Java互助课堂（周一

4. 继承

徐延楷 a.k.a. Froster
20级的老东西

。。。我讲不明白这个

枚举

先来点开胃菜

举个例子：

- 一个游戏角色有三种状态：走，跑，站着不动。
- 一个棋子有三种颜色：黑或者白或者红

我们要用一个变量记录上面的内容。上节课的例子里用`int`来记录，0代表一种，1代表另一种。这个得用大脑翻译一下，不太方便。

```
public enum Color {
    BLACK,
    WHITE,
    RED
}
Color color = Color.BLACK;
```

这种情况需要定义一个枚举类。color变量只有三种情况：黑，白，红。非常直观。

ArrayList

变长数组

```
ArrayList<Integer> a = new ArrayList<>(); // a: []
a.add(1); // a: [1]
a.add(2); // add是添加到末尾。a: [1, 2]
a.size(); // a里面有两个元素，返回2
a.get(0); // 等价于数组的arr[0]
a.indexOf(2); // 返回第一个值为2的下标。第一个2在下标为1的地方，所以返回1.
a.remove(0); // 去掉0号元素。 a: [2]
a.clear(); // 清空。 a: []
```

注意：循环删ArrayList的元素要从后往前删

```
ArrayList arr = ...; // arr: [1, 2, 3, 4, 5, 6]
for (int i = arr.size(); i >= 0; i--) {
    if (...) arr.remove(i);
    // 假设删除元素3: [1, 2, 4, 5, 6]。
    // 此时i为2，循环结束。如果是从前往后删，直接i++，i为3。arr.get(3)是5，4被跳过去了!
}
```

尖括号和Integer稍后再讲。我是搞不懂为啥老师在讲继承之前就讲wrapper class。。。

Package

包

其实就是文件目录。想象一下你的项目里有114514个文件，把它全堆在一个文件夹里非常难找。

显然，分个类会好一点。所以我们就有了诸如`java.lang.System`和`java.lang.Math`等等。

这个本质上是一堆文件夹和文件：

```
java
├─ lang
│  ├─ System.java
│  ├─ Math.java
...
```

Package

IDEA会帮你管好一切，除了考试

对于你自己在idea里新建的项目：

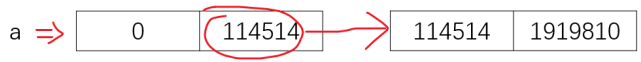
```
src
├─ Main.java
└─ utils
      └─ Test.java
```

java的根目录（从哪个文件夹开始找文件）是src。所以：

- Main.java在根目录，不属于任何包，不用写package。
  - 相对路径：./Main.java
  - 不属于任何包，不能import
- Test.java在utils文件夹下，属于`utils`包，开头写`package utils;`。
  - 相对路径：./utils/Test.java
  - `import utils.Test;`
- `import utils.*;` import utils文件夹里面所有的类

堆内存和栈内存

你们竟然讲这个了，还好我前两节就讲过了



回到这个图。。。

基本类型变量直接代表一段内存，引用类型变量先代表一个地址，通过这个地址再代表一段内存。(1)

还记得这个不?

所有变量**直接**代表的内存都在栈（stack）里，**间接**代表的内存都在堆（heap）里。

由于(1)这句话，也可以说所有的基本类型变量的值都在栈里，所有引用类型变量的值都在堆里。

why? （以下不考）

栈内存适合放生存时间短的，长度比较小而且长度确定的东西，堆内存相反。

基本类型变量和内存地址（“变量”本身）占用内存比较小的，而且是定长的。生存时间很短，出作用域就没了。

引用类型变量的值由于可以被多个引用类型变量指向，生存时间可能很长，而且可能会很大（比如长数组）。

继承

难以理解的东西

假设你在写一个国际象棋程序，你要写一个方法，返回这个棋子能走到哪些格子里。

```
public class Piece {
    private Type type;
    Point2D[] canMoveTo() {
        if (type == Type.KNIGHT) {
            // ...
        } else if (type == Type.BISHOP) {
            // ...
        } else ...
    }
}
```

这太不优雅了，`canMoveTo`方法可能会有几百行。把每种棋子单独变成一个类会好一些。

另一个问题：兵是可以升变的，其他棋子不行。如果给Piece类加入`promote`方法，会是一个不太好的设计：

- 成员方法用于实现类型应该支持的操作。换言之，类型不应该支持的操作不应该有一个成员方法。
- 举个例子：一个不懂国际象棋的人看到棋子有`promote`方法，直觉上会认为所有棋子都能升变。这不好。

继承

难以理解的东西

所以：

```
public class Pawn {
    Point2D[] canMoveTo() {
        // ...
    }
    void promote() {
        // ...
    }
}
```

```
public class Knight {
    private Type type;
    Point2D[] canMoveTo() {
        // ...
    }
}
```

现在每个棋子的move方法都被拆出来了。简洁明了方便修改。之前的问题解决了。

但是...

```
Piece[][] board;
```

棋盘咋办？数组只能存一种数据类型。不能既存`Pawn`类又存`Knight`类。

继承

难以理解的东西

java解决这个问题的方法是，额外声明`Pawn`是一种棋子，`Knight`也是一种棋子。

```
public class Piece { public abstract void move(); }
public class Pawn extends Piece {}
public class Knight extends Piece {}
Piece[][] board;
board[0][0] = new Pawn();
board[0][0] = new Knight();
```

所谓的 "is a" 关系。我们管这个叫 **继承**（inherit）。

虽然它叫继承，但我不太喜欢这个名字...只给人一种接受的感觉。可能特化（specialization）更准确一点。

建议大家忘记继承在中文里的意思，这么来理解这个概念：

- `Pawn`和`Knight`是**特殊**的`Piece`，`Piece`是**一般**的`Pawn`和`Knight`。
- `Piece`是`Pawn`和`Knight`共有的部分（都能动，都有xy坐标）。
- `Pawn`和`Knight`是具体的独特的`Piece`（移动方式不同，兵能升变）。

一些概念

我不知道先讲例子再讲概念是不是好的讲法，但是无所谓了

```
public class Pawn extends Piece {}
public class Knight extends Piece {}
```

- `extends`关键字：用来声明`Pawn`继承`Piece`。
- `Pawn`是`Piece`的子类（subclass）。
- `Piece`是`Pawn`的父类（superclass）。

继承都继承些什么？

继承都继承些什么？

所有变量和方法...

理清关系！

- 但是有些不可访问
- 父类的private成员变量和方法不可访问
- 父类的protected和public成员可以访问
- 类的外面不能访问protected成员

为什么是“所有”？

- 刚才说子类"is a"父类，或者子类是父类的特殊化
- 那子类理应拥有父类的全部特性
- 比如，学生是人，那学生必然有人的全部特征

```
class A {
    public int x;
    protected int y;
    private int z;
    public void f1() {}
    private void f2() {}
}

class B {extends A
    public f() {
        sout(x); // ok
        sout(y); // ok
        // sout(z); // err, 子类不能访问父类的private成员
        f1(); // ok
        // f2(); // err, 子类不能访问父类的private成员
    }
}

psvm() {
    B b = new B();
    // b.y; // err, 只有自己和子类内部能访问protected成员
}
```

protected

还是想不出副标题

何时使用protected?

- 一个成员变量需要子类来修改

```
class Car {
    protected int velocity;
    public abstract void accelerate();
}
```

速度是车共有的，应该在父类`Car`里。加速是每种车（子类）不同的，改`velocity`是在子类改。
所以子类需要能访问父类的velocity，这个成员得是protected。

子类允许修改

- 一个方法仅供子类使用
  - 想不出好的例子，总之就是不想让外界看到但是想让子类看到

和这两件事无关的成员变量全部private

@Override

方法重写

```
class Base {
    public void f() {} // f1
}
class Derived extends Base {
    @Override
    public void f() {} // f2
    public void test() {
        super.f(); // f1
        f(); // f2
    }
}
Base b = new Base();
b.f(); // f1
Base b1 = new Derived();
b1.f(); // f2
```

父类本身存在一个方法f。但是，子类和父类不同，方法f要表现的和父类不一样。（比如equals）

要做到这个，子类需要重写方法f。重写(字面义)。调用的时候，子类实例调用子类的f，父类实例调用父类的f。注意，是**实例**，不是实例被装在哪里（**被装在**Base**变量里的**Derived**实例还是个**Derived**实例**

@Override

方法重写

```
class Base {
    int x; // x1
    public void f() {} // f1
}
class Derived extends Base {
    int x; // x2
    @Override
    public void f() {} // f2
    public void test() {
        x; // x2
        super.x; // x1
        f(); // f2
        super.f(); // f1
    }
}
```

`@Override`是个注解，写在方法前面，用来提示程序员这个方法是重写的。可写可不写。

如果要在子类内部调用f，由于子类里面有两个f，重名了，我们需要加一个super来区分是子类的还是父类的。变量同理。

考试卷子可能会用静态方法恶心你

哎，出题人

```
class Base {
    public static void f() {} // f1
}
class Derived extends Base {
    public static void f() {} // f2
    public void test() {
        f(); // f2
        super.f(); // f1
    }
}
Base b = new Base();
b.f(); // f1
Base b1 = new Derived();
b1.f(); // f1
```

静态方法

调方法

变量类型

注意，这里的f变成了静态方法。静态方法和实例没有关系——

刚才是子类实例调用子类的f，父类实例调用父类的f，现在没有实例了，所以是子类**变量**调用子类的f，父类**变量**调用父类的f。或者说，什么变量调用什么的f。把b1替换成它的变量类型`Base.f()`，就搞得清楚了。

方法重写的细节

```
class Base {
    protected Object f(int x) {} // f1
}
class Derived extends Base {
    @Override
    public String f(int x) {} // f2
}
```

f2的访问控制符可以比f1更宽容，反之不行（private < protected < public

- 这是因为子类是一个父类，父类能干的事子类也必须能干
- 假设f1是public，f2是private，那父类能干的事子类干不了了（子类的同一方法不再能从外部访问），所以不能有更严格的限制。

两个f的参数必须完全一致

f2的返回值可以比f1更具体，即，f2的返回值可以是f1的返回值的子类

- 你说得对，但是我也是做课件的时候才知道。写代码从来没这么写过

子类构造器

其实继承就是子类实例里装了一个父类实例

```
public class Base {
    public Base(int x) {}
}

public class Derived {
    public Derived(int x, int y) {
        super(x);
        // ...
    }
    public Derived(int x) {
        this(x, 1);
        // ...
    }
}
```

如果父类有构造方法，子类实例化的时候需要调用父类构造方法。

在每个构造函数的第一行写一个`super()`来调用。

继承的用处

由于继承的特性，首先我们又可以少写代码了...

- 子类继承父类的成员，所以不需要把重复的部分在子类里写很多遍，只在父类里写就ok
  - 比如棋子的坐标和颜色，get、set坐标和颜色的方法
- 可以把子类特殊的部分安排到子类里面，方便找
  - 你也不想debug一个几百行的方法吧.jpg

因为"is a"这个关系：

- 外界调用成员方法的时候，不需要知道某个特定种类的棋子怎么动，只需要知道它是个棋子可以动就ok了
  - 有一个`Piece p`。`p`可能是任何种类的棋子。你在写`p.canMoveTo()`的时候不用关心到底是哪种棋子。
- 一个类可以方便的被更换和拓展
  - 我们在用一个类`A`的时候，一般只通过一系列方法和它交互
  - 如果有另一个类`B`，同样有这一系列方法，名字还一模一样，那它可以无缝的替换`B`。

下节课会复读一遍，因为我写完这个之后发现它有个名字叫多态。你们还没讲。

里式替代原则（Liskov Substitution Principle）

放在这，理解就好

这个是刚才讲的那一堆东西的不说人话版本：

如果有`class Derived extends Base {}`：

- 每一个类型为`Derived`的对象同时也是一个类型为`Base`的对象，反之不成立
- `Base`比`Derived`表现更一般化的概念，`Derived`比`Base`表现更特殊化的概念
- `Base`对象可派上用场的任何地方，`Derived`对象一样可以派上用场，因为每一个`Derived`对象都是一个`Base`对象
- 反之，如果你需要一个`Derived`对象，`Base`对象不能派上用场

应该算好理解的...吧？

写代码的时候尽量不要违反这个。

类型转换

你们下节lec的东西

```
class Base {/* 空的 */}
class Derived extends Base {
    public void f() {}
}
Base b = new Derived();
// b.f(); // err!
```

虽然b里面是Derived实例，Derived类有f方法，但是我们不能调用，因为Base里没有f方法。

在把Derived装进b的时候，丢失了它的特殊性。

如果一定要调用f，得类型转换一下

```
((Derived) b).f(); // ok!
```

类型转换

...但是尽量少用

```
Base b = new Base();
// ((Derived) b).f(); // err! ClassCastException
```

但是，Base里不总是一个Derived实例。如果不是，转换时会抛出ClassCastException。你的程序爆炸了。

如果要检查这个，有一个instanceof操作符：

```
if (b instanceof Derived) {
    ((Derived) b).f();
}
```

这个用来检查你的b里面是不是一个Derived实例。如果是，返回true。有点像`==`。

强烈不推荐使用这两页的东西。尽量不要用，尽量不要用，尽量不要用。这个代表你的代码出现了设计问题。

一般对于这种问题的解决方法是，加个`interface`，你们还没学。

但是设计问题还挺常见的。如果没有其他的解决方法，或者你们ddl快到了，还得是它。

Object类，包装类，ArrayList，以及泛型的介绍

我不知道你们会不会讲泛型，但我得回收开头的那个遗留问题

```
ArrayList<Integer> arr;
```

我们讲piece的时候说过，需要用继承来解决/用一个类的变量/装/很多不同类的实例/的问题。（斜线是断句

arraylist啥都能装，用的也是一样的机制。所有java类（即使你没extends）都继承自`Object`类。

Object类自带一些方法，比如`equals`等。

ArrayList里面其实就是一个`Object[]`，这样就啥都能装了。

这样带来两个问题：

- int啥的是基本类型，基本类型不是类放不进去
  - 有包装类`Integer`。它和int完全一样，也会自动互相转换（所谓拆箱装箱）。
  - 把基本类型用包装类装一下，这样就能放进Object数组里了。
- 这个数组里面真的啥都能装。如果你的队友偷偷塞了一个String进去，你当成Integer处理，程序爆炸。
  - 泛型。`ArrayList<Integer>`。尖括号里代表这个ArrayList只能装Integer。

画饼环节

鸽子的自我修养

虽然有两周没开互助课堂，但有一名助教还是在拖ddl

我的意思是，他拖到了上课的当天才开始做ppt

我不说这个助教是谁

作为填充上课内容的补救，他决定给大伙看看这次的assignment

这次assignment应该已经面向对象了，让我康康你们的设计正不正常（确信

还没写完的可以跑路了，不想给我看的也可以跑路了

以及，等讲完面向对象的所有课（抽象类，接口啥的），我应该会翻一个陈年老项目出来，带大火从头设计一下里面的每个类

饼得画在这，要不我会鸽的