一、Junit单元测试

测试分类：

1.黑盒测试：不需要写代码，给输入值，看程序是否能够输出期望值。

2.白盒测试：需要写代码。关注程序具体的执行流程。

Junit测试：白盒测试

使用步骤：

1.定义一个测试类（测试用例）

\* 建议：

\*测试类名：被测试的类名Test CalcTest

\*包名：xxx.xxx.xx.test cn.itcast.test

2.定义测试方法：可以独立运行

\*建议：

\*方法名：test测试的方法名 testAdd()

\*返回值：void

\*参数列表：空参

[3.给方法加@Test](mailto:3.给方法加@Test)注解

4.判定结果：

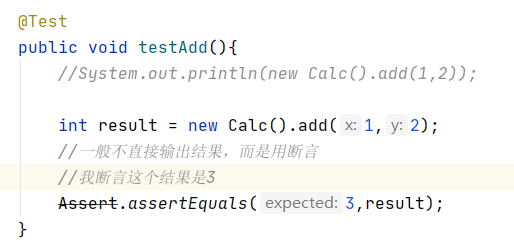
\*红色：失败

\*绿色：成功

\*一般我们会使用断言操作来处理结果

\*Assert.assertEquals(预期的结果，实际的结果);

/\*一般不会再单元测试中输出结果，而是做一个断言的操作\*/

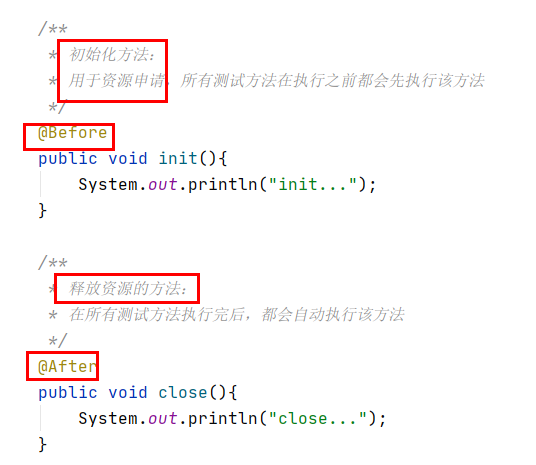


两个注解@Before和@After

无论是否出现异常，被这两个注解修饰的方法都会执行。

@Before： \*修饰的方法会在测试方法之前被自动执行。

@After： \*修饰的方法会在测试方法之后被自动执行。



二、反射

反射：也被称为框架设计的灵魂。

\*框架：半成品软件。可以在框架的基础上进行软件的开发，简化编码。

\*反射：将类的各个部分封装为其他对象，这就是**反射机制**。

\*好处：

1.在程序的运行过程中，操作这些对象。

2.可以**解耦**，**提高**程序的**可扩展性**。

\*获取Class对象的方式

1.Class.forName(**“全限定类名”**)：将字节码文件**加载进内存**，返回**Class对象**。

\*多用于配置文件，将类名定义在配置文件中。读取文件，加载类

2.类名.class:通过**类名的属性class**获取。

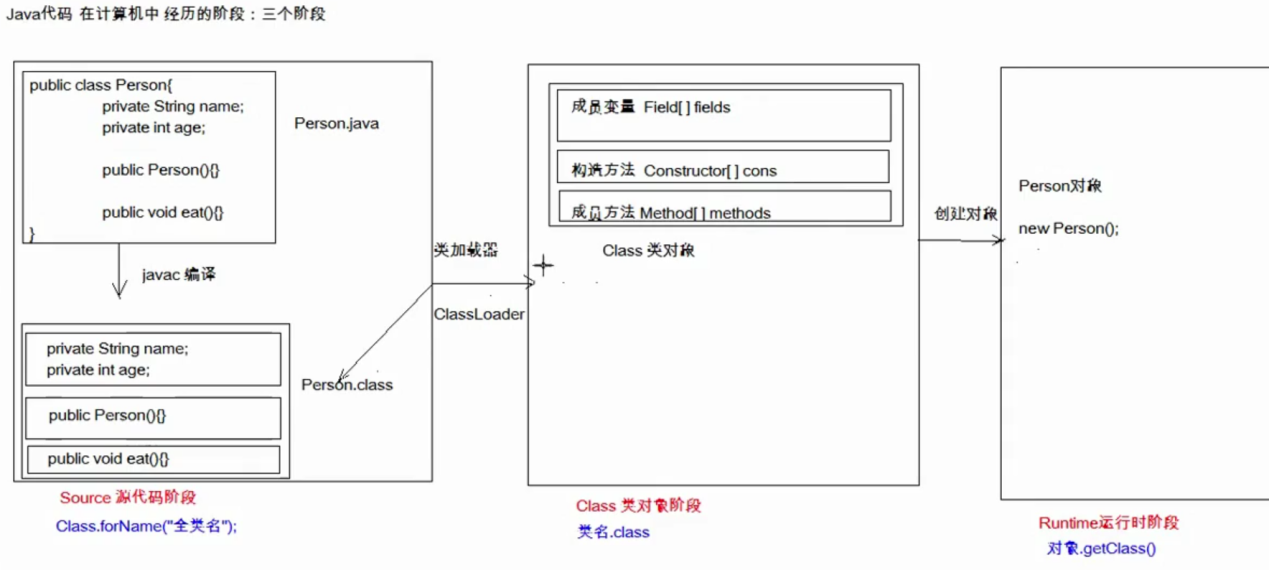
\*多用于参数的传递

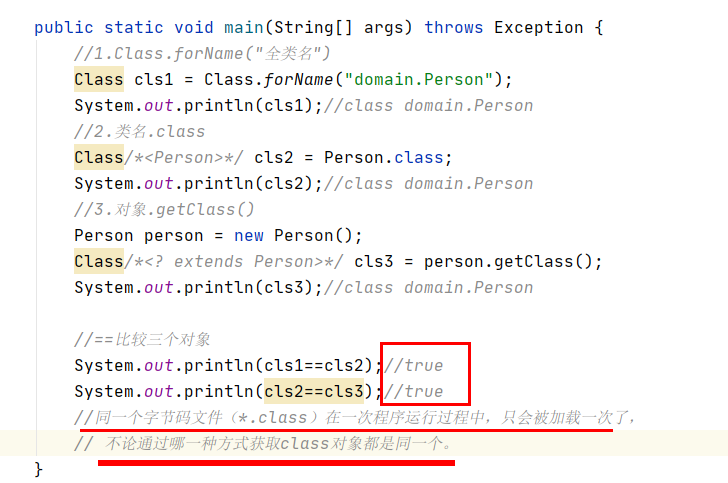
3.**对象.**getClass():getClass()方法在Object类中定义着。

\*多用于对象的获取字节码的方式

**\*结论：**

同一个字节码文件（\*.class）在一次程序运行过程中，只会被加载一次了，不论通过哪一种方式获取class对象都是同一个。





\*class对象的功能：

\*获取的功能：

1、获取成员变量们

\*Field[] getFields()：获取所有**public**修饰的成员变量

\*Field getField(String name)：获取指定**public**修饰的成员变量

\*Field[] getDeclaredFields():获取所有成员变量，不考虑权限修饰符

\*Field getDeclaredField(String name)

2、获取构造方法们

\*Constructor<?>[] getConstructors()

\*Constructor<T> getConstructor(类<?>… parameterTypes)

\*Constructor<?>[] getDeclaredConstructors()

\*Constructor<T> getDeclaredConstructor(类<?>… parameterTypes)

3、获取成员方法们

\*Method[] getMethods()

\*Method getMethod(String name,类<?>… parameterTypes)

\*Method[] getDeclaredMethods()

\*Method getDeclaredMethod(String name,类<?>… parameterTypes)

4、获取类名

\*String getName()

\*Field：成员变量

\*操作：

1.设置值

\*void set(Object obj,Object value)

2.获取值

\*get(Object obj)

3忽略访问权限修饰符的安全检查 （一般用于带declare的方法）

\*setAccessible(true):**暴力反射**

\*Constructor：构造方法

\*创建对象

\*T newInstance(Object… initargs)

\*如果使用空参数构造方法创建对象，操作可以简化:Class对象的newInstance方法

\*Method：方法对象

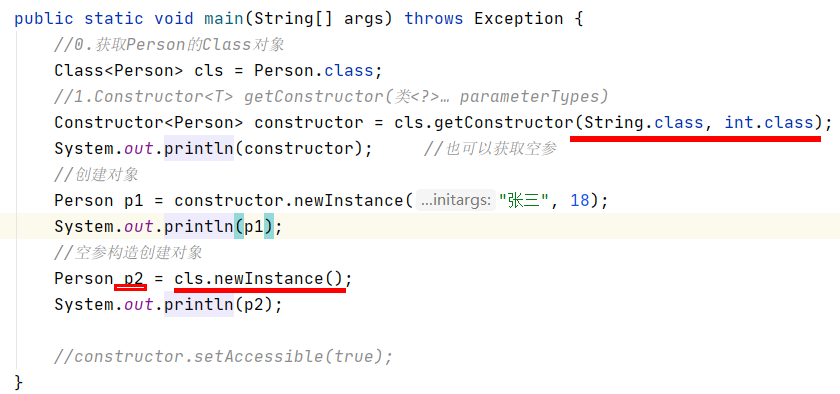
\***执行方法**：

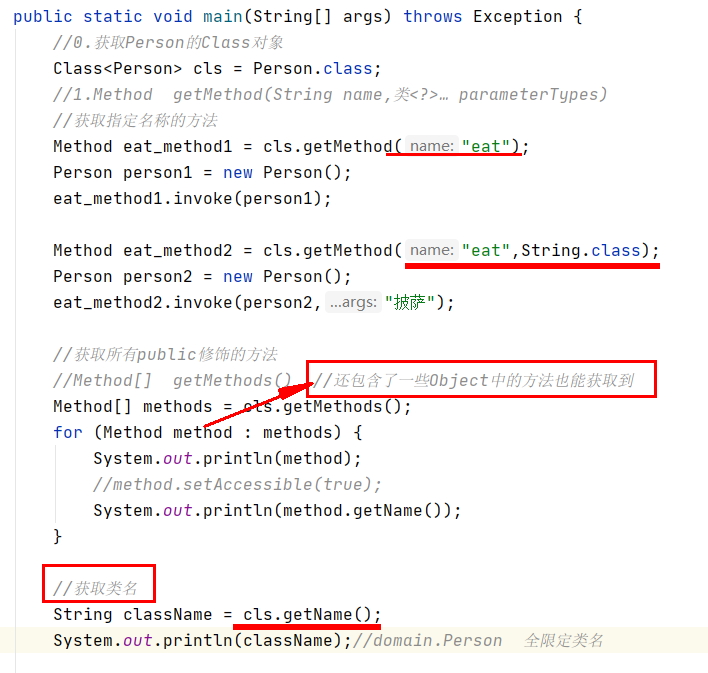
\*Object invoke(Object obj,Object… args)

\*获取方法名称：

\*String getName:获取方法的名称



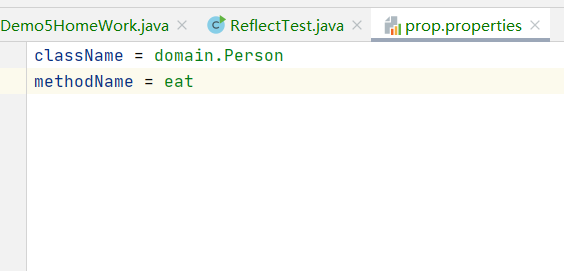




案例：

\*需求：写一个“框架”，可以帮我们创建任意类的对象，并且执行其中任意方法。





三、注解（Annotation）【也叫元数据】

\*概念：说明程序的，给计算机看的。

\*注释：用文字描述程序的。给程序员看的。

\*概念描述：它是JDK1.5后版本引入的一个特性。说明程序的。使用注解：@注解名称

\*作用分类：

编写文档：通过代码里标识的元数据（即注解）生成文档[生成文档javadoc文档]

代码分析：通过代码里标识的元数据对代码进行分析[使用反射]

编译检查：通过代码里标识的元数据让编译器能够实现基本的编译检查

\*JDK中预定义的一些注解

@Override ：检测被该注解标注的方法，是否是继承自父类（父接口）。

@Deprecated：将该注解标注的内容，已过时。

@SupperssWarnings：压制警告

\*一般传递参数all：@SupperssWarnings（“all”）

\*自定义注解

\*格式：

元注解

public @interface 注解名称{

属性列表

}

\*本质：注解本质上就是一个接口，该接口默认继承Annotation接口

\* public interface MyAnno extends java.lang.annotation.Annotation{}

（反编译 ： javap MyAnno.class）

**\*属性**：**接口中的抽象方法**

\*要求：

1.属性的返回值类型有下列：

\*基本数据类型

\*String

\*枚举

\*注解

\*以上类型的数组

2.定义了属性，在**使用**时需要**給属性赋值**

1.如果定义属性时，使用default关键字给属性默认初始化值，则使用注解时，可以不进行属性的赋值。

2.如果只有一个属性需要赋值，并且属性的名称是value，则value可以省略，直接定义值即可。

3.数组赋值时，值使用{}包裹。如果数组中只有一个值，则{}可以省略。

**\*元注解**：**用于描述注解的注解**

1、@Target：描述注解能够作用的位置

\*ElementType取值：

\*TYPE：可以作用于类上

\*METHOD：可以作用于方法上

\*FIELD：可以作用于成员变量上

2、@Retention：描述注解被保留的阶段

@Retention（RetentionPolicy.RUNTIME）：当前被描述的注解，会保留到class字节码文件中，并被JVM读取到。（一般自定义的注解都是保留在这个阶段）

3、@Documented：描述注解是否被抽取到api文档中

4、@Inherited：描述注解是否被子类继承

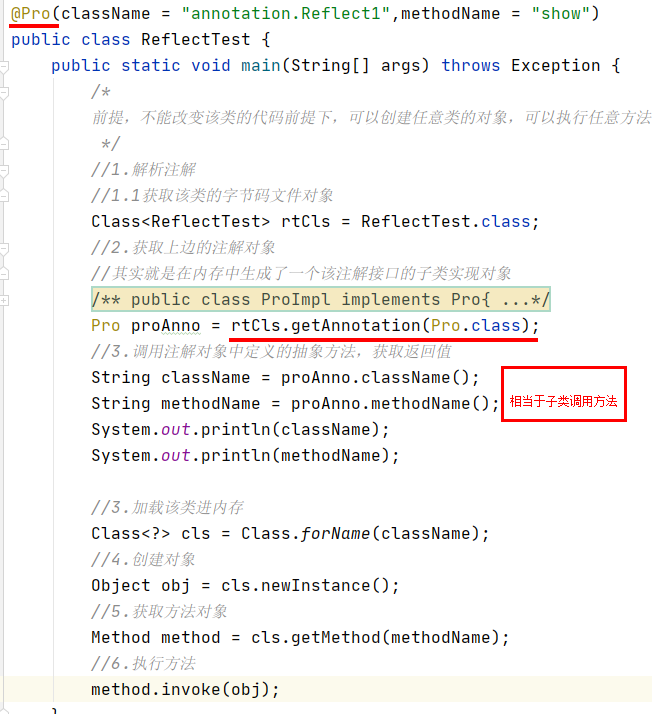
\*在程序中解析（使用）注解：获取注解中定义的属性值

1.获取注解定义的位置的对象（Class,Method,Field）

2.获取指定的注解 \*getAnnotation（Class）

*//其实就是在内存中生成了一个该注解接口的子类实现对象  
/\*\*  
 \* public class ProImpl implements Pro{  
 \* public String className(){  
 \* return "annotation.Reflect1";  
 \* }  
 \* public String methodName(){  
 \* return "show";  
 \* }  
 \* }  
 \*/*

3.调用注解中的抽象方法获取配置的属性值



\*小结：

1.以后大多数时候，我们会使用注解，而不是自定义注解

2.注解给谁用？

1、编译器

2、给解析程序用

3.注解不是程序的一部分，可以理解为注解就是一个标签。

案例：简单的测试框架

