Java互助课堂（周一

6. 先进行代码鉴赏，然后讲抽象类和接口

徐延楷 a.k.a. Froster
20级的老东西

。。。我仍然讲不明白这个

先来点开胃菜

关于怎么写规范代码

这周终于有人找我debug了，泪目

然后我就可以给大火进行不太好的代码鉴赏环节了，引以为鉴！

以下代码均不是各位同志的原始代码，我都自己改过

一些奇怪的类

```
public class MyCalender {
    private ArrayList<MyCalender> events;
    public MyCalender(int cap) {}
    public MyCalender(String name, MyDate date) {}
}
```

这位同志以为oj只能传两个类，于是ta的MyCalender又是日历又是Event

一个类只能干一件事！！！！！！！！！

btw，一个类`Cls`里面存储自己的引用（有一个成员变量类型是自己（`Cls`）或自己的数组），这是非常罕见的情景，应该只会在数据结构课上见到...

超纲一下（
成员变量其实可以分成两种模型：

- Has-a，你们课上讲的聚合。一辆汽车由轮子组成，少了轮子跑不起来。`class Car { Wheel[4] wheels; }`
- Association（关联）。一个停车场里停着很多车，但是停车场里没车也不妨碍它是个停车场。
  `class ParkingLot { ArrayList<Car> cars; }`

一些奇怪的类

```
public class MyCalender {
    private ArrayList<String> eventNames;
    private ArrayList<MyDate> eventDates;
    // ...
    public String finishNextEvent() {
        String finishEvent = "NONE";
        if (!myList.isEmpty()) {
            finishEvent = eventNames.get(0);
            myList.remove(0);
        }
        return finishEvent;
    }
}
```

这位同志把event的名称和日期分开存了。这么做是没问题的，就是容易出问题。

比如说ta在写`finishNextEvent`的时候只删了`eventNames`，然后debug几小时。

一定要把互相之间关联性很强的数据提取成一个类（

一些奇怪的if语句

```
// addEvent()
if (n ≠ events.size()) {
    // ...
}
if (n == events.size() && (MyDate.difference(event.date, events.get(events.size() - 1)) > 0 ||
    (MyDate.difference(event.date, events.get(events.size() - 1)) == 0
        && event.name.compareTo(events.get(events.size() - 1).name) ⩾ 0))) {
    // ...
}
if (n == events.size() && (MyDate.difference(event.date, events.get(events.size() - 1)) > 0 ||
    !(MyDate.difference(event.date, events.get(events.size() - 1)) == 0
        && event.name.compareTo(events.get(events.size() - 1).name) ⩾ 0))) {
    // ...
}
return true;
```

我看到这一堆的时候非常崩溃，给这位同志简化这段代码的时候还简化错了。

同志们，`else if`和`else`是好东西。一个if里面装着一眼望不到头的一堆条件和好几个括号的时候就该简化了，试着简化逻辑或者提取变量（比如说这一坨中的`events.get(events.size() - 1)`）。

这样除了很容易写错之外，还容易漏情况，比如几个分支都没进去，event没添加就return true了。

一些奇怪的getter和setter

```
public class MyCalender {
    public void setCapacity(int capacity) {}
    public int getCapacity() {}
    public void setEvents(ArrayList<Event> event) {}
    public ArrayList<Event> getEvents() {}
    public void setCount(int count) {}
    public int getCount() {} // 这个和getCapacity问题一样
    // ..............
}
```

这么写的人还挺多。**不是所有的成员变量都需要Getter和Setter的！！！！！！！！不是！！！！！！！！！**

这几个东西的问题如下（count在这里用来记录当前event数量，这个也是有问题的，一会说）。

- `getCapacity`：用户需要知道Capacity吗？题目要求里没有，不需要。这个方法除了冗余以外没其他错。
- `setCapacity`：capacity是只能在构造函数里初始化，之后不能修改吗？如果是，加final，然后删这个。
- `getEvents`：它返回一个ArrayList，引用类型。你的队友拿到之后可以把ArrayList清空，然后你就爆炸了。
- `setEvents`，`setCount`：events和count是这个类里面自己维护的。比如添加一个event就count++。这两就不应该能单独修改！！！！！！！

一些奇怪的getter和setter

```
public class MyCalender {
    public void setCapacity(int capacity) {}
    public int getCapacity() {}
    public void setEvents(ArrayList<Event> event) {}
    public ArrayList<Event> getEvents() {}
    public void setCount(int count) {}
    public int getCount() {} // 这个和getCapacity问题一样
    // ..............
}
```

总而言之，这几个东西破坏了**封装**。

一个实例里面各个`private`成员变量的状态是只能由成员方法维护的。很多变量根本不应该从外部修改（这就是private的意义）。

比如说一台发动机，可以有个`setSpeed()`方法给它调转速，没问题。但不能有个`setGear`在它跑的时候给它把齿轮换了吧（悲

总之上面代码里的`setEvents`，`setCount`一旦调用，百分百破坏程序正常状态。`getEvents`作为返回一个数组的方法，也是有点危险的。如果要获取某个event，建议写`getEvent(int idx)`。

一些奇怪的冗余

```
public String finishNextEvent() {
    String finishedEvent = "";
    if (events.isEmpty()) {
        return "None";
    } else {
        finishedEvent = //  ...
    }
    return finishedEvent;
}
```

```
public String finishNextEvent() {
    if (events.isEmpty()) {
        return "None";
    }
    String finishedEvent = //  ...
    //  ...
    return finishedEvent;
}
```

```
public String finishNextEvent() {
    String finishedEvent = "";
    if (!events.isEmpty()) {
        String finishedEvent = //  ..
        //  ...
        return finishedEvent;
    }
    return "None";
}
```

最左边是原始代码。看上去会有点怪。问题：

- 引用类型变量的初始值应该写null。
- 习惯上来说，变量的定义应该尽量延后。比如这个`finishEvent`在if语句里完全没起作用，但是直觉上会感觉它被赋值了。
- 不是写了if就得写else的。如果在if的最后return了，函数的其余部分其实就是else块。

右边俩是推荐的写法。本质上没区别。

guard clause（卫语句）

```
public boolean addEvent(/* ... */) {
    if (events.size() < capacity) {
        // 很长的东西
        // 很长的东西
        // 很长的东西
        // 很长的东西
        // 很长的东西
        while (/* ... */) {
            // 很长的东西
            // 很长的东西
            if (/* ... */) {
                // 很长的东西
                // 很长的东西
                if (/* ... */) {
                    // 很长的东西
                    // 很长的东西
                }
            }
        }
        // 很长的东西
        // 很长的东西
    }
    return false;
}
```

不说别的，这么一堆缩进看着是不是很乱！

不知道一行代码究竟是属于哪个块，
是最外面的if还是while还是里面的哪个if，
尤其是屏幕装不下整个方法的时候。

在写代码的时候缩进越少越好，并且要尽量避免嵌套超过三四层的缩进。可以这么写：

```
if (events.size() ≥ capacity) { // 注意这里的逻辑反过来了
    return false;
}
// 左边if里面的那一堆
while (/* ... */) {
    if (/* ... */) {
        continue;
    }
}
return true;
```

把条件在开头就判断可以省下很多缩进。更清楚。

英语教学（变量名鉴赏）

非常抱歉！！！！！！！！！！！！！

大伙自己写的时候按自己的习惯就好。我在这举的例子只是不符合我个人习惯的（叠甲

再强调一遍大小写问题，类名首字母大写，变量名和方法名首字母小写

`public int Y()`：还以为是个类

`ArrayList<Event> myList`：改成`events`更好，大概。不是什么东西都要加个my的。

- 代表数组的变量名最好用复数形式，或者写xxxList

`ArrayList<MyDate> calender`：一堆date不是日历...

`String finishEvent`：第一眼还以为是个方法。这里要表述已经完成的event，应该是`finishedEvent`

`int temp`，`int a`（并且在接下来的三十行反复出现）：希望这几位同志第二天记得temp和a啥意思

警惕static

```
public class MyCalender {
    public static ArrayList<Event> events;
}

// oj里面:
MyCalender a = new MyCalender(), b = new MyCalender();
a.addEvent()
b.addEvent()
//  ...
// expect <true> but was <false>
```

大火应该都知道static的意义吧（大概）

注意一下就好

静态绑定和动态绑定

上节课讲完有人问我...我发现直接讲过程更好一点

```
class A {public static a() {} public b() {} }
class B extends A {
    public static a() {}
    @Override
    public b() {}
}

A x = new B();
B y = new B();
```

声明类型

- 调用x.a：a是静态方法。java找x变量的类型，是A。直接调用A.a()。
- 调用x.b：b是成员方法。java找x变量里面**实例**的类型，是B。调用B.b()。
- 调用y.a：a是静态方法。java找y变量的类型，是B。直接调用B.a()。
- 调用y.b：b是成员方法。java找y变量里面**实例**的类型，是B。调用B.b()。

一个面向对象程序设计的例子

先不讲知识点，先吃点栗子

你在写一个2D战棋游戏。游戏包含如下元素：

- 地图，就是一堆格子
- 玩家人物，有血量和护盾，能放技能，站在地图上，能动（有自己的回合）
- 敌人，有血量和护盾，能放技能，有AI，站在地图上，能动（有自己的回合）
- 地图上有障碍物，有的可以破坏（有血量和护盾），有的不可以，站在地图上，不能动（没有自己的回合）
- 一系列炫酷的技能

你找到了一个游戏引擎。它可以帮你做：

- 在每个物体对应的回合中调用一个共有的方法`update()`
- 处理玩家输入

现在需要用一些类来把这个程序写出来（

注意：下文的“是一个”都是“is a”的意思

提取同类项（1）

首先，得先把地图做出来，然后把玩家，敌人，障碍物都放上去

这是一个2D游戏，地图是一堆格子，所以站在地图上的东西都有...xy坐标。

很显然，玩家人物，敌人和障碍物是不同的东西，所以是不同的类。

- 但它们有相同的一点：站在地图上，即，有xy坐标，所以应该有一个基类

```
public abstract class Entity {
    private int x, y;  // and getter
    boolean moveTo(int x, int y) {}
}
// ——————
Entity[][] map;
class Player extends Entity {}
class Enemy extends Entity {}
class Obstacle extends Entity {}
class SomeSkill {}
```

玩家，敌人，障碍物都是（"is a"）能放在地图上的实体。但是技能不是。

anyway，地图有了，各种东西能被放上去了。继续。

提取同类项（2.1）

然后，得让放上去的这堆东西能被打。也就是给它们添加血量和护盾。

和上一张ppt一样：

```
public class Destroyable {
    private int armor, shield; // and getter
    public boolean onHit(int damage) {}
}
// ——————
// class Player extends Entity {} ....?
```

然后就遇到了一个问题：java没有多继承。

Player既是一个可被摧毁的东西，又是一个能放在地图上的实体。咋办？

提取同类项（2.2）

回到需求。我们发现能被摧毁的东西一定是一个能放在地图上的实体。所以：

```
public class Destroyable extends Entity {
    private int armor, shield; // and getter
    public boolean onHit(int damage) {} // 返回true代表这个实体被摧毁了
}
// ——————
class Player extends Destroyable {}
// class Obstacle extends Destoryable {} ....?
```

wait...障碍物不一定是可摧毁的。

```
class DestoryableObstacle extends Destoryable {}
class NonDestoryableObstacle extends Entity {}
```

没完...如果两种障碍物也有同类项呢？比如有一个变量代表障碍物的颜色？

重写，和设计上的取舍（1）

欢迎来到面向对象设计

有很多种方法来不完美的解决上述问题（越上面的越不推荐）：

- ~~java不支持多继承，劳资写c++去了，那个支持~~
- 把障碍物共有的代码复制粘贴——但是这个绝对不好，会改了一个忘改另一个
- 把Destroyable改成**接口**——稍后再讲
- 稍微改变一下“不可被摧毁”的定义。
  - 刚才的设计中，不可被摧毁指完全不能被打。
  - 做一个取舍，把不可被摧毁的意思变成“能被打，但是无敌”

重写，和设计上的取舍（2）

欢迎来到面向对象设计

做一个取舍，把不可被摧毁的意思变成“能被打，但是无敌”

```
class Obstacle extends Destoryable { /* ... */}
class NonDestoryableObstacle extends Obstacle {
    @Override
    public boolean onHit(int damage) { // 返回true代表这个实体被摧毁了
        sout("无效");
        return false;
    }
}
```

也可以：

```
public class Destroyable extends Entity {
    public boolean invincible;
    public boolean onHit(int damage) { // 返回true代表这个实体被摧毁了
        if (invincible) {
            sout("无效");
            return false;
        }
        // ...
    }
```

重写，和设计上的取舍（3）

欢迎来到面向对象设计

如果做了上一页的取舍，我们会发现所有的`Entity`都是`Destroyable`。可以直接把它们俩合并了。

以及，上一页的两种方法的区别：

- 下面那一种直接改了`Destroyable`，它同时影响到了玩家和敌人。
  - 如果后面要设计无敌技能，这样会非常方便
- 上面那一种更像是个临时的取舍。
  - 如果我们的游戏里没有无敌技能，也没有其他不可摧毁的东西，那这种完全够用

写代码突出一个够用就行——

- 如果你的游戏是个课程project，要求已经确定好了，或者你只是随手加一个，其他的事情以后遇到了再改
  - 那想到哪个用哪个
- 如果真的有写无敌技能的需求，或者要添加无敌的中立角色（不是障碍物），同时项目刚启动有时间做设计
  - 后者，或者其他更复杂的解法

接口（1）

先暂停举例子环节，讲一下知识点

```
public interface Comparable<T> {
    public abstract int compareTo(T other);
}

public class Point2D implements Comparable<Point2D> {
    int x, y;
    @Override
    public int compareTo(Point2D other) {
        return (x - other.x) + (y - other.y);
    }
}
```

相比于继承的"is a"，接口的语义更多代表的是“能干什么，但是具体怎么干不知道”

这个语义体现在抽象方法上。比如，人能移动，车也能移动，但是人和车除了能动完全没有相似点，所以移动方法的实现（具体怎么动）得在人和车自己的类里面写。

接口也可以理解为...插座。在使用插座的时候，两根间距合适的条状物都可以被插进插座，无论是充电器还是铁丝还是冰糕棍。插座只给这两个东西提供220V交流电，这个电是用来充电还是电到人还是根本就不起作用，插座不知道。

接口（2）

插座那个例子其实就是多态。同一操作（供电）作用于不同的对象，可以有不同的解释，产生不同的执行结果。

回到我们的例子。玩家和敌人能动，有自己的回合。

一般的游戏引擎里面会这么写：

```
public interface Updateable {
    public abstract void update();
}

ArrayList<Updateable> entities;
int now = 0;
public void nextTurn() {
    entities.get(now).update();
    now = (now + 1) % entities.size();
}
```

对于游戏引擎，它只需要在合适的时机调用“有自己的回合”的东西的`update`方法，其余一概不管

对于写游戏的人，只需要让自己的类实现`Updateable`接口，其余全都交给游戏引擎。

合作起来非常方便。

接口（3）

对于我们的玩家和敌人：

```
class Player extends Entity implements Updateable {
    @override
    public void update() {
        while (true) {
            Point2D pos = getInputPos();
            if (this.moveTo(pos)) break;
        }
        Skill skill = getInputSkill();
        skill.use();
    }
}
class Enemy extends Entity implements Updateable {
    @override
    public void update() {
        Point2D pos = ai.getNextMove();
        this.moveTo(pos);
        Skill skill = ai.getNextSkill();
        skill.use();
    }
}
```

接口（4）

刚才说到，把Destroyable改成接口。

```
// public class Destroyable {
//     private int armor, shield; // and getter
//     public boolean onHit(int damage) {}
// }
public interface Destroyable {
    public boolean onHit(int damage);
}

public class DestroyableObstacle extends Obstacle implements Destroyable {}
```

这样就绕开多继承，解决了又是障碍物又能被摧毁的问题。同时它带来了如下改变：

- 由于接口不能有成员变量，armor和shield得在各个类里自己管理。
  - 造成了代码的冗余，但是...
  - 自由度增加了。现在可以写出一种没有护盾只有血量的东西。

实际的设计中很多这种情况...接口的限制比抽象类要小。灵活使用吧。我总结不出什么时候该用什么。

抽象类（1）

该写技能了...我们的技能长这样

```
public abstract class Skill {
    int damage, range;
    protected boolean checkRange() {}
    public abstract boolean use(Point2D dst, Entity src);
}
public class SkillA {
    @Override
    public boolean use(Point2D dst, Entity src) {
        if (!checkRange(dst, src))
            return false;
        Map.get(dst).onHit(damage);
    }
}
public class SkillB {
    @Override
    public boolean use(Point2D dst, Entity src) {
        foreach (i : src.pos → dst) { // 伪代码，意思到了就行
            Map.get(i).onHit(damage);
        }
    }
}
```

抽象类（1）

该写技能了...我们的技能长这样

```
public abstract class Skill {
    int damage, range;
    protected boolean checkRange() {}
    public abstract boolean use(Point2D dst, Entity src);
}
public class SkillA {
    @Override
    public boolean use(Point2D dst, Entity src) {
        if (!checkRange(dst, src))
            return false;
        Map.get(dst).onHit(damage); // 应该检查是否为null
    }
}
public class SkillB {
    @Override
    public boolean use(Point2D dst, Entity src) {
        foreach (i : src.pos → dst) { // 伪代码，意思到了就行
            Map.get(i).onHit(damage); // 应该检查是否为null
        }
    }
}
```

抽象类（2）

抽象类代表...一个一般的概念。和父类一样。

不过这个概念太一般了，以至于不应该被实例化。

你知道它的一些属性，一些应该有的方法。有的方法可以被实现，有的方法太不具体了，它没有一个默认行为。

比如，上节课例子里象棋的棋子。你知道棋子有一个坐标，你也知道棋子能动。

- 但是“棋子”怎么动？没法知道“棋子”怎么动，它只是个概念。
- 我们可以知道“车”“马”“象”这种更具体的棋子怎么动。

所以“动”是个抽象方法。放在这里等待子类去实现。这一块和接口的抽象方法是一样的。

也有没有抽象方法的抽象类。比如我们的`Entity`。

- `Entity`只有xy坐标，它也知道自己怎么移动。
- 但是它只是个一般化的概念，正如我们不能拿出一个抽象的象棋棋子。所以它应该是abstract class。

强制类型转换和instanceof

如果我们的Destroyable是接口，那技能的代码会出问题：

```
Map.get(pos).onHit(damage);
```

`Map.get(i)`返回一个Entity。在当前的设计里，Entity不一定Destroyable。怎么办？

```
Entity e = Map.get(pos);
if (e instanceof Destroyable d) {
    d.onHit(damage);
}
```

用instanceof做一个检查。

```
if (e instanceof Destroyable d)
```

是

```
if (e instanceof Destroyable)
Destroyable d = (Destroyable)e;
```

的简写。

public static变量（1）

又是tradeoff

最后来个和继承没有关系的话题。简单的深入一下`moveTo`方法：

```
public class Entity {
    int x, y;
    public void moveTo(Point2D pos) { //忽略掉边界和重叠检查
        map[pos.x][pos.y] = map[x][y];
        map[x][y] = null;
    }
}
```

等下，map从哪来？

- 作为Entity的成员变量，但是创建Entity的时候map从哪来？
  - 也许其他地方也要用map。比如ai，地图绘制，游戏初始化。把map作为很多东西的成员变量会很复杂
- 作为全局变量`public static Entity[][] map;`
  - (+)在程序的所有地方都能找到，非常方便，不需要管理传参和成员变量
  - (-)在程序的所有地方都能修改。会比较混乱。还可能会出现调用时没有初始化的问题。

public static变量（2）

又是tradeoff

可能我在这说没法说明白，等你们写project的时候就会懂了...

总之，这两种方法都有其复杂性和不好的地方。

但是我个人推荐开全局变量，然后在写代码的时候小心管理它。

如果要对map做控制（避免被你的队友`Map.map = null`毁掉程序，可以用private static配上public static方法：

```
public class Map {
    private static Entity[][] map;
    public static Entity get(int x, int y) { return map[x][y]; }
}
```