实验三　单元测试及JUnit的应用

一、实验目的

1、掌握单元测试的基本理论和作用。

2、掌握典型单元测试工具 JUnit 的使用。

二、实验类型

验证加设计。

三、实验内容

请按照以下关于单元测试的讲解说明，完成两个测试任务（即“实验任务 1”和“实验任务 2”），并在有时间的情况下，阅读后面关于单元测试和 JUnit 高级特性的内容，进一步了解单元测试的理论和 JUnit 测试的应用。

测试对于保证软件开发质量有着非常重要的作用，单元测试更是必不可。JUnit 是一个非常强大的单元测试包，可以对一个或多个类的单个或多个方法进行测试，还可将不同的 TestCase 组合成 TestSuit，使测试任务自动化。Eclipse 集成了 JUnit，可以非常方便地编写 TestCase。

我们在编写大型程序的时候，需要编写成千上万个方法或函数，这些函数的功能可能很强大，但我们在程序中只用到该函数的一小部分功能，并且经过调试可以确定，这一小部分功能是正确的。但是，我们同时应该确保每一个函数都完全正确，因为如果我们今后如果对程序进行扩展，用到了某个函数的其他功能，而这个功能有 bug 的话，将会造成很多问题。所以说，每编写完一个函数之后，都应该对这个函数的方方面面进行测试，这样的测试我们称之为单元测试。 传统的编程方式，进行单元测试是一件很麻烦的事情，你要重新写另外一个程序，在该程序中调用你需要测试的方法，并且仔细观察运行结果，看看是否有错。正因为如此麻烦，所以程序员们编写单元测试的热情不是很高。而类似 JUnit 这样的单元测试包的推出，大大简化了进行单元测试所要做的工作。

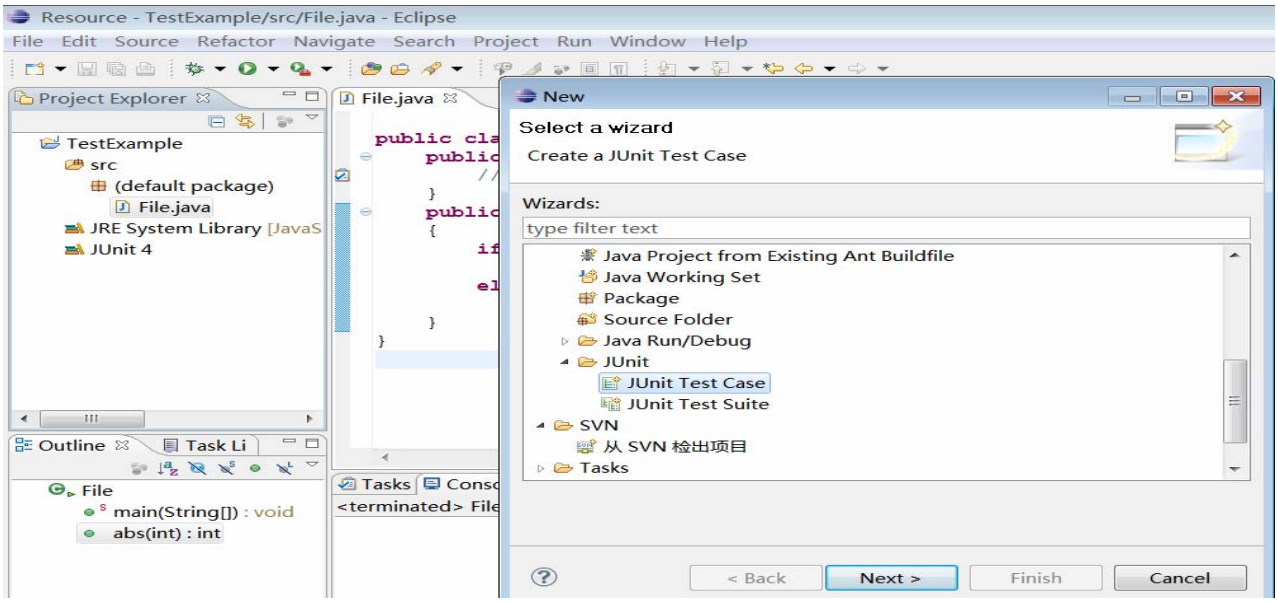
实验任务 1：

请按以下说明完成一个基本的 JUnit4 测试实例。

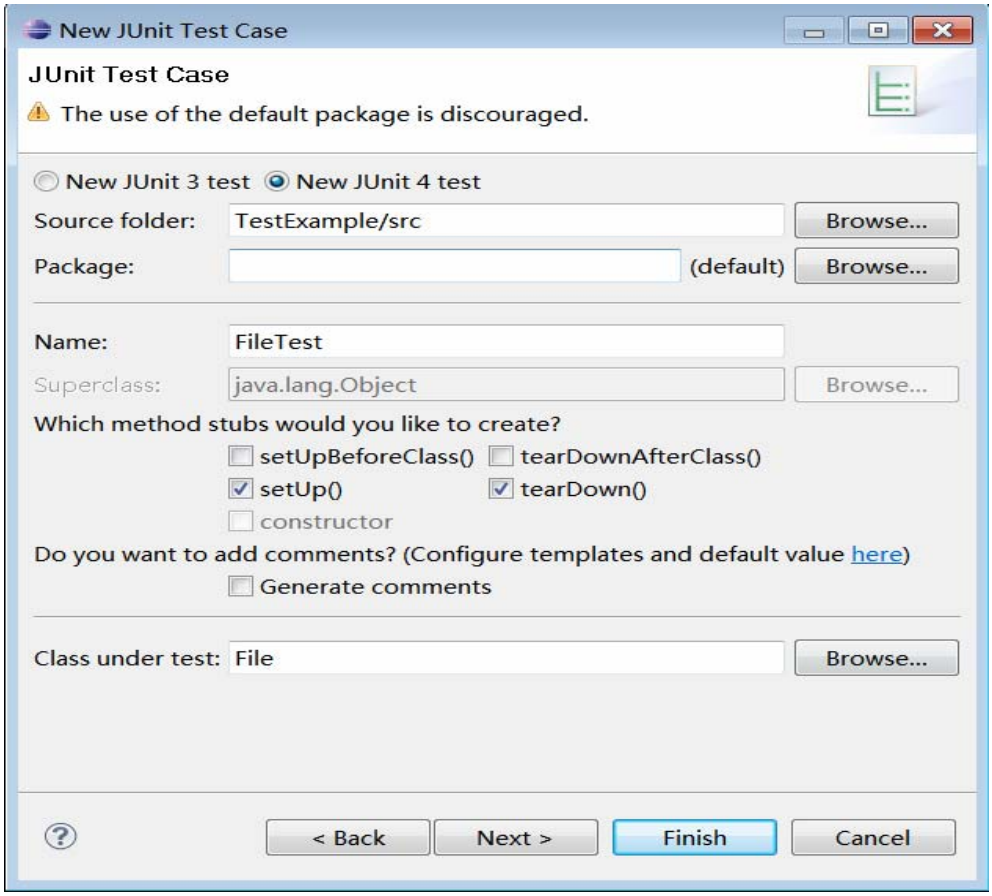
首先打开 Eclipse 编译器，创建一个 Java 项目，并在其中创建一个由 File 类构成的程序 File.java，给该类添加一个 abs 方法，作用是返回一个给定整数参数的绝对值。其参考代码如下：

```
public class File {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
    public int abs(int n)
    {
        if(n>=0)
            return n;
        else
            return (-n);
    }
}
```

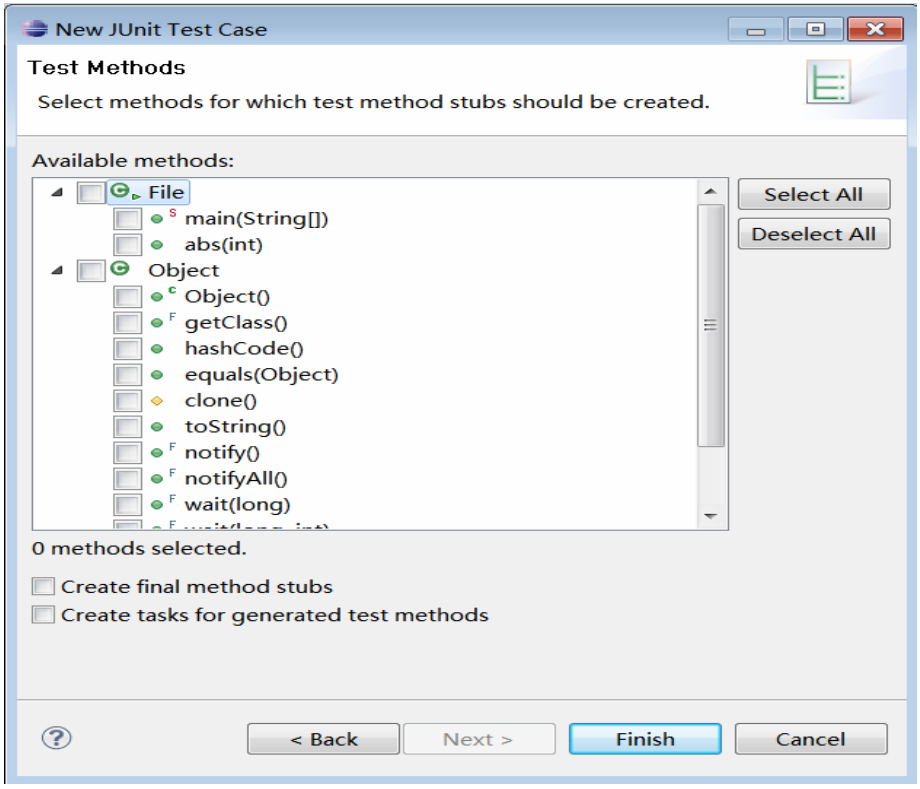
2、下面我们准备对这个方法 abs 进行测试，确保其功能正常。右键点击 File.java，选择 new—>other… ，弹出如下对话框：



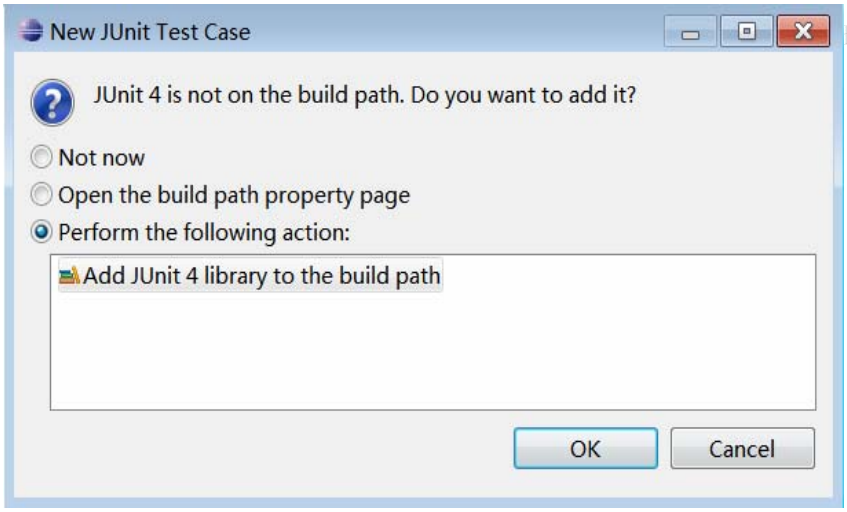
选择 JUnit Test Case。然后点击 next 到下一步，弹出如下窗口：



默认本页内容（或者选上面的 New JUnit 3 Test，此时是使用 JUnit3 进行测试，生成的测试类会继承自系统包里面的 junit.framework.TestCase 类，自己可以试试）。其中的 setUp 方法和 tearDown 方法一般要选上（默认会勾选）。然后点 next，弹出如下窗口：



这个窗口主要是让我们选择要对哪些方法进行测试，选中以后就会在测试类中生成其对应的测试方法。这里我们主要是要测试 abs 方法，所以在其前面勾选即可。点击 Finish，弹出最后的窗口：



是询问是否现在添加 JUnit 的库到编译路径下，选择添加（默认），点击 ok 即可。

结果会发现项目类生成了 FileTest 文件，测试类的代码如下：

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class FileTest {
    @Before
    public void setUp() throws Exception {
    }
```

```
    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testAbs() {
        fail("Not yet implemented");
    }
}
```

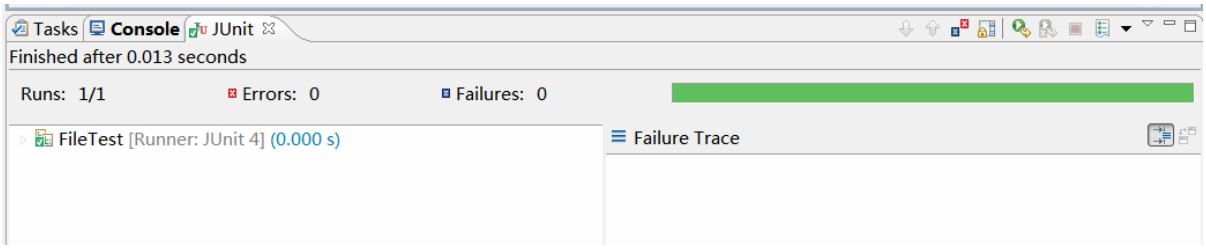
setUp()是建立测试环境，一般我们可以在 setUp 方法里面创建一个被测试类的实例；tearDown()用于清理资源，如释放打开的文件等等。以 test 开头的方法被认为是测试方法，JUnit 会依次执行 testXXX()方法。也就是说，JUnit 大致会以如下流程进行执行：

```
try {
    MyTestClass Test = new MyTestClass ();  //建立测试类实例
    Test.setUp();    //初始化测试环境
    Test.testAbs();  //测试某个方法
    Test.tearDown(); //清理资源
}
catch(){
}
```

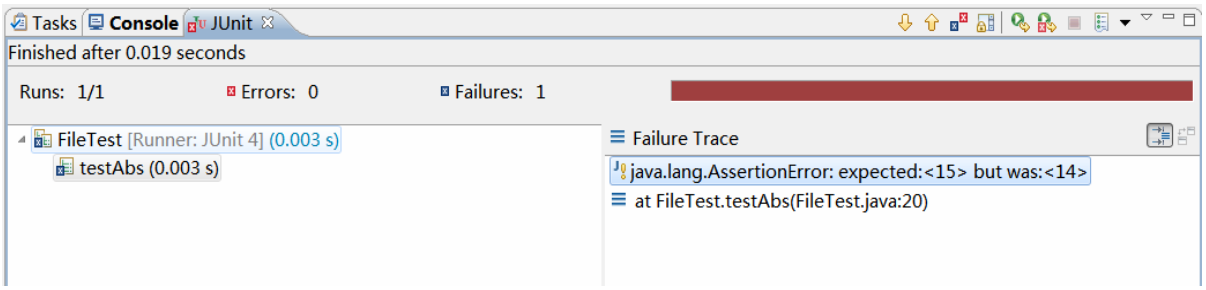
在 testAbs()方法中，我们对 abs()的测试分别选择正数，负数和 0，如果方法返回值与期待结果相同，则 assertEquals 不会产生异常。将其代码完善为如下的测试代码：


```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class FileTest {
    private File file;
    @Before
    public void setUp() throws Exception {
        file = new File();
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testAbs() {
        assertEquals(file.abs(14),14);
        assertEquals(file.abs(-5),5);
        assertEquals(file.abs(0),0);
    }
}
```

在测试类上右键单击，选择 Run As——>JUnit Test，进行运行。如果没问题，则出现如下所示的提示：



我们故意将 File 类中要测试的方法 abs 的 return n; 改为 return n+1; 再运行，则结果如下：



请比较和体会一下有问题和没问题的测试提示有何不同。

JUnit 通过单元测试，能在开发阶段就找出许多 Bug，并且，多个 Test Case 可以组合成 Test Suite，让整个测试自动完成，尤其适合于 XP 方法。每增加一个小的新功能或者对代码进行了小的修改，就立刻运行一遍 Test Suite，确保新增和修改的代码不会破坏原有的功能，大大增强软件的可维护性。

至此实验任务 1 结束。下面继续实验任务 2。

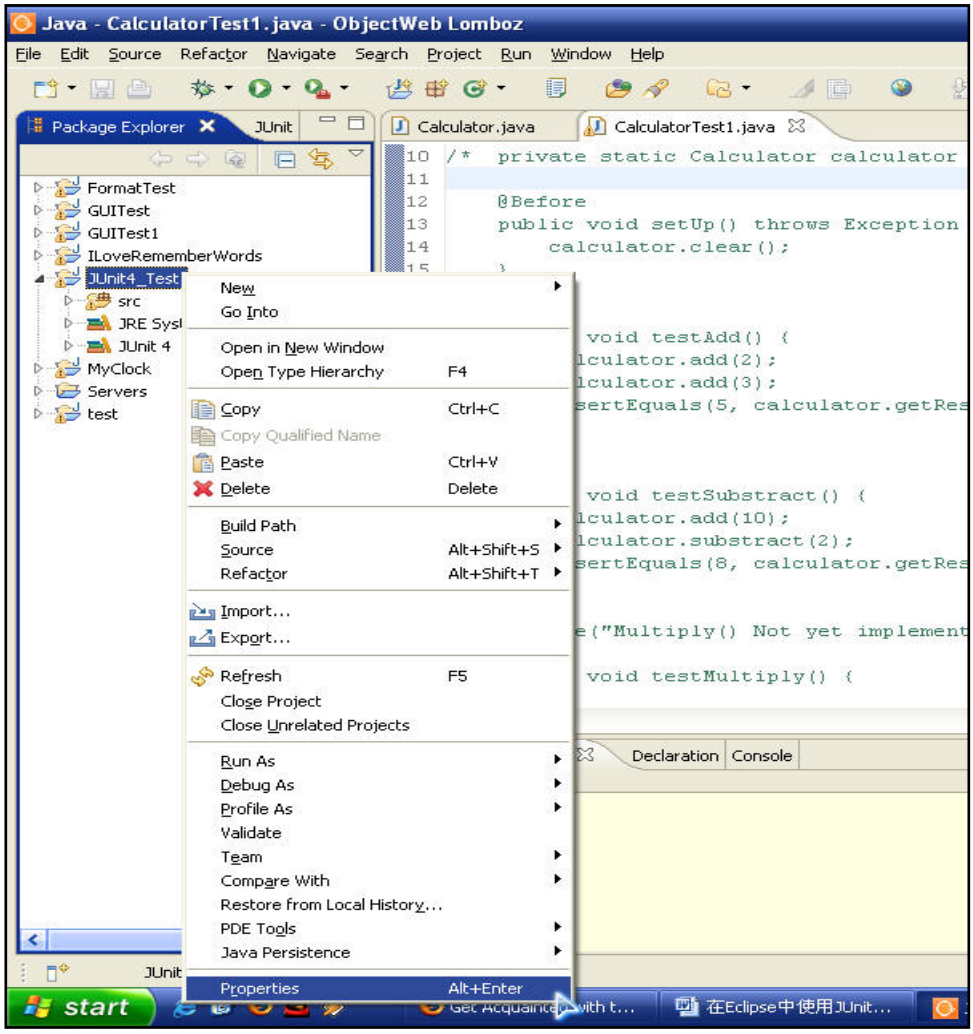
实验任务 2：

根据下面的说明，建立一个基于 JUnit4 的测试项目，对一个类当中的多个方法进行单元测试，进一步体验一下单元测试的作用和 JUnit 测试的应用。

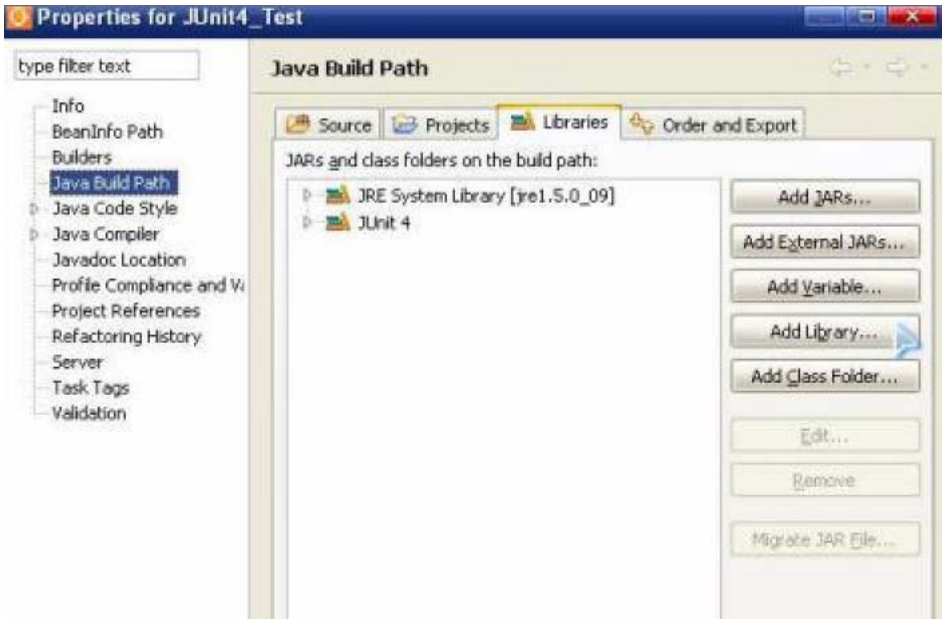
首先新建一个项目叫 JUnitTest，我们编写一个 Calculator 类，这是一个能够简单实现加减乘除、平方、开方的计算器类，然后对这些功能进行单元测试。这个类并不是很完美，我们故意保留了一些 Bug 用于演示，这些 Bug 在注释中都有说明。该类代码如下：

```
public class Calculator{
    private static int result;  // 静态变量，用于存储运行结果

    public void add(int n) {
        result = result + n;
    }

    public void substract(int n) {
        result = result - 1; //故意的Bug，应该是 result =result-n
    }

    public void multiply(int n) {

    }  // 假设此方法在项目完成过程中尚未写好

    public void divide(int n) {
        result = result / n;
    }

    public void square(int n) {
        result = n * n;
    }

    public void squareRoot(int n) { //求平方根
        for (; ;) ;   //Bug : 死循环
    }

    public void clear() {  // 将结果清零
        result = 0;
    }

    public int getResult(){
        return result;
    }
}
```

第二步，将 JUnit4 单元测试包引入这个项目：在该项目上点右键，点“属性”，如图：



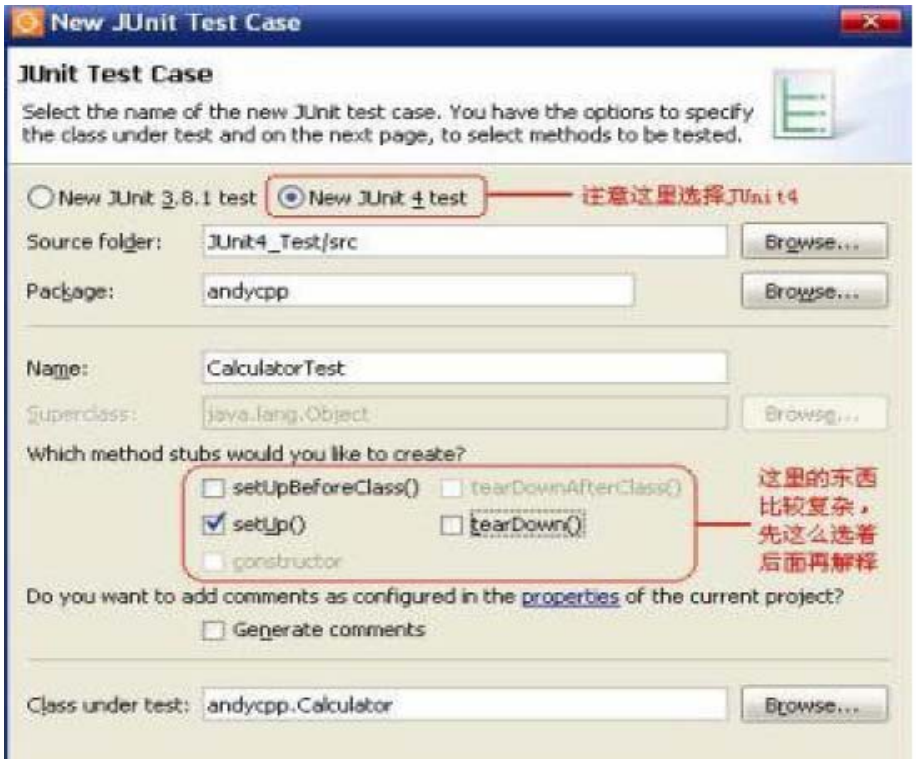
在弹出的属性窗口中，首先在左边选择“Java Build Path”，然后到右上选择“Libraries”标签，之后在最右边点击“Add Library…”按钮，如下图所示：



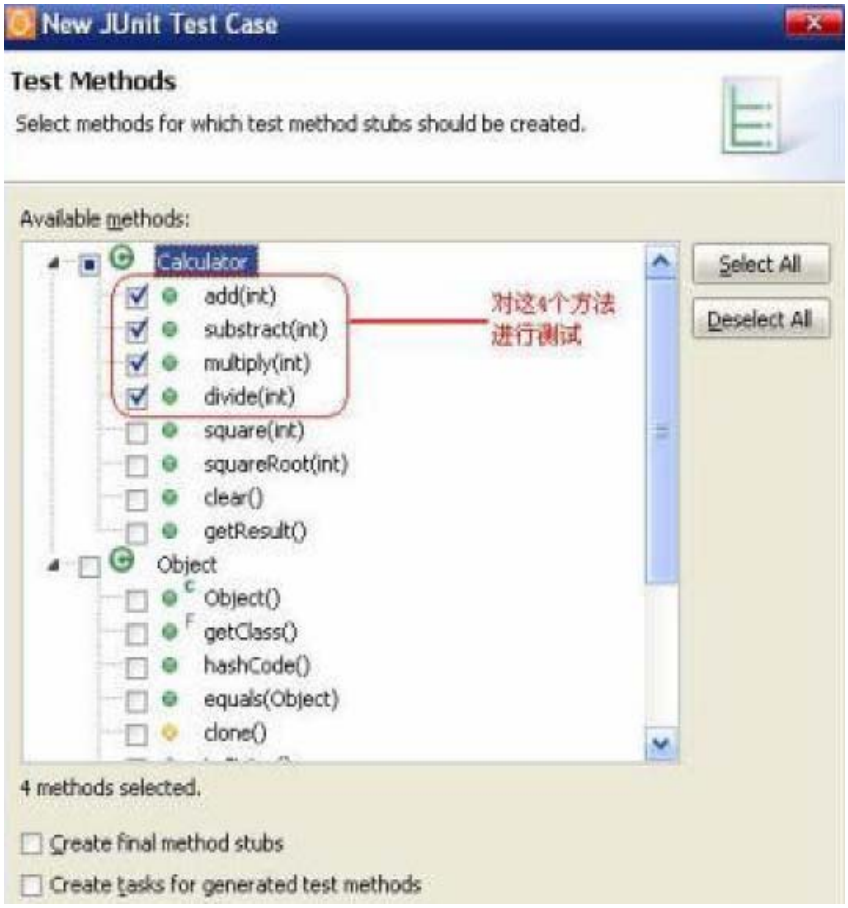
然后在新弹出的对话框中选择 JUnit4 并点击确定，如上图所示，JUnit4 软件包就被包含进我们这个项目了（实际上在后面的第三步生成测试用例的过程中 Eclipse 这样的编译器也会让我们选择添加 JUnit4 包的）。

第三步，生成 JUnit 测试框架：在 Eclipse 的 Package Explorer 中用右键点击该类弹出菜单，选择“JUnit 测试用例”。如下图所示：

在弹出的对话框中，进行相应的选择，如下图所示：



点击“下一步”后，系统会自动列出你这个类中包含的方法，选择你要进行测试的方法。此例中，我们仅对“加、减、乘、除”四个方法进行测试。如下图所示：

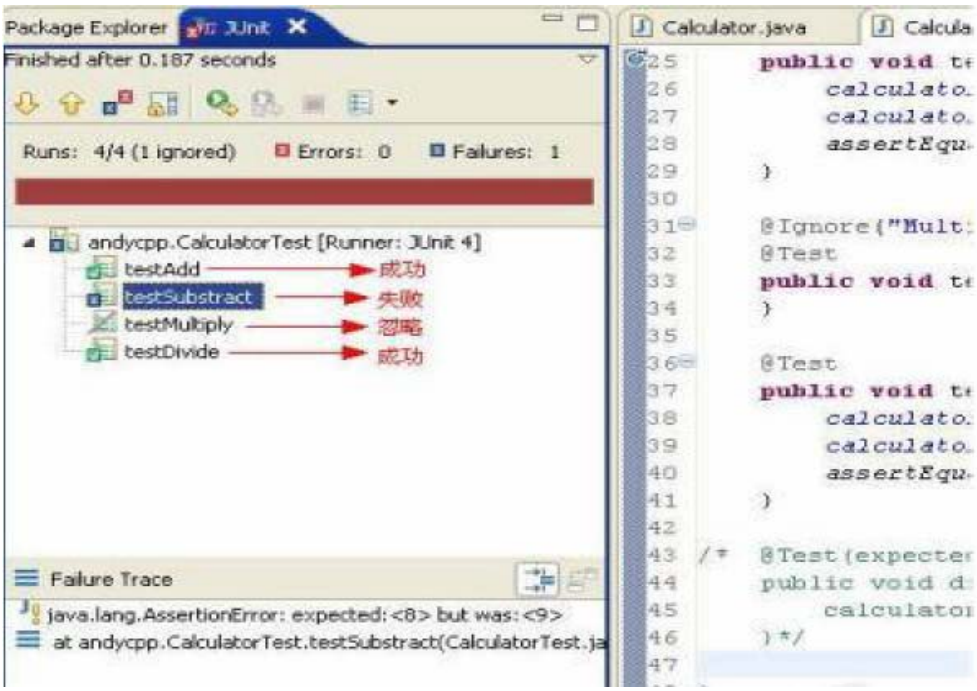


之后系统会自动生成一个新类 CalculatorTest，里面包含一些空的测试用例。你只需要将这些测试用例稍作修改即可使用。完整的 CalculatorTest 代码如下：

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
public class CalculatorTest{
    private static Calculator calculator = new Calculator();

    @Before
    public void setUp() throws Exception{
        calculator.clear();
    }

    @Test
    public void testAdd(){
        calculator.add(2);
        calculator.add(3);
        assertEquals(5, calculator.getResult());
    }

    @Test
    public void testSubstract(){
```

```
        calculator.add(10);
        calculator.substract(2);
        assertEquals(8, calculator.getResult());
    }

    @Ignore("Multiply() Not yet implemented")
    @Test
    public void testMultiply(){

    }

    @Test
    public void testDivide(){
        calculator.add(8);
        calculator.divide(2);
        assertEquals(4, calculator.getResult());
    }
}
```

第四步，运行测试代码：按照上述代码修改完毕后，我们在 CalculatorTest 类上点右键，选择“Run As——>JUnit Test”来运行我们的测试，运行结果如下：



进度条是红颜色表示发现错误，具体的测试结果在进度条上面有表示“共进行了 4 个测试，其中 1 个测试被忽略，一个测试失败”。

至此通过实验任务 1 和实验任务 2 所进行的两次单元测试均结束。

下面我们对 JUnit 进一步进行说明，并进一步了解 JUnit 的其它高级特性。

2、JUnit4 的高级特性

通过上述 2 个测试实例，我们已经比较完整地体验了在 Eclipse 中使用 JUnit 测试的方法。接下来进一步解释测试代码中的每一个细节，了解这个测试过程是如何一步步完成的。

一、导入必要的系统包

在测试类中用到了 JUnit4 框架，自然要把相应地 Package 包含进来。最主要地一个 Package 就是 org.junit.*。把它包含进来之后，绝大部分功能就有了。还有一句话也非常地重要“import static org.junit.Assert.*;”，我们在测试的时候使用的一系列 assertEquals 方法就来自这个包。大家注意一下，这是一个静态包含(static)，是 JDK5 中新增添的一个功能。也就是说，assertEquals 是 Assert 类中的一系列的静态方法，一般的使用方式是 Assert. assertEquals()，但是使用了静态包含后，前面的类名就可以省略了，使用起来更加的方便。

二、测试类的声明

大家注意到，我们的测试类是一个独立的类，没有任何父类。测试类的名字也可以任意命名，没有任何局限性。所以我们不能通过类的声明来判断它是不是一个测试类，它与普通类的区别在于它内部的方法的声明，我们接着会讲到。

三、创建一个待测试的对象

你要测试哪个类，那么你首先就要创建一个该类的对象。正如前面的代码：

private staticCalculator calculator =newCalculator();

为了测试 Calculator 类，我们必须创建一个 calculator 对象。

四、测试方法的声明

在测试类中，并不是每一个方法都是用于测试的，你必须使用 Annotation（一般译为“元数据”或“注解”）来明确表明哪些是测试方法。Annotation 也是 JDK5 的一个新特性，用在此处非常恰当。我们可以看到，在某些方法的前有@Before、@Test、@Ignore 等字样，这些就是标注，以一个“@”作为开头。这些标注都是 JUnit4 自定义的，熟练掌握这些标注的含义非常重要。

JUnit4 与 JUnit3.X 及之前的版本相比，正是增加了 Annotation（一般译为“元数据”或“注解”）等各项特性，使得其功能更为强大。JUnit4 是 JUnit 框架有史以来的最大改进，其主要目标便是利用 Java5 的 Annotation 特性简化测试用例的编写。

先简单解释一下什么是 Annotation，这个单词一般是翻译成元数据或注解。元数据是什么？元数据就是描述数据的数据。也就是说，这个概念在 Java 里面可以用来和 public、static 等关键字一样来修饰类名、方法名、变量名。修饰的作用是描述这个数据是做什么用的，类似于用 public 描述这个数据是公有类型一样。

1. 测试方法：

Junit 中还因 JDK5 而增加了一项新特性，静态导入（static import）。

我们先看一下在 JUnit 3 中我们是怎样写一个单元测试的。比如下面一个类：

```
public class AddOperation {
        public int add(int x,int y){
            return x+y;
        }
}
```

我们要测试 add 这个方法，我们写单元测试得这么写：

```
import junit.framework.TestCase;
import static org.junit.Assert.*;
public class AddOperationTest extends TestCase{

    public void setUp() throws Exception {
    }

    public void tearDown() throws Exception {
    }

    public void testAdd() {
        System.out.println(\"add\");
        int x = 0;
        int y = 0;
        AddOperation instance = new AddOperation();
        int expResult = 0;
        int result = instance.add(x, y);
        assertEquals(expResult, result);
    }
}
```


可以看到上面的类使用了 JDK5 中的静态导入，这个相对来说就很简单，只要在 import 关键字后面加上 static 关键字，就可以把后面的类的 static 类型的变量和方法导入到这个类中，调用的时候和调用自己的方法没有任何区别。

我们可以看到上面那个单元测试有一些比较霸道的地方，表现在：

1.单元测试类必须继承自 TestCase。

2.要测试的方法必须以 test 开头。

如果上面那个单元测试在 JUnit 4 中写就不会这么复杂。代码如下：

```
import junit.framework.TestCase;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class AddOperationTest {
    public AddOperationTest() {
    }

    @Before
    public void setUp() throws Exception {
    }
```

```
    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void add() {
        System.out.println(\"add\");
        int x = 0;
        int y = 0;
        AddOperation instance = new AddOperation();
        int expResult = 0;
        int result = instance.add(x, y);
        assertEquals(expResult, result);
    }

}
```

我们可以看到，采用 Annotation 的 JUnit 已经不会霸道的要求你必须继承自 TestCase 了，而且测试方法也不必以 test 开头了，只要以@Test 元数据来描述即可。

从前面的 2 个测试实例中，我们可以看到在 JUnit 4 中还引入了一些其它的 Annotation（注解），下面对它们和其它一些注解分别进行介绍：

@Before：
使用了该元数据的方法在每个测试方法执行之前都要执行一次(即有多个方法它就要运行多次)。

@After：
使用了该元数据的方法在每个测试方法执行之后要执行一次(即有多个方法它就要运行多次)。
注意：@Before 和@After 标示的方法只能各有一个。这个相当于取代了 JUnit 以前版本中的 setUp 和 tearDown 方法，当然你还可以继续叫这个名字，不过 JUnit 不会霸道的要求你这么做了。

@Test(expected=*.class)
“(expected=*.class)”不是必须的。
在 JUnit4 之前，测试类通过继承 TestCase 类，并使用命名约束来定位测试，测试方法必须以“test”开头。Junit4 中使用注释类识别：@Test，而且也不必约束测试方法的名字。当然，TestCase 类仍然可以工作，只不过不用这么繁琐而已。
在 JUnit4.0 之前，对错误的测试，我们只能通过 fail 来产生一个错误，并在 try 块里面 assertTrue（true）来测试。现在，可以通过@Test 元数据中的 expected 属性来解决问题。expected 属性的值是一个异常的类型。

@Test(timeout=xxx):
该元数据传入了一个时间（毫秒）给测试方法，防止一个方法无限期地执行下去。
如果测试方法在指定的时间之内没有运行完，则测试也以失败结束。

@ignore：
该元数据标记的测试方法在测试中会被忽略。当测试的方法还没有实现，或者测试的方法已经过时，或者在某种条件下才能测试该方法（比如需要一个数据库联接，而在本地测试的时候，数据库并没有连接），那么使用该标签来标示这个方法。同时，你可以为该标签传递一个 String 的参数，来表明为什么会忽略这

个测试方法。比如：@Ignore(“该方法还没有实现”)，在执行的时候，仅会报告该方法没有实现，而不会运行测试方法。

五、编写一个简单的测试方法

首先，你要在方法的前面使用@Test 标注，以表明这是一个测试方法。对于方法的声明也有如下要求：名字可以随便取，没有任何限制，但是返回值必须为 void，而且不能有任何参数。如果违反这些规定，会在运行时抛出一个异常。至于方法内该写些什么，那就要看你需要测试些什么了。比如：

```
@Test
public voidtestAdd(){
calculator.add(2);
calculator.add(3);
assertEquals(5, calculator.getResult());
}
```

我们想测试一下“加法”功能时候正确，就在测试方法中调用几次 add 函数，初始值为 0，先加 2，再加 3，我们期待的结果应该是 5。如果最终实际结果也是 5，则说明 add 方法是正确的，反之说明它是错的。assertEquals(5, calculator.getResult());就是来判断期待结果和实际结果是否相等，第一个参数填写期待结果，第二个参数填写实际结果，也就是通过计算得到的结果。这样写好之后，JUnit 会自动进行测试并把测试结果反馈给用户。

六、 忽略测试某些尚未完成的方法

如果你在写程序前做了很好的规划，那么哪些方法是什么功能都应该实现定下来。因此，即使该方法尚未完成，他的具体功能也是确定的，这也就意味着你可以为他编写测试用例。但是，如果你已经把该方法的测试用例写完，但该方法尚未完成，那么测试的时候一定是“失败”。这种失败和真正的失败是有区别的，因此 JUnit 提供了一种方法来区别他们，那就是在这种测试函数的前面加上@Ignore 标注，这个标注的含义就是“某些方法尚未完成，暂不参与此次测试”。这样的话测试结果就会提示你有几个测试被忽略，而不是失败。一旦你完成了相应函数，只需要把@Ignore 标注删去，就可以进行正常的测试。

七、 Fixture(暂且翻译为“固定代码段”)

Fixture 的含义就是“在某些阶段必然被调用的代码”。比如我们上面的测试，由于只声明了一个 Calculator 对象，他的初始值是 0，但是测试完加法操作后，他的值就不是 0 了;接下来测试减法操作，就必然要考虑上次加法操作的结果。这绝对是一个很糟糕的设计!我们非常希望每一个测试都是独立的，相互之间没有任何耦合度。因此，我们就很有必要在执行每一个测试之前，对 Calculator 对象进行一个“复原”操作，以消除其他测试造成的影响。因此，“在任何一个测试执行之前必须执行的代码”就是一个 Fixture，我们用@Before 来标注它，如前面例子所示：

```
@Before
public void setUp() throws Exception ...{
   calculator.clear();
}
```

这里不在需要@Test 标注，因为这不是一个 test，而是一个 Fixture。同理，如果“在任何测试执行之后需要进行的收尾工作”也是一个 Fixture，使用@After 来标注。由于本例比较简单，没有用到此功能。

下面再来探讨一下 JUnit4 中其它一些高级特性。

一、 高级 Fixture

前面部分我们介绍了两个 Fixture 标注，分别是@Before 和@After，我们来看看他们是否适合完成如下功能：有一个类是负责对大文件(超过 500 兆)进行读写，他的每一个方法都是对文件进行操作。换句话说，在调用每一个方法之前，我们都要打开一个大文件并读入文件内容，这绝对是一个非常耗费时间的操作。如果我们使用@Before 和@After，那么每次测试都要读取一次文件，效率及其低下。这里我们所希望的是在所有测试一开始读一次文件，所有测试结束之后释放文件，而不是每次测试都读文件。JUnit 的作者显然也考虑到了这个问题，它给出了@BeforeClass 和 @AfterClass 两个 Fixture 来帮我们实现这个功能。从名字上就可以看出，用这两个 Fixture 标注的函数，只在测试用例初始化时执行 @BeforeClass 方法，当所有测试执行完毕之后，执行@AfterClass 进行收尾工作。在这里要注意一下，每个测试类只能有一个方法被标注为 @BeforeClass 或 @AfterClass，并且该方法必须是 Public 和 Static 的。

二、 限时测试

最前面给出的 Calculator 类例子中，那个求平方根的函数有 Bug，是个死循环：

```
public voidsquareRoot(intn) ...{
    for(; ;) ; //Bug : 死循环
}
```

如果测试的时候遇到死循环，可以想象会出现什么结果。因此，对于那些逻辑很复杂，循环嵌套比较深的程序，很有可能出现死循环，因此一定要采取一些预防措施。限时测试是一个很好的解决方案。我们给这些测试函数设定一个执行时间，超过了这个时间，他们就会被系统强行终止，并且系统还会向你报告该函数结束的原因是因为超时，这样你就可以发现这些 Bug 了。要实现这一功能，只需要给@Test 标注加一个参数即可，代码如下：


```
@Test(timeout = 1000)
public voidsquareRoot(){

calculator.squareRoot(4);
assertEquals(2, calculator.getResult());
}
```

Timeout 参数表明了你要设定的时间，单位为毫秒，因此 1000 就代表 1 秒。

三、 测试异常

JAVA 中的异常处理也是一个重点，因此你经常会编写一些需要抛出异常的函数。那么，如果你觉得一个函数应该抛出异常，但是它没抛出，这算不算 Bug 呢?这当然是 Bug，而 JUnit 也考虑到了这一点，来帮助我们找到这种 Bug。例如，我们写的计算器类有除法功能，如果除数是一个 0，那么必然要抛出“除 0 异常”。因此，我们很有必要对这些进行测试。代码如下：

```
@Test(expected = ArithmeticException.class)
public void divideByZero() ...{
calculator.divide(0);
}
```

如上述代码所示，我们需要使用@Test 标注的 expected 属性，将我们要检验的异常传递给它，这样 JUnit 框架就能自动帮我们检测是否抛出了我们指定的异常。

四、 Runner (运行器)

大家有没有想过这个问题，当你把测试代码提交给 JUnit 框架后，框架如何来运行你的代码呢?答案就是——Runner。在 JUnit 中有很多个 Runner，他们负责调用你的测试代码，每一个 Runner 都有各自的特殊功能，你要根据需要选择不同的 Runner 来运行你的测试代码。可能你会觉得奇怪，前面我们写了那么多测试，并没有明确指定一个 Runner 啊?这是因为 JUnit 中有一个默认 Runner，如果你没有指定，那么系统自动使用默认 Runner 来运行你的代码。换句话说，下面两段代码含义是完全一样的：

```
import org.junit.internal.runners.TestClassRunner;

import org.junit.runner.RunWith;
//使用了系统默认的 TestClassRunner，与下面代码完全一样

public class CalculatorTest{
...
}

@RunWith(TestClassRunner.class)
public class CalculatorTest{
...
}
```

从上述例子可以看出，要想指定一个 Runner，需要使用@RunWith 标注，并且把你所指定的 Runner 作为参数传递给它。另外一个要注意的是，@RunWith 是用来修饰类的，而不是用来修饰函数的。只要对一个类指定了 Runner，那么这个类中的所有函数都被这个 Runner 来调用。最后，不要忘了包含相应的 Package 哦，上面的例子对这一点写的很清楚了。接下来，我们展示其他 Runner 的特有功能。

五、 参数化测试

你可能遇到过这样的函数，它的参数有许多特殊值，或者说他的参数分为很多个区域。比如，一个对考试分数进行评价的函数，返回值分别为“优秀，良好，一般，及格，不及格”，因此你在编写测试的时候，至少要写 5 个测试，把这 5 中情况都包含了，这确实是一件很麻烦的事情。我们还使用我们先前的例子，测试一下“计算一个数的平方”这个函数，暂且分三类：正数、0、负数。测试代码如下：

```
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class AdvancedTest{
    private static Calculator calculator = new Calculator();

    @Before
    public void clearCalculator(){
        calculator.clear();
    }

    @Test
```

```
    public void square1(){
        calculator.square(2);
        assertEquals(4, calculator.getResult());
    }

    @Test
    public void square2(){
        calculator.square(0);
        assertEquals(0, calculator.getResult());
    }

    @Test
    public void square3(){
        calculator.square(-3);
        assertEquals(9, calculator.getResult());
    }
}
```

为了简化类似的测试，JUnit4 提出了“参数化测试”的概念，只写一个测试函数，把这若干种情况作为参数传递进去，一次性的完成测试。代码如下：

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import java.util.Arrays;
import java.util.Collection;

@RunWith(Parameterized.class)
public class SquareTest{
    private static Calculator calculator = new Calculator();
    private int param;
    private int result;

    @Parameters
    public static Collection data() {
        return Arrays.asList(new Object[][]{{2, 4},{0, 0},{-3, 9},});
    }

    //构造函数，对变量进行初始化
    public SquareTest(int param, int result){
        this.param = param;
        this.result = result;
    }

    @Test
```

```
    public void square(){
        calculator.square(param);
        assertEquals(result, calculator.getResult());
    }
}
```

下面我们对上述代码进行分析。首先，你要为这种测试专门生成一个新的类，而不能与其他测试共用同一个类，此例中我们定义了一个 SquareTest 类。然后，你要为这个类指定一个 Runner，而不能使用默认的 Runner 了，因为特殊的功能要用特殊的 Runner 嘛。@RunWith(Parameterized.class)这条语句就是为这个类指定了一个 ParameterizedRunner。第二步，定义一个待测试的类，并且定义两个变量，一个用于存放参数，一个用于存放期待的结果。接下来，定义测试数据的集合，也就是上述的 data()方法，该方法可以任意命名，但是必须使用@Parameters 标注进行修饰。这个方法的框架就不予解释了，大家只需要注意其中的数据，是一个二维数组，数据两两一组，每组中的这两个数据，一个是参数，一个是你预期的结果。比如我们的第一组{2, 4}，2 就是参数，4 就是预期的结果。这两个数据的顺序无所谓，谁前谁后都可以。之后是构造函数，其功能就是对先前定义的两个参数进行初始化。在这里你可要注意一下参数的顺序了，要和上面的数据集合的顺序保持一致。如果前面的顺序是{参数，期待的结果}，那么你构造函数的顺序也要是“构造函数(参数， 期待的结果)”，反之亦然。最后就是写一个简单的测试例了，和前面介绍过的写法完全一样，在此就不多说。

六、打包测试。

通过前面的介绍我们可以感觉到，在一个项目中，只写一个测试类是不可能的，我们会写出很多很多个测试类。可是这些测试类必须一个一个的执行，也是比较麻烦的事情。鉴于此，JUnit 为我们提供了打包测试的功能，将所有需要运行的测试类集中起来，一次性的运行完毕，大大的方便了我们的测试工作。具体代码如下：


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

@RunWith(Suite.class)
@Suite.SuiteClasses({
        CalculatorTest.class,
        SquareTest.class
        })

public class AllCalculatorTests {
}
```

大家可以看到，这个功能也需要使用一个特殊的 Runner，因此我们需要向@RunWith 标注传递一个参数 Suite.class。同时，我们还需要另外一个标注@Suite.SuiteClasses，来表明这个类是一个打包测试类。我们把需要打包的类作为参数传递给该标注就可以了。有了这两个标注之后，就已经完整的表达了所有的含义，因此下面的类已经无关紧要，随便起一个类名，内容全部为空既可。

3、JUnit4 新特性的进一步说明

因 jdk5 中的新特性，JUnit4 也因此有了很大的改变。确切的说，Junit4 简直就不是 3 的扩展版本，而几乎是一个全新的测试框架。下面详细介绍 JUnit4 的使用方法

1、 固件测试

所谓固件测试(Fixture)，就是测试运行运行程序（test runner）会在测试方法之前自动初始化、和回收资源的工作。JUnit4 之前是通过 setUp、TearDown 方法完成。在 Junit4 中，仍然可以在每个测试方法运行之前初始化字段，和配置环境，当然也是通过注释完成。Junit4 中，通过@befroe 替代 setUp 方法；@After 替代 tearDown 方法。在一个测试类中，甚至可以使用多个@Before 来注释多个方法，这些方法都是在每个测试之前运行。说明一点，@Before 是在每个测试方法运行前均初始化一次，同理@After 是在每个测试方法运行完毕后，均运行一次。也就是说，经这两个注释的初始化和注销，可以保证各个测试之间的独立性而互不干扰，它的缺点是效率较低。另外，不需要在超类中显式调用初始化和清除方法，只要他们不被覆盖，测试运行程序将根据需要自动调用这些方法。超类中的@Before 方法在子类的@Before 方法之前调用（与构造函数调用顺序一致），@After 方法是子类中的在超类之前运行。

在 JUnit4 中加入了一项新特性。加入了两个注解：@BeforeClass 和@AfterClass，使用了这两个注解的方法，在该测试类中的测试方法之前、后各运行一次，而不是像@Before 和@After 那样，在每个方法运行之前和之后都要运行一次。对于资源消耗很大的方法，可以使用这两个注解。

2. 异常测试

因为使用了注解特性，JUnit4 测试异常非常的简单和明了。通过对@Test 传入 expected 参数值，即可测试异常。通过传入异常类后，测试类如果没有抛出异常或者抛出一个不同的异常，本测试方法就将失败。见代码：


```
package cn.hrmzone.junit;

import java.io.File;
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.io.SAXReader;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;

public class ExceptionTest {
File f;
Document doc;
@Before
public void init() {
f=new File("output"+File.separator+"test.xml");
}
@Ignore("not run")
@Test(expected=DocumentException.class)
public void read() throws DocumentException {
SAXReader reader=new SAXReader();
```

```
doc=reader.read(f);
}
@Test(expected=ArithmeticException.class)
public void divideZero() {
int n=2/0;
}
}
```

在第二个测试方法中，用 0 做除数，将会抛出 ArithmeticException 异常，所以对 expected 参数传入该类。测试抛出此异常，说明本次测试成功。如图：

测试成功后，方法前会出现一个小勾，大家可能注意到了，read( )方法，代码前有一个奇怪的@Ignore 注解，测试后也会出现一个标识。@Ignore 注解表示忽略，运行测试类时，被@Ignore 注解的方法将不会被测试，所以运行测试类后，会出现一个奇怪的标识。

3. 超时测试

通过在@Test 注解中，为 timeout 参数指定时间值，即可进行超时测试。如果测试运行时间超过指定的毫秒数，则测试失败。超时测试对网络链接类的非常重要，通过 timeout 进行超时测试，简单异常。如下例子：

```
package cn.hrmzone.junit;

import static org.junit.Assert.assertTrue;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.junit.Before;
import org.junit.Test;

public class RegularExpressionTest {
private String dateReg;
private Pattern pattern;

@Before
public void init() {
dateReg="^\\d{4}(\\-\\d{1,2}){2}";
pattern=Pattern.compile(dateReg);
}
// timeout 测试是指在指定时间内完成就正确
@Test(timeout=1)
public void verifyReg() {
Matcher matcher=pattern.matcher("2010-10-2");
boolean isValid=matcher.matches();
// 静态导入功能
assertTrue("pattern is not match",isValid);
}
}
```


4. 测试运行器

测试运行器：JUNit 中所有的测试方法都是由它负责执行。JUnit 为单元测试提供了默认的测试运行器，

但是没有限制必须使用默认的运行器。自己定制的测试运行器必须继承自 org.junit.runner.Runner。而且还可以为每一个测试类指定某个运行器：@Runwith(CustomTestRunner.class)。在 JUnit 中，有两个高级特性会需要自定义运行器。

5. 测试套件

JUnit4 中最显著的特性是没有套件(套件机制用于将测试从逻辑上的分组并将这这些测试作为一个单元测试来运行)。为了替代老版本的套件测试，套件被两个新注解代替：@RunWith、@SuteClasses。通过@RunWith 指定一个特殊的运行器：Suite.class 套件运行器，并通过@SuiteClasses 注释，将需要进行测试的类列表作为参数传入。

使用方法为：

a.创建一个空类作为测试套件的入口

b.将@RunWith、@SuiteClasses 注释修饰这个空类；

c.吧 Suite.class 作为参数传入@RunWith 注释，以提示 JUnit 将此类指定为运行器；

d.将需要测试的类组成数组作为@SuiteClasses 的参数。

注意：这个空类必须使用 public 修饰符，而且存在 public 的无参构造函数（类的默认构造函数即可）。

测试代码如下：将先前的两个测试类：RegularExpressionTest,Exception 作为一个测试套件进行测试：

```
package cn.hrmzone.junit;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({RegularExpressionTest.class,ExceptionTest.class})
public class SuiteTest {

}
```


@SuiteClasses 参数为需要测试类的数组，可以使用{}，将两个测试类传入。

测试套件类为空类，需要一个无参的 public 构造函数，使用默认构造函数即可。

运行结果如图，可以带到作为参数的两个测试类均运行，与单独测试一致。图 2

6. 参数化测试

为测试程序健壮性，可能需要模拟不同的参数对方法进行测试，如果在为每一个类型的参数创建一个测试方法，呵呵，人都疯掉了。幸好有参数化测试出现了。它能够创建由参数值供给的通用测试，从而为每个参数都运行一次，而不必要创建多个测试方法。注：测试方法（@Test 注释的方法）是不能有参数的。

参数化测试编写流程如下：

a.为参数化测试类用@RunWith 注释指定特殊的运行器：Parameterized.class；

b.在测试类中声明几个变量，分别用于存储期望值和测试用的数据，并创建一个使用者几个参数的构造函数；

c.创建一个静态（static）测试数据供给（feed）方法，其返回类型为 Collection，并用@Parameter 注释以修饰；

d.编写测试方法（用@Test 注释）。

测试示例代码如下：

```
package cn.hrmzone.junit;

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

```
import java.util.Arrays;
import java.util.Collection;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class ParameterTest {
private String dateReg;
private Pattern pattern;
// 数据成员变量
private String phrase;
private boolean match;

// 使用数据的构造函数
public ParameterTest(String phrase, boolean match) {
super();
this.phrase = phrase;
this.match = match;
}


@Before
public void init() {
dateReg="^\\d{4}(\\-\\d{1,2}){2}";
pattern=Pattern.compile(dateReg);
}

// 测试方法
@Test
public void verifyDate() {
Matcher matcher=pattern.matcher(phrase);
boolean isValid=matcher.matches();
assertEquals("Pattern don't validate the data format",isValid,match);
}
// 数据供给方法（静态，用@Parameter 注释，返回类型为 Collection
@Parameters
public static Collection dateFeed() {
return Arrays.asList(new Object[][] {
{"2010-1-2",true},
{"2010-10-2",true},
```

{"2010-123-1",false},
{"2010-12-45",false}
});
}
}
运行结果如图：

因为代码只测试日期格式，并未测试日期范围的正确性，2010-12-45 这数值测试与期望值不符。

7. 数组断言

JUnit4 中添加了一个用于比较数组的新断言（Assert），这样不必使用迭代比较数组中的条目。

public static void assertEquals(Object[] expected, Object[] actual)

public static void assertEquals(String message, Object[] expected,Object[] actual)

JUnit4 是向前兼容的，在 JUnit4 中依然可以使用 JUnit3 的测试代码，而不需要做任何改动。

JUnit 测试框架将测试更加便捷和容易，编写测试代码也是简单、明了，但功能依然强大。

4、最后了解一点关于 JUnit 的最佳实践（供参考体会）

1.使用 Junit 的最佳实践：

①新建一个名为 test 的 source folder，用于存放测试类源代码.

②目标类与测试类应该位于同一个包下面，这样测试类中就不必导入源代码所在的包，因为他们位于同一包下面

③测试类的命名规则:假如目标类的是 Calculator,那么测试类应该命名为 TestCalculator 或者是 CalculatorTest

2.Junit 的口号是: keep the bar green to keep the code clean.

3.Junit:单元测试不是为了证明您是对的，而是为了证明你没有错误.

4.测试用例(Test Case)是单元测试的一个很重要的方面.

5.单元测试主要是用来判断程序的执行结果与自己期望的结果是否一致.

6.测试类必须要继承与 TestCase 父类:

7.在 Junit3.8 中,测试犯法需要满足如下原则:

①public

②void 的

③无方法参数

④方法名称必须以 test 开头.

8.Test Case 之间一定要完全的独立性，不允许出现任何的依赖关系.

9.我们不能依赖于测试方法的执行顺序.

10.DRY(Don't Repeat Yourself):

11.关于 setUp 与 tearDown 方法的执行顺序:

①setUp

②testAdd

③tearDown

12.测试类的私有方法有时,可以采取两种方法:

①修改方法的访问修饰符，将 private 修改为 default 或 public(但不推荐采取这种方法.)

②使用反射在测试类中调用目标类的私有方法(推荐)

13. TestSuite(测试套件): 可以将多个测试组合在一起,同时进行多个测试.

14.RepeatedTest():重复执行测试，注意重复的是测试方法而不是测试类.

15. JUnit4 全面引入了 annotation 来执行我们编写的测试.

16.JUnit4 不要求测试类必须继承 TestCase 父类:

17.虽然 JUnit4 并不要求测试方法名以 test 开头，但我们最好还是按照 JUnit3.8 的要求那样，以 test 作为测试方法的名开头.

18. 在 JUint4 中 通过 @Befor 修饰的方法 对应于 3.8 中 setUp( )方法

19. 在 JUnit4 中, @After 修饰的方法相当于 3.8 中 tearDown( )方法.

20. 在 JUnit4 中，@BeforeClass 修饰静态方法，在所有的方法执行之前执行，可以进行一些代价较高的初始化工作.

21.在 JUnit4 中，@AfterClass 修饰静态方法，在所有方法执行之后执行.所有@AfterClass 修饰的方法理所当然的执行,即使 BeforeClass 方法抛出了异常. superclass 的@AfterClass 会在当前类后执行.

22.在 JUnit4 中，可以使用@BeforeClass 与@AfterClass 注解修饰一个 public static void no-arg 的方法,这样被@Before 注解所修饰的方法会在所有测试方法执行之前执行，被@AfterClass 注解所修饰的方法会在所有测试方法执行之后执行.

23. JUnit4 中, @Ignore:既可以用在测试方法上面也可以用在测试类上面，当修饰测试类时,表示忽略掉类中的所有测试方法；当修饰测试方法时，表示忽略掉该测试方法.其具有一个可选值.

24.JUnit4 的新功能:参数化测试:当一个测试类使用参数化运行器运行时,需要在类的声明除加上@RunWith(Parameterized.class)注解，表示该类将不使用 JUnit 内建的运行器运行，而使用参数化运行器运行,参数化运行器提供参数的方法上要是用@Parameters 注解来修饰，同时在测试类的构造方法中为各个参数赋值(构造方法是由 JUint 调用的)，最有编写测试类，它会根据参数的组数来运行测试多次.

25. 在 JUnit4 中，如果想要同时运行多个测试，需要使用两个注解：@RunWith(Suite.class)以及@Suite.SuiteClasses()，通过这两个注解分别制定使用 Suite 运行期来运行测试，以及制定了运行哪些测试类，其中@SuiteClasses 中可以继续制定 Suite，这样 JUint 会再去寻找里面的测试类，一直找到能够执行的 TestCase 并执行之.

26. JUnit 的设计模式:

①模板方法模式(Template Method)

定义一个操作中的算法骨架，而将一些步骤延伸到子类中去，是的子类可以不改变一个算法的结构，即可重新定义该算法的某些特定步骤，这里需要服用的是算法结构，也就是步骤，而步骤的实现可以在子类中完成.

使用场合:

1)一次性实现一个算法的不变部分，并且将可变部分的行为留给子类来完成,

2) 各子类公共的行为应该被提取出来并集中到一个公共父类中以避免代码的重复. 首先识别现有代码的不同之处，并且把不同部分分离为新的操作，最后，有一个调用这些新的操作的模板方法来替换这些不同的代码.

3)控制子类的扩展.

模板方法模式的组成:

父类角色:提供模板

子类角色: 为模板提供实现.

②适配器(Adapter)模式

在软件系统中,由于应用环境的变化，常常需要将"一些现存的对象"放在心的环境中应用，但是新环境要求的接口时这些现存对象所不能满足的. 那么如何应对这种"迁移的变化"? 如何既能利用现有对象的良好实现，同时又能满足新的的应用环境所要求的接口? 这就是本文所要说的 Adapter 模式.

意图: 将一个类的接口转换成客户希望的另外一个接口，Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作.

适配器(Adapter)模式的构成:

目标抽象角色(Target)

--定义客户要用的特定领域的接口.

适配器(Adapter)

--调用另一个接口，作为一个转换器

适配器(Adaptee)

--定义一个接口, Adapter 需要接入.

客户端(Client)

--协同对象符合 Adapter 适配器.

适配器的分类:有两种类型的适配器模式:

--类适配器(采取继承的方式)

--对象适配器(采取对象组合的方式)推荐

适用性:

--对象需要利用现存的并且接口不兼容的类.

--需要创建可重用的类以协调其他接口可能不兼容的类.

在 JUnit4 中的应用:

runBare( )方法中使用的 runTest( )方法.

```
public void runBare() throws Throwable {
    setUp();
    try {
        runTest();
    }
    finally {
        tearDown();
    }
```

在 runBare 方法中，通过 runTest 方法将我们自己编写的 testXXX 方法进行了适配，使得 JUnit 可以执行我们自己编写的 TestCase, runTest 的方法实现如下:

```
protected void runTest() throws Throwable {
    assertNotNull(fName);
    Method runMethod= null;
    try {
        // use getMethod to get all public inherited
        // methods. getDeclaredMethods returns all
        // methods of this class but excludes the
        // inherited ones.
        runMethod= getClass().getMethod(fName, null);
    } catch (NoSuchMethodException e) {
        fail("Method \""+fName+"\" not found");
    }
    if (!Modifier.isPublic(runMethod.getModifiers())) {
        fail("Method \""+fName+"\" should be public");
    }

    try {
        runMethod.invoke(this, new Class[0]);
    }
    catch (InvocationTargetException e) {
        e.fillInStackTrace();
        throw e.getTargetException();
    }
    catch (IllegalAccessException e) {
        e.fillInStackTrace();
        throw e;
    }
}
/**
```

在 runTest 方法中，首先获得我们字节编写的 testXXX 方法所对应的 Method 对象(不带参数)，然后见笑哈该 Method 所对应的方法是否是 public 的，如果是则调用 Method 对象的 invoke 方法来去执行我们自己编写的 testXXX 方法.

使用对象组合的方式实现适配器模式:

缺省的适配器模式:(AWT , Swing 事件模型所采用的适配器模式)

27. 命令模式:

JUnit 的设计使用 Pattern Generate Architectures 的方式来架构系统，其设计思想史通过从零才是来应用设计模式，然后一个接一个，直至你获得最终合适的系统架构.

Command 模式:将一个请求封装成一个对象，从而使你可用

不同的请求对客户经行参数化，对请求经行排队或记录请求日志..." Command 模式告诉我们可以为一个操作生成一个对象并给出它的一个"execute"(执行)方法.

意图: 将一个请求封装成一个对象，从而使你可用不同的请求对客户经行参数化；对请求排队或记录请求日志，以及支持可撤销的操作.

命令模式带来的效果:

① Command 模式将实现请求的一方和调用一方经行解耦.

②Command 模式使新的 TestCase 很容易加入，无需改变已有的类，只需要继承 TestCase 类即可，这样方便了测试人员.

③Command 模式可以将多个 TestCase 进行组合成一个命令, TestSuite 就是它的复合命令. 当然它使用了 Composite 模式.

④Command 模式容易把请求的 TestCase 组合成请求队列，这样是接受请求的一方(Junit Framework)，容易决定是否执行请求，一旦发现测试用例失败或者错误可以立即停止，进行报告.

命令模式的构成:

①客户角色:创建一个具体命令对象，并确定其接受者.

②命令角色: 声明一个给所有具体命令类的抽象接口，这是一个抽象角色，通常由一个接口或抽象类实现.

③具体命令角色: 定义一个接受者和行为之间的弱耦合，实现 execute 方法，负责调用接受者的响应操作.

④请求者角色: 负责调用命令对象执行请求.

⑤接受者角色:负责具体实现和执行一个请求.

28.我们学习过的设计模式:

①单例模式(singleton)

②策略模式(Strategy)

③代理模式(static proxy, dynamic proxy)

④观察者模式(Oberver)

⑤装饰模式(Decorator)

⑥工厂模式(FactoryMethod)

⑦模板模式(Template Method)

⑧适配器模式(Adapter)

⑨命令模式(Command)

⑩组合模式(Composite)

29. 组合模式(Composite):

组合模式有时候叫做部分-整体模式，它使我们属性结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦.

意图: 将对象组合成树形结构，以表示"部分-整体"的层次结构. Composite 模式使得用户对单个对象和组合对象的使用具有一致性.

组合模式的构成:

Component(抽象构件接口)

- 为组合的对象声明接口.
- 在某些情况下实现从此接口派生出的所有类共有的默认行为.
- 定义一个接口可以访问及管理它的多个子部件.

Leaf(叶部件)

- 在组合中表示叶节点对象，叶节点没有子节点.

- 定义组合中接口对象的行为.

Composite(组合类)

- 定义所有子节点(子部件)的部件的行为

- 存储子节点(子部件)

- 在 Component 接口中实现与子部件相关的操作.

Client(客户端):

-通过 Component 接口控制组合部件的对象.

30. 组合模式(Composite):组合模式有两种实现方式:

① 将管理子元素的方法定义在 Composite 类中,

② 将管理子元素的方法定义在 Component 接口中，这样 Leaf 类就需要对这些方法空实现.

30. Junit 的源代码:

分析源代码比较好的方法是使用调试的方式.

31. Juint 中的错误与失败:

① 错误指的是代码中抛出了异常等影响代码正常执行的情况，比如抛出了 ArrayIndexOutOfBoundsException，这就叫做错误.

② 失败指的是我们断言所期待的结果与程序实际执行的结果不一致，或者是直接调用了 fail() 方法，这叫做失败.

32. JUnit 中所使用的观察者模式:

33. 对于测试类来说，如果一个测试类中有 5 个测试方法，那么 JUnit 就会创建 5 个测试的对象，每一个对象只会调用一个测试方法(为了符合命令模式的要求)，在添加方法之前，需要首先判断测试方法是否满足 public，void, no-arg, no-return 这些条件，如果满足则添加则添加到集合当中准备作为测试方法去执行.