speed after accelerate: 42.50
Battery left: 89.41
Estimated range: 804.67
Electric car stopped. Running: false
---------------------------------

 HybridCar: Prius
Average Efficiency: 8.00
Fuel after refuel: 80.00
Battery after recharge: 70.00
Total range before eco mode: 1190.00
Eco mode activated: true
Speed after accelerate: 51.00
Fuel left: 80.00
Battery left: 69.26
Balancing power...
Fuel after balance: 74.63
Battery after balance: 74.63
New estimated range: 1194.07
Hybrid car stopped. Running: false
```

آنچه باید آپلود کنید:

پروژه اولیه را کامل کنید و به صورت فایل زیپ آپلود کنید:

```
<zip_file_name.zip>
    ├── Controllable.java
    ├── ElectricCar.java
    ├── ElectricPowered.java
    ├── FuelBased.java
    ├── GasCar.java
    ├── HybridCar.java
    └── Vehicle.java
```

Javagram (The Messenger)

- سطح : سخت
- طراح : آریا زریاب

جزئیات برنامه

ابتدا پروژه اولیه را از این لینک دانلود کنید.

این تمرین یک سیستم پیام‌رسان را شبیه‌سازی می‌کند که در آن کاربران می‌توانند پیام‌هایی را با استفاده از پیام‌رسان‌های مختلف ارسال و دریافت کنند. سیستم شامل سه پیام‌رسان است: **تلگرام**، **واتساپ** و **اینستاگرام**. کاربران می‌توانند پیام‌ها را از طریق هر یک از این پیام‌رسان‌ها به دیگر کاربران ارسال کنند و سیستم پیام‌ها را با تایم‌استمپ‌ها و وضعیت تحویل ردیابی می‌کند.

▼ ساختار فایل پروژه

```
<zip_file_name.zip>
      ├── Messenger.java
      ├── Telegram.java
      ├── Whatsapp.java
      ├── Instagram.java
      ├── Message.java
      ├── User.java
      └── MessageStatus.java
```

اینترفیس `Messenger`

این اینترفیس نوع پیام رسان را مشخص میکند.

۱. متد `sendMessage` : پیام را به گیرنده ارسال می‌کند.

```java
void sendMessage(String message, User receiver);
```

۲. متد `receiveMessage` : دریافت پیام از فرستنده را مدیریت می‌کند

```
String receiveMessage();
```

۳. متد `getMessengerName` : نام پیام‌رسان را برمی‌گرداند.

```
String getMessengerName();
```

۴. متد `setLastMessage` : آخرین متن مسیج پیام رسان را آپدیت میکند.

```
public void setLastMessage(String message);
```

کلاس `Telegram`

- این کلاس اینترفیس `Messenger` را `implement` میکند.

پراپرتی ها

۱. `lastMessage` : آخرین متن پیام را نگه میدارد و از جنس `String` و سطح دسترسی `private` میباشد

متود ها

۲. `sendMessage` : باید متن زیر را *چاپ* کند.

`Message sent via Telegram: + message`

▼ مثال

به طور مثال اگر متن پیام `hi` باشد خروجی `Message sent via Telegram: hi` میباشد

۲. `receiveMessage` : باید متن زیر را *برگرداند*.

`Message received via Telegram: + lastMessage`

۳. `getMessengerName` : نام کلاس که همان `Telegram` است را *برمیگرداند*.

کلاس Whatsapp

- این کلاس اینترفیس `Messenger` را `implement` میکند.

پراپرتی ها

۱. `lastMessage` : آخرین متن پیام را نگه میدارد و از جنس `String` و سطح دسترسی `private` میباشد

متود ها

۲. `sendMessage` : باید متن زیر را *چاپ* کند.

```
Message sent via Whatsapp: + message
```

▼ مثال

به طور مثال اگر متن پیام `hi` باشد خروجی `Message sent via Whatsapp: hi` میباشد

۲. `receiveMessage` : باید متن زیر را *برگرداند*.

```
Message received via Whatsapp: + lastMessage
```

۳. `getMessengerName` : نام کلاس که همان `Whatsapp` است را *برمیگرداند*.

کلاس Instagram

- این کلاس اینترفیس `Messenger` را `implement` میکند.

پراپرتی ها

۱. `lastMessage` : آخرین متن پیام را نگه میدارد و از جنس `String` و سطح دسترسی `private` میباشد

متود ها

۲. `sendMessage` : باید متن زیر را *چاپ* کند.

```
Message sent via Instagram: + message
```

▼ مثال

به طور مثال اگر متن پیام `hi` باشد خروجی `Message sent via Instagram: hi` میباشد

۲. `receiveMessage` : باید متن زیر را *برگرداند*.

```
Message received via Instagram: + lastMessage
```

۳. `getMessengerName` : نام کلاس که همان `Instagram` است را *برمیگرداند*.

کلاس Message

هر پیام را تعریف میکند.

پراپرتی ها

۱. `content` : متن پیام را نگه میدارد که از جنس `String` است.
۲. `sender` : فرستنده پیام که از جنس `User` است.
۳. `receiver` : گیرنده پیام که از جنس `User` است.
۴. `messenger` : نوع پیام رسان که از جنس `Messenger` است.
۵. `sentTimestamp` : زمان فرستادن پیام که از جنس `LocalDateTime` است.
۶. `deliveredTimestamp` : زمان رسیدن پیام به گیرنده که از جنس `LocalDateTime` است.
۷. `readTimestamp` : زمان دیدن پیام توسط گیرنده که از جنس `LocalDateTime` است.
۸. `status` : وضعیت پیام که از جنس `MessageStatus` است.

تمام پراپرتی ها سطح دسترسی `private` را دارند.

کانستراکتور

این کلاس تنها یک متود سازنده دارد که به ترتیب مقادیر `content` و `sender` و `receiver` و `messenger` را ورودی میگیرد و آن هارا مقداردهی میکند، همچنین زمان فرستادن پیام و وضعیت آن را مشخص میکند.

متود ها

۱. `markAsDelivered` : پیام را به وضعیت `DELIVERED` تغییر میدهد.

۲. `markAsRead` : پیام را به وضعیت `READ` تغییر میدهد.

توجه : متود `toString` را تغییر ندهید.

کلاس `User`

پراپرتی ها

۱. `name` : نام کاربر را نگه میدارد که از جنس `String` است.

۲. `sentMessages` : تمام مسیج های ارسال شده کاربر را نگه داری میکند و از جنس `List<Message>` است.

۳. `receivedMessages` : تمام مسیج های دریافت شده کاربر را نگه داری میکند و از جنس `List<Message>` است.

تمام پراپرتی ها سطح دسترسی `private` را دارند.

کانستراکتور

این کلاس تنها یک متود سازنده دارد که `name` را ورودی میگیرد و آنرا مقداردهی میکند.

متود ها

۱. `send` : مسیجی را با پیام رسان دلخواه به گیرنده دلخواه میفرستد.

۲. `receive` : کاربر مسیج را دریافت میکند.

۳. `readMessage` : کاربر مسیج ورودی گرفته شده را میخواند. (صرفا وضعیت پیام را آپدیت کنید)

توجه : فقط قسمت های `//TODO` ۳ متود بالا را کامل کنید.

۴. `deleteSentMessage` : با استفاده از این متود کاربرمیتواند مسیج ورودی گرفته شده را از لیست مسیج های ارسالی اش پاک کند.

۵. `deleteReceivedMessage` : با استفاده از این متود کاربرمیتواند مسیج ورودی گرفته شده را از لیست مسیج های دریافتی اش پاک کند.

در ۳ متود بالا نیازی به چک کردن وجود داشتن پیام در لیست پیام های کاربر نیست، تضمین میشود که پیام در لیست وجود دارد.

▼ مثال

برای مثال با اجرای main زیر:

```
public static void main(String[] args) {
    Messenger telegram = new Telegram();
        User aria = new User("Aria");
        User arman = new User("Arman");

        aria.send("Hello Arman!", arman, telegram);
    }
```

خروجی به صورت زیر است:

```
Aria: Sending "Hello Arman!" to Arman via Telegram...
Message sent via Telegram: Hello Arman!
Arman: Received "Hello Arman!"
```

کلاس MessageStatus

از جنس ENUM است که وضعیت های پیام را نگه میدارد.

```
 public enum MessageStatus {
    SENT,
    DELIVERED,
    READ
}
```

نکات

- شما اجازه‌ی اضافه کردن پراپرتی دیگری غیر از پراپرتی‌های خواسته‌ی سوال ندارید.
- گتر و ستر های مورد نیاز را پیاده سازی کنید.

▼ نحوه نام گذاری متود های گتر و ستر

نامگذاری باید به شکل Camel case باشد. به طور مثال برای نامگذاری متد *setter* و *getter* فیلدی به نام `name` به ترتیب باید به صورت `setName` و `getName` نامگذاری شود.

▼ نحوه استفاده از کلاس LocalDateTime

برای مطالعه متدها و نحوه‌ی استفاده از این کلاس می‌توانید به اینجا مراجعه کنید..

آنچه باید آپلود کنید

ساختار فایل `zip` ارسالی باید به صورت زیر باشد:

```
<zip_file_name.zip>
    ├── Messenger.java
    ├── Telegram.java
    ├── Whatsapp.java
    ├── Instagram.java
    ├── Message.java
    ├── User.java
    └── MessageStatus.java
```

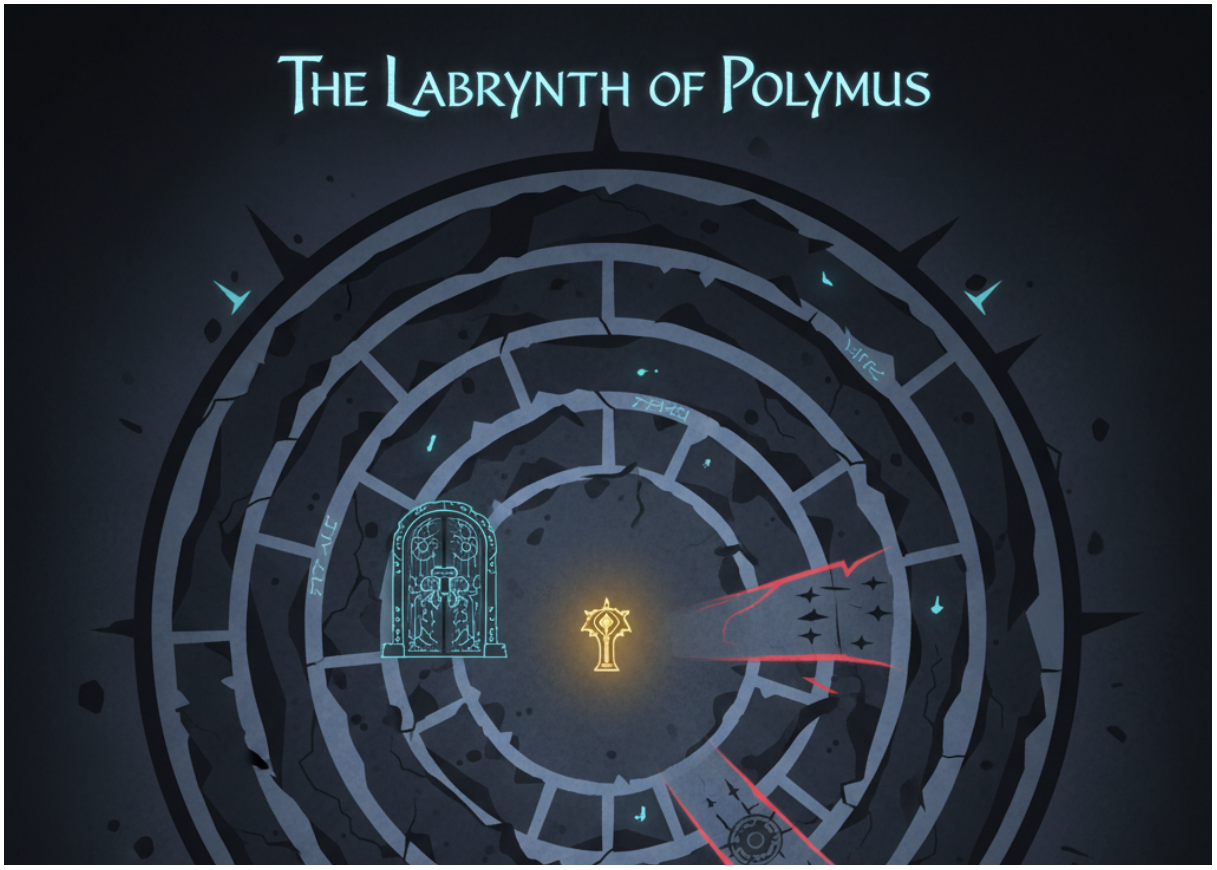
هزارتوی پلیموس (امتیازی)

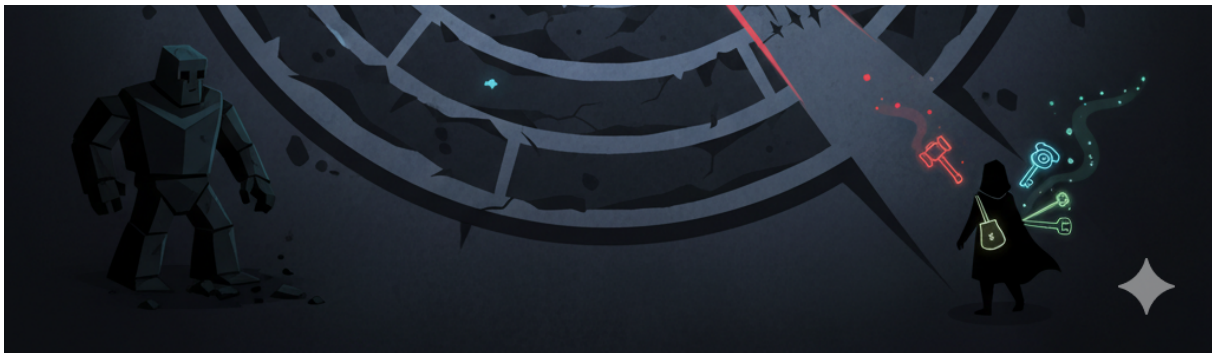
- سطح: سخت
- طراح: علیرضا متقی

در اعماق کوهستان‌های مه‌آلود، "هزارتوی پُلیموس" قرار دارد؛ یک مارپیچ جادویی که توسط یک جادوگر باستانی برای محافظت از یک شیء افسانه‌ای ساخته شده است. این هزارتو یک مکان ثابت نیست؛ تالارها و موانع درون آن بسته به ابزارها و مهارت‌های کسانی که جرأت ورود به آن را دارند، واکنش‌های متفاوتی نشان می‌دهند.

شما یک ماجراجوی شجاع هستید که به کیفی از ابزارهای قدرتمند و جادویی مجهز شده‌اید. هدف شما عبور از این هزارتو و رسیدن به مرکز آن است. موفقیت شما نه به زور بازو، بلکه به خرد شما در انتخاب ابزار مناسب برای هر چالش بستگی دارد. ماهیت تعامل شما با هر مانع باید چندریختی (Polymorphic) باشد و بر اساس مانعی که با آن روبرو می‌شوید و ابزاری که به کار می‌برید، تطبیق یابد.

پروژه اولیه را از این لینک دانلود نمایید.





وظیفه شما مدل‌سازی سفر ماجراجو در این هزارتو است. شما باید سیستمی را طراحی کنید که در آن انواع مختلف ابزارها با انواع مختلف موانع به شیوه‌ای انعطاف‌پذیر و قابل توسعه تعامل داشته باشند و از به‌کارگیری روش هایی مانند `if-else` پرهیز شود. راه‌حل باید زیبا و منطبق بر «اصل باز-بسته» (Open/Closed Principle) باشد؛ یعنی امکان افزودن موانع و ابزارهای جدید با حداقل تغییر در کدهای موجود فراهم باشد.

پیاده‌سازی

شما باید اینترفیس‌ها و کلاس‌های زیر را پیاده‌سازی کنید.

اینترفیس Obstacle (مانع):

این اینترفیس نماینده هر چالشی است که ماجراجو ممکن است با آن روبرو شود و اساس طراحی هزارتو را تشکیل می‌دهد.

```
1 public interface Obstacle {
2     boolean overcome(AdventurerTool tool);
3 }
```

- `overcome` : تلاش برای غلبه بر مانع با استفاده از ابزار ماجراجو (tool) که در صورت موفقیت *true* و در غیر این صورت *false* برمیگرداند.

اینترفیس AdventurerTool (ابزار ماجراجوی):

این اینترفیس قرارداد قابلیت‌های یک ابزار را تعریف می‌کند. به جای یک متد عمومی `use()` ، این اینترفیس متدهای مشخصی برای هر نوع مانعی که می‌تواند با آن تعامل داشته باشد، دارد. این بخش، قلب تعامل چندریختی ما است.

```java
public interface AdventurerTool{
    boolean visit(Golem golem);
    boolean visit(MagicLock lock);
    boolean visit(SpikeTrap trap);
    boolean visit(DungeonRoom room); // Added for composite handling
}
```

موجودیت و گنجینه ها (کلاس های اصلی):

حال، پیاده‌سازی‌های مشخصی از موانع و ابزارها را ایجاد کنید.

موانع اصلی

سه کلاس `Obstacle` پیاده‌سازی کنید:

- `Golem` : یک موجود سنگی بزرگ که در برابر خُرد شدن آسیب‌پذیر است.
- `MagicLock` : یک قفل جادویی روی یک در که فقط با امضای جادویی صحیح باز می‌شود.
- `SpikeTrap` : یک تله نیزه کلاسیک با صفحه فشاری روی زمین که باید با دقت خنثی شود.

هر یک از این کلاس‌ها باید اینترفیس `Obstacle` را پیاده‌سازی کنند. متد `overcome` در هر کلاس باید فقط یک خط باشد.

ابزارهای اصلی

ابزارهای شما اینترفیس `AdventurerTool` را پیاده‌سازی خواهند کرد. برای هر مانعی که یک ابزار *نمی‌تواند* با آن کار کند، متد `visit` مربوط به آن مانع باید به سادگی `false` برگرداند. برای این کار به جای آنکه برای هر یک از ابزار به تفکیک موانع، تابع لازم برای خروجی `true` و یا `false` را پیاده‌سازی کنیم، میتوانیم یک کلاس abstract از `AdventurerTool` بسازیم که به طور پیشفرض برای همه مقادیر `false` برمیگرداند و سپس در هر یک از ابزار ها برای موانع صحیح متناظر تابع جدیدی را `@Override` میکنیم که `true` برمیگرداند. *با این مدل از پیاده سازی و مدل های دیگر در قسمت بعدی یعنی Design Pattern ها بیشتر آشنا میشوید!*

برای پیاده‌سازی پیشنهادی نیاز به یک کلاس پایه داریم که به طور پیشفرض برای هر زوج ابزار و مانعی `false` بازگرداند، این کلاس abstract را `AdventurerToolBase` مینامیم.

ابزار ها:

- Warhammer (پتک جنگی): از AdventurerToolBase ارث بری می‌کند. متد visit(Golem golem) باید پیامی مانند "!پتک جنگی گولم را به تکه‌های سنگ تبدیل کرد" چاپ کرده و true برگرداند. تمام متدهای دیگر visit باید false برگردانند.
- EnchantedKey (کلید جادویی): از AdventurerToolBase ارث بری می‌کند. متد visit(MagicLock lock) آن باید موفقیت‌آمیز باشد و پیامی چاپ کند. این ابزار در برابر سایر موانع شکست می‌خورد.
- MasterThiefsTools (ابزار شاه‌دزد): از AdventurerToolBase ارث بری می‌کند. این مجموعه ابزار همه‌کاره می‌تواند با موفقیت SpikeTrap را (با خنثی کردن) و MagicLock را (با باز کردن قفل) visit کند، اما در برابر Golem کارایی ندارد.

تالار پژواک‌ها (الگوی Composite)

یک ماجراجو تنها با یک مانع روبرو نمی‌شود، بلکه وارد اتاق‌هایی پر از چالش می‌شود.

کلاس DungeonRoom را پیاده‌سازی کنید: این کلاس نیز باید اینترفیس Obstacle را پیاده‌سازی کند، که آن را به یک مانع ترکیبی (Composite) تبدیل می‌کند. این کلاس باید شامل موارد زیر باشد:

ویژگی ها:

- name : نام تالار
- obstaclesInRoom : یک لیست از تمامی موانع در تالار

کانستراکتور:

- کانستراکتور این کلاس به ترتیب اسم تالار و لیست موانع داخل آن را دریافت میکند

متد ها:

- getObstacles()
- getName()
- overcome(AdventurerTool tool) :این متد باید تلاش کند تا بر *تمام* موانع داخل اتاق غلبه

کند. اگر ابزار بتواند بر تمام موانع لیست غلبه کند، متد `true` برمی‌گرداند. اگر حتی در یک مورد شکست بخورد، باید متوقف شده و `false` برگرداند.

وظیفه قهرمان (کلاس Adventurer)

در نهایت، کلاس `Adventurer` را برای اتصال همه چیز به یکدیگر ایجاد کنید. این کلاس باید شامل موارد زیر باشد:

ویژگی:

- `toolSatchel` : یک لیست از تمام ابزار های موجود در کیف ماجراجو

کانستراکتور:

- با صدا کردن کانستراکتور این کلاس باید به طور پیشفرض یک ماجراجو با کیفی شامل همه ابزار های ممکن ساخته شود.

متد ها:

- `setTools()` : اگر این متد صدا زده شود ابزار داخل کیف ماجراجو با لیست ورودی این تابع جایگزین میشود.
- `traverse(Obstacle obstacle)` : همه ابزار ماجراجو روی مانع مورد نظر امتحان میشود و اگر هیچ کدام کار نکرد در نهایت `false` برمیگرداند.
- `clearLabyrinth(List<Obstacle> labyrinthPath)` : موانع هزار تو را به ترتیب رد میکند.
- `clearRoom(DungeonRoom room)` : برای پاکسازی موانع یک تالار از این تابع استفاده میکنیم که از داخل متد `clearLabyrinth` باید صدا زده شود.

توجه داشته باشید که یک مانع خود میتواند یک تالار یا لیستی از موانع یا تالار های تو در تو باشد!!

```java
public class Adventurer {
    private List<AdventurerTool> toolSatchel;

```

```
    public Adventurer() {
        // Initialize the satchel with one of each tool.
    }
    public boolean traverse(Obstacle obstacle) {
        // Implementation needed.
        // The adventurer should iterate through their tools and try each one
    }

    public boolean clearLabyrinth(List<Obstacle> labyrinthPath){
    }
    private boolean clearRoom(DungeonRoom room){
    }
```

آنچه باید آپلود کنید

ساختار فایل زیپ ارسالی باید به صورت زیر باشد:

```
<zip_file_name.zip>
    ├── Obstacle.java
    ├── AdventurerToolBase.java
    ├── Golem.java
    ├── MagicLock.java
    ├── SpikeTrap.java
    ├── DungeonRoom.java
    ├── AdventurerTool.java
    ├── Warhammer.java
    ├── EnchantedKey.java
    ├── MasterThiefsTools.java
    └── Adventurer.java
```