Java互助课堂（周一

4. 面向对象

徐延楷 a.k.a. Froster
20级的老东西

抽象，继续抽象

为啥要面向这个对象。。。

其实是面向数据和类型

上次课的这个例子：这个方法要返回一个二维的点。

```
int[] getNewPosByChar(char command, int x, int y) {
  switch(command) {
    case 'q':
      return new int[]{x - 1, y - 1};
    ...
  }
}
```

我们在这里用一个`int[]`代表点。这时候就出现了一个问题：这玩意不是数组吗？？？

数组是用来表示一堆**不定长**的数的，在这里表示点不合适。

而且这个数组的第0项是x还是第1项是x，这个点是二维的还是三维的，这个返回值完全看不出来。

为啥要面向这个对象。。。

其实是面向数据和类型

第二节课讲类型的时候有提到：

```
char是字符，short是数。

当我们看到一个变量的时候，它的类型会给我们一些额外的信息，比如：
- 它里面存储什么数据？
- 它在这里是被用来干什么的？
- 它能支持什么操作？
```

`int[]`在这个场景给不出什么信息。 很自然的，我们在这里需要一个新类型，代表：

- 这个类型里面存储int类型的坐标`x`，`y`。
- 这个类型用来表示一个二维的点
- 没啥操作，我只想表示一个点

基本类型和数组干不了这个事情。该**创建新数据类型了**。后文会把这些数据类型统称为“类”(class)。

class：数据类型的定义

这时候还没到OOP，我在讲class的先人——struct

```
public class Point2D {
    int x, y;
}

Point2D getNewPosByChar(char command, Point2D currPos) { /* ... */ }
```

清楚多了。我们还可以：

```
Point2D[] points = new Point2D[10];
// int[][] points = new int[][10];
```

同样，任何人看到`int[][]`脑子里都是个棋盘一类的东西，而非一堆点。

我们还可以：

```
currPos.x; // currPos[0]
```

一眼就知道`currPos.x`是x坐标，但`[0]`就看不出来。

数据的聚合

点是x, y两个int，其他东西呢？

```
public class Point2D {
    int x, y;
}
```

可以看到Point2D里有两个成员变量(instance variables, member, field, ...)，说明一个点由两个数组成。

假设要定义一个棋类游戏的游戏玩家类。一个抽象的正在下棋的游戏玩家由什么“数据”组成呢？

- 昵称：`String`
- 先手还是后手：`boolean`，代表是否为先手
- 此前的胜场：`int`
- 是否为AI：`boolean`
- ...

把这些数据**统一管理**起来，就是我们的`Player`类。

还是数据的聚合

class套class

一个复杂的项目会由很多类组成...

玩家定义完了，该棋盘了。棋盘这东西简单，一堆格子，格子里面能装棋子`Piece`。

这时候就要看需求了——

- 你在下国际象棋。格子就是格子，除了放棋子没有其他用处。
  - 二维数组就是个很好的格子。数组的每一项要么有东西要么没有。
  - `Piece[][]`。格子里面要么有棋子，要么没有（即`null`）。
  - 棋子的颜色和类型怎么看？棋盘就是个数组而已，问里面装的棋子（即`pieces[0][0].color`）。
- 你在下斗兽棋。格子分陆地河流陷阱还有大本营。
  - `Piece[][]`不够用了。这个只能表示某个位置有没有棋子。
  - 来个Grid类。然后`Grid[][]`。`Grid.type`表示格子类型，`Grid.piece`表示格子里面的棋子（可为`null`）。

还是数据的聚合

class套class

全写下来，就是：

```
class Piece { /* ... */ }

class Grid {
    int type; // bad, 正确写法暂时超纲了
    Piece piece; // 可以为null
}

class Board {
    // Piece[][] pieces;
    Grid[][] grids;
}
```

一点有趣的东西：grids里的每一项都不能为`null`。棋盘怎么能缺格子呢？

但是piece可以，格子上没棋子太正常了。

写代码的时候脑子里要有这个意识...关于什么能是null什么不能是。如果能是，最好写个注释。

实例

int和x的区别

```
Point2D point;
```

我们刚才说的class是类型，和int一样。类型是个概念，不是实在的东西。

举个例子：

- 一个点的x坐标值是什么？不知道。我只知道一个点有xy，具体哪个点的坐标？
- `point`的x坐标值是什么？`point.x`。

实例化(Instantiation)：把这个具体的东西创建出来。就是，`new`一个。

下面会称这个具体的东西为实例(Instance / Object)。

btw，湾湾那边的翻译。class：类别。(Instance / Object)：物件。

构造器(constructor)

数据怎么装进去?

```
public class Point2D {
    int x, y;
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

Point2D point = new Point2D(0, 1);
```

构造器/构造方法用来初始化一个实例。

比如说这个`Point2D`，一个点肯定要有它的xy坐标，所以我们写一个构造器，传入x，y两个值，然后把它放进成员变量里。

构造器

初始化

```
public class Board {
    Grid[][] grids;
    public Board() {
        this.grids = new Grid[10][10];
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                this.grids[i][j] = new Grid(i, j);
            }
        }
    }
}
```

再比如说这个Board。假定我们默认棋盘是10*10的。

每一个board被创建的时候，它的格子需要被初始化好（棋盘必须要有格子，不存在没有格子的棋盘）。

我们在构造器里做这个。确保每一个实例创建好之后不会有格子不存在的情况。

构造器的作用...初始化你操作这个实例所需要的一切东西。

很多构造器

默认参数

有时候你只是想创建一个点，xy坐标过会再填进去（bad idea）

或者一个类可以用默认值初始化，但有时候也需要传进去参数
（比如你要画一个矩形，这个矩形默认是红色，但你可以指定其他颜色

```
public class Rectangle {
    Color color;
    public Rectangle() {
        this(Color.red);
    }
    public Rectangle(Color color) {
        this.color = color;
    }
}

Rectangle rect1 = new Rectangle(Color.blue);
Rectangle rect2 = new Rectangle();
```

重载一个构造器就行。调用哪个构造器取决于new的时候传什么参数。和重载方法一样。

this？？？

zssm

上几页出现的this是指“当前实例”。

```
public class Point2D {
    int x, y;
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

Point2D point1 = new Point2D(0, 1);
Point2D point2 = new Point2D(1, 3);
point.x;
```

在构造方法外面，你知道新创建的这个实例叫`point`。在构造方法（以及所有成员方法）里面，你不知道。

所以需要this。注意，`new Point2D(0, 1)`和`new Point2D(1, 3)`是两个不同的实例。
所以这里this指代的是两个不同的东西。

成员方法

这里开始OOP

回到灵魂三问：

```
char是字符，short是数。

当我们看到一个变量的时候，它的类型会给我们一些额外的信息，比如：
- 它里面存储什么数据？
- 它在这里是被用来干什么的？
```

它能支持什么操作？

拿之前的类型举点例子：

- `int` 可以加减乘除
- `String` 可以取某一个字符`charAt()`，可以取子串`substr()`
- `point` 目前啥也做不了。
- `Board` 目前也啥都做不了，这肯定不行。

成员方法

这里开始OOP

成员方法用于实现类型应该支持的操作。

比如，我们的`Grid`类需要能够判断自己上面有没有棋子：

```
class Grid {
    int type; // bad, 正确写法暂时超纲了
    Piece piece; // 可以为null
    public boolean hasPiece() {
        return piece == null;
    }
}
```

这里的`hasPiece`方法使用实例的成员变量piece，判断这个变量是否为null。逻辑很简单。

```
Grid grid = board.grids[0][0];
grid.hasPiece(); // ok
// Grid.hasPiece(); // err
```

需要一个实例来调用成员方法。上面的“实例”一页讲了为什么。（需要一个具体的格子才能知道里面有无棋子）

等一下...为什么要成员方法?

OOP的理念其一

```
class Grid {
    int type; // bad, 正确写法暂时超纲了
    Piece piece; // 可以为null
}

public static boolean hasPiece(Grid grid) {
    return grid.piece == null;
}

hasPiece(grid);
```

我们完全可以写一个非成员方法来做这件事，为什么要搞成员方法?

因为...OOP的理念是把**数据**和**对数据的操作**都放在一起，扔进一个类里。

这么做的优点:

- 方便找（对这个类型的所有操作都写在这个类里）
- 访问控制
- 封装

访问控制

不造屎山的尝试

```
public class Board {
    public Grid[][] grids;
}
```

这个类设计的时候，对于grid变量的使用方法只有一种：每回合把里面的棋子搬来搬去。

```
board.grids[i][j].piece = board.grids[m][n].piece;
board.grids[m][n].piece = null;
```

你的队友有一天酒后写代码，写出了惊世骇俗的：

```
board.grids = new Grid[10][10];
```

你合并了他的代码，观察到玩家走了几步棋之后，整个棋盘都被清空了！！！！！！

于是你debug十小时，发现了这一行。你的心态崩了。

程序员们喜欢偷懒，于是有了一劳永逸解决问题的方法：让`board.grids =` 报错。只能通过特定方法改grids。

访问控制

不造屎山的尝试

```
public class Board {
    private Grid[][] grids;
    public void movePiece(int i, int j, int m, int n) {
        grids[i][j].piece = board.grids[m][n].piece;
        grids[m][n].piece = null;
    }
}

// board.grids[i][j] // err! private access.
board.movePiece(0, 0, 1, 1);
```

`private`关键字：让一个成员变量只能被成员方法访问。

然后，通过限制成员方法能做的事，就实现了...对“如何访问和修改数据”的限制。

就像`String`没有`setChar()`方法，不支持改字符串里的单个字符一样。

btw，能private变量自然也能private方法，意思是这个方法只能被成员方法访问。

Getter & Setter

还是访问控制

```
public class Player {
    private String name;
}
```

Getter和Setter本质上就是上一页的内容——你的成员变量在访问和修改时需要受限。

比如说，这个name只能在构造器里被设置，一旦被设置就不能被修改。那么就是只有Getter没有Setter

```
public class Player {
    private String name;
    public String getName() { return name; }
}
```

又比如说，name还是不能被修改，这次获取只能统一获取information而非单独的name，那么就是：

```
public class Player {
    private String name;
    private int id;
    public String getInformation() { return name + " " + id; }
}
```

Getter & Setter

还是访问控制

又比如说，name能单独获取能被修改，但修改时需要检查格式：

```
public class Player {
    private String name;
    public String getName() { return name; }
    public void setName(String name) {
        if (checkFormat(name))
            this.name = name;
    }
}
```

以及很常见的name能单独获取能被修改并且没有任何其他限制：

```
public class Player {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

这里不把name直接搞成public的原因...习惯，以及如果以后突然要改getter和setter的行为，方便改。

final

好用，多用

这个name只能在构造器里被设置，一旦被设置就不能被修改。那么就是只有Getter没有Setter

```
public class Player {
    private String name;
    public String getName() { return name; }
}
```

但是有一个问题：成员方法还能修改name。

```
public class Player {
    private final String name;
    // public void method() { name = ""; } // err!
}
```

把name设为final，就完全满足“必须在构造器里被设置，一旦被设置就不能被修改”。

好用。防止你队友乱改。

封装

和刚才的差不多

这个还是上面的内容——你的成员变量在访问和修改时需要受限。

不过除了防止你队友以你意想不到的方式修改成员变量，它的另一个好处是节约你队友的脑细胞。

你的队友不需要知道你的类是怎么运作的，只需要知道它暴露了哪些方法就可以了。

还是拿`String`举例子，你不需要知道它是怎么存的字符串。你只需要知道你可以charAt，可以拼接就ok了。

对封装正式一点的说法：

- 将使用者所不需要知道的细节隐藏，只暴露需要了解的内容和功能。
- 类的内部数据及操作细节在类内完成，不允许外部干涉，仅对外暴露必要的方法用于使用

非成员变量和方法

把不知道怎么排顺序的内容全都放到最后

讲OOP之前的方法都是这么写的：

```
public static void f() {

}
```

`static`关键字代表“不访问成员变量，和实例没有关系”。

比如`Math.abs`。它只是接受一个数输出一个绝对值，和什么类的什么实例一点关系都没有。

`System.out.println`也是。

调用这种方法只需要`Class.methodName()`就好，类名+方法名。类名只是用来定位的。

另外，还有全局变量`public static int a;`这个代表“该变量全局可见”。

访问这个变量：`Class.a`，类名同样只是用来定位的。

如果你有一个程序到处都用到的工具方法，或者到处都用到的变量，static！

成员变量和方法（作为上一页的对比）

FSM...

`static` 关键字代表“不访问成员变量，和实例没有关系”，
普通方法就代表“访问成员变量，和实例有关系”

我们可以把一个类看成**一种**物品，实例看成**一个**物品的当前总的状态（不同物品当然状态不同）

那成员变量可以看作这些状态的描述，成员方法用来改变这些状态。

比如：

```
public class Car {
    float velocity;
    int passeger;
    void accelerate();
    void addPassenger();
}
```

描述一辆车的状态：当前速度，上面坐着几个人。通过两个方法更改这两种状态。

大半夜写的，很抽象，领会一下意思就行，没领会也没关系。

注意事项

希望你们没忘上节课的内容...

```
public class A {
    int[] ref;
    public A(int[] a) { ref = a; }
}

int[] arr = new int[]{1, 2, 3};
A a1 = new A(arr);
A a2 = new A(arr);

a1.ref[0] = 10;
// a2.ref; {10, 2, 3}
```

成员变量和普通变量还有函数传参什么的一样，该引用类型还得是引用类型！！！！！！