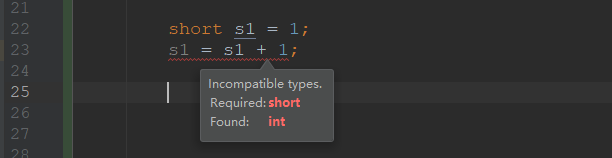
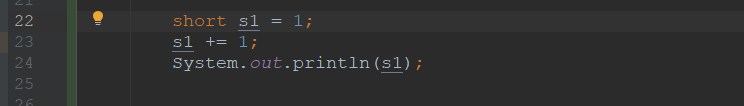
1. **对比short s1=1;s1=s1+1;与short s1=1;s1+=1;**

对于short s1=1;s1=s1+1;代码中，因为1是int类型，所以s1+1也是类型，而前面定义了是short类型，不能将int的值赋给short类型，所以该代码不能编译通过，需要强转:s1=(short)(s1+1)。如图：



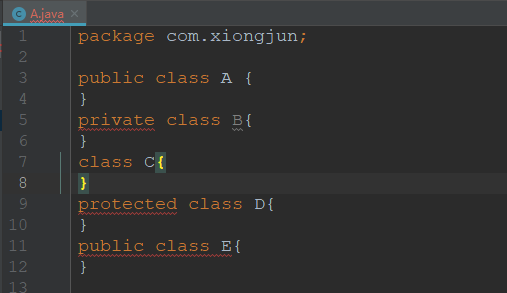
而short s1=1;s1+=1;代码中,+=会做隐式强转，s1+=1相当于s1=(short)(s1+1),所以该代码能编译通过。



1. **访问修饰符private、public、protected及缺省**



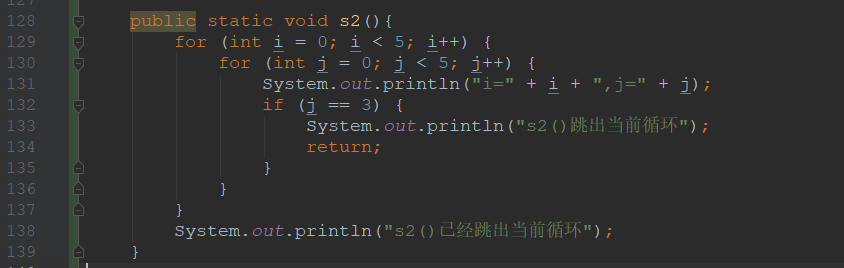
外部类是指在同一个.java文件中，有多个class,除了文件名之外的类，都是外部类，外部类不能被任何修饰符修饰。如图：



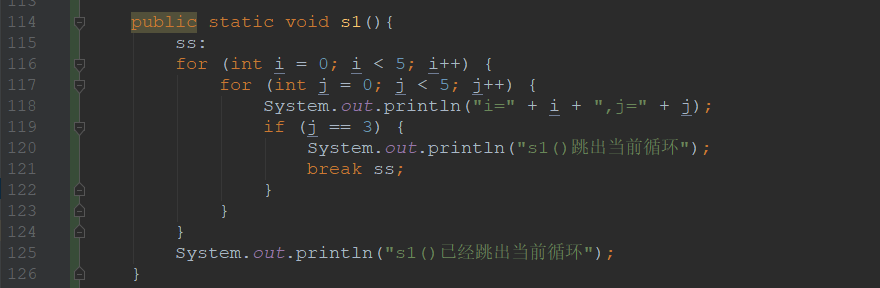
1. **多重嵌套循环如何退出循环**

可以使用return或者在外层循环前定义一个标号，然后break标号即可，后一种方法更灵活一点。如下图：

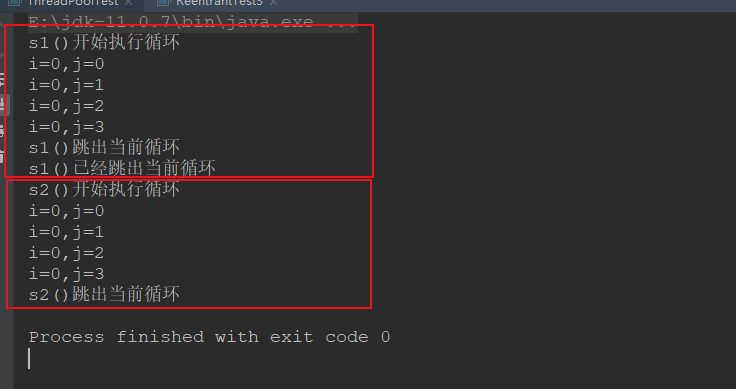
方法1：使用return，return后，for循环后面的代码不执行。



方法2：使用标号，标号后的代码可以继续执行，灵活度更高。



执行结果如下：



1. **实例对象与引用对象区别**

用关键字new一个实例对象，存放在堆内存中，而引用对象是指向实例对象的地址，是存放在栈内存中的。

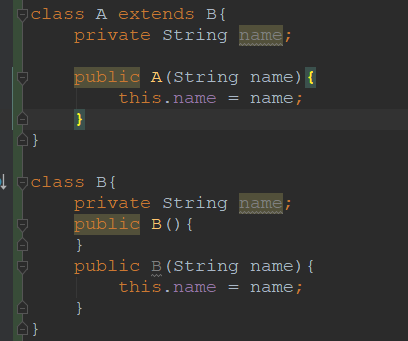
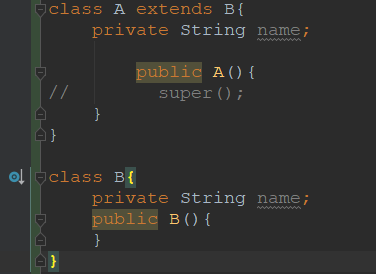
1. **堆内存与栈内存的区别**

堆是一个程序运行动态分配的内存区域。在java中，构建对象时所需的内存从堆中分配。对内存使用完毕后，是由GC回收的。它的存取速度相对于栈慢一点。

栈是由编译器自动分配和释放的一块内存区域，主要存放一些基本类型(如int、float等)的变量、指令代码、常量、及对象句柄(即对象的引用地址)。栈内存是先进后出。他的存取速度比堆快，仅次于寄存器，当超过变量的作用域后，java会自动释放掉为该变量锁分配的栈内存。

1. **子类继承父类，构造函数的解析**

我们先来看下，两段不同的代码：



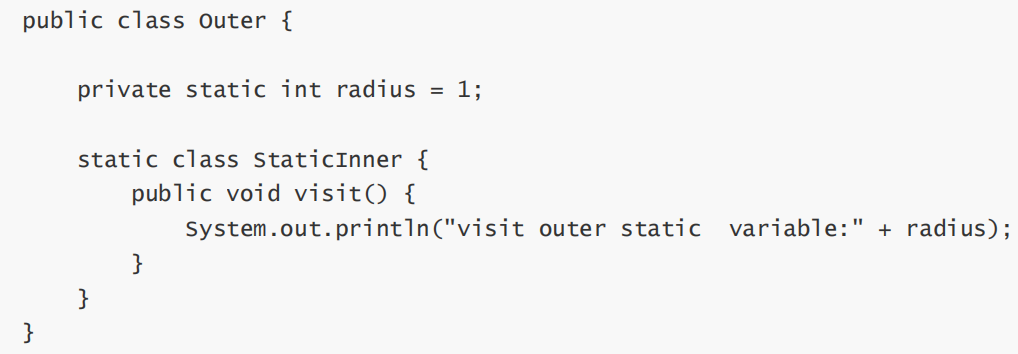
图一中，类A的构造方法中，可以不需要super().因为如果自类在自己的构造方法中不显示调用super(),会自动隐式调用super()。

图二中，类A的构造器中会隐式调用super()方法来调用父类B中的无参构造函数，帮助子类进行初始化，而父类B中有了有参的构造函数，所以父类B中必须要又无参构造函数，否则会编译错误。

1. **内部类**

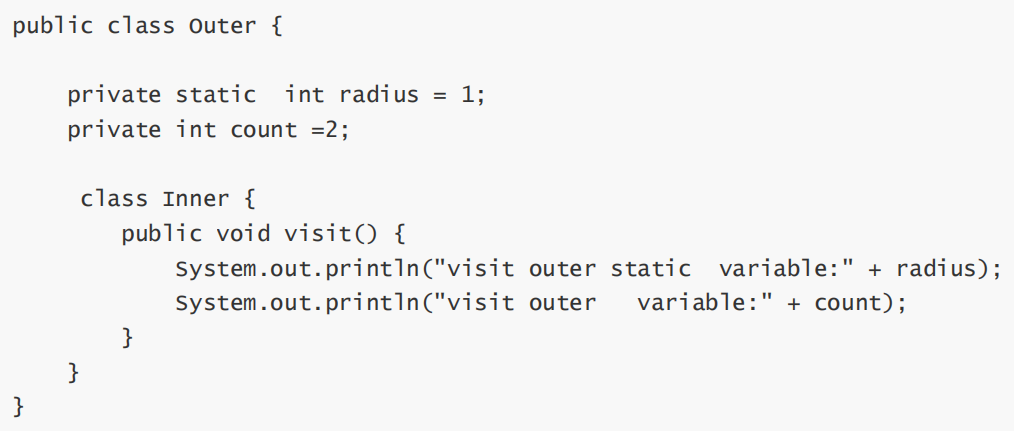
内部类有静态内部类、成员内部类、局部内部类、匿名内部类

·静态内部类



静态内部类只能访问外部类所有的静态成员变量和静态方法，使用new Outer.StaticInner();创建内部类的对象；

·成员内部类



成员内部类可以访问外部类所有的成员变量和方法，使用new Outer.new Inner();创建内部类的对象；

·局部内部类

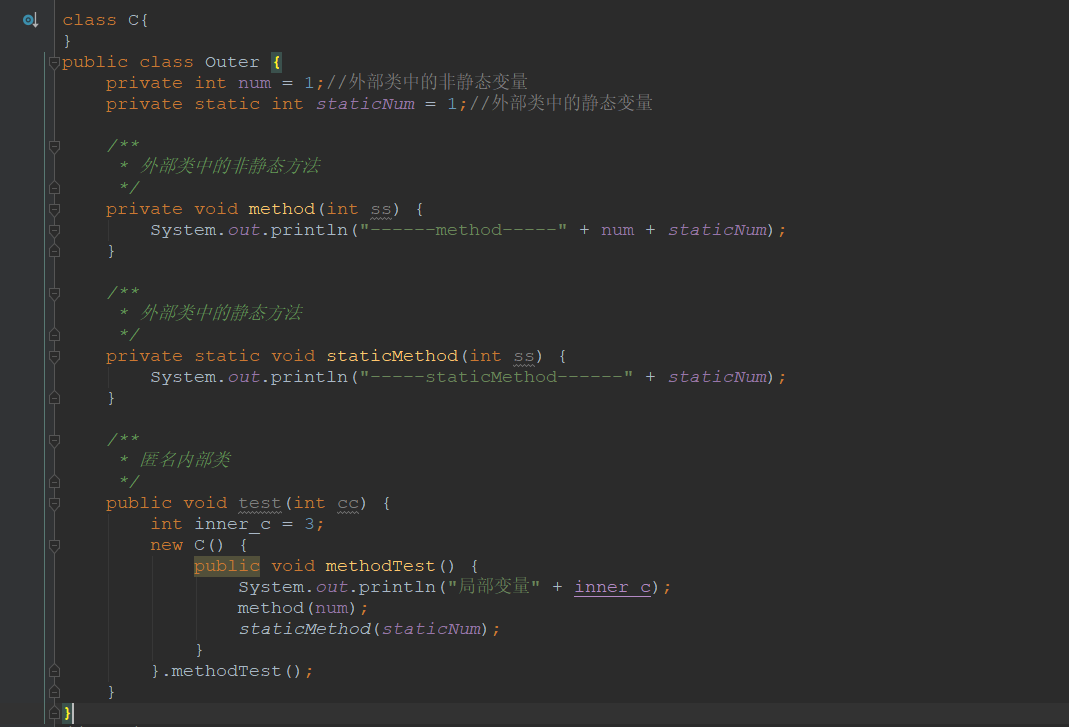
定义在方法内部的类就是局部内部类，如下图：



局部内部类一般只在所属的方法中调用，局部内部类可以调用外部类(方法所属类)的所有成员变量和方法，但是局部类中不能有静态方法，否则会编译报错。静态方法的局部类中不能调用外部类(方法所属类)的非静态方法。使用new Inner().funciont()。局部内部类中不能对方法中的变量进行修改。

·匿名内部类

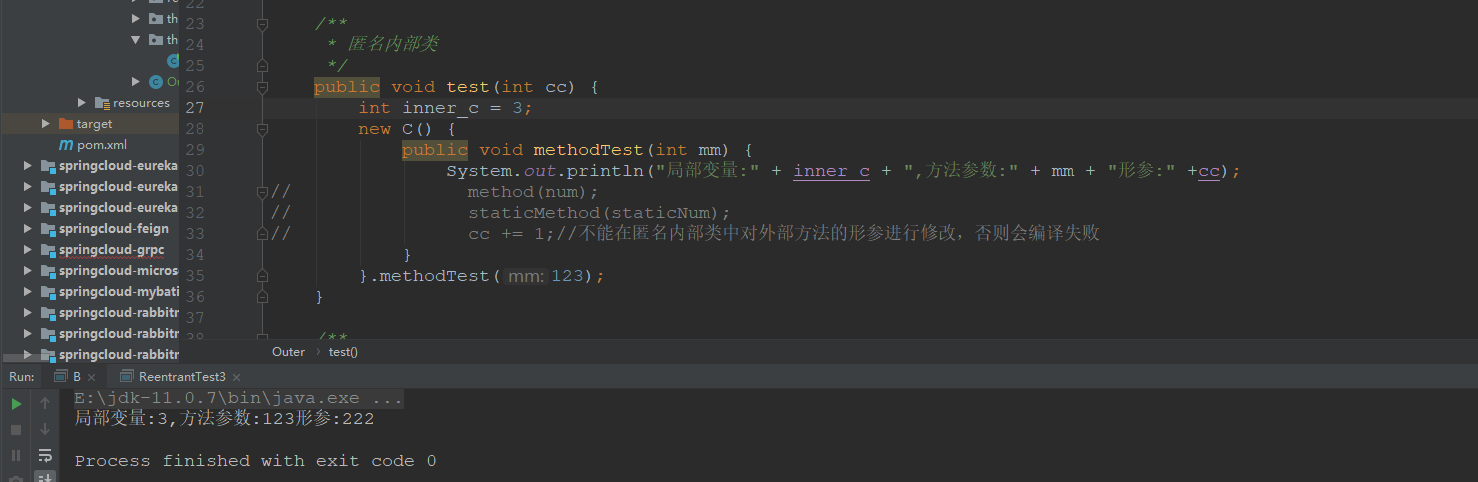
匿名内部类就是没有名字的内部类，其实也是一种局部内部类。



匿名内部类必须继承一个类或实现一个接口;

匿名内部类不能定义任何静态成员和静态方法；

如果方法的形参不能在匿名内部类中做修改，否则会编译错误，同样也不能对方法的局部变量进行修改。如下图：



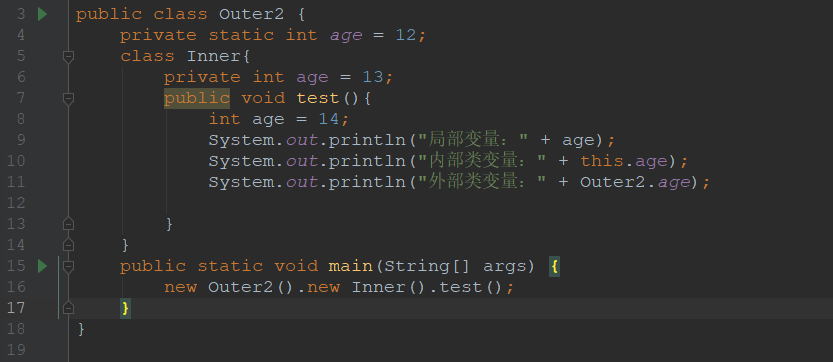
匿名内部类中的方法不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

局部内部类和匿名内部类中访问局部变量的时候，必须要加上final。为什么呢？

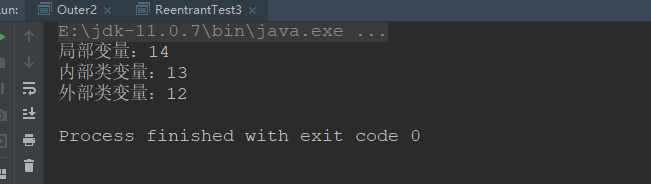
因为生命周期不一致，局部变量存储在栈中，当内部内中起了一个异步线程，将局部变量传入了线程中，那么当方法执行结束后，非final的局部变量就会被消费，而线程还没跑完，此时对局部变量的引用依然存在，此时就会报错。加了final，可以确保局部内部类使用的变量与外层的局部变量区分开，解决了这个问题。



下面我们来看一下一段代码：



执行结果：



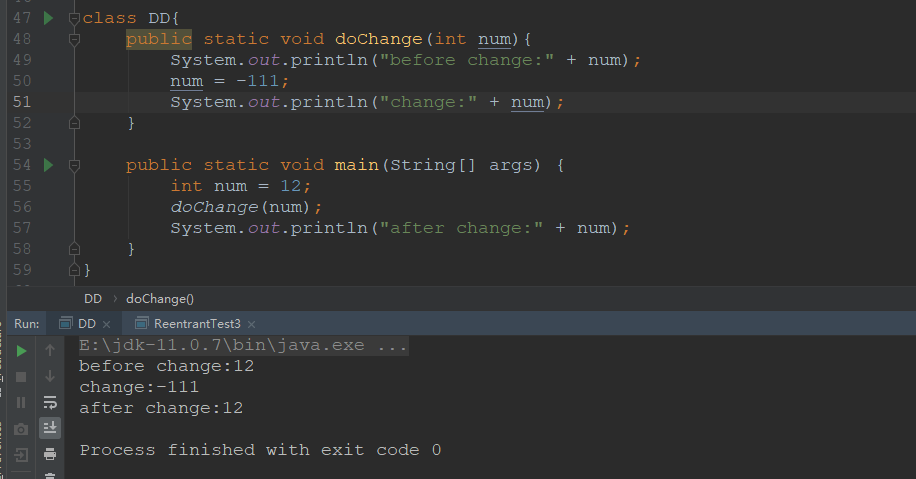
1. **java中的值传递和引用传递，为什么Java中只有值传递？**

值传递：指的是在方法调用时，传递的参数是按值的拷贝传递，传递的是值的拷贝，也就是说传递后就互不相关了。

引用传递：指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引用的地址，也就是变量所对应的内存空间的地址。传递的是值得引用，也就是说传递前和传递后都指向同一个引用(同一个内存空间)。

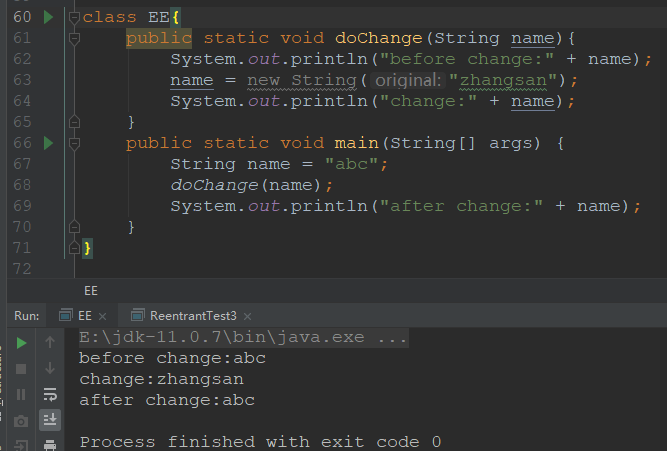
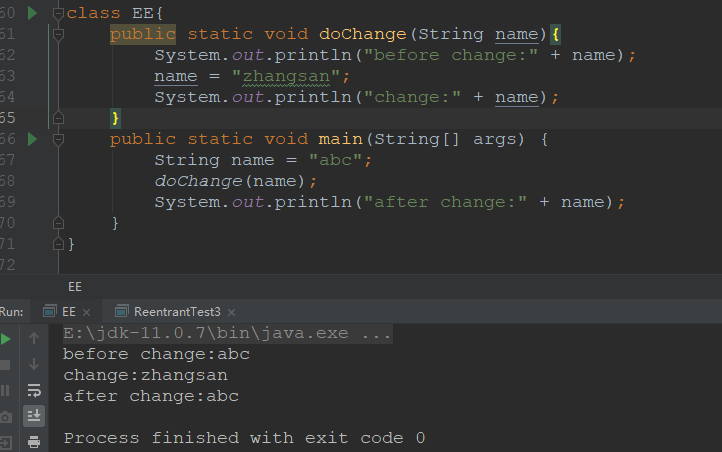
我们下面看几个例子：

Example1:基础数据类型作为参数

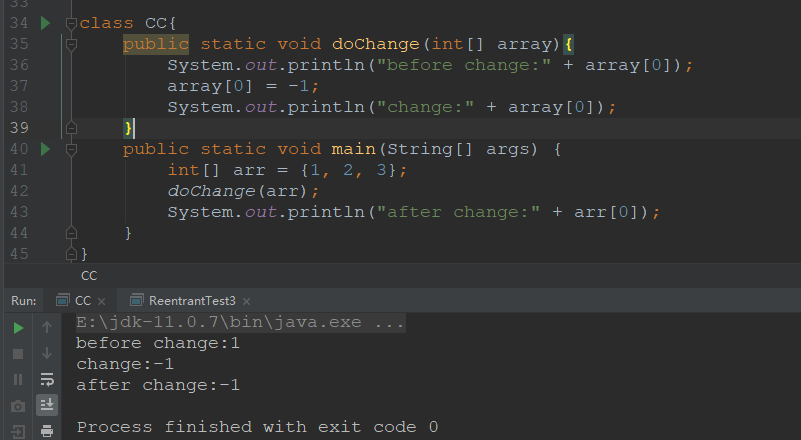


在doChange(int num)方法中，num=-111;是将num的拷贝赋值为-111。因为doChange(int num)方法的num的值只是从main()方法中的int num=12拷贝过来的，相当于doChange(int num)方法的num只是main()方法中的int num的一个副本；所以副本无论怎么修改都不会改变原件本身。

Example2:String类型数据类型作为参数

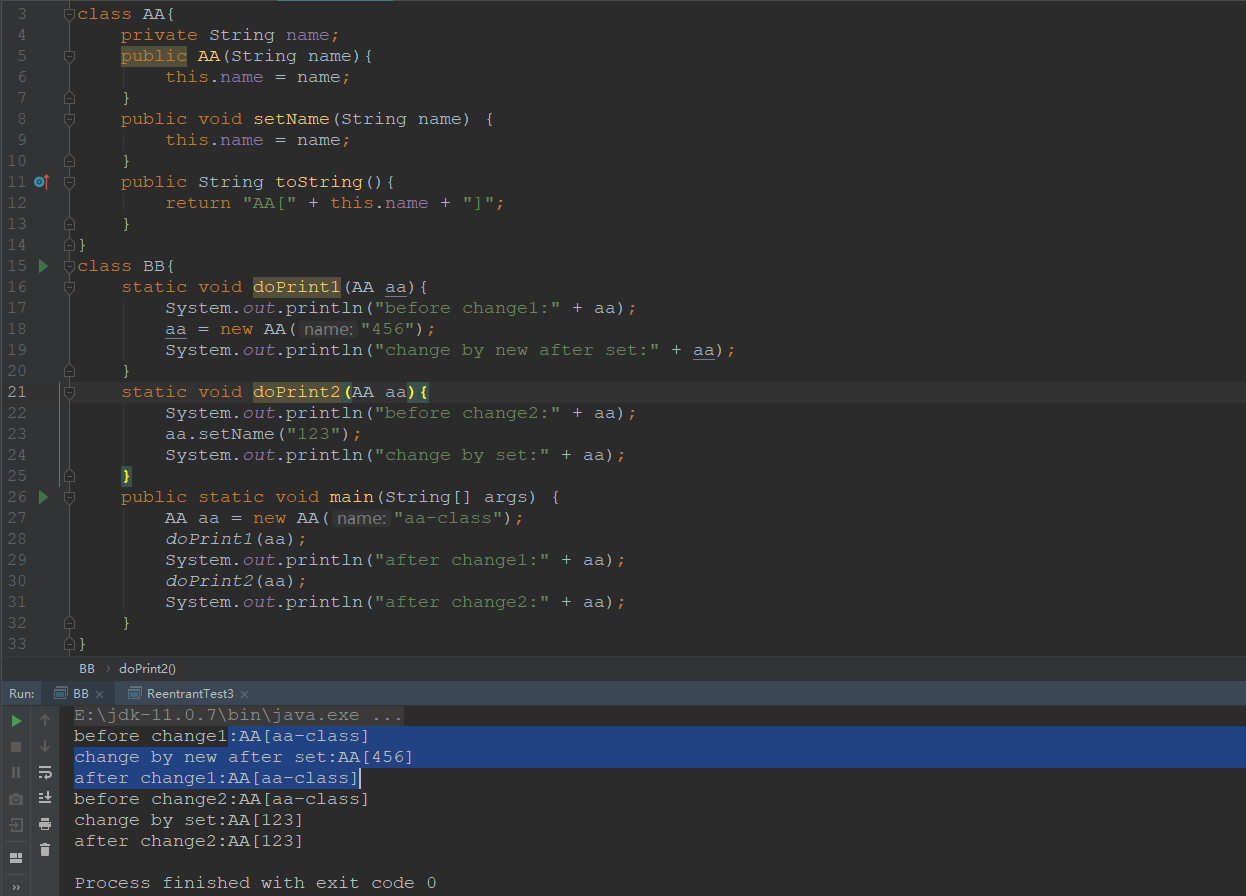


Example3:数组类型数据类型作为参数，List与数组同理



array的值从arr拷贝过来的，只不过，拷贝过来的值是一个对象的引用，array和arr指向同一个数组地址，因此外部对象的改变也会体现到原件。所以数组类型参数也是值传递。

Example4:自定义对象类型数据类型作为参数



对象与数组，都是引用类型，所以对象作为参数，传递的值也是引用，所有当改变拷贝对象的元素时，也会影响原件。

总结：

当方法的参数是基础数据类型或者是String类型时，不管在方法内部如何修改参数，在方法外传入的参数值都不会改变；

当方法的参数是除了String类型的对象类型(引用变量)时，也是值传递，只是这里的值是对象的引用，当修改拷贝对象的变量(属性)时，在方法外对象参数对应的变量(属性)值也会修改，如果是在方法内部直接改变变量的地址(即使用object=new Object())，那么在方法对象参数对应的变量(属性)不会修改。

1. **Java常用工具包**

Java.lang:系统的基础类；

Java.io:所有的输入输出有关的类，比如文件操作等；

Java.net：所有与网络有关的类；

Java.util:系统辅助类，特别是集合类。