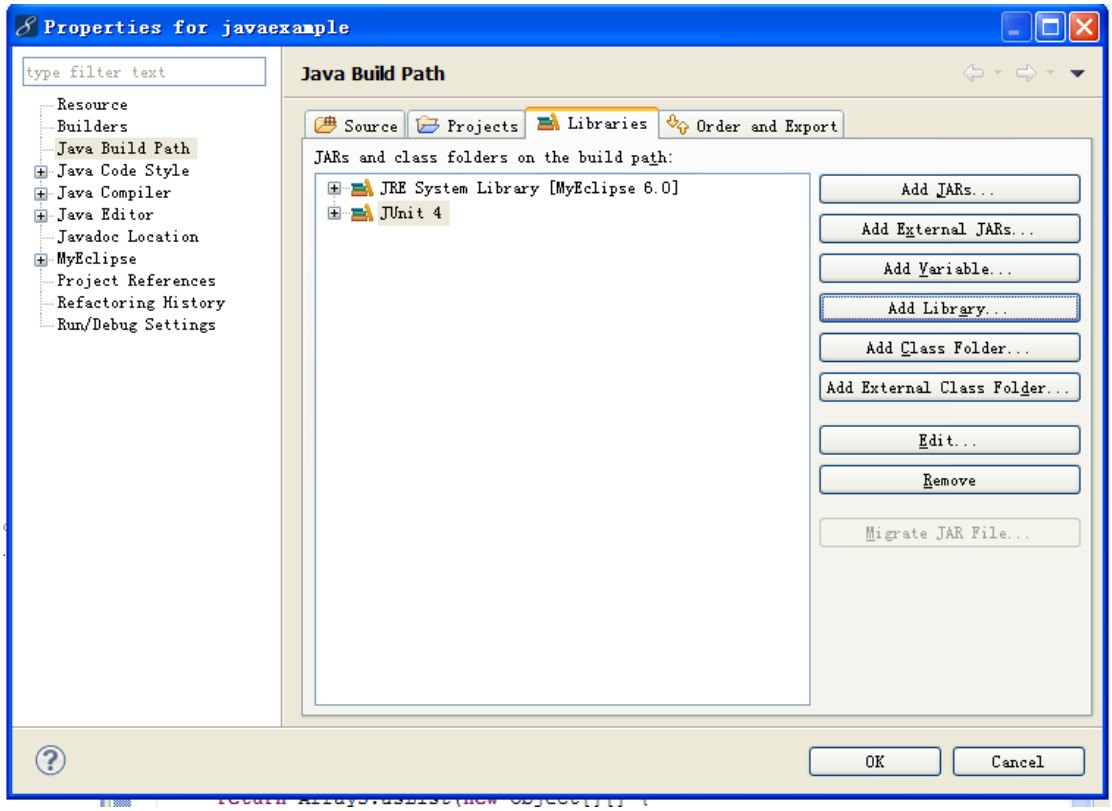
JUnit 4 学习笔记——入门篇

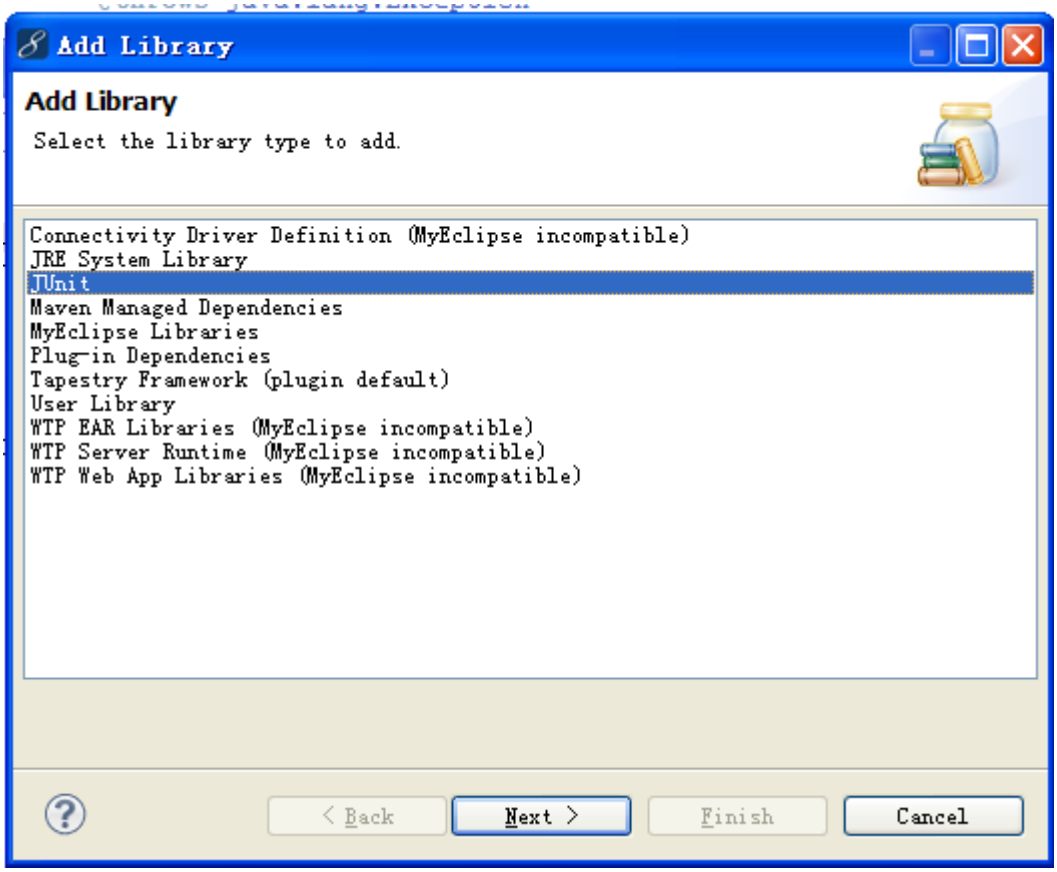
陈九思（**Sirius**）

目　录

一、配置Myeclipse在项目中引入JUnit4.jar包

Myexlips 自带了 Junit 4 和 Junit 3 的 jar 包，只需引入工程即可。
Properties -> add Library 选择 JUNIT。





二、Eclipse中JUnit的用法

先以一个简单的例子说明 Eclipse 中 JUnit 的用法：

一个简单的Demo：

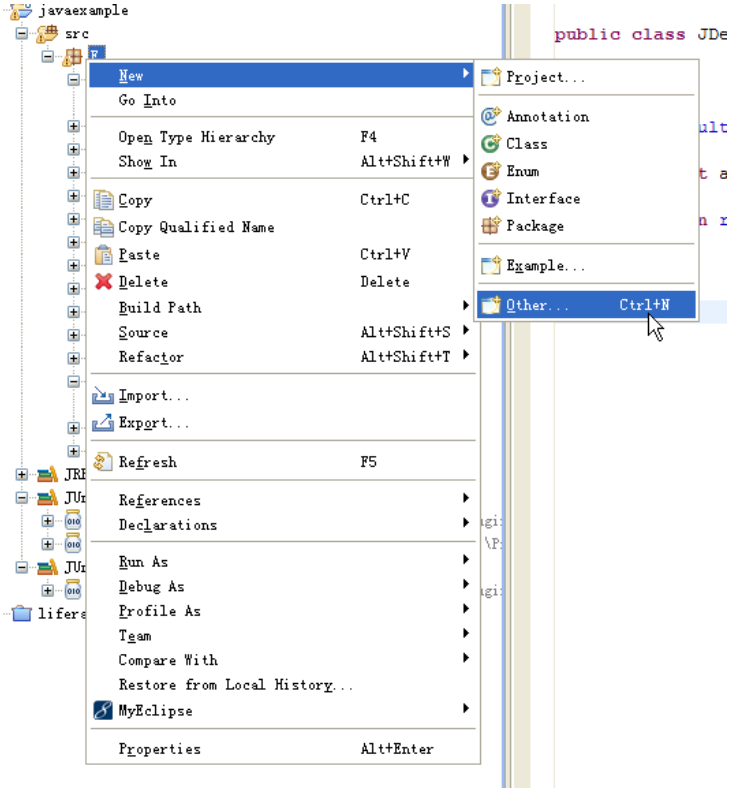
1、创建 JDemo 类

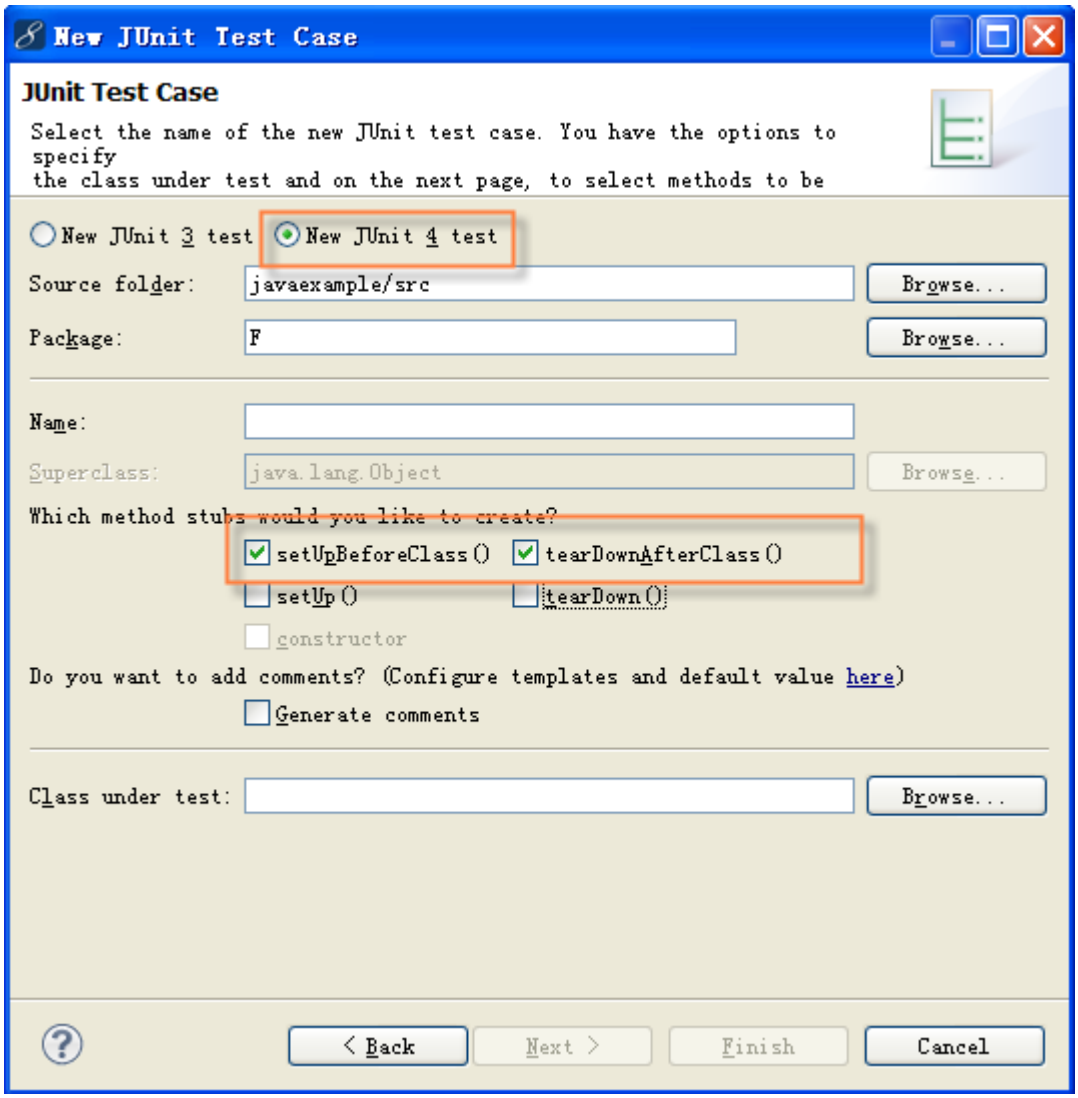
创建 JDemo 类，并创建一个简单的加法方法 ADD

```
public class JDemo {

    int  a;
    int  b;
    int  result;

   public int add(int a,int b){

       return result=a+b;
   }

}
```

2、建立测试用例

右键选择 new->other 选择 TESTCASE



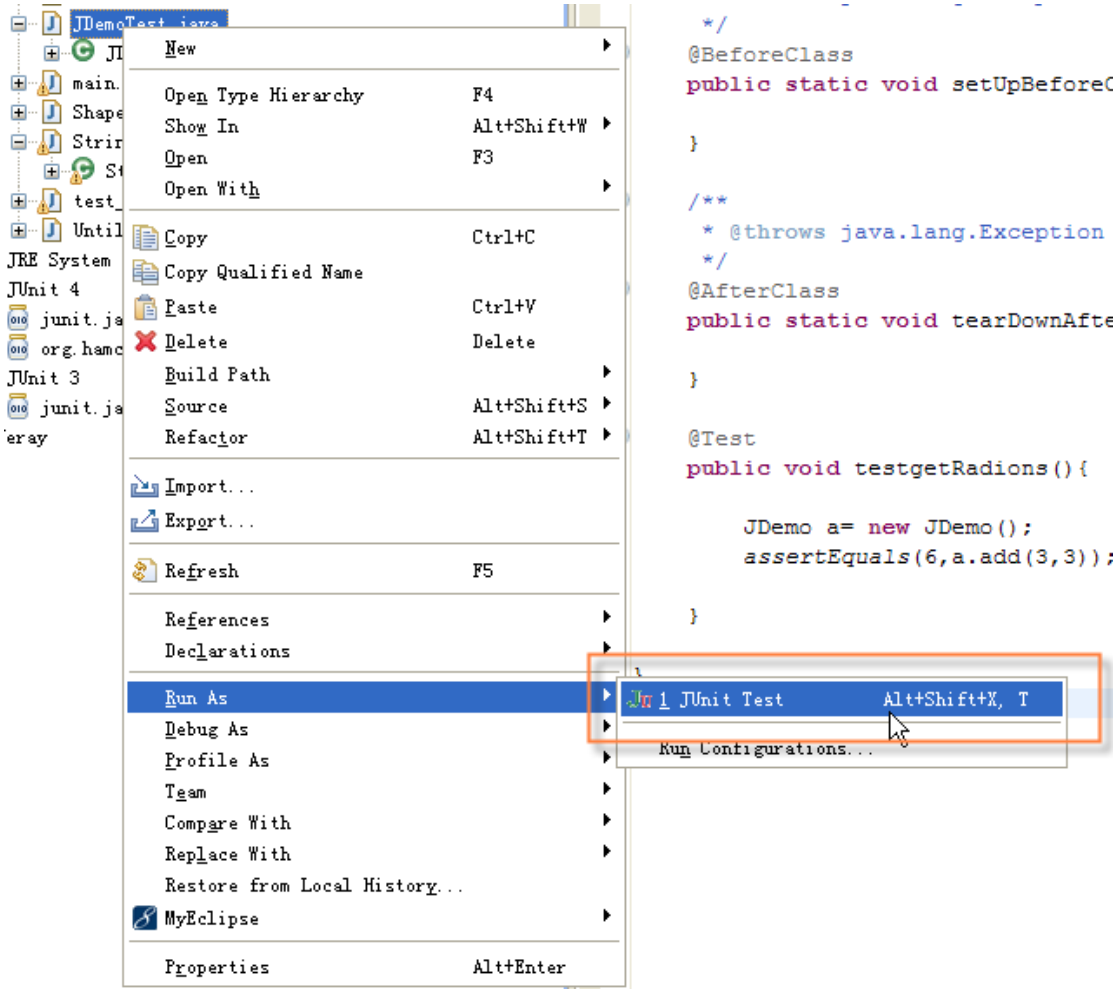


setUp()方法在测试方法前调用，主要用来做测试准备工作。
tearDown()方法在测试方法后调用，主要用来做测试的清理工作。
setUpBeforeClass()方法在整个类初始化之后调用，主要用来做测试准备工作。
tearDownAfterClass()方法在整个类结束之前调用，主要用来做测试清理工作。
constructor()为是否包含构造方法。
这个例子中我们选择 setUpBeforeClass（），tearDownAfterClass（）方法即可

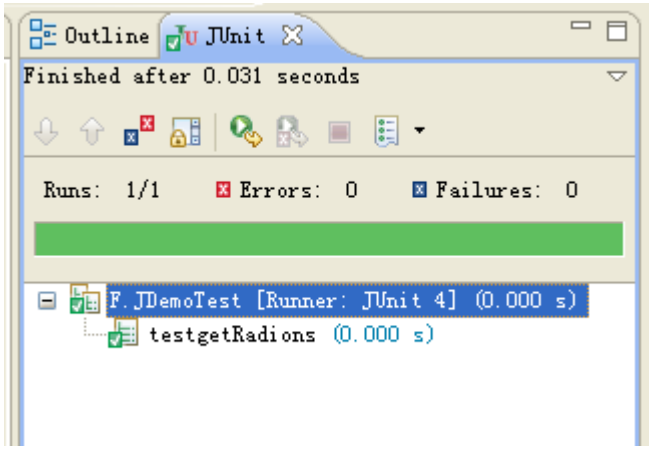
3、添加测试方法

```
import static org.junit.Assert.assertEquals;

/**
 * @author Administrator
 *
 */
public class JDemoTest {

    /**
     * @throws java.lang.Exception
     */
    @BeforeClass
    public static void setUpBeforeClass() throws Exception {

    }

    /**
     * @throws java.lang.Exception
     */
    @AfterClass
    public static void tearDownAfterClass() throws Exception {

    }

    @Test
    public void testgetRadions(){

        JDemo a= new JDemo();
        assertEquals(6,a.add(3,3));

    }
}
```

4、执行测试



5、查看测试结果



以上一个简单的测试用例执行完毕，

注：@Test 标明测试方法

assertEquals 断言判断是否得到预期的结果

三、Junit4 入门

1、annotation 介绍

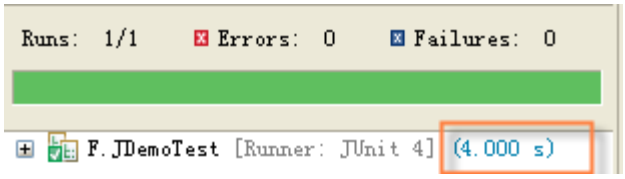
@Test（timeout，expected）
该注释修饰类为具体测试类，执行测试时 Junit 会自动加载它。
timeout: 规定该方法的执行时间，超时将抛异常 （最短时间 1ms）
例修改 之前的 JDemo ，继承 Thread 类，这样在运行加法运算时进程将休眠 2 秒钟，整个方法的执行时间将是 4 秒钟。

```
public class JDemo extends Thread{

    private int a;
    private int b;
    int result;

    /*public JDemo(String str) {
        super(str);
    }*/

    public int add(int a,int b){
            try
            {
                System.out.println("Hello,Sirius. Add is Begining," +
                                   " but please wait!");
                sleep(2000);
                result=a+b;
                System.out.println("Result is "+result);
                sleep(2000);
                System.out.println("End!");
            }
            catch (InterruptedException e) {
            }
            return result;
    }
    public void setB(int b) {
        this.b = b;
    }
    public int getB() {
        return b;
    }
    public void setA(int a) {
        this.a = a;
    }
    public int getA() {
        return a;
    }
}
```

修改 JDemoTest 测试 ADD 方法超时时间为 4s

```
@Test (timeout=4000)
public void testgetRadions(){

    JDemo a= new JDemo();
    assertEquals(6,a.add(3,3));

}
```

测试结果

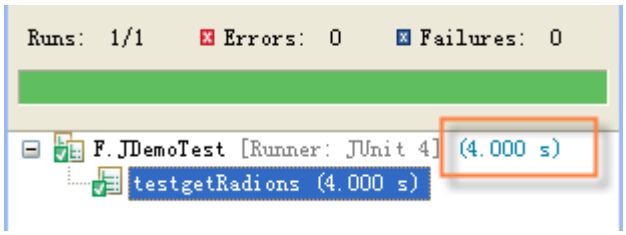


现在将测试超时时间设置为 3999ms

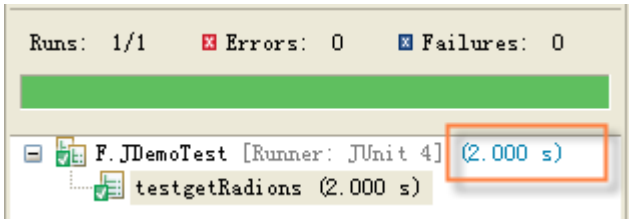
```
@Test (timeout=3999)
public void testgetRadions(){

    JDemo a= new JDemo();
    assertEquals(6,a.add(3,3));

}
```

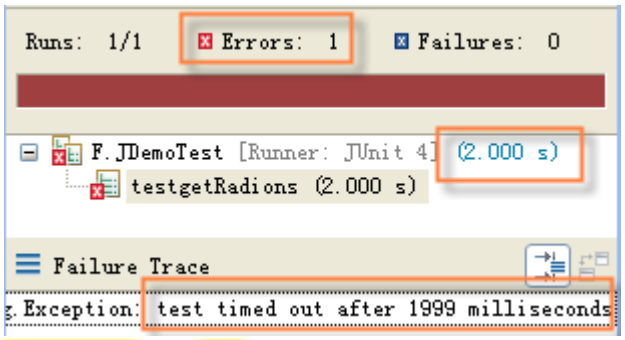
执行测试，这时会发现测试结果依然是通过。且结果显示验证方法执行时间确实是 4s

产生这种现象的原因其实是因为@Test 的 timeout 计算的是我们验证的得到断言结果的时间。也就是在这个例子中 add 方法得到 result 结果的时间。



根据上面描述分别修改 timeout 为 2000 和 1999 得到结果，注意在 1999ms 提示已经 Test time out。





Expected：修饰 Test 方法后，抛出相同异常才算通过测试

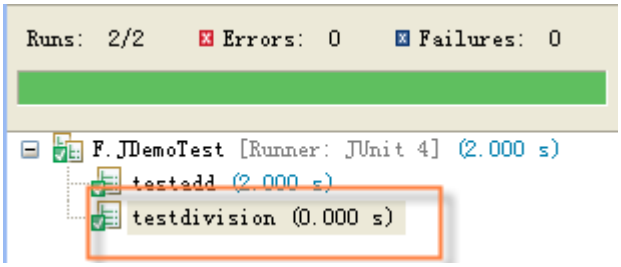
JDemo 添加除法方法

```
public int division(int a,int b){

    return result=a/b;
}
```

添加除法异常测试 注意：expected 为抛出异常则测试通过

```
@Test (expected=ArithmeticException.class)
public void testdivision(){

    JDemo a= new JDemo();
    a.division(2, 0);

}
```

查看测试结果



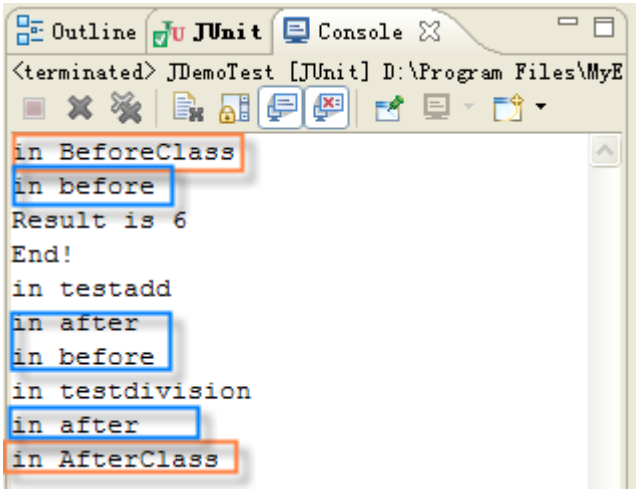
2、@Before、@After @BeforeClass @AfterClass 的区别。

@BeforeClass@AfterClass 标签注释的方法用于在整个类测试过程的初始化仅调用一次，@Before、@After 标签组合在每个测试方法前后都调用。

例：修改 JDemotest.java 程序 插入控制台输出（System.out.println();）表示调用规则。

```
JDemo a= new JDemo();
@BeforeClass
public static void setUpBeforeClass() throws Exception {
    //JDemo a= new JDemo();
    System.out.println("in BeforeClass");
}
/**
 * @throws java.lang.Exception
 */
@AfterClass
public static void tearDownAfterClass() throws Exception {
    System.out.println("in AfterClass");
}
@Before
public  void before(){
    System.out.println("in before");
}
@After
public  void after(){
    System.out.println("in after");
}
@Test (timeout=4000)
public void testadd(){
     //JDemo a= new JDemo();
    assertEquals(6,a.add(3,3));
    System.out.println("in testadd");
}
@Test (expected=ArithmeticException.class)
public void testdivision(){
    //JDemo a= new JDemo();
     System.out.println("in testdivision");
    a.division(2, 0);
}
```

执行测试用例并查看结果：
@Befort@After 被调用两次。

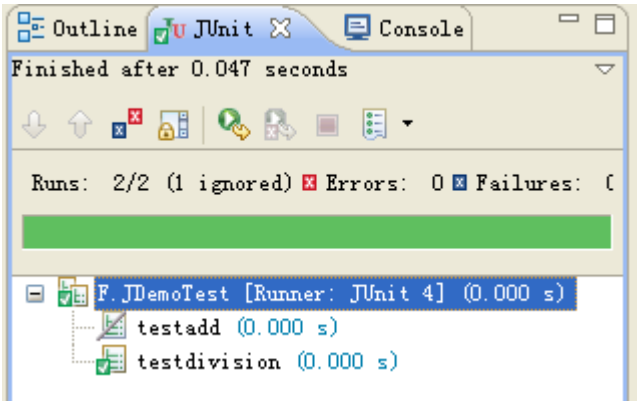


3、@Ignore 忽略某@Test 的测试

修改 JDemo.java

```
@Ignore("忽略add测试")
@Test (timeout=4000)
public void testadd(){
      //JDemo a= new JDemo();
     assertEquals(6,a.add(3,3));
     System.out.println("in testadd");
}
```

测试结果：



4、@Parameters 参数化测试数据，用于编辑测试用例中的参数

```
@Parameters
//测试数据准备
public static Collection multipleValues() {
 return Arrays.asList(new Object[][] {
  {6,3,3},
  {7,3,4},
 });
```

5、@RunWith（）：是 JUnit 为单元测试提供了默认的测试运行器，在其中可以选择使用具体的运行器 如：@RunWith(Parameterized.class) ，@RunWith(Suite.class)。

2、参数化测试用例JDemotest

以下以测试加法为例，参数化加法测试方法。修改 JDemotest.java
步骤：
添加新测试用例构造方法

```
//添加新构造方法，参数化。
int result;
int adddate_a;
int adddate_b;

public JDemoTest(int result,  int adddate_a,int adddate_b) {
    this.adddate_a =  adddate_a;
    this.adddate_b =  adddate_b;
    this.result = result;
}
```

设置测试数据，使用@Parameters 元注释，设置两组测试数据 数据一： a=3 b=3,期望结果 6 数据二：a=3 b=4, 期望测试结果 7

```
@Parameters
//测试数据准备
public static Collection multipleValues() {
 return Arrays.asList(new Object[][] {
  {6,3,3},
  {7,3,4},
 });
```

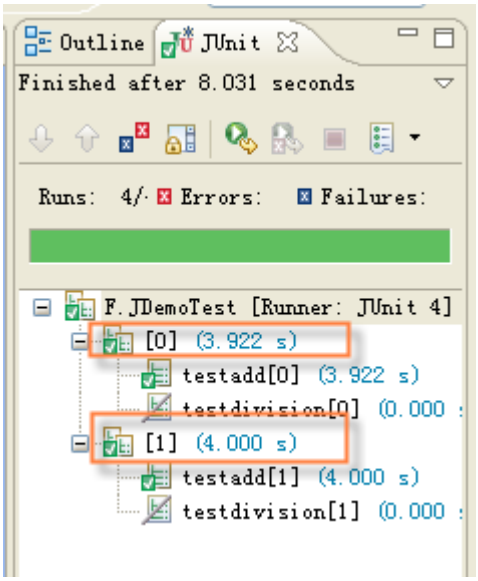
参数化测试步骤，改写元加法测试方法

```
@Test (timeout=4000)
public void testadd(){
     //JDemo a= new JDemo();
    assertEquals(result,a.add(adddate_a,adddate_b));
    System.out.println("in testadd");
}
```

添加@RunWith(Parameterized.class) 声明

```
@RunWith(Parameterized.class)
```

执行测试，查看结果
由测试结果可以看到，testadd 分别被调用了 2 次并测试通过。



3、测试套件testsuite设置

Testsuite 的目的是为了将多个 test 文件（如 JDemo.java）统一调度执行。

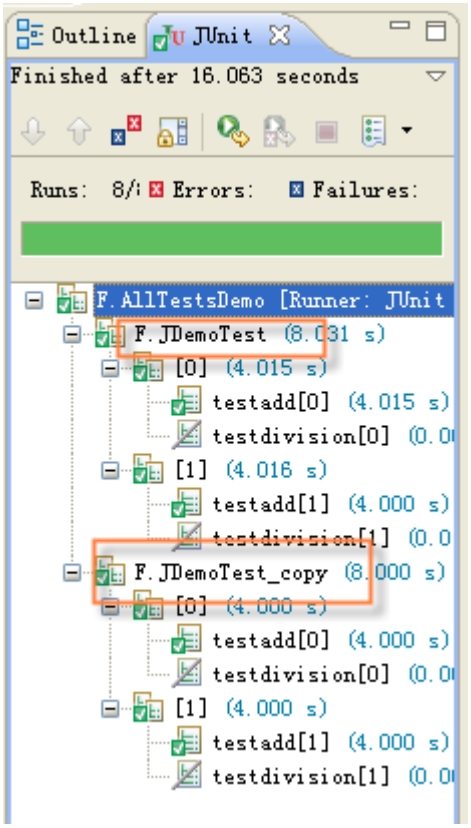
例：

1、复制 JDemo.java 文件命名为 JDemoTest_Copy.java

新建 AllTestDemo.java 文件 创建空的 AllTestDemo 插入

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({JDemoTest.class,
               JDemoTest_copy.class})
public class AllTestsDemo {

}
```

执行查看结果，展示出两个测试文件执行情况



四、总结：

JUnit4 较之前的 JUnit3 有了很大的改变，很大程度上源于引入了许多元注释的方法，使得具体的测试逻辑及步骤更为清晰，并且进一步简化了测试案例编写的难度。学习了几天的 JUnit4 总体有如下感受

1、编程能力必不可少。

JUnit 是一个基于 Java 的单元测试框架，应用具体测试项目时了解 JAVA 编程语言是必不可少的，否则在编写 TestCase，到判断错误原因均会带来很大困难。并且如果组织测试用例及测试数据的过程中也时刻贯彻着面向对象方法和思想的应用。

总之练好编程的基本功才是向更高级测试迈进的根本，也是常说开发与测试最终殊途同归的原因之一吧。

2、测试用例思想的转换

对于习惯于编写 EXCALE 版测试用例的我们，测试用例与测试数据的设计不是问题，但是如何将其进行整合与转化成，简洁高效的单元测试代码也是一个值得学习和思考的问题。

3、这仅仅只是个开始

无论是自动化测试还是单元测试，学习某一个工具的使用，仅仅是个开始，如果能有效的应用于对应的项目，产生真正的效益才是最终的目的。

五、待继续学习的问题：

关于单元测试自动化测试的一些构想中如何完成以下两点内容也是测试成败的关键。
1、与 ANT 或其它部署工具集成，应用于自动化测试
2、如何自动执行测试用例并生成测试报告