public static void main(String[] args) {

Integer a = 1;  
 Integer b = 2;  
 System.*out*.printf("a = %s, b = %s\n", a, b);  
 *swap*(a, b);  
 System.*out*.printf("a = %s, b = %s\n", a, b);  
}  
  
*/\*\*  
 \* 错误的方式  
 \*/*public static void swap(Integer a, Integer b) {  
 int temp = a;  
 a=b;  
 b=temp;  
}

### 1、形参和实参

public void test() {

int shi\_can = 0;

testA(shi\_can);

}

public void testA(int xing\_can) {

}

在调用testA时，传递的就是实参，而在testA方法签名中的参数为形参。

从作用域上看，形参只会在方法内部生效，方法结束后，形参也会被释放掉，所以形参是不会影响方法外的。

### 2、值传递和引用传递

* 值传递：传递的是实际值，像基本数据类型。
* 引用传递：将对象的引用作为实参进行传递。

java基本类型数据作为参数是值传递，对象类型是引用传递

实参是可以传递给形参的，但是形参却不能影响实参，所以，当进行值传递的情况下，改变的是形参的值，并没有改变实参，所以无论是引用传递还是值传递，只要更改的是形参本身，那么都无法影响到实参的。对于引用传递而言，不同的引用可以指向相同的地址，通过形参的引用地址，找到了实际对象分配的空间，然后进行更改就会对实参指向的对象产生影响。

// 仅仅是一个java对象

public class IntType {

private int value;

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

}

// main方法

public class IntTypeSwap {

public static void main(String[] args) {

// CODE\_1

IntType type1 = new IntType();

type1.setValue(1);

IntType type2 = new IntType();

type2.setValue(2);

// CODE\_1

swap1(type1, type2);

System.out.printf("type1.value = %s, type2.value = %s", type1.getValue(), type2.getValue());

swap2(type1, type2);

System.out.println();

System.out.printf("type1.value = %s, type2.value = %s", type1.getValue(), type2.getValue());

}

public static void swap2(IntType type1, IntType type2) {

int temp = type1.getValue();

type1.setValue(type2.getValue());

type2.setValue(temp);

}

public static void swap1(IntType type1, IntType type2) {

IntType type = type1;

type1 = type2;

type2 = type;

}

}

输出结果：

type1.value = 1, type2.value = 2

type1.value = 2, type2.value = 1

从输出结果来看swap1并没有达到目的，

public static void swap1(IntType type1, IntType type2) {

IntType type = type1;

type1 = type2;

type2 = type;

}

从值传递的角度来看，对象参数传递采用的是引用传递，那么type1和type2传递过来的是指向对象的引用，在方法内部，直接操作形参，交换了形参的内容，这样形参改变，都是并没有对实参产生任何影响，也没有改变对象实际的值，所以，结果是无法交换

public static void swap2(IntType type1, IntType type2) {

int temp = type1.getValue();

type1.setValue(type2.getValue());

type2.setValue(temp);

}

而对于swap2，对象引用作为形参传递过来后，并没有对形参做任何的改变，而是直接操作了形参所指向的对象实际地址，那这样，无论是实参还是其他地方，只要是指向该对象的所有的引用地址对应的值都会改变。

### 3、自动装箱

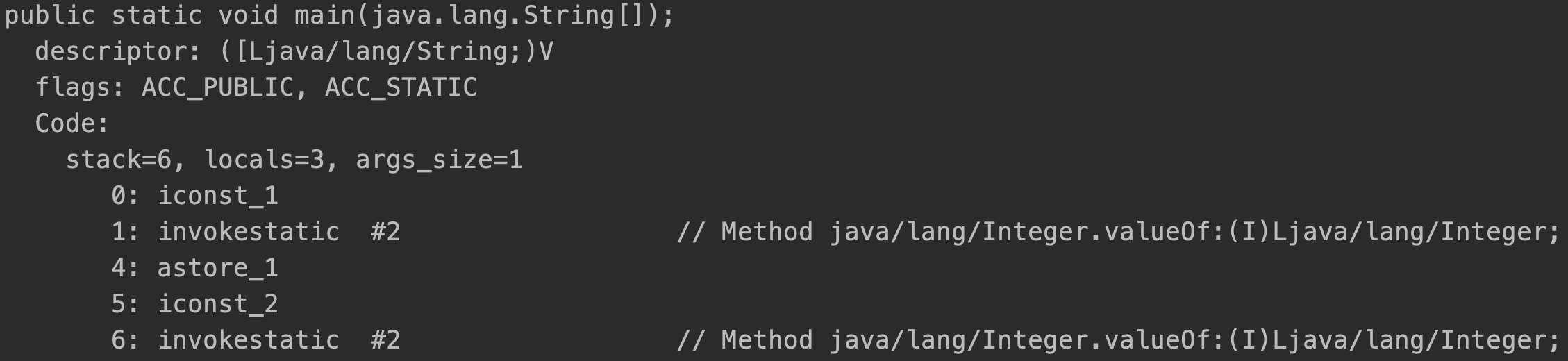
什么是自动装箱呢？jdk到底做了什么事？

自动装箱就是jdk调用了Integer的valueOf(int)的方法，看源码：

public static Integer valueOf(int i) {

if (i >= IntegerCache.*low* && i <= IntegerCache.*high*)  
 return IntegerCache.*cache*[i + (-IntegerCache.*low*)];  
 return new Integer(i);  
}

new出来一个对象，那么怎么知道自动装箱调用的是valueOf方法，通过查看反编译的字节码指令可以知道：



valueOf(int)方法被调用。

### 4、回归正题

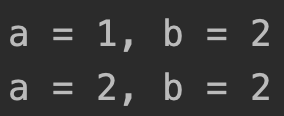
直接操作形参无法改变实际值，而Integer又没有提供set方法，那是不是无解了呢？既然出了肯定是有解的，可以实现，在上面swap2的例子中是通过set方法来改变值的，那么Integer有没有提供呢？没有。

Integer的构造函数，可以看到Integer对象实际值是用value属性来存储的，但是这个value是被final修饰的，没办法继续找，value没有提供任何的set方法。既然在万法皆不通的情况下，那就只能动用反射来解决问题：

public static void swap1(Integer a, Integer b) {

int temp = a.intValue();  
 try {  
 Field value = Integer.class.getDeclaredField("value");  
 value.setAccessible(true);  
 value.set(a, b);  
 value.set(b, temp);  
 } catch (NoSuchFieldException e) {  
 e.printStackTrace();  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 }  
}

输出结果：



wtf！！！

为什么会出现这种情况，无奈，调试会发现是在value.set的时候将Integer的缓存值改变了，因为value.set(Object v1, Object v2)两个参数都是对象类型，所以temp会进行自动装箱操作，会调用valueOf方法，这样会获取到错误的缓存值，所以，为了避免这种情况，就只能不需要调用缓存值，直接new Integer就可以跳过缓存，所以代码改成如下即可

public static void swap2(Integer a, Integer b) {

int temp = a.intValue();  
 try {  
 Field value = Integer.class.getDeclaredField("value");  
 value.setAccessible(true);  
 value.set(a, b);  
 value.set(b, new Integer(temp));  
 } catch (NoSuchFieldException e) {  
 e.printStackTrace();  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 }  
}