单元测试框架系列之-JUnit

什么是 JUnit？

JUnit是一个用 Java编写而成的单元测试框架，其早先的作者是大名鼎鼎的 Erich Gamma和 Kent Beck（后文对此还会详述）。利用JUnit，程序员可以通过编写简单的测试代码，方便的进行白盒测试，亦即：在了解被测代码如何工作的前提下，对其内部结构的正确性进行自动化的测试。
在 JUnit的官方主页上，还可以找到有关 JUnit的更为正统的释义：
JUnit是一个开放源代码的简单框架，用来编写和运行可重复的测试（注：也被称为可回归测试）。它是致力于单元测试框架的 xUnit架构的一种实现。其中包含了：

1. 用于检测预期结果的
   Assertions
2. 用于共享测试用数据的
   Test Fixtures
3. 用于运行测试的
   Test Runners

在没有 JUnit的年代，本分的程序员也会对自己开发的代码编写测试程序。但是，这种"ad hoc"的手段多缺乏通用性，无法重用。出自名家手笔的 JUnit，强大而趁手。遵照几条简单易学的规则写就的测试代码，其中富含了各种断言，在 IDE环境下，只消鼠标轻轻点击，便可一蹴而就——测试成功与否全凭一个直观的"green bar"或是"red bar"。程序员的目标很简单，那就是——"Keep the bar green to keep the code clean"。不仅如此，JUnit的出现还将 Java程序员们带入了敏捷开发和测试驱动的时代。通过测试代码的快速反馈来驱动开发过程，业已成为敏捷开发者们编写单元测试的首选方法。而"测试"与"重构"交替进行的"敏捷韵律操"也已经为大家所熟识。JUnit让众多程序员更加认可和信赖了敏捷开发，从这一点来看，JUnit的出现对技术社群的影响，已非一个简单的单元测试框架这么简单了。

最新版本及新版本最大变化

目前大家所使用的JUnit版本多是 3.8.x，不过今年初 JUnit发布了 4.0版本，紧接着又在 5月份发布了最新的 4.1版。新版
JUnit以 Java 5.0为支撑，使用了像 annotations，static import这样的新语法特性。较之以往的版本，变得更加简洁，更加丰富，更加易于使用。具体而言，
JUnit4提供的特色大致包括如下：

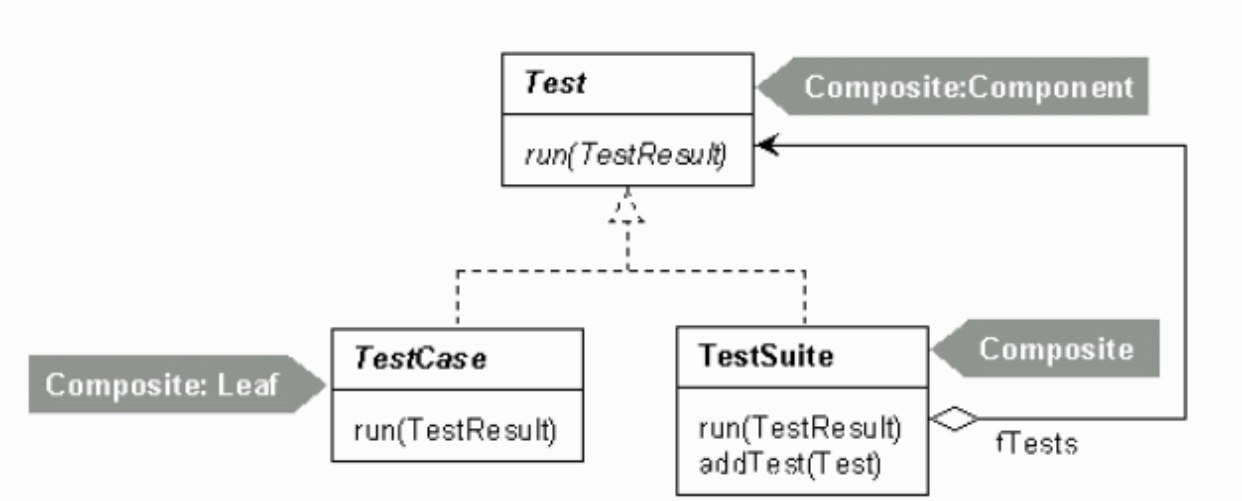
.
测试类不必再从 junit.framework.TestCase派生了；
.
测试方法也不必再以"test"作为前缀，而是代之以 @Test注解来标示；
.
作为 Fixtures的 setUp与 tearDown也不再强制使用这两个方法名了，只要在任何方法名称前冠以@Before或@After，即可达到一样的效果；
.
对 setUp/tearDown的一大改进还包括，可以限定二者只在整个 test case范围内执行一次，这是通过@BeforeClass和@AfterClass注解达成的；
.
@Test注解还可以带上 timeout参数和 expected参数，前者代表测试方法超过指定时间即被认为失败，后者则声明了预期被抛出的异常类型；此外，为了和以前版本的 Test Runner兼容，JUnit4提供了一个 JUnit4Adapter。有了它，用 JUnit4写的测试代码就可以运行于旧版本的 Test Runner下了。当然，以前写的测试代码在 JUnit4的 Test Runner里是可以直接运行的。耳听为虚，眼见为实。让我们以一个简单的例
子来给大家演示一下新特性的使用方法：

```
package example.junit4;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;

public class LibraryTest {不必从
TestCase派生
private Library library;
@BeforeClass public void runOnceBeforeAllTests() {仅在所有测试方法执行前执行
// …
}
@Before public void runBeforeEachTest() {不必以
setUp作为方法名称
library = new Library();
}

@Test public void bookAvailableInLibrary () {不必以
test打头
boolean result = library.checkAvailabilityByTitle(“Webster’s Dictionary”);
assertEquals (“Our Library should have the standard Dictionary”,

true,
result);
}

@Test(expected=BookNotAvailableException.class)指定预期抛出的异常
public void bookNotAvailableInLibrary(){
library.checkAvailabilityByTitle(“Some book that does not exist”);


}
@After public void runAfterEachTest() {不必以
tearDown作为方法名称
Library = null;
}

@AfterClass public void runAfterAllTests() {仅在所有测试方法执行后执行
// …
}

public static junit.framework.Test suite() {
return new JUnit4TestAdapter(LibraryTest.class);
}
}
```

有关JUnit组织方式与设计架构的讨论，应当首推 Erich Gamma与 Kent Beck合写的"JUnit： A Cook's Tour"（可以在 JUnit的官方主页上找到）。虽然这篇文章是针对 JUnit 3.8.x写的，但却颇具研习的价值。作者在文中带领着大家从零开始，通过逐个运用模式，最终构造出完整的 JUnit。此处，笔者就着 JUnit的主体设计，为大家介绍两个出现其中的常用模式。更为详细的内容，请读者参阅相关资源1。值得一提的是，小巧精致的 JUnit是学习设计模式的绝好素材。

Composite模式

在 JUnit中，有一个贯穿始终的设计模式，那就是 Composite模式。在《设计模式》一书中对该模式是这么解释的：将对象组合成树形结构以表示"部分－整体"的层次结构。Composite使用户对单个对象和组合对象的使用具有了一致性。在 JUnit的框架里，测试类分为两种，某些测试类代表单个测试，称为 TestCase，另一些则由若干测试类组合而成，称为 TestSuite。彼此相关的 TestCase共同构成一个 TestSuite，而 TestSuite也可以嵌套包含。两者分别对应了 Composite模式中的 Leaf和 Composite。如下所示，TestCase和 TestSuite共同派生自 Test接口：



图：Composition模式

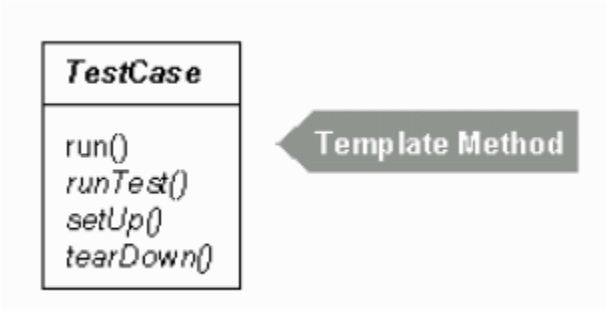
Template Method模式

我们知道，在每一个测试方法执行之前，可能需要准备测试用的数据，而在测试方法执行完毕之后，也可能需要做些清理工作。借助 Template Method模式，JUnit为我们提供了所谓的 Test Fixture，使得多个测试方法可以共享同样的测试环境，并且每个测试方法的执行环境彼此独立，互不影响。在 TestCase的 run方法中，执行逻辑大致如下：

```
public void run() {
setUp();
runTest();
tearDown();
}
```

其中，setUp用于在测试方法执行之前初始化测试环境，tearDown则在测试方法执行之后清理测试环境。如此，每个测试方法在执行前都会重新建立测试环境，使得当前测试方法的执

行不依赖于其他测试方法。在 TestCase中，setUp与 tearDown的默认实现不做任何事情。具体实现逻辑要由测试编写人员在 TestCase的子类中来决定。这正是 Template Method模式的主旨所在：在父类中定义算法执行步骤，将步骤的具体执行逻辑推迟到子类中实现。实际上，该模式在一般的 framework中十分常见。



图：Template Method模式

主要人物及相关故事

JUnit最初是由 Erich Gamma和 Kent Beck二人合力打造而成的。说起这两位大师级人物，想必大家都不会陌生。

Erich Gamma是 Eclipse的架构师之一，也是技术畅销书《设计模式》（Addison-Wesley, 1995）的合著者之一。该书针对软件领域通常遇到的设计问题，采用面向对象的设计思想分类理出了 23种特定的解决方案。这本书自出版以来虽十年有余，却依旧能时常见诸 Amazon的畅销书排行榜。正是这本经久不衰的里程碑式的书籍，让 Erich Gamma跃上了软件业界的舞台，位列大师之中。他与该书的另三位作者常被拥趸们戏称为"Gang of Four"。Erich Gamma目前是 IBM的一名杰出工程师，在瑞士苏黎世的 IBM Object Technology International（OTI）实验室工作。他还担任着 Eclipse社区的领导工作，负责 Eclipse平台上与 Java开发相关的事宜。
JUnit的另一位作者 Kent Beck也非等闲之辈。他被人们称为软件开发方法学的泰斗，长期致力于软件工程的理论研究和实践。作为极限编程（eXtreme Programming）和测试驱动开发的创始人，软件业界最富创造力，最有口碑的领导者之一，Kent Beck极力推崇设计模式、极限编程和测试驱动开发，同时他还是多部经典技术书籍的作者或合作者，包括：《Smalltalk Best Practice Patterns》（Prentice Hall，1996）、《解析极限编程——拥抱变化》（Addison-Wesley，
2nd Edition，2004）、《规划极限编程》（Addison-Wesley，2000），《Test-Driven Development: By
Example》（Addison-Wesley，2002），《Contributing to Eclipse》（Addison-Wesley，2003，该
书的另一位作者即是 Gamma）。Kent Beck目前是 Three Rivers Institute（TRI）的总裁，TRI 主要从事技术和商业接合的应用研究。从某种程度上讲，两位大师的联袂注定了 JUnit在软件开发领域不可撼动的地位和影响力。自 1998年诞生以来，根正苗红的 JUnit深受 Java程序员们的好评：2001及2002年连续两届"Java World编辑选择奖"，2003年"Java World最佳测试工具"，2003年"Java Pro最佳 Java测试工具"。现在，JUnit已然成为 Java社区单元测试的"事实标准"。不仅如此，JUnit还影响到了许多其他技术领域和技术平台，以至于人们会将其称为庞大的"xUnit家族"。在这里，我们可以列出一串长长的名单：HttpUnit，HtmlUnit，DBUnit，CppUnit，NUnit，JUnitPerf， accessUnit，AS2Unit，CUnit，……，这其中有些是 JUnit的直接"翻版"，而另一些则依托于 JUnit之上，有着特殊的用途。

Step by Step

一、准备工作

JUnit的官方网站提供了最新版本的 JUnit下载。不过，由于 JUnit已经相当普及，一般的 IDE开发环境，比如：Eclipse，JBuilder，NetBeans，IntelliJ等等，都缺省提供了对 JUnit的支持，而不必单独去下载。若非使用 JUnit4的新功能（目前流行的 IDE开发环境多以支持3.8.x版本为主），否则可以直接使用 IDE中内置的 JUnit。此外，由于一般的 IDE都有着良好的 GUI功能，因此 JUnit的使用会变得更加方便。后文将以 Eclipse 3.1.2为例，向读者演示 JUnit的使用方法，该版本缺省内置了 3.8.1版的 JUnit插件。读者可以从 Eclipse的官方网站（http://www.eclipse.org/）下载到最新版本的 Eclipse。至于其他的 IDE环境，使用方法大同小异，读者可以查阅相关资源。在这里，我们使用一个简单的具有绝对值计算功能的数学运算工具类作为被测试对象，在演示如何使用 JUnit的同时，也向读者演示一个简单的测试驱动开发过程。限于篇幅，有关测试驱动开发的知识请读者参阅其他相关资料。

二、**MathUtil**的"需求"

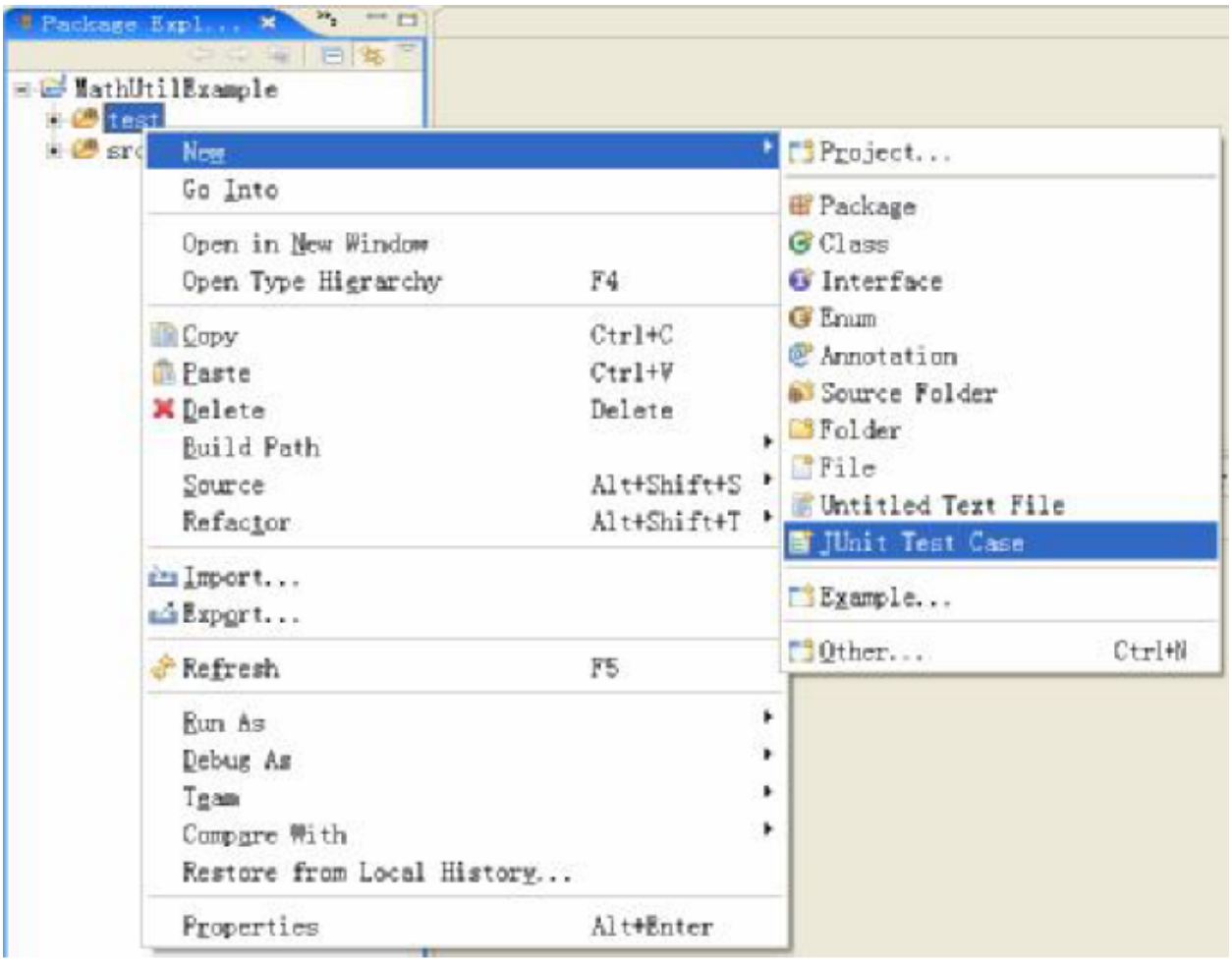
假设我们要实现一个提供了若干常用数学运算功能的工具类，名叫 MathUtil。首先，我们为其加入一个绝对值计算函数。不过，在开始编写实现代码之前，我们依据测试先行的建议，先来考虑一下绝对值运算的"需求"。一个绝对值运算，应该包含如下几项"需求"：

l当给定一个正数时，应该返回同样大小的正数；
l当给定一个负数时，应该返回该负数的相反数；
l当给定 0时，应该返回 0；

下面，我们即将开始编写第一个测试用例，测试用例的目的就是要将上述"需求"以代码的形式表达出来。不过此前，我们还需要先在 Eclipse中创建一个名叫 MathUtilExample的新 Java工程，并在工程中新建两个目录：用于存放测试代码的 test目录和用于存放源文件的 src目录。

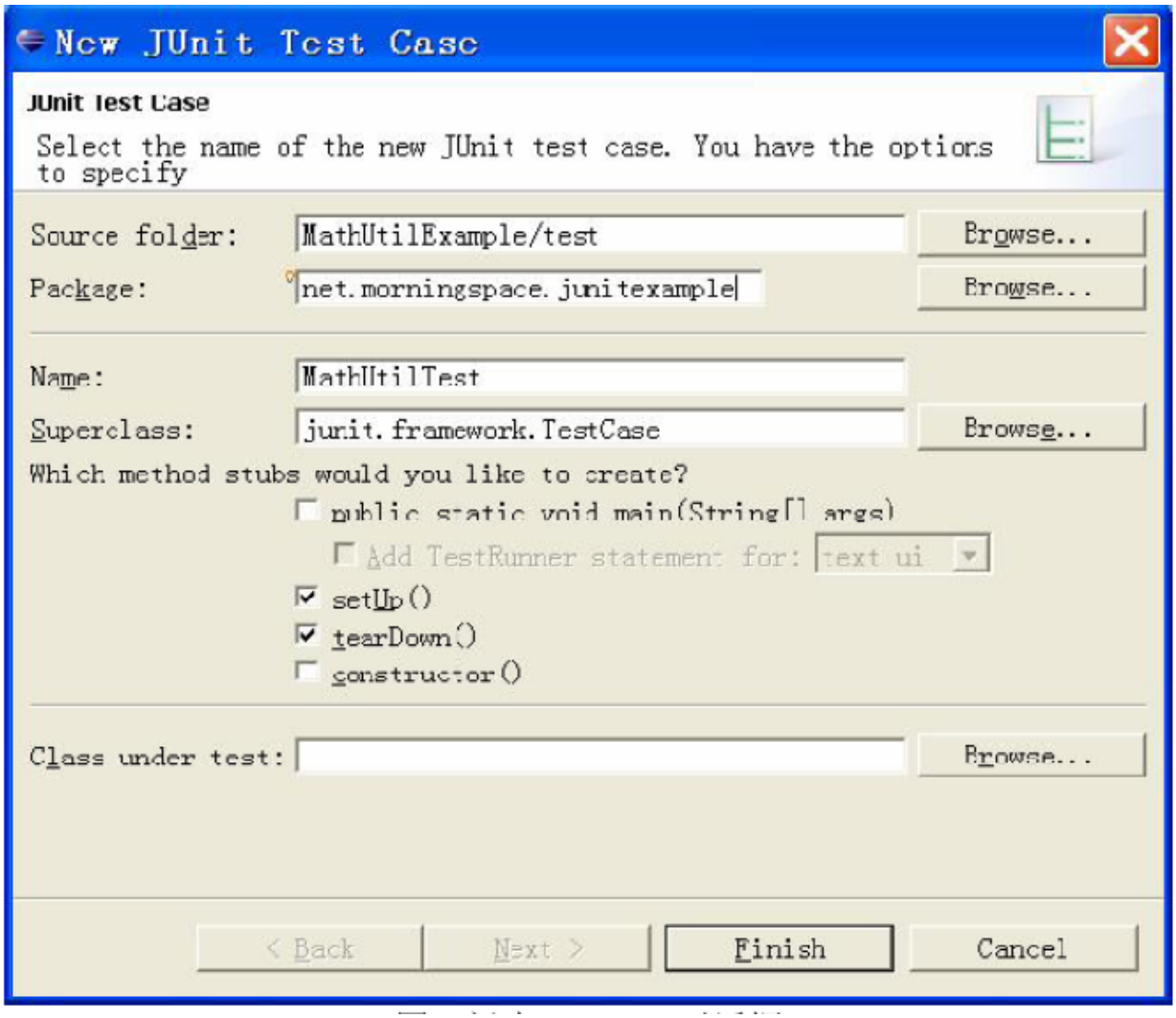
三、编写第一个 **TestCase**

在工程中选中 test目录，右键点击，选择 ***New->JUnit Test Case***。此时，Eclipse会询问是否在工程中加入 junit.jar，点击确定即可。



图：新建 JUnit TestCase

在随后打开的新建 Test Case对话框中，我们按照对话框的提示新建一个测试类：net.morningspace.junitexample.MathUtilTest。然后选中 setUp()和 tearDown()，并点击“Finish”。



图：新建 Test Case对话框

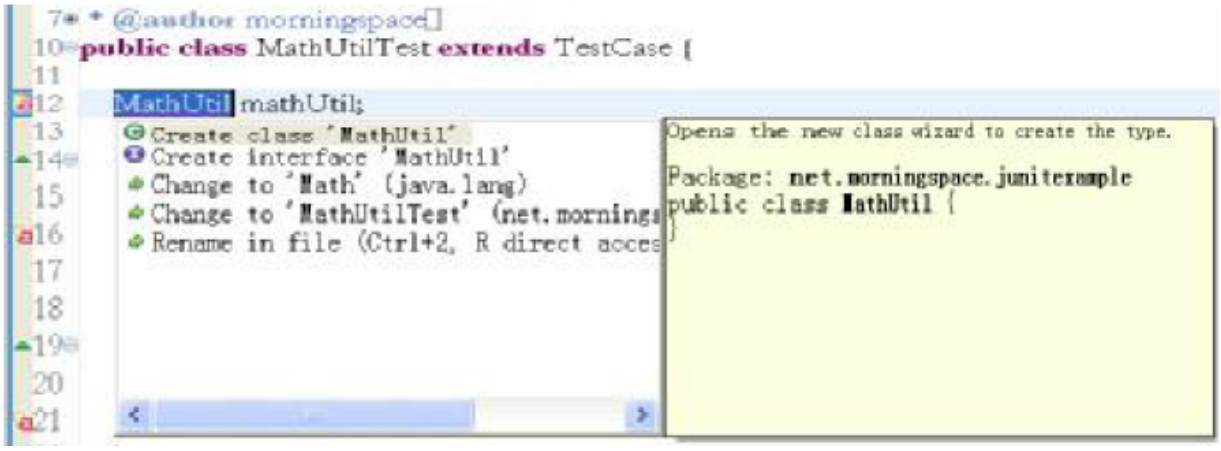
现在我们可以开始在新建的
MathUtilTest类中添加测试代码了。根据前面所列的"需求"，我们将在MathUtilTest中逐步添加测试方法，随着开发的推进，依次满足每项"需求"。首先我们添加第一个测试方法，以满足正数的情形：

MathUtilTest.java

```
1  package net.morningspace.junitexample;
2
3  import junit.framework.TestCase;
4
7  * @author morningspace
10 public class MathUtilTest extends TestCase {
11
12    MathUtil mathUtil;
13
14    protected void setUp() throws Exception {
15       super.setUp();
16       mathUtil = new MathUtil();
17    }
18
19    protected void tearDown() throws Exception {
20       super.tearDown();
21       mathUtil = null;
22    }
23
24    public void testAbsShouldReturnPositiveWhenGivenPositiveNumber() {
25       assertEquals(15, mathUtil.abs(15));
26    }
27 }
```

图：添加第一个测试方法

此处，我们使用了 JUnit的断言 assertEquals。通过该断言，我们确保了当传入的数值为 15 时，返回值必是 15。另外，利用 setUp()，我们在每个测试方法执行前都创建了新的 MathUtil实例，同时利用 tearDown()，在执行完毕后又将该实例销毁。编辑窗口左侧的红叉告诉我们，此时还没有创建 MathUtil类及其 abs函数。因此，接下来我们要新建一个 MathUtil类，并以最简单的方法让测试通过。利用 Eclipse提供的便捷功能，我们可以很方便的新建 MathUtil类：

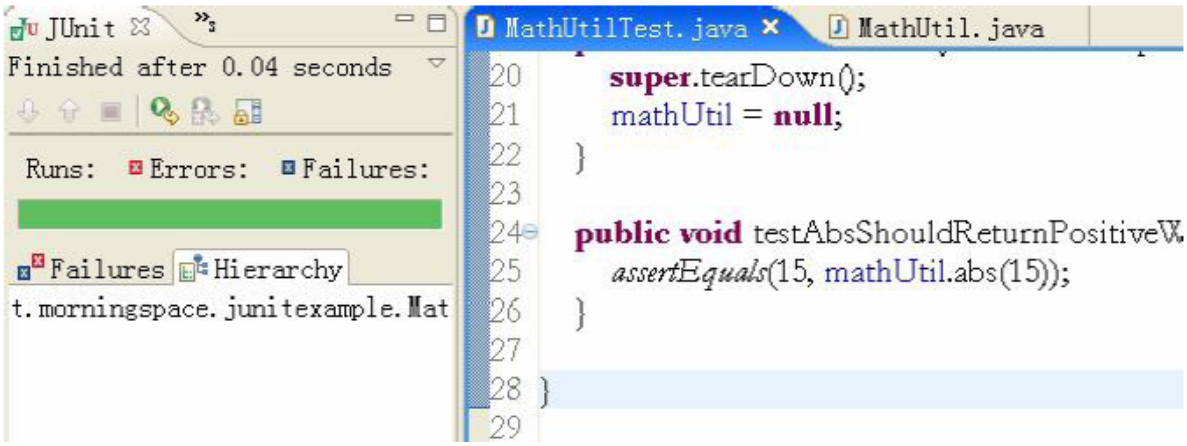


图：新建 MathUtil类

```
1  package net.morningspace.junitexample;
2
5  * @author morningspace
8  public class MathUtil {
9
10   public int abs(int number) {
11     return number;
12   }
13 }
```

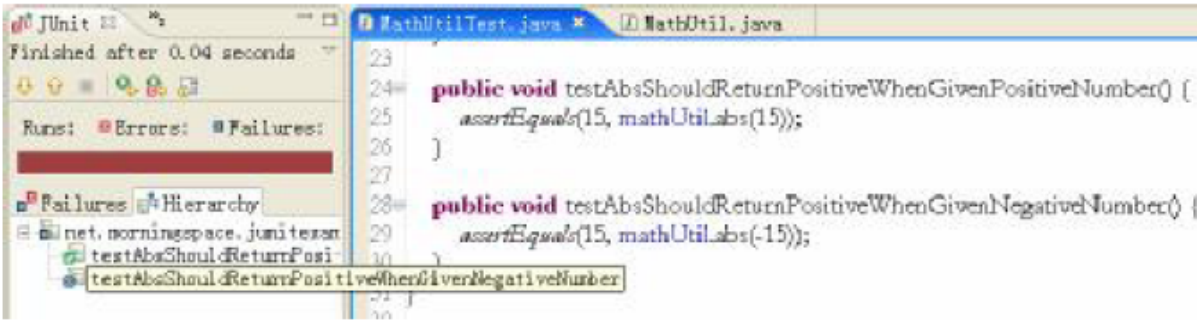
图：尝试让测试通过的最简单做法，未必是正解

随后我们运行一下单元测试，选择 Run->Run As->JUnit Test，和预想的一样，测试顺利通过了。



图：第一个测试方法通过

JUnit单元测试的执行速度一般都很快，所以运行后立刻就能看到测试结果。JUnit以其经典的“green bar/red bar”来表示测试通过与否。绿色表示测试通过，红色表示测试失败，假如有多个测试方法时，只要有一个测试方法没有通过，就会显示红色并列出未通过测试的方法。接下来我们再加入第二个测试方法，以满足负数的情况。再次执行单元测试，此时的结果显示，新添加的测试方法没有通过。

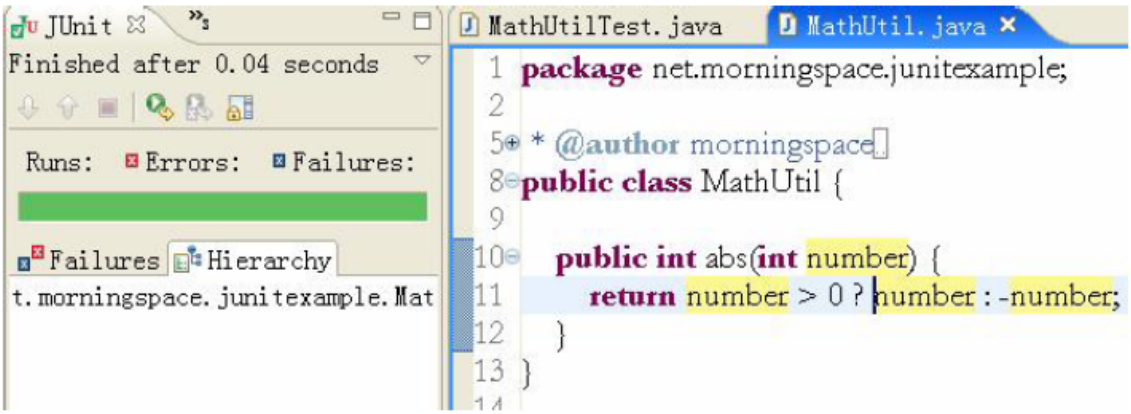


图：第二个测试方法没有通过

为了让测试通过，我们不得不修改MathUtil.abs()原来的实现，不过这次依然采用的是让测试通过的最简单办法，我们试着在原有返回值前增加一个负号：

```
public int abs(int number) {
return –number;
}
```

很遗憾，这次没有那么走运，再次运行测试后发现，结果依然是“red bar”。不过，与上次有所不同的是，这回没有通过测试的不是新增的测试方法，而是前一个已经通过了的测试方法。这一结果启发了我们，应该考虑引入一种 if...else的结构，来同时应对正数和负数这两种情况：



图：前两个测试方法顺利通过

这一回，两个测试都通过了。最后，我们加入第三个测试方法，以验证传入参数为0的情况。
结果显示为"green bar"。至此，一个简单的测试驱动过程完成了。

四、小结

在上面的例子里，我们简单演示了在 Eclipse环境下，如何利用 Eclipse与 JUnit的良好集成能力，以测试驱动的方式，编写 JUnit单元测试。也向大家演示了如何使用 setUp/tearDown，以及如何使用断言。不过由于例子十分的简单，所以无法演示 JUnit的所有使用技巧。更为高级的使用方法，读者可以查阅相关资源。另外，后文的 FAQ中对此也会有所涉及。

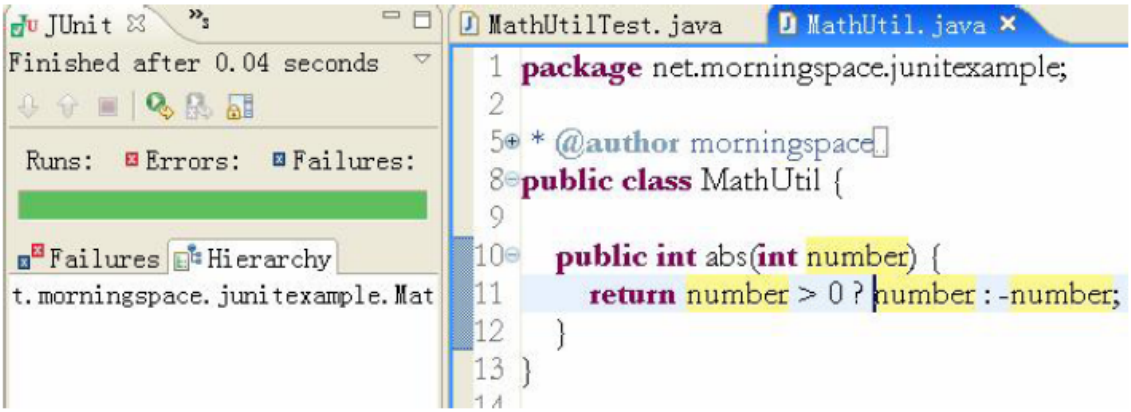
五、与**Ant**结合使用

与 JUnit一样，另一个开源项目——自动化编译部署工具 Ant，也是大多数 Java程序员的必备工具。利用 Ant，我们可以将许多繁琐工作自动进行，从而极大的简化了开发过程。此外，目前像极限编程这样的敏捷方法中时常倡导的持续集成，也要求有能够持续执行的自动化编译、测试和部署机制。而将 Ant与 JUnit结合使用，会使这一机制发挥更大的威力，产生更大的价值：只消简单的配置，从编译代码到打包部署，从自动测试到报告生成，可谓一气呵成。
因为 Ant早已广为流传，所以主流 IDE对它也都有很好的内置支持，使用起来十分的方便。这里，我们仍已 MathUtil为例，为大家演示一下在 Eclipse环境中，如何编写 Ant脚本，利用 Ant提供的相关 task自动执行单元测试，并生成测试报告。篇幅所限，有关 Ant的详细使用方法，推荐读者阅读《Java Development with Ant》一书（Manning，2002）。在 MathUtilExample工程的根目录下新建一个 build.xml文件，该文件的内容摘要如下：

```
<project name="MathUtilExample" basedir="." default="compile" >
……

<target name="compile" depends="init" >编译 Java源文件
<javac srcdir="${src.dir}" destdir="${build.dir}" >
<include name="**/*.java"/>
</javac>
</target>


<target name="compile-tests" depends="compile">编译测试代码
<mkdir dir="${test.build.dir}"/>
<javac srcdir="${test.src.dir}" destdir="${test.build.dir}">

<classpath>
<pathelement location="${build.dir}" />
</classpath>
<include name="**/*.java"/>

</javac>
</target>

<target name="test" depends="compile-tests">执行单元测试
<mkdir dir="${test.data.dir}" />创建用于保存单元测试中间结果的目录

<junit printsummary="yes" errorProperty="test.failed"
failureProperty="test.failed" fork="yes">
```

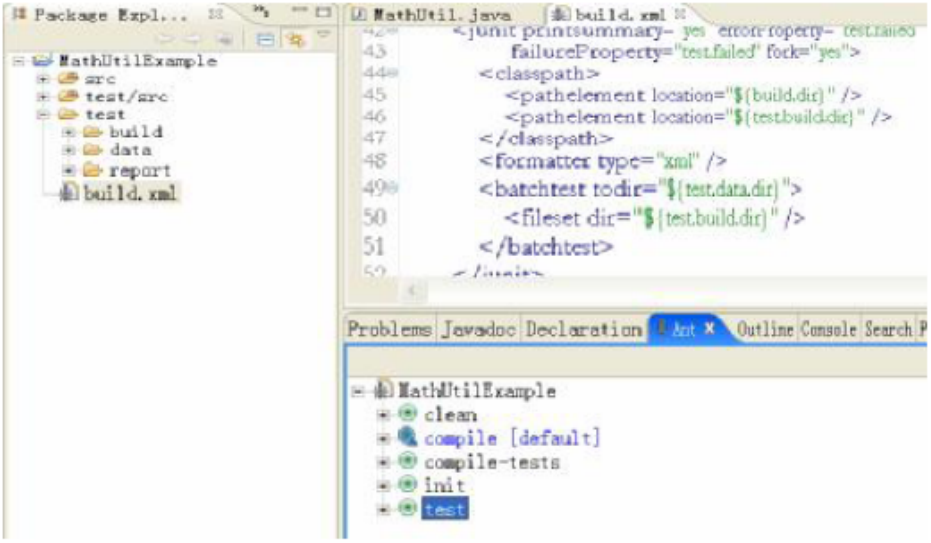
```
<classpath>
<pathelement location="${build.dir}" />
<pathelement location="${test.build.dir}" />

</classpath>
<formatter type="xml" />
<batchtest todir="${test.data.dir}">批量执行指定目录下的单元测试

<fileset dir="${test.build.dir}" />
</batchtest>
</junit>

<mkdir dir="${test.report.dir}" />创建用于存放测试报告的目录

<junitreport todir="${test.report.dir}">生成测试报告
<fileset dir="${test.data.dir}">

<include name="TEST-*.xml"/>
</fileset>
<report format="frames" todir="${test.report.dir}"/>
</junitreport>
<fail message="Tests failed. Check reports"如果测试失败则终止构建过程
if="test.failed" />
</target>
</project>
```

待
build.xml编写完后，选择
Window->Show View->Ant，打开 Ant视窗，然后将 build.xml文件"托拽"到该视窗中。



图：位于 Ant视窗中的 build脚本此时，双击 Ant视图中名为 test的 target，即可在控制台视图中得到结果

此外，我们还可以在${test.report.dir}目录下找到一组漂亮的 html格式的测试报告：



图：html格式的测试报告

FAQ

问：从哪里可以下载到 **JUnit**的最新版本？
答：最新版本的
JUnit可以在
http://sourceforge.net/project/showfiles.php?group_id=15278下载
到。
问：在哪里可以找到**JUnit**的文档资源？
答：在 JUnit的官方主页上，你可以找到丰富的在线文档。你还可以通过网络搜索到大量有关 JUnit使用的资源。此外，与 JUnit相关的技术书籍也层出不穷，对此，读者可以参考后文的推荐书目。
问：我应该把测试代码放在哪里？
答：可以把测试类与待测类放在同一个包下，这对于小规模系统而言应该足够了。例如：

src
com

.junitexample
MathUtil.java
MathUtilTest.java

如果你觉得这种方法会造成源代码目录的混乱，并且会让发布部署变得复杂，那么不妨把测试代码单独放到一个目录下，并保持与被测代码相同的包结构。例如：

src
com
junitexample
MathUtil.java
test
com

junitexample
MathUtilTest.java

问：什么是**Test Fixture**？
答：Test Fixture是一组测试所需的公共数据和协作对象，它为若干测试所共享。通常用测试类的实例变量（也就是数据成员）来表示。
问：**Failure**和 **Error**有什么区别？
答：JUnit将测试不通过的结果分为两种类型：Failure和Error。其中，Failure指的是断言结果为 false，亦即你所预期的结果没有满足，从而说明测试失败；Error则是指意料之外的异常，这在测试运行之前是无法预期的，例如：ArrayIndexOutOfBoundsException。
问：如何测试保护类型的成员函数？
答：只要保持测试代码的包路径与待测类的包路径一致即可。保护类型的成员函数在包范围内是可见的。
问：如何测试私有类型的成员函数？
答：一般来说，测试私有类型的成员函数很可能暗示着代码需要重构，这些方法往往应该被移入另一个类中，以提高代码的重用。不过，如果你真的想测试私有类型的成员函数，并且你用的是 JDK 1.3及其以后的版本，那么你可以用 Java的反射机制，配合 PrivilegedAccessor3来突破 Java访问控制的限制。

问：如何编写一个测试，让其在预期异常抛出的时候通过（或失败）？
答：如果你用的是 JUnit4，可以利用@Test的 expected属性定义预期抛出的异常，一旦异常如期抛出，测试就通过了。假如你用的是 JUnit 3.8.x，则可以采用类似如下的方法来达到同样的目的：

```
public void testIndexOutOfBoundsException() {
try {
ArrayList emptyList = new ArrayList();
Object o = emptyList.get(0);
fail("Should throw IndexOutOfBoundsException!");

} catch (IndexOutOfBoundsException e) {
// gotha...
}
}
```

假如预期异常抛出时，希望测试失败，则只需要简单的在测试方法签名中加上throws声明，
并且确保不在方法体内
catch异常即可。例如：

```
@Test public void testIndexOutOfBoundsExceptionNotRaised()
throws IndexOutOfBoundsException {
ArrayList emptyList = new ArrayList();
Object o = emptyList.get(0);
}
```

问：如何测试必须运行在 **J2EE**容器内的代码（例如：**servlets**，**EJBs**）？
答：对J2EE组件进行重构，将绝大部分执行逻辑移到普通Java对象中（POJO），它们可以脱离容器执行，这是一种有效改善设计和系统可测试性的手段。此外，Apache Cactus是一个开源的JUnit扩展框架，使用它可以模拟容器内的组件测试。

问：在什么时候应该测试 **getter/setter**方法？
答：多数情况下没有必要测试
getter/setter，因为这两个方法十分简单，不会导致程序错误。
假如
getter/setter有可能导致程序运行错误，那么你可以考虑为它们编写测试代码。例如，
下面的代码中，我们希望验证在
getX()调用的时候，X的值已经通过构造函数被正确设置了。
这样的测试也许是有价值的，尤其在
MyClass有多个版本的构造函数时：

```
@Test public void testCreate() {
assertEquals(23, new MyClass(23).getX());
}
```

问：我是否要为每个待测试的类都编写一个**TestCase**？
答：用不着，虽然通常的习惯是一个待测类对应一个测试类，但这并非是必须的。TestCase 仅仅提供了一种组织测试的方法。也许开始的时候你会用一个测试类来对应待测类，但随后你也许会发现，测试类中有一组测试方法具有共同的 Test Fixture，此时你可以将这些测试重构为一个新的测试类。
问：我应该多久运行一次单元测试？
答：尽可能频繁的运行单元测试，确保所有的单元测试都被执行。频繁的测试可以为你修改既有代码增加自信心，你不再担心因为不小心误改代码而破坏原有程序的功能了。对于较大规模的软件系统，你可以运行和目前工作相关的测试集，并且至少每天运行一次完整的测试集。
问：为什么不应该简单的使用 **System.out.println()**？
答：在代码中插入调试用的
System.out.println()是一种最为原始的程序调试手段。你需要通
过肉眼观察输出的调试结果，来判断代码执行是否正确。这种输出结果无法用
JUnit提供的断言机制来表达，因而也就无法充分有效的利用 JUnit。
问：为什么不应该仅仅使用调试手段？
答：人们常常使用调试的方式来深入代码内部，查看执行逻辑和变量取值情况。但是这种方式只能手工进行，本质而言，它等同于手工检查"实际值与预期值是否一致"。而且，一旦程序改动之后，我们就必须再次从头开始一步步调试。将"实际值与预期值是否一致"的判断以自动化方式执行，会弥补手工调试的不足。并且，每当这种自动判断很难编写时，往往预示着你的设计有待改进。