回调：A调用B，B也调用A。

Java中的回调分为：

1. 同步回调：一种阻塞调用，A调用B的方法，等待B执行完，A才继续。
2. 异步回调：类似打电话，A告诉B一声，然后A、B各干各的事。异步回调可以是多线程，也可以是接口。

## 1、什么是回调函数

有些库函数（library function）要求应用先传给它一个函数，在合适的时候调用，以完成目标任务。这个被传入的、后又被调用的函数就称为回调函数（callback function）

## 2、例子一

上文中的接口回调，理解为异步回调，就是我想完成一件事，不用自己做，告诉别人一下，让他去做，我自己可以做别的事。

场景：幼稚园的小朋友刚刚学习了10以内的加法。

1）幼师在黑板上写一个式子“1+1=”，由小明同学来填空。

由于已经学习了10以内的加法，小明同学完全可以靠自己来计算这个题目，模拟该过程：

public class Student {

private String name = null;

public Student(String name) {

// TODO Auto-generated constructor stub

this.name = name;

}

public void setName(String name) {

this.name = name;

}

private int calcADD(int a, int b) {

return a + b;

}

public void fillBlank(int a, int b) {

int result = calcADD(a, b);

System.out.println(name + "心算:" + a + " + " + b + " = " + result);

}

}

小明同学在填空（fillBlank）的时候，直接心算（clacAdd）了一下，得出结果是2，并将结果写在空格里，测试代码如下：

public class Test {

public static void main(String[] args) {

int a = 1;

int b = 1;

Student s = new Student("小明");

s.fillBlank(a, b);

}

}

运算结果如下：

小明心算：1+1=2

整个过程完全由Student类的实例对象单独完成，并未涉及回调机制。

2）下课后，老师突发奇想在黑板上写了“168+291=”让小明完成，然后会办公室了。

这是问题超纲，小明不能像上面心算那样完成了，正在懵逼的时候，小红通过递过来一个只能计算加法的计算器，而小明同学恰好知道怎么用计算器，于是通过计算器计算得到结果并完成了填空。

计算器代码：

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

修改Student类，添加使用计算器的方法：

public class Student {

。。。

private int useCalculator(int a, int b) {

return new Calculator().add(a, b);

}

public void fillBlank(int a, int b) {

int result = useCalculator(a, b);

System.out.println(name + "使用计算器:" + a + " + " + b + " = " + result);

}

}

运行结果如下：

小明使用计算器：168+291=459

该过程中仍未设计到回调机制，但是部分小明的工作已经实现了转移，有计算器来协助实现。

3）幼师回来了

发现小明完成了3位数的加法，老师觉得小明很聪明。于是又在黑板上写下了“26549+16487=”，让小明上课之前完成填空，然后又回办公室去了。

小明心生一计：让小红代劳。

小明告诉小红题目是“26549+16487=”，然后指出填写结果的具体位置，然后就出去玩了。

这里不要小红单独实现出来，而是把这个只能算加法的计算器和小红看成一个整体，一个会算且还会填空的超级计算器。这个超级计算器需要传的参数是两个加数和要填空的位置，而这些内容需要小明提前告知，也就是小明要把自己的一部分方法暴露给小红，最简单的方法就是把自己的引用和两个加数一块告诉小红。

因此，超级计算器的add方法应该包含两个操作数和小明自身的引用，代码：

public class SuperCalculator

{

public void add(int a, int b, Student xiaoming)

{

int result = a + b;

xiaoming.fillBlank(a, b, result); //回调函数

}

}

小明这边现在已经不需要心算，也不需要使用计算器了，因此只需要有一个方法可以向小红寻求帮助就行了，代码如下：

public class Student

{

。。。

public void callHelp (int a, int b)

{

new SuperCalculator().add(a, b, this);

}

public void fillBlank(int a, int b, int result)

{

System.out.println(name + "求助小红计算:" + a + " + " + b + " = " + result);

}

}

测试代码如下：

public class Test {

public static void main(String[] args) {

int a = 26549;

int b = 16487;

Student s = new Student("小明");

s.callHelp(a, b);

}

}

运行结果为：

小明求助小红计算：26549+16487=43036

执行流程为：小明通过自身的callHelp方法调用了小红（new Sup而Calculator()）的add方法，在调用的时候将自身的引用（this）当做参数一并传入，小红在使用计算器得出结果之后，回调了小明的fillBlank方法，将结果填在了黑板的空格里。

此时，小明的fillBlank方法就是我们常说的回调函数。

通过这种方式，可以看出，对于完成老师的填空题这个任务，小明已经不需要等待到加法做完且结果填写在黑板上才能去跟小伙伴们撒欢了，填空这个工作由超级计算器小红来做了。回调的优势已经开始体现。

实质：小明调用了小红超级计算器的add(int a, int b, Student xiaoming)方法并出传入数值和填写的位置，同时小红超级计算器的add方法调用了小明的fillBlank方法。

## 3、例子二

幼稚园的门口有一个老婆婆，每天风雨无阻在那里摆着地摊卖一些快过期的垃圾食品。由于年纪大了，脑子有些糊涂，经常算不清自己挣了多少钱。有一天，她无意听到小明和小伙伴吹嘘自己如何在小红的帮助下与幼师斗智斗勇。于是婆婆决定找到小红牌超级计算器来做自己的小助手，并提供一包卫龙辣条作为报酬。小红经不住诱惑答应了。

回看例子一的代码，发现小红牌超级计算器的add方法需要的参数是两个整型变量和一个Student对象，但是老婆婆她不是学生，是个小商贩，这里肯定要做修改。这种情况下我们很自然想到了继承和多态。如果让小明这个学生和老婆婆这个小商贩从一个父类进行继承，那么我们只需给小红牌超级计算器传入一个父类的引用就可以了。

不过，实际使用中，考虑到Java的单继承，以及不希望把自身太多东西暴露给别人，这里使用从接口继承的方式配合内部类来做。

换句话说，小红希望以后继续向班里的小朋友提供计算服务，同时还能向老婆婆提供算账服务，甚至以后能够拓展其他人的业务，于是她向所有的客户约定了一个方法，用于统一的处理，也就是自己需要的操作数和做完计算之后应该怎么做。这个统一的方法，小红做成一个接口，提供给大家。

public interface doJob {

public void fillBlank(int a, int b, int result);

}

因为灵感来自小明填空，所以小红保留了初心，把所有业务都当做填空（fillBlank）来做。同时小红修改了自己的计算器，使其可以同时处理不同的实现了doJob接口的人。代码如下：

public class SuperCalculator {

public void add(int a, int b, doJob customer) {

int result = a + b;

customer.fillBlank(a, b, result);

}

}

小明和老婆婆拿到这个接口之后，只要实现了这个接口，就相当于按照统一的模式告诉小红得到结果之后的处理办法，按照之前说的使用内部类来做：

//小明

public class Student {

private String name = null;

public Student(String name) {

// TODO Auto-generated constructor stub

this.name = name;

}

public void setName(String name) {

this.name = name;

}

//内部类

public class doHomeWork implements doJob {

@Override

public void fillBlank(int a, int b, int result) {

// TODO Auto-generated method stub

System.out.println(name + "求助小红计算:" + a + " + " + b + " = " + result);

}

}

public void callHelp (int a, int b) {

new SuperCalculator().add(a, b, new doHomeWork());

}

}

//老婆婆

public class Seller {

private String name = null;

public Seller(String name) {

this.name = name;

}

public void setName(String name) {

this.name = name;

}

//内部类

public class doHomeWork implements doJob {

@Override

public void fillBlank(int a, int b, int result) {

System.out.println(name + "求助小红算账:" + a + " + " + b + " = " + result + "元");

}

}

public void callHelp (int a, int b) {

new SuperCalculator().add(a, b, new doHomeWork());

}

}

测试：

public class Test {

public static void main(String[] args) {

int a = 56;

int b = 31;

int c = 26497;

int d = 11256;

Student s1 = new Student("小明");

Seller s2 = new Seller("老婆婆");

s1.callHelp(a, b);

s2.callHelp(c, d);

}

}

运行结果如下：

小明求助小红计算:56 + 31 = 87

老婆婆求助小红算账:26497 + 11256 = 37753元

## 4、例子的理解

1）约定接口（计算完后要做的事）：

这里就是填写结果，具体写谁的？填什么？怎么填，都是传过来的。。。谁要找我帮忙，你的给我传个你的权限和信息，比如（我帮你填写结果，你告诉我怎么填，就是接口中这个方法，方法的具体实现规则是由对方来完成）

**public** **interface** **doJob** {

**public** **void** **fillBlank**(**int** a, **int** b, **int** result);

}

2）计算方法：

根据传过来的值和对象，计算，计算完并根据约定，调用接口的方法做操作。

**public** **class** **SuperCalculator** {

**public** **void** **add**(**int** a, **int** b, doJob customer) {

**int** result = a + b;

customer.fillBlank(a, b, result);

}

}

学生，找人帮忙计算，把数据和约定的实现方法传过去，告诉他，这几个数，你给我计算，计算完按照我的方法给我填了。

小贩，找人帮忙，告诉你数值和，具体计算完应该怎么写的方法，计算完按照这个方法帮我填了。

学生给了你数值，和怎么填写的，然后你计算完，用学生的对象调用了下方法，把数据传进去，填写，这就是所谓的回调。