**今天完成的事情：**

**第一部分：JAVA基础知识**

**学习预览：**

* Junit单元测试方法
* Java异常处理

**第二部分：修真任务**

**学习预览：**

* 上传项目至服务器，执行main方法

**第一部分**

1. **Junit单元测试**

**单元测试又称模块测试，属于白盒测试，是最小单位的测试**

**JUnit是一个回归测试框架（regression testing framework）**

模块分为程序模块和功能模块：

* **功能模块**

功能模块指实现了一个完整功能的模块（单元）

一个完整的程序单元具备输入、加工和输出三个环节

而且每个程序单元都应该有正规的规格说明，使之对其输入、加工和输出的关系做出明确的描述

* **程序模块**

1. **简单单元测试入门**

编程是智力活动，编程前需要把干什么、如何干等问题想清楚

因此当想用程序解决问题时，要会写三种码：

* **伪代码**
* **产品代码**
* **测试代码**

1. **实例解释三类代码**

**需求**：在Calculator类中实现简单计算器的功能

1. **伪代码：**

* **简单计算器：**

加法：返回两个数相加后的结果

减法：返回两个数相减后的结果

乘法：返回两个数相乘后的结果

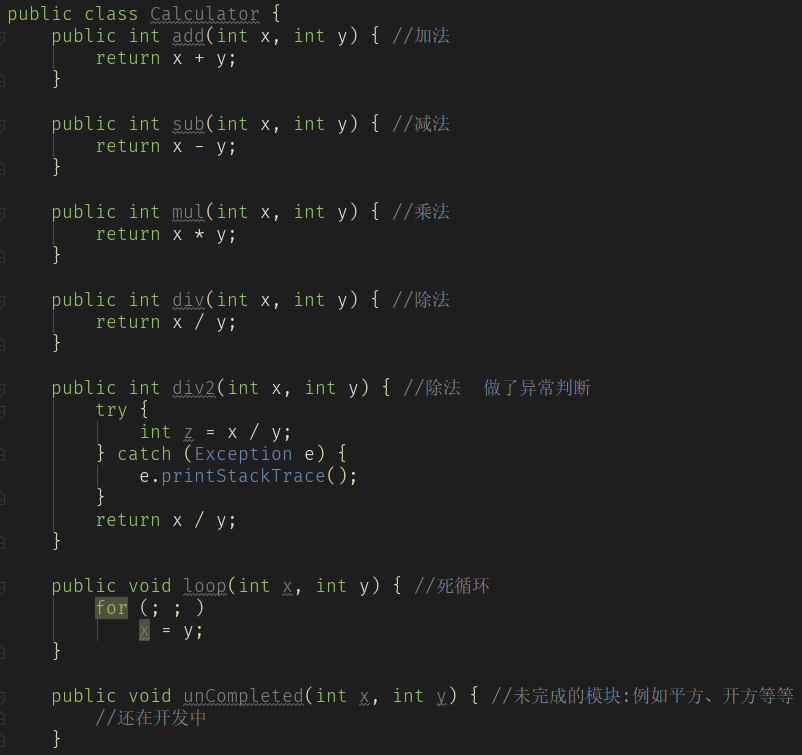
除法：除数不为0，返回两个数相除后的结果

除数为0，打印异常信息在程序中出错的位置及原因

其他：未完成的模块例如平方、开方等等

1. **产品代码：**

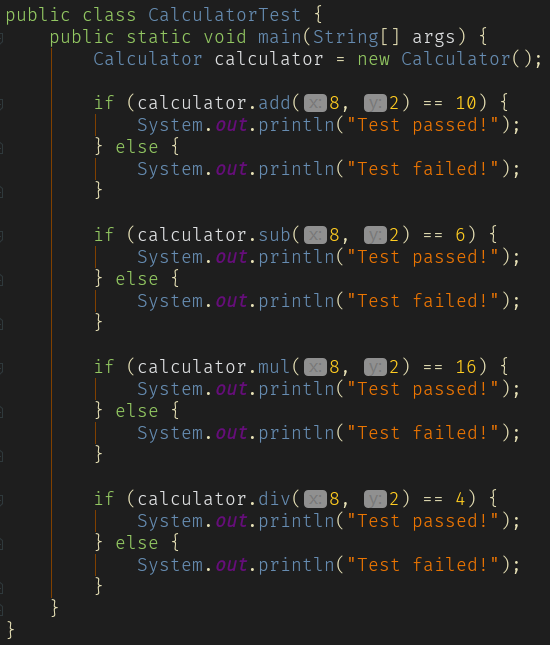
* 有了**伪代码**，用特定语言翻译一下，就是可用的**产品代码**了
* 翻译之后的产品代码Calculator.java:



1. **测试代码：**

XXX类的测试代码，一般建立XXXXTest的类写入测试代码

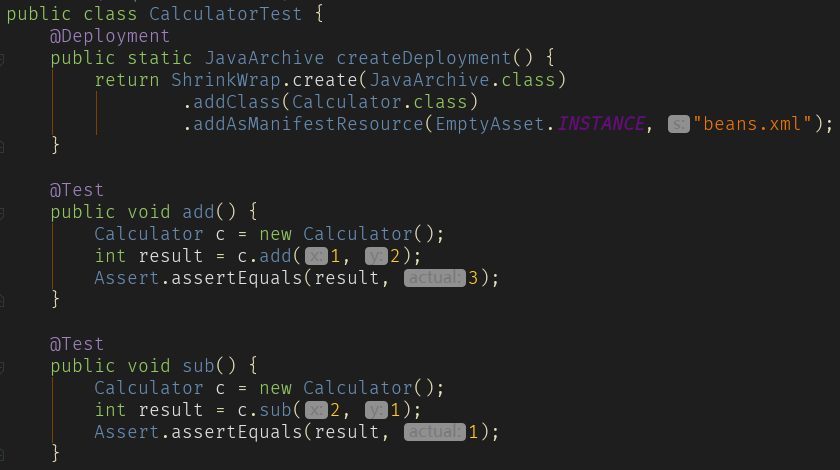
**1# 传统main()方法测试代码为**



这里设计了一个**测试用例（Test Case）**

**测试用例**是为某个特殊目标而编制的一组：测试输入、执行条件、预期结果，以便测试某个程序路径或核实是否满足某个特定需求

**2# 使用Junit方法测试代码为**



好处为可以单独运行每一个测试单元

1. **Junit测试注解**
2. **@Test**

注解参数

* **(expected=XXException.class)**

如果程序的异常和XXException.class一样，则测试通过

* **(timeout=100)**

如果程序的执行能在100毫秒之内完成，则测试通过

1. **@Ignore**

被忽略的测试方法：加上之后，暂时不运行此段代码

1. **@Before**

每一个测试方法之前运行

1. **@After**

每一个测试方法之后运行

1. **@BeforeClass**

方法必须必须要是静态方法（static 声明）

所有测试开始之前运行，注意区分before，是所有测试方法

1. **@AfterClass**

方法必须要是静态方法（static 声明）

所有测试结束之后运行，注意区分 @After

1. **Junit测试代码改进**

可以看到测试代码中，每个@Test方法中都由类的实例化代码

Calculator c = new Calculator()

可以将其放入@Before中，在每个测试单元前执行

1. **编写测试类的原则**
2. 测试方法上必须使用@Test进行修饰
3. 测试方法必须使用public void 进行修饰，不能带任何的参数
4. 新建一个源代码目录来存放我们的测试代码

用来将测试代码和项目业务代码分开

1. 测试类所在的包名应该和被测试类所在的包名保持一致
2. 测试单元中的每个方法必须可以独立测试，测试方法间不能有任何的依赖
3. 测试类使用Test作为类名的后缀（不是必须）
4. 测试方法使用test作为方法名的前缀（不是必须）
5. **Java异常处理**

**异常是程序中的一些错误，但并不是所有错误都是异常：**

* 代码少了一个分号，运行结果报错为java.lang.Error
* 输出System.out.println(11/0)，会抛出异常java.lang.ArithmeticException

**异常发生的原因很多，通常包含以下几类：**

* 用户输入了非法数据
* 要打开的文件不存在
* 网络通信时连接中断
* JVM内存溢出

**要理解Java异常处理的工作原理，需要掌握以下三种类型的异常：**

* **检查性异常**

用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略

* **运行时异常**

运行时异常是可能被程序员避免的异常。

与检查性异常相反，运行时异常可以在编译时被忽略

* **错误**

错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。

例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的

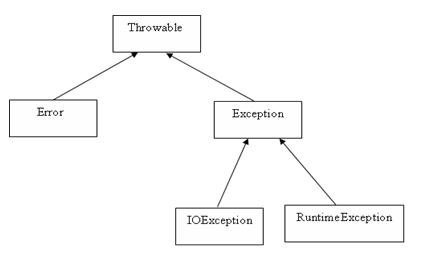
1. **Exception类的层次**

异常类是从java.lang.Exception类继承的子类

Exception类是Throwable类的子类，Error类也是Throwable类的子类

**异常类有两个主要的子类：**

* IOException 类
* RuntimeException 类



1. **Java内置异常类**

Java 语言定义了一些异常类在 java.lang 标准包中

由于 java.lang 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用

**异常详解见：**http://www.runoob.com/java/java-exceptions.html

1. **异常方法**

**详细介绍Throwable类的主要方法：**

1. public String getMessage()

返回关于发生的异常的详细信息

这个消息在Throwable 类的构造函数中初始化了

1. public Throwable getCause()

返回一个Throwable 对象代表异常原因

1. public String toString()

使用getMessage()的结果返回类的串级名字

1. public void printStackTrace()

打印toString()结果和栈层次到System.err，即错误输出流

1. public StackTraceElement [ ] getStackTrace()

返回一个包含堆栈层次的数组

下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底

1. public Throwable fillInStackTrace()

用当前的调用栈层次填充Throwable对象栈层次

添加到栈层次任何先前信息中

1. **捕获异常**

**Java使用try---catch或try---catch---finally代码块来捕获异常**

**程序会捕获try代码块里面的代码，若捕获到异常则进行catch代码块处理**

**若有finally则在catch处理后执行finally里面的代码**

**但是有个问题：**

**finally永远都会在catch的return前被执行**

具体语法如下：

**try{**

**// 程序代码**

**}**

**catch (ExceptionName e1) {**

**//Catch 块**

**}**

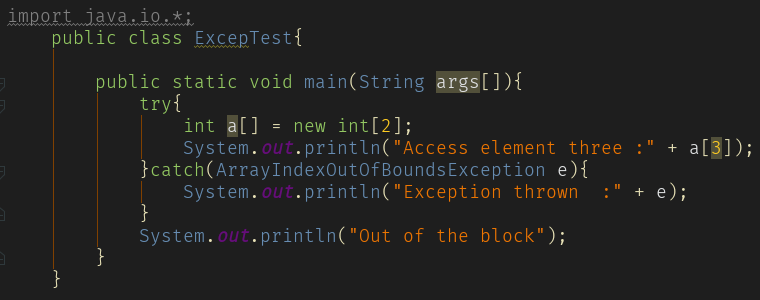
* **Catch 语句包含要捕获异常类型的声明**

当保护代码块中发生一个异常时，try 后面的 catch 块就会被检查

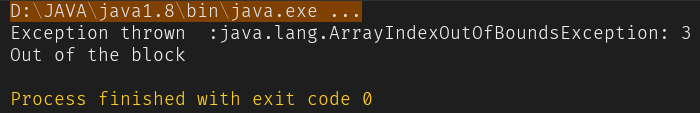
1. **实例说明**

下例声明一个数组，该数组拥有两个元素

当代码试图访问第三个元素时会抛出一个异常，使用try catch捕获



以上代码编译运行输出结果如下：



1. **多重捕获**

一个 try 代码块后面跟随多个 catch 代码块的情况就叫多重捕获

代码如下：

**try{**

**// 程序代码**

**}**

**catch(异常类型1 异常的变量名1){**

**// 程序代码**

**}**

**catch(异常类型2 异常的变量名2){**

**// 程序代码**

**}**

**catch(异常类型2 异常的变量名2){**

**// 程序代码**

**}**

1. **throws和throw关键字**

**区别：**

* **throws表示一个方法声明可能抛出一个异常，关键字放在方法签名的尾部**
* **throw表示此处抛出一个已定义的异常**

**（可以是自定义需继承Exception，也可以是java自己给出的异常类）**

1. **finally关键字**

finally 关键字用来创建在 try 代码块后面执行的代码块

无论是否发生异常，finally 代码块中的代码总会被执行

在finally代码块中，可以运行清理类型等收尾善后性质的语句

语法如下：

**try{**

**// 程序代码**

**}**

**catch(异常类型1 异常的变量名1){**

**// 程序代码**

**return 1；**

**}**

**catch(异常类型2 异常的变量名2){**

**// 程序代码**

**return 2；**

**}**

**finally{**

**// 程序代码**

**// 永远在匹配的catch中的return之前执行finally中的代码**

**// 并最后返回finally中的return值**

**return 3；**

**}**

**明天计划完成的事情：**

昨天部署完服务器的服务后，今天按照计划学习了Junit单元测试方法，异常处理方法

明天计划继续完成

1. 测试不关闭连接池的时候，在main方法中写1000个循环调用
2. 数据库的批量操作

**遇到的问题：**

在服务器上执行main方法时一直无法执行，原因是需要绝对路径

在学习异常处理的时候，一开始没有明白异常处理的意义，最后在其他资料中看到了详细的解释

**收获：**

1. 学会了Junit的测试方法，包括注释、断言测试
2. 学会了异常处理的相关概念和操作