Java语法糖与Java编译器

## 自动拆、装箱

Auto-boxing、Auto-unboxing

之所以需要包装类型，是因为Java核心类库的API都是面向对象的。举个栗子，java核心类库中的容器类，就只支持引用类型。

当需要一个能够存储数值的容器类时，我们需要定义一个存储包装类对象的容器。

**public int** foo1() {  
 **ArrayList**<**Integer**> list = **new** ArrayList<>();  
 list.add(0);  
 **int** result = list.get(0);  
 **return** result;  
}

$javap –c FooV1

public int foo1();

Code:

0: new #2 // class java/util/ArrayList

3: dup

4: invokespecial #3 // Method java/util/ArrayList."<init>":()V

7: astore\_1

8: aload\_1

9: iconst\_0

10: invokestatic #4 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

13: invokevirtual #5 // Method java/util/ArrayList.add:(Ljava/lang/Object;)Z

16: pop

17: aload\_1

18: iconst\_0

19: invokevirtual #6 // Method java/util/ArrayList.get:(I)Ljava/lang/Object;

22: checkcast #7 // class java/lang/Integer

25: invokevirtual #8 // Method java/lang/Integer.intValue:()I

28: istore\_2

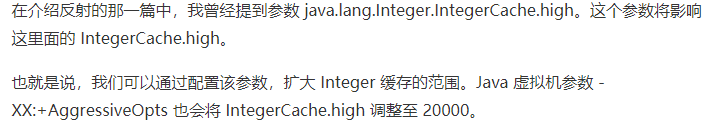
29: iload\_2

30: ireturn

在字节码偏移量为10的指令，我们调用了Integer.valueOf方法，将int类型转化为Integer类型，再存储在容器类中。

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

这是Integer.valueOf方法，当请求int值在某个范围内时，我们会返回缓存了的Integer对象，而请求int缓存范围之外的值时候，我们会新建一个Integer对象。



但是java并不支持对IntegerCache.low的更改，也就是说，对于-128的整数，我们无法直接使用由java核心类库所缓存的Integer对象。

25: invokevirtual java/lang/Integer.intValue:()I

这个是自动拆箱指令，这是一个实例方法，直接返回Integer中存储的int值。

## 泛型与类型擦除

13: invokevirtual #5 // Method java/util/ArrayList.add:(Ljava/lang/Object;)Z

…..

19: invokevirtual #6 // Method java/util/ArrayList.get:(I)Ljava/lang/Object;

22: checkcast #7 // class java/lang/Integer

在list.get方法中，其实获得的是Object对象，而我们在这里要进行向下转换，将Object强转为Integer，然后才能进行拆箱。

之所以出现这种情况，是因为**泛型擦除**，java程序中出现的泛型，在java虚机全部丢失，主要是为了兼容引入泛型的之前的代码。

并不是所有的泛型都被擦除成Object，对于限定了继承类的泛型参数，经过擦除之后，所有的泛型参数都将变成所限定的继承类。也就是java编译器会选取该泛型所能指代的所有类中层次最高的那个，作为替换泛型的类。

**class GenericTest** <T **extends** Number>{  
 T foo(T t) {  
 **return** t;  
 }  
}

$javap –c GenericTest

T foo(T);

descriptor: (Ljava/lang/Number;)Ljava/lang/Number;

flags: (0x0000)

Code:

stack=1, locals=2, args\_size=2

0: aload\_1

1: areturn

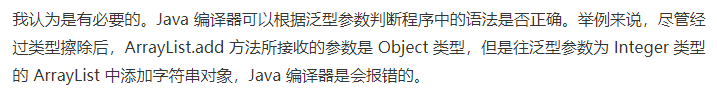
Signature: (TT;)TT;

可以看到foo的接收类型和和返回类型都是Number类型。



既然泛型类型会被擦除，那为什么还需要泛型呢？

### 需要泛型原因



ArrayList<Integer> list = new ArrayList<>();

list.add("0"); // 编译出错

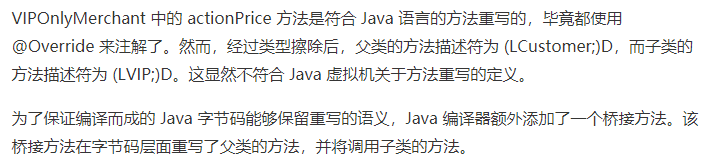
## 桥接方法

（在字节码层面重写了父类的方法，并将调用子类的方法）

泛型擦除带来不少问题，其中一个就是方法重写。

### 第一种

**class Merchant**<T **extends** Customer> {  
 **public double** actionPrice(T customer) {  
 **return** 0.0d;  
 }  
}  
  
**class VIPOnlyMerchant extends Merchant**<**VIP**> {  
 @Override  
 **public double** actionPrice(**VIP** customer) {  
 **return** 0.0d;  
 }  
}



cn.lxj.jvm40courses.course15.VIPOnlyMerchant();

descriptor: ()V

flags:

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method cn/lxj/jvm40courses/course15/Merchant."<init>":()V

4: return

LineNumberTable:

line 35: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcn/lxj/jvm40courses/course15/VIPOnlyMerchant;

public double actionPrice(cn.lxj.jvm40courses.course4.VIP);

descriptor: (Lcn/lxj/jvm40courses/course4/VIP;)D

flags: ACC\_PUBLIC

Code:

stack=2, locals=2, args\_size=2

0: dconst\_0

1: dreturn

LineNumberTable:

line 38: 0

LocalVariableTable:

Start Length Slot Name Signature

0 2 0 this Lcn/lxj/jvm40courses/course15/VIPOnlyMerchant;

0 2 1 customer Lcn/lxj/jvm40courses/course4/VIP;

public double actionPrice(cn.lxj.jvm40courses.course4.Customer);

descriptor: (Lcn/lxj/jvm40courses/course4/Customer;)D

flags: ACC\_PUBLIC, ACC\_BRIDGE, ACC\_SYNTHETIC

Code:

stack=2, locals=2, args\_size=2

0: aload\_0

1: aload\_1

2: checkcast #2 // class cn/lxj/jvm40courses/course4/VIP

5: invokevirtual #3 // Method actionPrice:(Lcn/lxj/jvm40courses/course4/VIP;)D

8: dreturn

LineNumberTable:

line 35: 0

LocalVariableTable:

Start Length Slot Name Signature

0 9 0 this Lcn/lxj/jvm40courses/course15/VIPOnlyMerchant;

}

Signature: #19 // Lcn/lxj/jvm40courses/course15/Merchant<Lcn/lxj/jvm40courses/course4/VIP;>;

SourceFile: "FooV1.java"

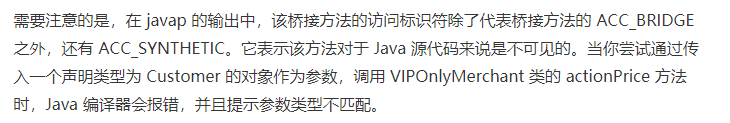
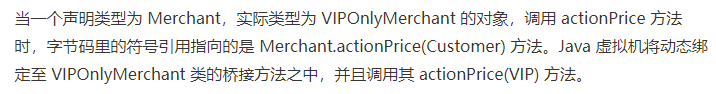
// 这个桥接方法等同于

public double actionPrice(Customer customer) {

return actionPrice((VIP) customer);

}

在例子中，VIPOnlyMerchant类将包含一个桥接方法，actionPrice（Customer），重写了父类同名的方法描述符方法。该桥接方法将传入的Customer参数强转为VIP类型，再调用原本的actionPrice（VIP）方法。



**VIP** vip = **new** VIP();  
Customer customer = **new** VIP();  
//new VIPOnlyMerchant().actionPrice(customer); // 报错  
**new** VIPOnlyMerchant().actionPrice(vip);

### 第二种

如果子类定义了一个与父类参数类型相同的方法，其返回类型为父类方法返回类型的子类，那么Java编译器也会生成桥接方法。

class NaiveMerchant extends Merchant

public java.lang.Double actionPrice(Customer);

descriptor: (LCustomer;)Ljava/lang/Double;

flags: (0x0001) ACC\_PUBLIC

Code:

stack=2, locals=2, args\_size=2

0: dconst\_0

1: invokestatic Double.valueOf:(D)Ljava/lang/Double;

4: areturn

public java.lang.Number actionPrice(Customer);

descriptor: (LCustomer;)Ljava/lang/Number;

flags: (0x1041) ACC\_PUBLIC, ACC\_BRIDGE, ACC\_SYNTHETIC

Code:

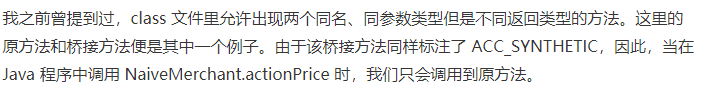
stack=2, locals=2, args\_size=2

0: aload\_0

1: aload\_1

2: invokevirtual actionPrice:(LCustomer;)Ljava/lang/Double;

5: areturn



## 其他语法糖

### Foreach

允许Java程序在for循环里遍历数组或者Iterable对象。对于数组来说，foreach循环将从0开始逐一访问数组中的元素，直至数组的末尾。

**public void** foo(**ArrayList**<**Integer**> list) {  
 **for** (**Integer** item : list) {  
 }  
}  
// 等同于  
**public void** bar(**ArrayList**<**Integer**> list) {  
 Iterator<**Integer**> iterator = list.iterator();  
 **while** (iterator.hasNext()) {  
 **Integer** item = iterator.next();  
 }  
}

字符串switch编译出来的字节码虽然看起来复杂，其实就是一个hash桶，由于每个case截取获取的字符串都是常量值，因此，java编译器会将原来的字符串switch转换为int值的switch，比较所输入的字符串hash值。

