

Android秋季第3节课-面向对象进阶

写在最前面：谈两句题外话

- 我们上传的一些资料下载量明显过少，同学们一定要主动的去学习
- 先搞懂逻辑，再去实现代码
- 这个课件里面有很多拓展的内容，现在不要求全部掌握（以后可以回来看看，也许会有新的收获？）

在开启接口和抽象类的飞行模式之前，先复习和补充一些上节课的内容，方便大家这节课理解

继承

对某一类事物的抽象

举个例子

移动开发部有不少学姐，她们有着不同的性格，但是也有着不少相同之处，比如她们都会写代码，都会一种或者几种代码，都有自己的名字

接下来开始对她们进行抽象！

让我们抽象出移动开发部成员的**属性和方法**：

Tip: 这里插句嘴，为啥要是属性和方法？因为这个是类的基本构成单位，可以说类和方法就是对象的血和肉

属性方面，我们六个人的共有特征有：年龄，姓名，会的代码种类。

方法上，我们六个人共有的特征有：吃饭，睡觉，写代码

基于上面这些我们抽象出来的共有特征，就可以创造出一个红岩Android开发者类

```
public class AndroidDeveloper {
    private String name; //开发者的名字
    private String[] programs; //会的编程语言
    private int age; //开发者的年龄

    //构造方法
    public AndroidDeveloper(String name, String[] programs, int age) {
        this.name = name;
        this.programs = programs;
        this.age = age;
    }

    //公有的写代码方法
    public void code(){
        System.out.println(name + "正在写代码");
    }

    //公有的吃饭方法
    public void eat(){
        System.out.println(name + "正在吃饭");
    }
}
```

```

//公有的睡觉方法
public void sleep(){
    System.out.println(name + "正在睡觉");
}

//get set方法
public void setName(String name) {
    this.name = name;
}

public void setPrograms(String[] programs) {
    this.programs = programs;
}

public void setAge(int age) {
    this.age = age;
}

public String getName() {
    return name;
}

public String[] getPrograms() {
    return programs;
}

public int getAge() {
    return age;
}
}

```

然后我就可以基于这个类（当作父类）去构造其他的类（比如张涛类，张煜类这些，可以继承自这个父类）

我来写一个张煜（RayleighZ）类

```

public class RayleighZ extends AndroidDeveloper {

    public RayleighZ(String[] programs, int age) {
        super("RayleighZ", programs, age);
    }

    //添加独特的方法
    public void lickSisterTaoTao(){
        System.out.println("涛の姐世界第一です");
    }

    //覆写父类中的方法
    @Override
    public void code(){
        //Tip: 这里不能通过this直接访问父类的私有域，必须要通过父类提供的get和set接口来获取
        System.out.println(this.getName()+"在划水");
    }

    //不覆写吃饭方法

```

```
//在原有的基础上拓展sleep方法
@Override
public void sleep(){
    System.out.println("睡之前听6小时摇滚");
    super.sleep();
}
}
```

这里针对三个方法我采用了三个不同的策略

code()方法：完全覆写，将父类代码中的东西一点不留

sleep()方法：进行补充，保留父类代码中的东西，并且在父类代码之前进行了补充

eat()方法：不做任何改变，就按照父类中的代码执行

Tip：关于@Override注解

它的作用如下

1. 提高可读性，告诉阅读代码的人这里是重写了父类的方法
2. 编译器会校验写的方法在父类中是否存在，如果这个方法在父类中不存在，就将报错

主函数中测试一下

```
public class Main {
    public static void main(String[] args) {
        //新建一个张煜
        RayleighZ rayleighZ = new RayleighZ(new String[]{"Java", "kotlin"} , 10);
        rayleighZ.eat();
        rayleighZ.sleep();
        rayleighZ.code();
        rayleighZ.licksisterTaoTao();
    }
}
```

输出结果如下：

```
RayleighZ正在吃饭
睡之前听6小时摇滚
RayleighZ正在睡觉
RayleighZ在划水
涛の姐世界第一です
```

为什么要这样写

为什么我需要对对象进行抽象？为什么我不直接写一个张煜类然后在里面定义这些方法？

通过写父类然后继承的优点大概有这些（不扯太玄乎的）

- 方便代码重复使用：就比如我和涛姐的吃饭方式是差不多的（虽然实际上不是这样），如果不使用父类，而是单独的定义张涛类和张煜类，就会发现张涛和张煜这两个类的吃饭方法是完全相同的，同样的代码写了两次，这样是很不**优雅**的，我们的程序中应当尽可能少的出现相同的代码，而如果将这种公共的代码写到父类中，子类继承过后就有了一个默认的实现，就不需要再去写这个方法，可以减少代码的重复
- 提供了一个共有的标签：这里举一个例子

红岩的产品大哥需要一位Android开发者去完成掌上重邮代码的编写，对他来说，不管你是谁，你多少岁，只要你会写Android代码（继承自AndroidDeveloper类），就可以完成任务

这时的代码要怎么写？自然是传递一个AndroidDeveloper进去

```
public class ProductManager { //产品经理类

    //安排某个人写代码的方法
    public void letSomeoneCode(AndroidDeveloper androidDeveloper){
        androidDeveloper.code();
    }
}
```

main函数中调用

```
ProductManager pm = new ProductManager();
pm.letSomeoneCode(rayleighZ); //安排张煜去写代码
```

这样写的好处在于，比如我写了一个产品经理类，然后布置作业让你们去自己写一个Android程序员来发给我，我在我的代码中运行，如果不使用父类，我在设计产品经理类的时候就必须确定要传进来的是什么类，举个例子，如果不使用父类，我就只能让张煜或者张涛去写，而不能让具有某种特征的一类人去写。

但是如果你使用了继承的方法去实现，首先你写的类可以千变万化，可以尽情拓展，再次只要是继承了AndroidDeveloper类，这个类就必然有code方法，就算你在子类里面一行都不写，父类中的code方法也会成为子类的一个默认实现。**父类是一种标签**，这个标签标志着子类一定有父类的一些特征，你可以信赖它。

最后一个优点可能稍微有点抽象，没关系，我们讲一讲接口，可以辅助大家理解这个优势所在

接口

什么是接口

先吹一吹概念（摘抄自[菜鸟教程](#)）：

接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过实现接口的方式，从而来实现接口的抽象方法。

看着似乎非常抽象，但是不用怕，咱用简单点的语言来描述一下，首先给大家假设一个场景：

场景A 剧情1：涛涛学姐昨天晚上熬夜熬得太久，需要一位工具人去帮她买一杯咖啡，对于涛涛学姐而言，她需要的是一个能购买咖啡的工具人，具体是谁无所谓，那么涛涛学姐咋知道谁能购买咖啡呢？这个时候就需要接口了

这里的接口更像是一种合同，一个标签，它表示拥有接口的对象具备接口定义的一种能力，比如在场景A中，我们可以定义一种接口，名叫**买咖啡**，如果某个人具有这个名叫买咖啡的接口，那就表明他有为涛涛学姐买咖啡的能力，让我们继续上面的例子

场景A 剧情2：涛涛学姐环顾四周，发现张煜身上赫然贴着一张标签：此人具有买咖啡接口，遂可以认定张煜具有买咖啡的能力，就委托他去买咖啡。

我们总结一下上面的故事

张涛学姐的需求

- 一杯咖啡

- 一个能帮她买咖啡的工具人

买咖啡接口规定了啥

- 所有具实现了咖啡接口的人都有能力帮别人买一杯咖啡

张煜的特点

- 他具有买咖啡接口

所以，张涛学姐就可以放心的找工具人张煜，让他帮自己买一杯咖啡

进一步抽象一下上面的故事，接口声明了一种能力，拥有某种接口的对象一定具备这种能力

代码实现

接下来上代码，看看上面的故事如何用代码实现

1 声明买咖啡的接口

```
public interface BuyCoffee { //买咖啡的接口
    void buyCoffee(); //定义了一个买咖啡的方法，也就是这个接口所声明的能力
}
```

2 新建张煜类，让他实现买咖啡接口，并让他实现买咖啡的能力

```
/**
 * 这个类代表张煜
 * implements BuyCoffee表示他实现了BuyCoffee接口
 * 说明他具备buyCoffee方法，也就是购买咖啡的能力
 */
public class ZhangYu implements BuyCoffee{
    //下面这个方法是BuyCoffee接口中定义的
    // 所有实现了BuyCoffee的类都必须实现这个方法
    // 以确保实现了BuyCoffee方法的类都拥有接口所定义的能力
    @Override
    public void buyCoffee() {
        System.out.println("张煜为涛涛学姐买了一杯卡布奇诺");
    }
}
```

3 新建涛涛学姐类，并定义让别人买咖啡的方法

```
/**
 * 代表张涛学姐的类
 */
public class SisterTaoTao {
    //这个方法描述的是张涛学姐请别人去给她买一杯咖啡
    //她需要一个实现了BuyCoffee的人去帮她买
    //所以这里需要的参数是一个实现了BuyCoffee的对象(toolsMan, 工具人)
    public void letSomeoneBuyCoffee(BuyCoffee toolsMan){
        //表示工具人为涛涛学姐买了一杯咖啡
        toolsMan.buyCoffee();
    }
}
```

4 在主函数中让张涛学姐调用张煜，去给她买一杯咖啡

```
//场景A，张涛学姐买咖啡
System.out.println("场景A，张涛学姐买咖啡");
ZhangYu zhangYu = new ZhangYu();//初始化张煜
SisterTaoTao sisterTaoTao = new SisterTaoTao();//初始化涛涛学姐
sisterTaoTao.letSomeoneBuyCoffee(zhangYu);//张涛学姐要求张煜去帮她买咖啡
```

这里用到了上节课徐国林学长讲的继承和多态的概念，让我们对其中的难点进行一下解析

实现接口

上节课徐国林学长提到了类之间的继承关系，类与类之间的继承使用的关键词为extends，而一个类实现一个接口使用的是implements关键词，具体用法为

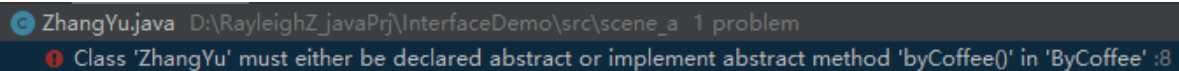
```
class A implements B , C , D {

}
```

在implements后面跟上接口的名称，如果需要实现多个接口，接口名称与接口名称之间要用“,”隔开（多个接口的同时引用会在后面提到）

接口方法的重写

上节课中大家应该都接触到了方法重写的概念，与类的继承不同，如果实现了一个接口，就势必要重写其中的方法。下面就是没有重写方法而引起的报错



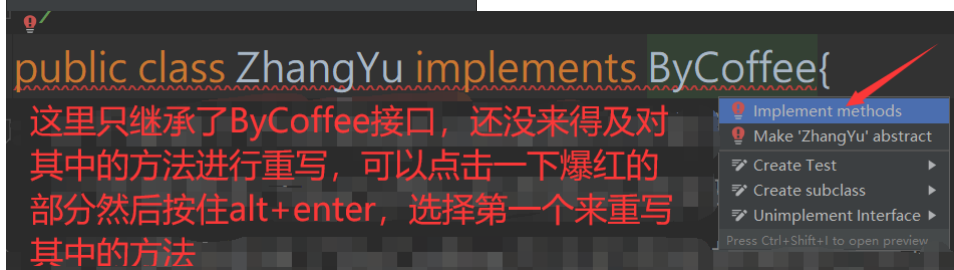
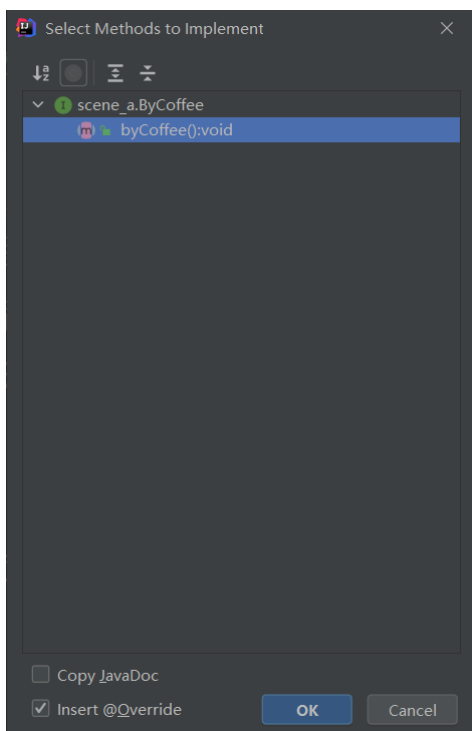
```
ZhangYu.java D:\RayleighZ_javaPrj\InterfaceDemo\src\scene_a_1 problem
❌ Class 'ZhangYu' must either be declared abstract or implement abstract method 'byCoffee()' in 'ByCoffee' :8
```

why?：接口里面定义的方法默认为public abstract，abstract限定了它没有一个默认的实现，必须要在子类中进行实现。当然你也可以按照报错的第一个方法将这个类声明为abstract。（有关abstract的更多知识等会会在抽象类的介绍中详解，这里大家可以先预先了解一下就好）

在实现了接口的子类中重写接口的中定义的方法与子类重写父类的方式类似

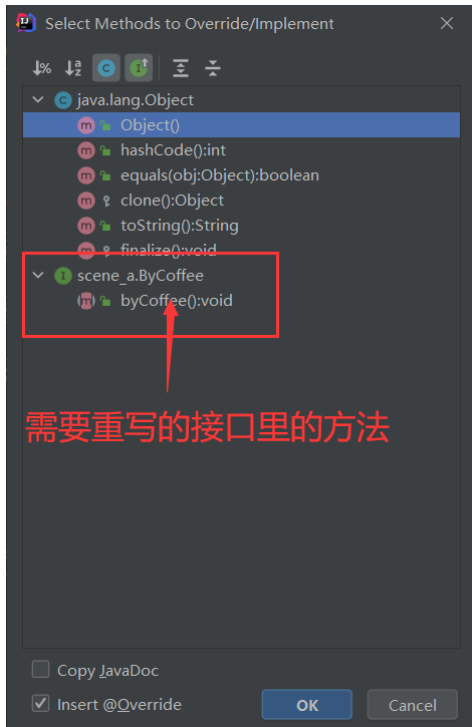
想要快速的重写父类接口中的方法？这里提供两种快捷路径，

一个是在实现接口之后在类名处按住alt+Enter进行快速修正，之后会出现右侧界面，点击第一个选项，选择需要重写的方法，然后点击OK，就会弹出空白的，需要复写的方法



这里只继承了ByCoffee接口，还没来得及对其中的方法进行重写，可以点击一下爆红的部分然后按住alt+enter，选择第一个来重写其中的方法

另一个是按住ctrl+O，快速选择需要重写的方法



需要重写的接口里的方法

更多用法：传递和返回参数

让我们构造一个新的场景

场景B：涛涛学姐还是想要找人买一杯咖啡，她还是想要委托张煜去给她买，但是张煜突然想起来在场景A里面涛姐好像没有给他钱，于是就要张涛给他点钱，自己再去买，于是涛涛学姐给了他一些钱，张煜也给涛涛学姐带回了咖啡

这里与场景A的唯一不同是，买咖啡这个动作需要传递一定的参数（钱）进去，也需要返回特定的参数（咖啡），这里的接口设计必须要考虑到这些，我们自然而然的也可以想到如何设计这个接口

```
//买咖啡的接口
public interface BuyCoffeeWithMoney {
    Coffee buyCoffee(int money); //定义了一个买咖啡的方法，需要传递一个整数（钱）进去，并且返回Coffee
}
```

其他的照搬原本的设计

定义一个咖啡类

```
//咖啡类，只有一个属性：名字
public class Coffee {
    String name ;
    public Coffee(String name) {
        this.name = name;
    }
}
```

张涛学姐类

```
public class SisterTaoTao {
    public void letSomeoneBuyCoffee(BuyCoffeeWithMoney toolsMan , int money){
        Coffee coffee = toolsMan.buyCoffee(money);
        System.out.println("涛涛学姐拿到了一杯"+coffee.name);
    }
}
```

张煜类

```
public class ZhangYu implements BuyCoffeeWithMoney{
    @Override
    public Coffee buyCoffee(int money) {
        return new Coffee("张煜花了"+money+"块钱买的摩卡");
    }
}
```

主函数中模拟这个场景

```
//场景B，张涛学姐给张煜钱，张煜给张涛带回咖啡
System.out.println("\n场景A，张涛学姐给张煜钱，张煜给张涛带回咖啡");
scene_b.SisterTaoTao sisterTaoTao1 = new scene_b.SisterTaoTao();
scene_b.ZhangYu zhangYu1 = new scene_b.ZhangYu();
sisterTaoTao1.letSomeoneBuyCoffee(zhangYu1 , 12); //涛涛学姐给张煜12块钱让他给自己买咖啡
```

以上就是如何在接口中传递参数和返回参数

更多用法：一个接口多个方法，一个类实现多个接口

继续之前的场景

场景C：林潼姐姐因为需要练习中门对狙，没有时间去食堂吃饭，需要一位工具人帮她购买可乐和汉堡，而她突然发现金牌工具人张煜具有购买快乐餐的接口，于是就委托张煜在为涛涛学姐购买咖啡的路上帮自己买可乐与汉堡，张煜也照做了

这里涉及到了两个内容，一个是接口内定义多个方法，一个是一个类实现多个接口，实现起来应该不难，具体代码可以下载我写的demo，这里只贴上重点的几个部分的写法了哦

```
//一个接口声明多个方法
public interface BuyColaAndHamburger {
    void buyCola();//买可乐
    void buyHamburger();//买汉堡
}
```

张煜实现多个接口

```
public class ZhangYu implements BuyCoffee , BuyColaAndHamburger{
    @Override
    public void buyCoffee() {
        System.out.println("张煜买了一杯咖啡");
    }

    @Override
    public void buyCola() {
        System.out.println("张煜买了一杯可乐");
    }

    @Override
    public void buyHamburger() {
        System.out.println("张煜买了一个汉堡");
    }
}
```

同时实现多个接口的问题

警告！即将开启飞行模式  20-iOS-直接飞行模式

首先是这里提到了一个类可以实现多个接口，但是上节课国林学长有提到，java中类不能多继承，这个是为什么呢？

防止冲突是一个主要原因，如果类A同时继承了A，B两个类，那么假设类A和类B中同时存在一个**名称相同**的方法，而子类A中不进行覆写，那么系统需要去执行那个类里面的方法呢？

但是如果是接口的话就没有这种顾虑了，因为接口没有方法的默认实现，所以不用考虑向上层查询来寻找方法的默认实现这件事情

这里举一个例子，我声明两个接口，并且让两个接口中的方法的名称完全相同，其中有一个方法的接收参数也完全相同

```
public interface InterfaceA {
    void sameNameSameFieldFunction();//两个接口中的同名同参数类型的方法
    void sameNameDifFieldFunction(int num);//两个接口中同名不同参数类型的方法
}
```

```
public interface InterfaceB {
    void sameNameSameFieldFunction(); //两个接口中的同名同参数类型的方法
    void sameNameDifFieldFunction(String info); //两个接口中同名不同参数类型的方法
}
```

接下来让一个类实现这两个接口

```
public class Test implements InterfaceA , InterfaceB {

    @Override
    public void sameNameSameFieldFunction() {
        //完全相同的方法将会被合并
        //这里的代码将同时被A, B两个接口所使用
    }

    @Override
    public void sameNameDifFieldFunction(String info) {
        //参数值不同的方法将会分开来写
    }

    @Override
    public void sameNameDifFieldFunction(int num) {
        //参数值不同的方法将会分开来写
    }
}
```

可以看到方法名和参数类型完全相同的方法将会被合并，两个接口的方法拥有相同的具体代码，但是参数不同的代码就将被分开，不同的接口的方法拥有不同的具体代码

这样还是会面临一个问题，如果我想要两个接口的同名同参的方法拥有不同的实现，那要咋办？

也许可以试一下内部类

更多用法：接口回调（相当重要）

芜湖金牌讲师大司码曾经说过：

“回首 掏，接口一开看不见 走位走位，得到数据，难受 ~”



为什么需要接口回调

继续之前的场景

场景D：张煜最近学会了制造咖啡的技巧，张涛姐姐为了节约资金，选择让张煜帮他免费制作咖啡，于是在一天下午找到张煜，让他帮自己制作一杯咖啡，但是由于张涛姐姐很忙，所以她决定先去帮忙，然后让张煜在制作好之后通知自己一声，张煜在制作好了咖啡之后就给张涛送了过去.....

总结一下，上面的故事是这样的一个场景：

- 涛姐姐找我要咖啡
- 我帮涛姐姐做咖啡，与此同时涛姐姐先去忙
- 我做好了咖啡
- 我通知涛姐姐他的咖啡做好了
- 涛姐姐拿到了自己的咖啡并享用

这里的核心点在于张煜做咖啡的**具体时间是不确定的**，所以需要张煜在做好了咖啡之后手动通知涛姐姐，这里的通知怎么实现呢？这里就需要接口回调了

可以这样理解，张煜需要知道涛姐姐的位置，这样才能最后通知到涛姐姐，用java的语言来说，就是张煜要持有一个张涛的引用，最后当自己做好咖啡之后调用涛姐姐的方法并且把做好的咖啡传递进去。

多说汉字无益，还是直接上code

咖啡制作者接口

```
public interface CoffeeMaker {  
    void makeCoffee();  
}
```

咖啡消费者接口

```
public interface CoffeeConsumer {  
    void getCoffee(Coffee coffee);  
}
```

咖啡制作者张煜

```
public class ZhangYu implements CoffeeMaker {  
  
    private CoffeeConsumer sisterTaoTao; //张涛学姐  
  
    public void setCoffeeConsumer(CoffeeConsumer coffeeConsumer) {  
        this.sisterTaoTao = coffeeConsumer;  
    }  
  
    @Override  
    public void makeCoffee() {  
        //以下为延时函数，暂时不要求掌握  
        try {  
            Thread.sleep(3000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        //为涛涛姐姐做好咖啡  
        Coffee coffeeForSister = new Coffee("摩卡");  
        System.out.println("张煜做好了"+coffeeForSister.name+"给涛涛学姐");  
        //通知涛涛姐姐她的咖啡做好了  
        //并且将咖啡递给涛涛姐姐  
        sisterTaoTao.getCoffee(coffeeForSister);  
    }  
}
```

咖啡接收者张涛

```

public class SisterTaoTao implements CoffeeConsumer{
    @Override
    public void getCoffee(Coffee coffee) {
        /*
        这里仅仅进行了一个简单的展示
        这里可以进行很多逻辑处理
        拿到咖啡之后的逻辑任你处理
        */
        System.out.println("张涛姐姐喝到了"+coffee.name);
    }
}

```

主函数调用

```

public class Main {
    public static void main(String[] args) {
        //声明涛涛姐姐
        SisterTaoTao sisterTaoTao = new SisterTaoTao();
        //声明张煜
        ZhangYu zhangYu = new ZhangYu();
        //相当于是涛涛姐姐来张煜这里订购了一杯咖啡
        //然后张煜拿到了涛涛姐姐的联系方式（引用）
        //以方便通知涛涛姐姐
        zhangYu.setSisterTaoTao(sisterTaoTao);
        //张煜在制作咖啡，制作好了之后会通知涛涛姐姐
        zhangYu.makeCoffee();
    }
}

```

输出结果

张煜做好了摩卡给涛涛学姐
张涛姐姐喝到了摩卡

这就是一个简单的接口回调，今后我们开发的时候将会在很多地方用到接口回调，举一个例子：

现在我们需要从网络上获取一个数据，等数据返回到我们的手机上之后将数据显示在屏幕上，因为网络请求获取数据的时间是不确定的，所以只有等数据获得了之后，才能通知屏幕展示数据，这里就要用到接口回调

更多用法：相当炫酷的lambda表达式

芜湖金牌讲师大司码曾经还说过

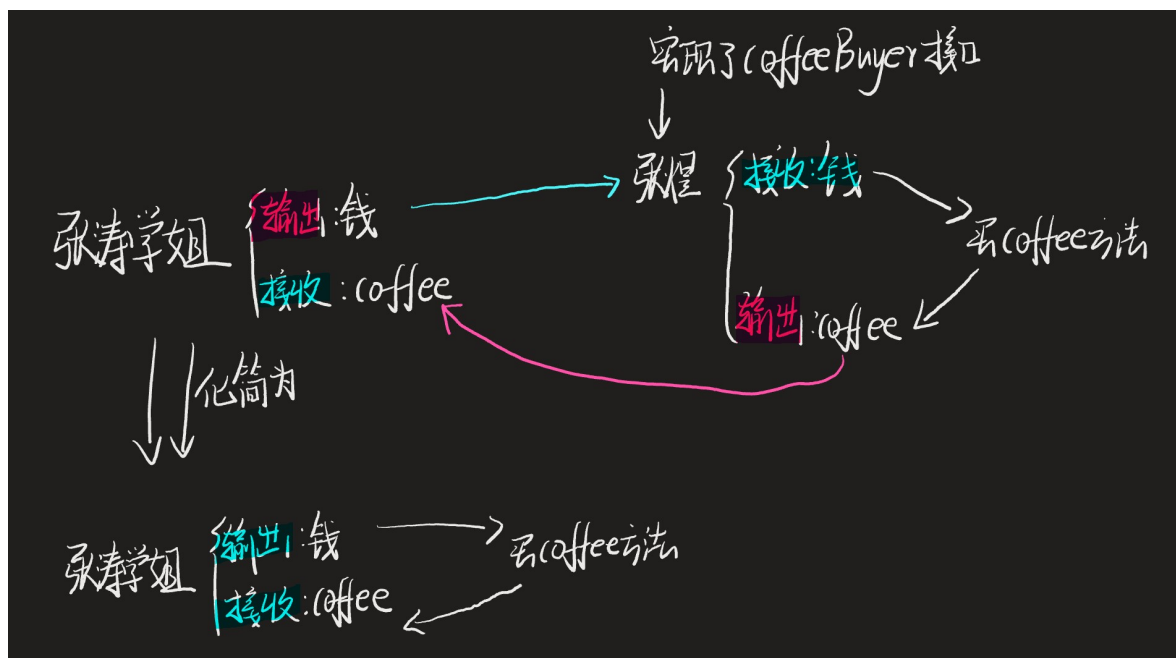
“各个程序员都有自己的理解，但是在我看来，要是你的代码用上了lambda表达式，你的代码就

chu~~，起飞，哎飞”



这里属于进阶内容，学有余力的同学可以听一下。

不知道在学习接口的时候，大家会不会问一个这样的问题，虽然接口回调很好用，但是感觉有时候超级麻烦，比如在场景A中，涛涛学姐只是想要买一杯咖啡罢了，居然需要找一个工具人实现接口，最后将这个工具人的实例再传递进去，废了半天周章才干了一行代码的事情，也就是下面这张逻辑图



本质上张涛学姐需要的仅仅是外界传递进来一个能将钱映射为咖啡的方法，我们也可以只给她一个方法，这样就够了

让我们看看代码

```
public class SceneMain {  
    public static void main(String[] args) {  
        //场景B lambda表达式实现，不需要工具人张煜  
        System.out.println("场景B lambda表达式实现，不需要工具人张煜");  
        SisterTaoTao sisterTaoTao = new SisterTaoTao();  
        sisterTaoTao.letSomeoneBuyCoffee(money-> new Coffee("花了"+money+"原买的咖啡"), 10);  
    }  
}
```

对比一下前后的差距，首先我们不再需要工具人张煜类，再次我们不需要再代码中实例一个工具人张煜，直接用一行

```
money-> new Coffee("花了"+money+"原买的咖啡")
```

代替了张煜类的所有作用，是不是便捷到飞起？

你们可能不知道，只用一行代码实现接口是什么概念，我们一般只用一个词形容这种写法：
lambda表达式

接下来康康如何使用这个6到起飞的工具

lambda表达式发挥了什么作用？

简而言之，用一行代码迅速实现一个接口，也就是完成输入到输出的映射

如何使用lambda表达式

lambda表达式可以公式化

```
(接口中的参数，可以为多个) -> {接口中需要具体实现的代码}
```

举个例子

```
//买咖啡的接口
public interface BuyCoffeeWithMoney {
    Coffee buyCoffee(int money); //定义了一个买咖啡的方法，需要传递一个整数（钱）进去，并且返回Coffee
}
```

上述接口使用lambda表达式进行实现

```
(money) -> {
    //在这两个花括号之间书写的就是接口中的buyCoffee方法
    return new Coffee("张涛姐姐花了"+money+"元买的咖啡");;
}
```

这句lambda表达式就等价于

```
//工具类中实现此函数
public class ZhangYu implements BuyCoffeeWithMoney{
    @Override
    public Coffee buyCoffee(int money) {
        return new Coffee("张涛姐姐花"+money+"块钱买的咖啡");
    }
}
```

lambda表达式可以直接写在需要实现某一个单方法接口的地方，学有余力的同学可以下课继续了解一下lambda表达式

更多用法：接口中声明成员变量

接口不仅仅可以声明函数，也可以在其中声明属性（成员变量），但是其默认会为public static final 类型

比如，如果我定义了这样的一个接口

```
public interface InterfaceWithField {
    int number = 1;
    void function();
}
```

如果在实现了的子类中这样写：

```
public class Test implements InterfaceWithField{
    static int test = 0;
    @Override
    public void function() {
        System.out.println(this.number); //注意！如果没有冲突才可以用this访问
        System.out.println(InterfaceWithField.number);
        InterfaceWithField.number++; //会报错，提示为final变量不能改变
    }
}
```

试图更改number是会报错的，报错提示为：Cannot assign a value to final variable 'number'，也就是final的变量不可再改变

Why? 凡事都要讲求一个为什么

- 假设此成员变量不是static的，因为java支持实现多个接口，那么不同接口中如果属性的名称相同，将无法判断是哪个接口的属性，举个例子，这里再写一个有属性的接口，并且属性名称与

```
public interface InterfaceWithField2 {  
    int number = 2;  
}
```

现在让Test同时实现两个接口

```
public class Test implements InterfaceWithField , InterfaceWithField2{  
    static int test = 0;  
    @Override  
    public void function() {  
        System.out.println(this.number); //会报错，提示为有变量同名  
        System.out.println(InterfaceWithField.number);  
        System.out.println(InterfaceWithField2.number)  
        InterfaceWithField.number++; //会报错，提示为final变量不能改变  
    }  
}
```

仅仅通过this关键字将无法区分number是来自哪一个接口之中，这时候只有通过接口的名称进行调用才能加以区分

- 假设此成员变量不是final，那这个变量就是可以修改的，这样如果在一个实现了接口的子类中修改了这个变量，其他实现了此接口的子类的此变量也会改变，这样不符合接口作为一个模板的设计理念（换句话说，要是允许这样写，那就不是接口，而是后面要讲的抽象类了）
- 假设此关键字不是public的，别的包的类将无法得到这个属性

更多用法：default关键字

default关键字允许给接口的方法一个默认的实现

```
default void function(){  
    //这样允许给function一个默认的方法  
}
```

但是这里就会出现类似类的多继承中的问题，需要注意同名方法默认实现的冲突问题

更多用法：接口中声明静态方法

可以在接口中声明静态方法，并且这个静态方法可以实例化

```
public interface InterfaceTest {  
    static void function(){  
        //此方法可以进行实例化  
    }  
}
```

抽象类

有些类就很抽象，比如女朋友，但是有些人总能找到并且实例化她，只可惜不是我

什么是抽象类

抽象类是很抽象的类

先看看比较官方的定义：

- 在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。
- 抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。
- 由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。也是因为这个原因，通常在设计阶段决定要不要设计抽象类。
- 父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。
- 在Java中抽象类表示的是一种继承关系，一个类只能继承一个抽象类，而一个类却可以实现多个接口。

接下来让我用人话说一下抽象类的这些概念（表达方式借鉴自丁德桥学长）

- **只能当爸爸**：生而为父，我很抱歉，伟大的丁神在课件中曾经说过：“它天生就是用来被别人来继承的，简而言之，只有当爸爸的命，不当儿子。可能有些事，必须得儿子去做，别直接叫我去做事。”，换句话说，抽象类不能被new出来，因为它只是一个模板，需要有人实现了这个模板才能被使用，它也许是别人的子类，但是想要它发挥作用就只能给他找个儿子
- **“有些事情只能儿子去做”**：抽象类中的方法是抽象的，就是没有实现的，就像是接口中的方法一样，需要他的儿子去继承并且实例化

与其说是抽象，我更愿意称之为模板，或者说是程序员在变成时的一种设计

接下来看看抽象类怎么声明

抽象类的声明

在声明类的时候，class前面加上abstract关键字修饰就可以了

```
//普通类
public class Test{

}

//抽象类
public abstract class Test {

}
```

还是举个例子，带大家写一下

场景：张煜想要成为一名摇滚乐队吉他手，但是他不知道一名优秀的吉他手要具备那些素质，于是决定按照一个摇滚吉他手的模板成为一名优秀的吉他手

他总结了一下，一名吉他手要有这些特性：

属性方面：

- 有一把电吉他
- 有个名字

方法方面：

- 会弹吉他

- 会编曲

根据这些可以抽象出一个摇滚吉他手类，按照之前的思路，我们可以轻松的设计一个父类，这里只需要在class签名加上一个abstract关键字，就可以生成一个抽象类

```
public abstract class GuitarPlayer {
    private Guitar guitar;//私有属性： 吉他
    private String name;//私有属性，姓名

    public GuitarPlayer(Guitar guitar, String name) {
        this.guitar = guitar;
        this.name = name;
    }

    //get set方法
    public Guitar getGuitar() {
        return guitar;
    }

    public String getName() {
        return name;
    }

    public void setGuitar(Guitar guitar) {
        this.guitar = guitar;
    }

    public void setName(String name) {
        this.name = name;
    }

    public abstract void playGuitar();//抽象方法： 弹吉他
    public abstract Music makeMusic(String inspire);//抽象方法： 作曲，需要传递灵感进来，返回一个音乐
    public void eat(){//普通方法，不抽象
        System.out.println(name+"在吃饭");
    }
}
```

留心这里的方法域，抽象类中可以声明两类方法，分别为抽象方法和与之对应的普通方法

- **抽象方法**：由abstract关键词修饰，只能声明接收的参数类型和返回的参数类型，不能书写方法的具体实现，这个方法子类**必须覆写**，以此实现此方法的具体功能。
- **普通方法**：常规的方法，具体是使用规则可以参考父类中的方法的继承规则。

```
public void function(){};//一个普通方法

public abstract void function();//一个抽象方法
```

继承抽象类和继承普通类的区别

从功能上看，似乎抽象类和父类并没有什么区别，二者都是声明了一种模板，重点实现和拓展还是交给了子类去做。但抽象类抽象的特征使得抽象类的抽象方法必须要被继承并实现

父类：内部方法为普通方法，子类可以不去修改，父类中会提供默认的实现

抽象类：内部方法为抽象方法，**非抽象**子类必须**强制进行覆写**，不然会报错

抽象类与接口的区别

语法层面上

- 接口支持**多继承**，抽象类不支持多继承（也就是说一个类可以继承多个接口，但是只能继承一个抽象类）
- 接口中只能声明**抽象方法**，抽象类中可以声明普通方法
- 接口中的成员变量只能是public static final的，抽象类的成员变量可以是各种类型的

设计层面上

接口对方法的抽象，而抽象类是对类的抽象

这是一个很核心的概念，需要着重梳理一下

接口是对一种动作的抽象，举个例子，如果我定义了一个Fly接口，用于抽象飞这个动作，飞机和鸟都会飞，那么飞机和鸟都继承了Fly接口并且实现了各自的fly方法。而飞行这个动作本身不宜设计为抽象类，因为鸟和飞机还是有本质上的不同的，它们只是恰巧飞这个方法上有相同之处罢了，所以这里的继承关系最好设计成这样：



去年我学长是这样说的：**继承**是一种**是不是**的关系，而**接口**是一种**有没有**的关系。就像是程序员和写代码之间的区别一样，会不会写代码是一个有没有这种能力的概念，而不是程序员则在更多的角度上加上了描述。

内部类

“你以为我在第一层，实际上我在第二层，甚至我可以在第三层”

———大司码

什么是内部类

内部类（inner class）是**定义在类中的类**，比如下面的InnerClass就是OuterClass的一个内部类

```
public class OuterClass {
    //OuterClass为外部类
    class InnerClass{
        //这个类就是在类中定义的类，称之为内部类
    }
}
```

上述仅仅是内部类的写法之一，接下来给大家展示一下几种内部类的写法

成员内部类

直接定义在外部类之中，并且在外部类方法之外的内部类

```
public class OuterClass {
    //OuterClass为外部类
    class InnerClass{
        //成员内部类
        //定义在方法之外，外部类之中
    }

    public void function(){

    }

}
```

成员内部类的注意事项：

- 成员内部类中不能存在任何static的变量和方法
- 成员内部类是依附于外围类的，所以只有先创建了外围类才能够创建内部类
如何实例化一个内部类对象？如果是在外部类中，可以直接

```
InnerClass innerClass = new InnerClass();
```

如果是在外部类之外的其他地方，实例化内部类之前必须要先实例化一个外部类，借由外部类来实例化内部类

```
//先实例化外部类对象，再实例化内部类对象
outerClass.InnerClass innerClass = outerClass.new InnerClass();
```

Why? 为啥要依附于外部类？不着急，等会会讲

为什么需要内部类？

- 内部类可以自由的**访问和修改**外部类的各种属性和方法，包括**private**修饰的属性和方法

```
public class OuterClass {
    //OuterClass为外部类
    private int num = 0;

    class InnerClass{
        //成员内部类
        //定义在方法之外，外部类之中
        public void innerFunction(){
            //内部类的方法中可以调用外部类的私有方法
            privateFunction();
            //内部类的方法中可以自由的访问和修改外部类的私有属性
            System.out.println(++num);
        }
    }

    //一个私有方法
    private void privateFunction(){

    }

}
```

- 内部类可以独立的继承父类和实现接口

```

public class OuterClass {
    //OuterClass为外部类
    private int num = 0;

    class InnerClass implements Comparable<InnerClass>{//实现独立的接口

        @Override
        public int compareTo(InnerClass innerClass) {
            return 0;
        }
    }
}

```

为啥内部类可以访问外部类的这些属性呢？内部类与外部类之间究竟有着怎样的联系呢？

警告！即将开启飞行模式

非静态内部类默认持有一个外部类的引用

静态内部类等会讲，现在这个内部类是非静态的

之所以可以访问这些private的属性，主要原因有两点

- 内部类里面有一个隐含的外部类对象，借此可以引用到外部类的属性
- 内部类在外部类的花括号之间，所以private属性是可以访问的

也就是说，应要写的话，这些属性应该是这么访问的

```

public class OuterClass {
    //OuterClass为外部类
    private int num = 0;

    class InnerClass{
        public void innerFunction(){
            privateFunction();
            OuterClass.this.privateFunction();//上面一行等价于这样调用
        }
    }
    //一个私有方法
    private void privateFunction(){

    }
}

```

其中OuterClass.this就相当于获取这个外部类是实例

那么这个外部类是什么时候传递进来的呢？又是怎么传递进来的呢？

这里再开一小段飞机

明确一点，内部类只是一个编程时的概念，当代码被编译后，内部类和外部类是两个类，我们编译一下这个包含了内部类的外部类，找到编译之后的内部类的.class文件

```

package inner_class;

class OuterClass$InnerClass implements Comparable<OuterClass$InnerClass> {
    OuterClass$InnerClass(OuterClass this$0) {
        //重点代码：这里系统默认提供了一个构造方法
        //传递了一个外部类的实例对象进来
        this.this$0 = this$0;
    }
}

```

```

    }

    public void innerFunction() {
        //可以看到这里是通过外部类的实例变量进行私有方法调用的
        this.this$0.privateFunction();
        this.this$0.privateFunction();
        System.out.println(++this.this$0.num);
    }

    public int compareTo(OuterClass$InnerClass innerClass) {
        return 0;
    }
}

```

好了，航班已到达目标机场，下飞机！

总结一下上面的代码，可以得出结论：

系统会默认给内部类一个构造方法，这个构造方法将给内部类传递一个外部类的实例化对象进去

所以之前方利斌同学在群里面提的问题是不是就可以解决了，静态方法是依附于类而存在的，其中不包含有对象的实例，所以就不能在静态方法中与内部类建立联系，自然也就无法实例化。

现在是不是对之前提到的内部类依附于外部类有了更深刻的理解？

局部内部类

定义在外部类的方法之中的内部类称之为局部内部类

```

public class OuterClass {
    //OuterClass为外部类

    public void function(){
        class LocalInnerClass{
            //定义在方法之内，叫做局部内部类
        }
    }
}

```

局部内部类的优势

局部内部类不仅具有**成员内部类**的所有优势，同时也可以自由的访问**局部变量**，但是一旦访问量，就不允许进行任何修改，他就变成了隐式的final变量。

```

public void function(){
    int localNumber = 0;
    class LocalInnerClass{
        //定义在方法之内，叫做局部内部类
        void innerFunction(){
            //可以正常的访问外部的局部变量
            System.out.println(localNumber);
            //但是不可以修改外部的局部变量
            localNumber++; //会报错
        }
    }
    localNumber++; //会报错，在内部类中调用之后就不能再修改
}

```

为什么这个地方是隐式final？来，要不继续开启飞行模式？

 20-iOS-直接飞行模式

不开了，飞不动了，这里贴一个老夫之前写的[文档](#)，大家有兴趣的可以看一下其中关于内部类和final的部分

静态内部类

用static关键字修饰的内部类

```

public class OuterClass {
    //OuterClass为外部类

    static class staticInnerClass{
        //此处为一个静态内部类
    }
}

```

静态内部类在之前有提到过，静态内部类**不持有**外部类对象的引用，所以它**不能**引用外部类的**非static**成员变量和方法

匿名内部类

匿名内部类就是没有类名的内部类。<——没有名字的是啥东西？张煜你一直在开玩笑

假设我有一个类，只需要使用一次，如果我大费周章的去新建一个类，耗时又耗力，不如在代码中就地新建一个类，并且生成一个它的对象，岂不美哉。

其实匿名内部类的作用和lambda表达式有些类似，只不过匿名内部类的使用范围更广一些

这里举个例子来写一下：

```

//一个非常简单的接口
public interface Interface {
    void function();
}

```

```

//一个非常简单的抽象类
public abstract class AbstractClass {
    abstract void function();
}

```

上面是两个模板，匿名内部类不是想些啥就写啥的，它需要基于**抽象类**或者**接口**来进行实现

接下来就是匿名内部类的写法

```
new AbstractClass(){
    @Override
    void function() {
        //在这里写上子类对这个抽象方法的具体实现
    }
};

new Interface(){
    @Override
    public void function() {
        //在这里写上子类对这个接口方法的具体实现
    }
};
```

上面的代码等价于先写两个类，分别继承这两个接口或抽象类

```
class InterfaceInit implements Interface{

    @Override
    public void function() {

    }

}

class AbstractInit extends AbstractClass{

    @Override
    void function() {

    }

}
```

再在函数中实例化

```
{
    new InterfaceInit();
    new AbstractInit();
}
```

匿名内部类和局部内部类一样，如果要访问外部方法的局部变量，这个变量必须是final的，或者说是隐式final的

写在最后：写的着急，内容较杂，如有bug，请联系我