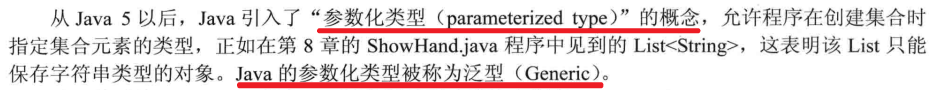
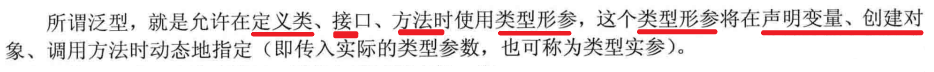
# 泛型-未完成待补充

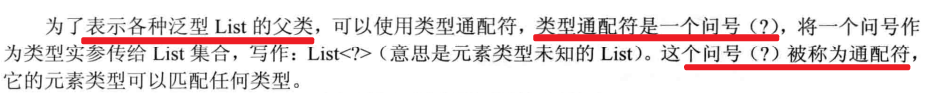
## 泛型类定义



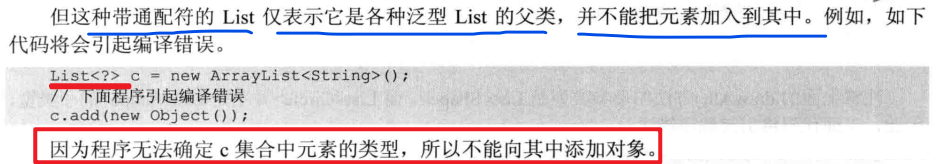


泛型的作用主要是用于动态的指定 类中参数的类型。此显示的方式指定了类型，避免的类型转换的操作。

## 通配符-形参



### 通配符的局限



**<T>** 表示一个具体类型的泛型，**[只能在定义类时指定]**。（T表示一个具体的类型可以是任意大写字母。）

**<T extends Shape>**表示Shape的任意一个子类的类型，**[只能在定义类时指定]**。此种指定方式指定一个范围内的一个具体类型**。**

**<?>** 表示的是任意一个泛型的父类，因此就无法确定具体的类型。不能添加数据，只能以父类（Object）的方式获取数据。[**只能在引用类型参数中使用，**]

## 泛型范围-上下限

**设定上限-表示确定了父类**

**<? extends Shape>** 表示Shape的任意一个子类泛型的父类，因此List<? extends Shape >只能以父类(Shape)的方式获取数据。[**只能在引用类型参数中使用**]



**设定下限-表示确定了子类**

<? supper Shape> 用法同上。

### 小结

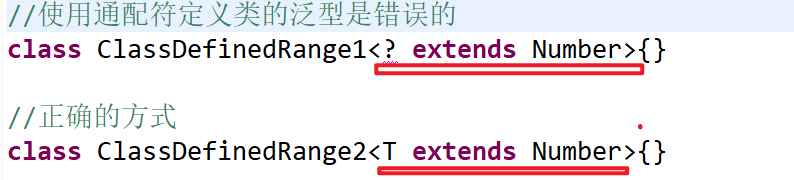
1.T表示一个具体的类型。?表示多个类型(也就是任意一个类型)，是T的父类。

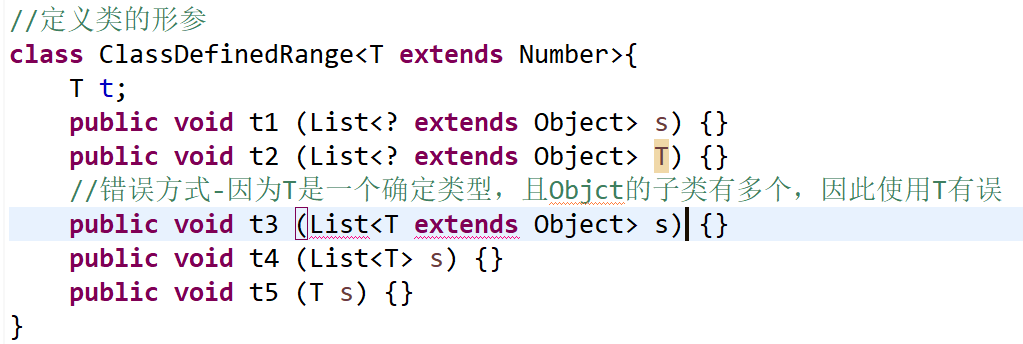
2.泛型中通配符(?)表示的是数据范围，因此不能在泛型类的定义中。【**因为泛型在类中定义需要指定确定的类型信息**】。只能用于形参中。

形参定义->是指在定义函数时使用的参数.即方法的参数都是形参。

3.而且通配符定义的类型只能查询数据，不能添加数据（**通配符指的就是 ?**）。

错误案例

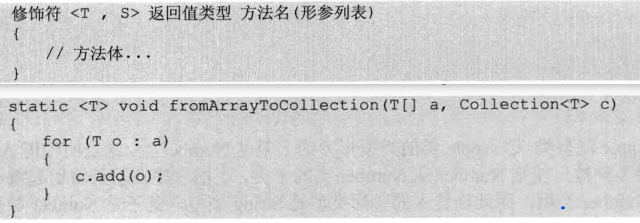




## 泛型方法

### 定义

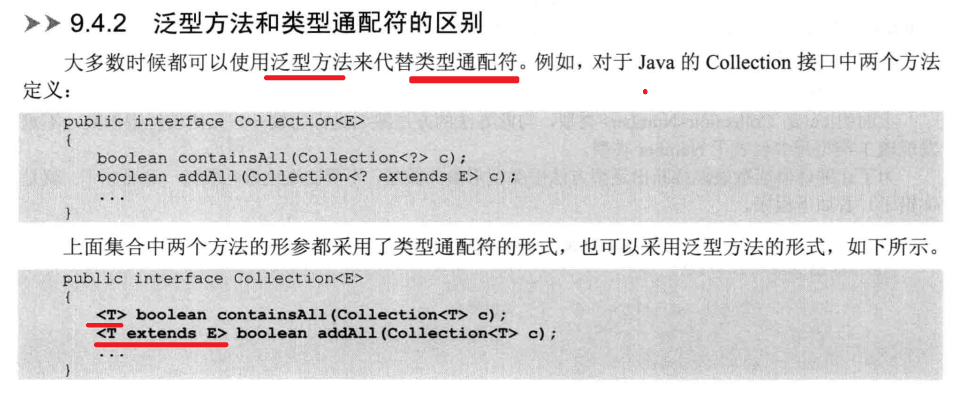
在类中没有定义泛型的时候，又想在方法中定义泛型，此时语法定义如下：



泛型方法定义： 返回值类型前边

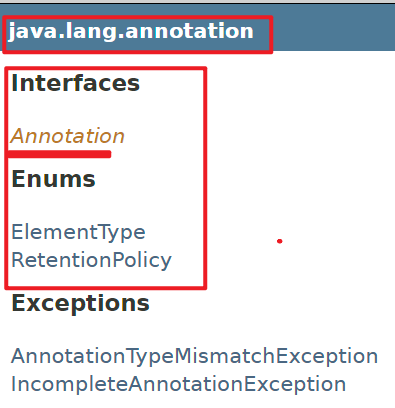
泛型类定义： 类名后边

public class A<T>{}



# 注解

### Java包



### 基本注解

**@Deprecated：**过时注解，用于标记已过时 & 被抛弃的元素（类、方法等）

**@Override：**复写注解，用于标记该方法需要被子类复写

**@SuppressWarnings：**阻止警告注解，用于标记的元素会阻止编译器发出警告提醒

**@SafeVarargs：**参数安全类型注解，用于提醒开发者不要用参数做不安全的操作 & 阻止编译器产生 unchecked警告，Java 1.7 后引入

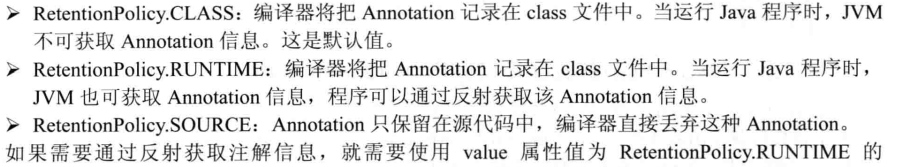
**@FunctionalInterface**: java8新增的函数式接口

### JDK元注解（meta-annotation）

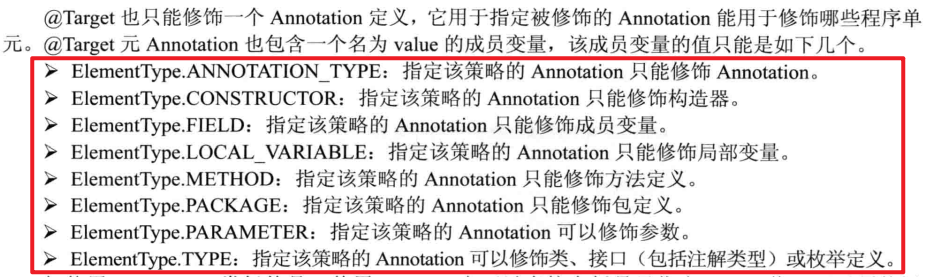
作用：对创建的注解进行范围限定

#### @Retention

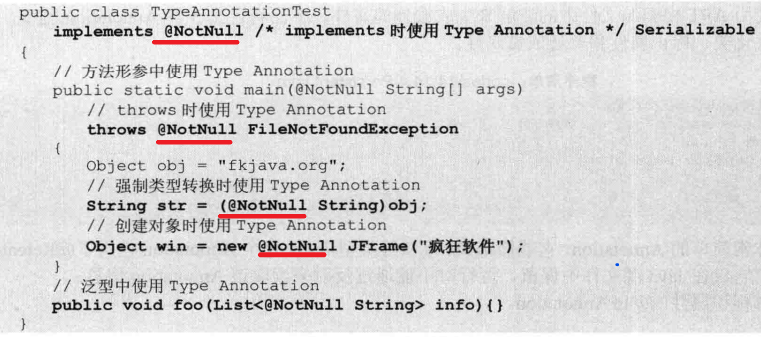
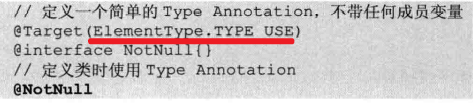
限定注解在JVM的保存范围



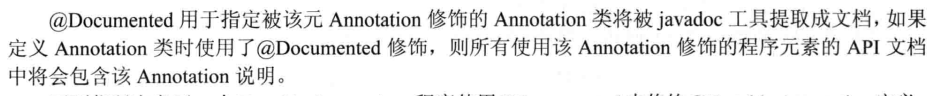
#### @Target



**ElementType.TYPE\_USE：称为类型注解，可用于任何地方.不受限于ElementType.TYPE的范围限定。**

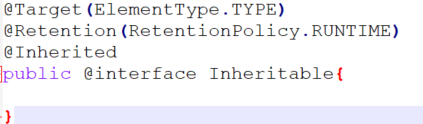
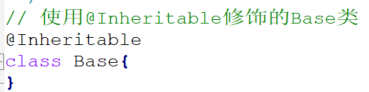


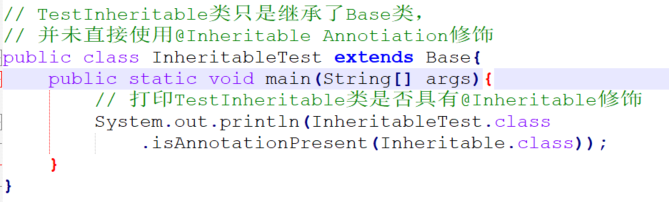
#### @Documented



#### @Inherited

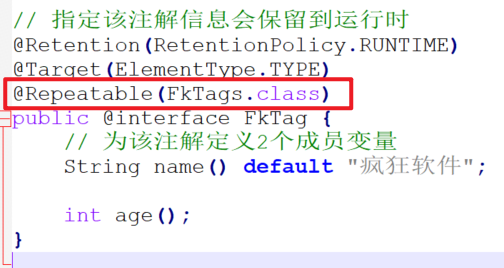


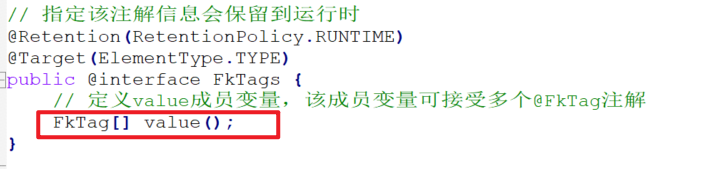
 



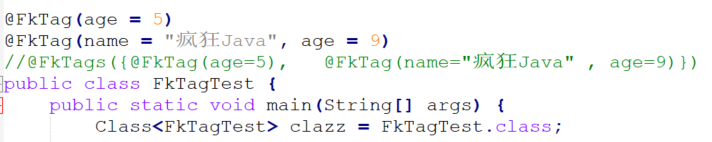
#### @Repeatable

1.定义重复注解需要定义两个注解

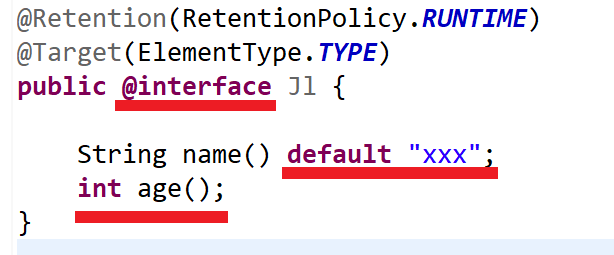




2.重复注解效果



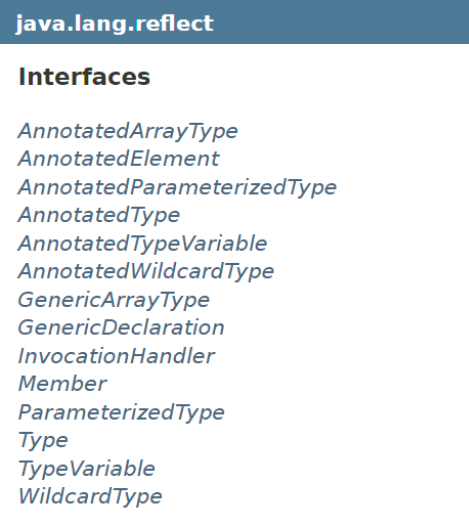
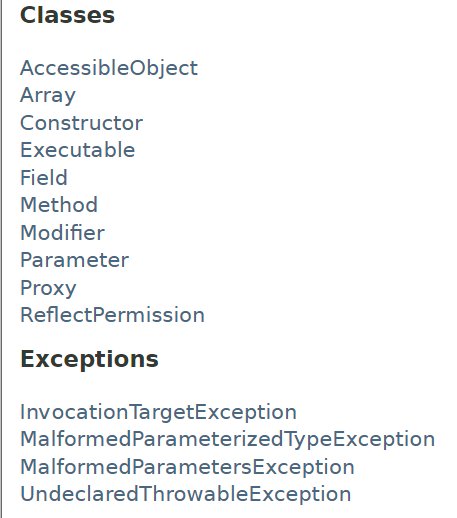
### 创建注解



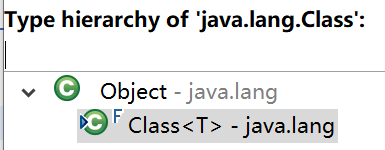
与接口定义相似,只不过可以设置默认值。

# 反射-base

## 位置

## Class对象API



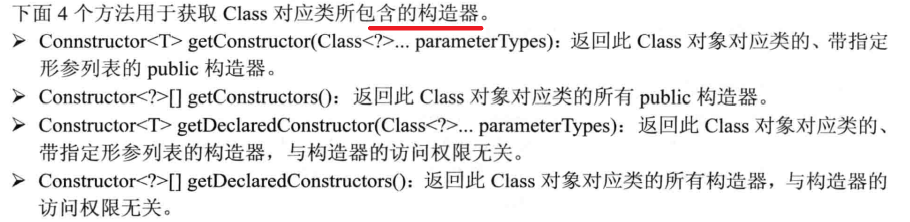
### 获取Class

对象： Class c = 对象.getClass()

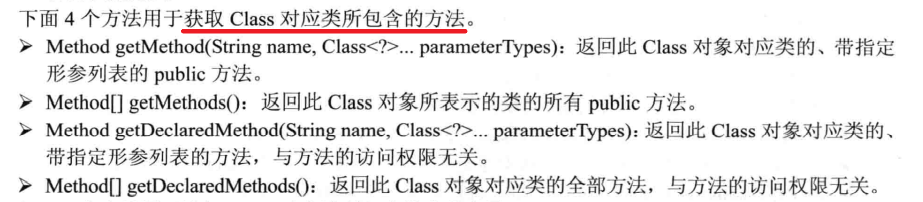
**类：** Class c = 类.class

**字符串格式类名称：** Class c = Class.forName(“”)

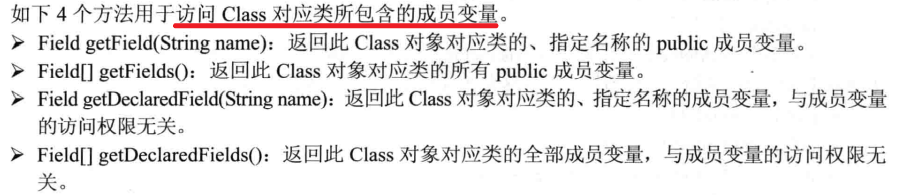
### 获取构造器



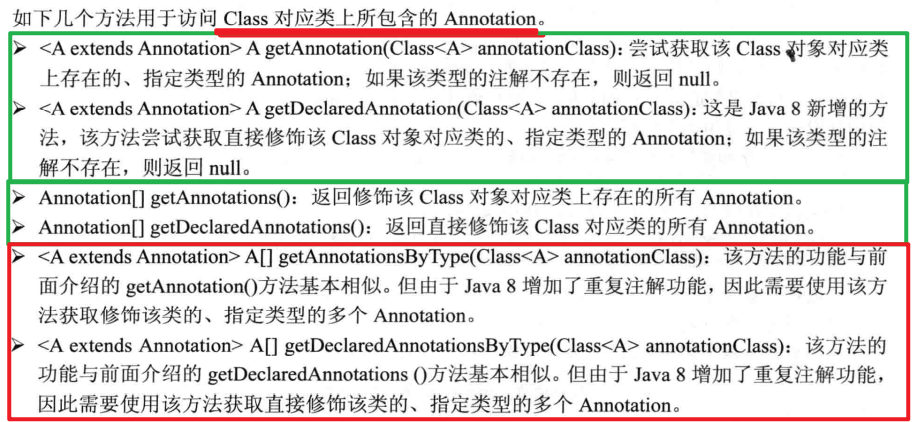
### 获取方法



### 获取成员变量



### 获取注解

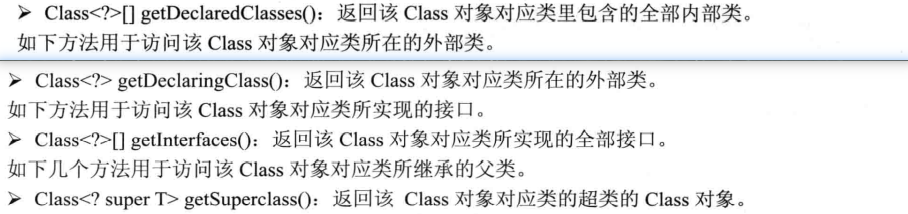




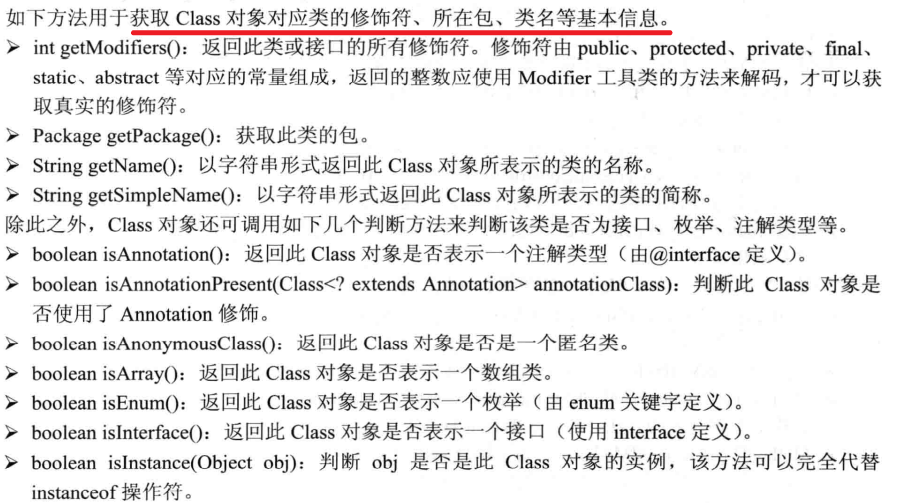
案例



### 获取内外部类-接口即超类

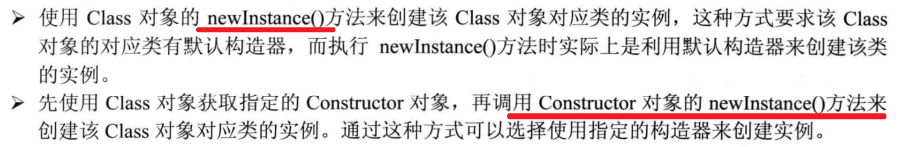


### 获取基本信息

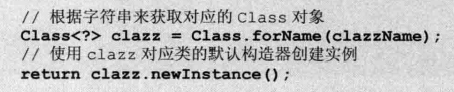


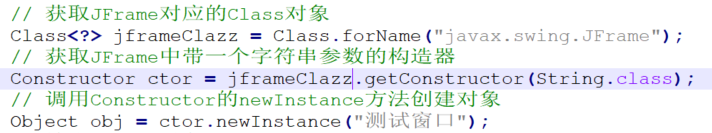
**isAssignableFrom()方法的功能: 用于检测一个类是否继承于一个类**

### 创建对象



案例





### 总结

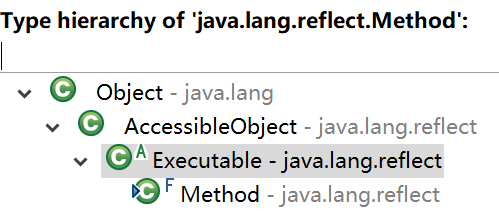
总的来说，getDeclaredXxxx()方法返回的是不包括继承的，和具体修饰符无关。getXxxx()包括继承的，但是只包括public修饰的。

[Java反射（2）getDeclaredXxxx()和getXxxx()的比较](https://www.jianshu.com/p/da4821bbe7bc)

[Java反射（1）总览](https://www.jianshu.com/p/92ac782fd92d)

## Method对象API

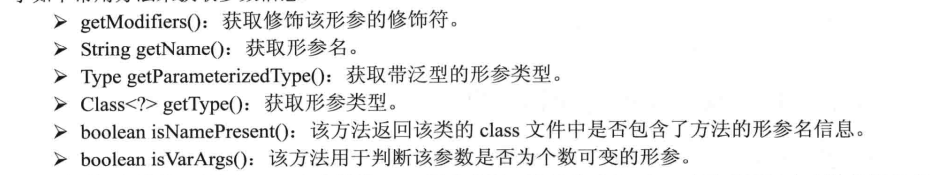
### 继承关系



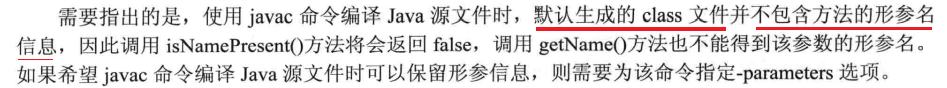
### 常用方法



Parameter（参数）提供的方法。

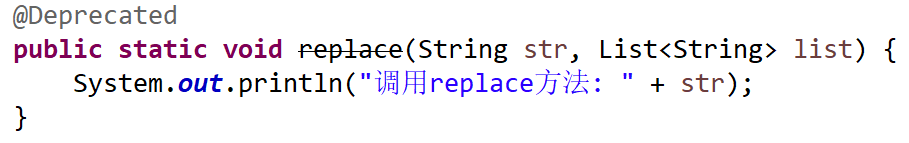


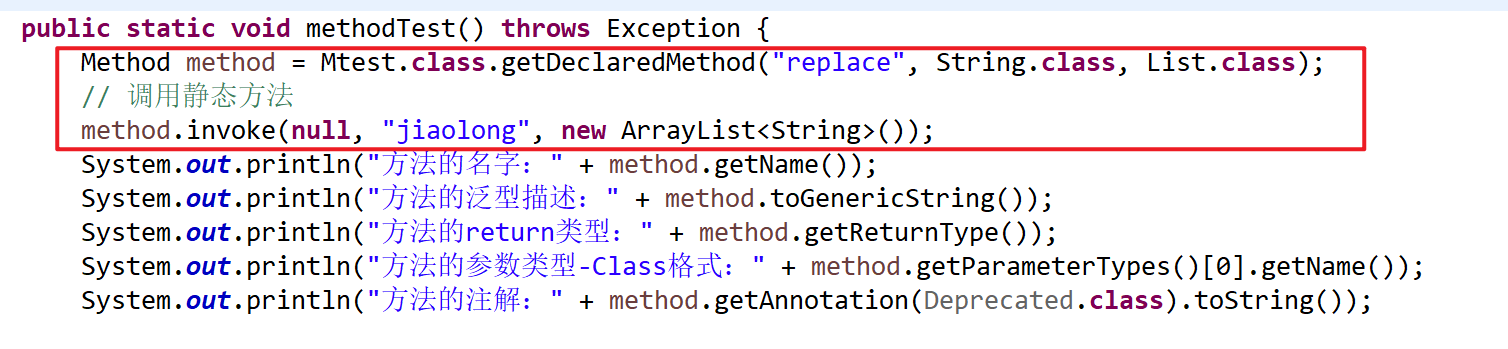
注意：

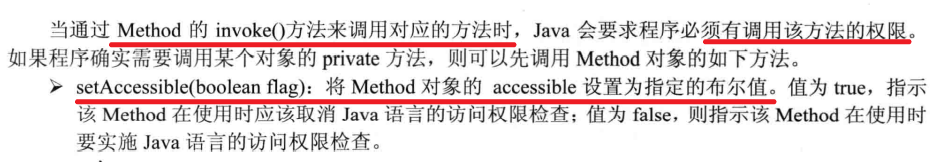


### 调用方法

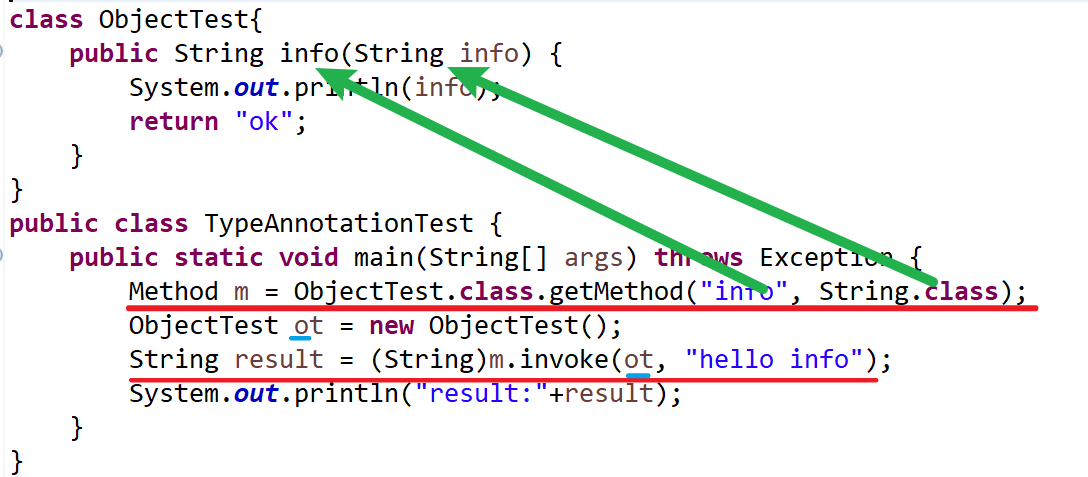
**调用静态方法**





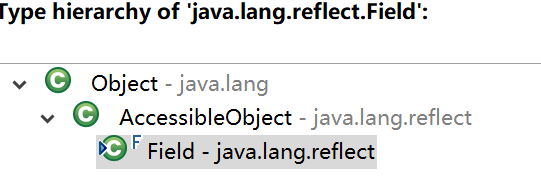


**调用对象方法**



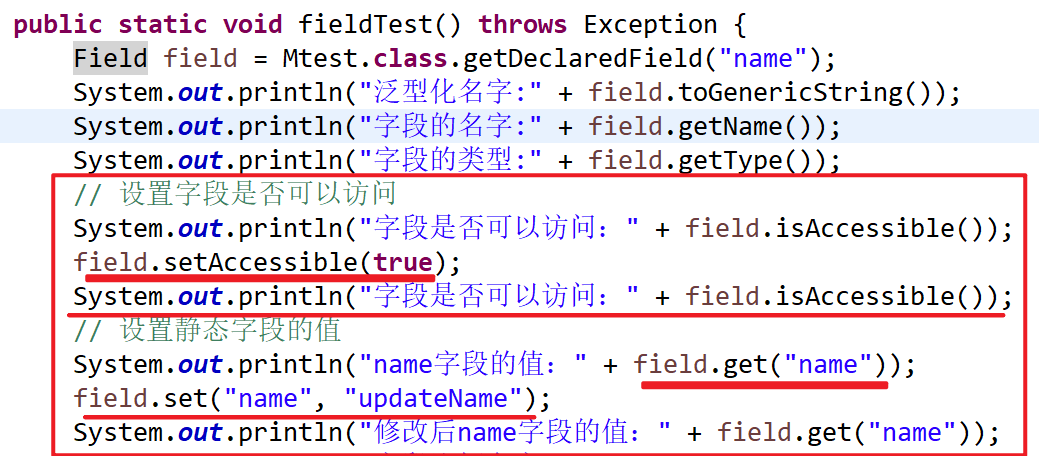
## Field对象API

位置



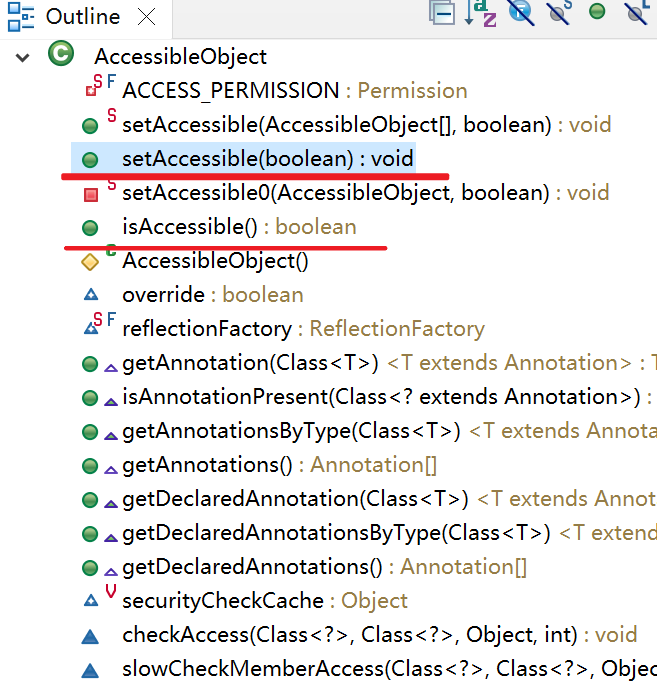
使用方法





私有方法访问控制





## 总结

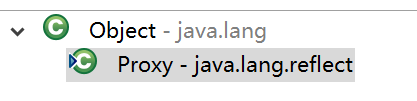
getDeclaredXxxx()方法返回的是不包括继承的，和具体修饰符无关。

getXxxx()包括继承的，但是只包括public修饰的。

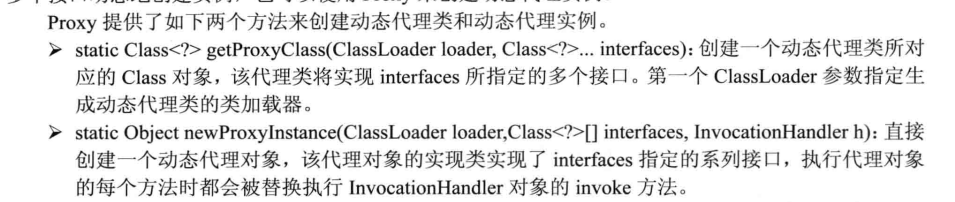
getGenericXxx()泛型相关方法

# 反射-动态代理

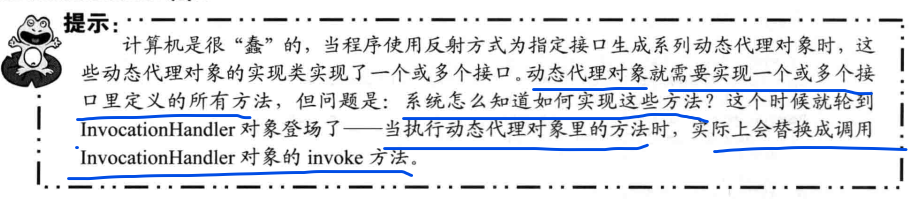
## 位置



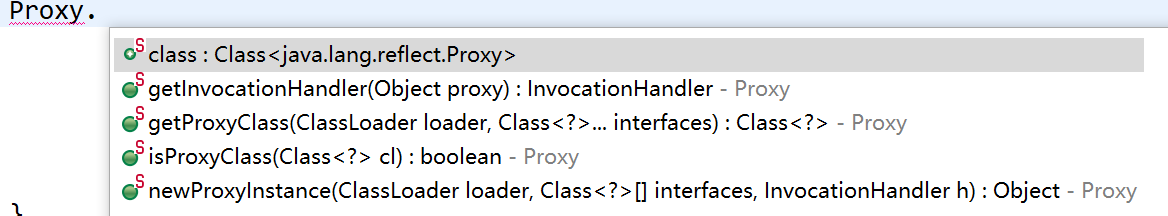
## 创建代理对象



Handler的作用



## Proxy静态方法



## 代理对象的特点

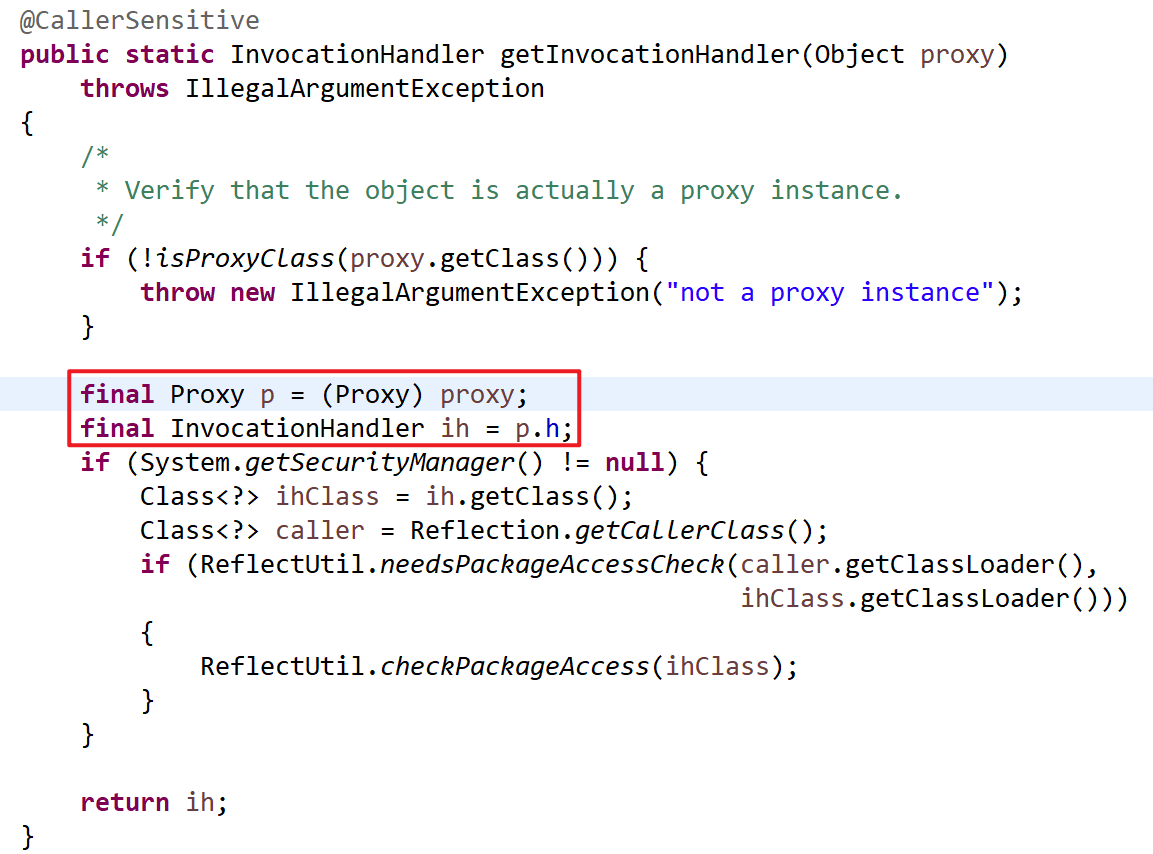
1.代理对象既有Proxy类的所有方法，也有被代理对象的所有方法。



2.因此可以将代理对象转换为接口实例或Proxy类的实例。



Proxy源码



1.说明生成的代理类包含接口的方法和Proxy的所有方法，当代理类执行接口的方法时，会调用InvocationHandler的Invoke方法去执行。

2.invoke方法中真正执行的接口方法的是方法通过反射调用 被代理对象的方法。

代理对象 就是代理 接口实现类对象去增加新的方法。也就是现实世界的 中间商赚差价，方便对一类的目标进行统一的服务。

# 反射泛型