**JUnit**

# JUnit - 概述

测试用例不是用来证明你是对的，而是证明你没有错。

测试用例用来达到想要预期的结果，但对于逻辑错误无能为力。

所谓单元测试是测试应用程序的功能是否能够按需要正常运行，并且确保是在开发人员的水平上，单元测试生成图片。单元测试是一个对单一实体（类或方法）的测试。单元测试是每个软件公司提高产品质量、满足客户需求的重要环节。

单元测试可以由两种方式完成

|  |  |
| --- | --- |
| 人工测试 | 自动测试 |
| 手动执行测试用例并不借助任何工具的测试被称为人工测试。  - 消耗时间并单调：由于测试用例是由人力资源执行，所以非常缓慢并乏味。  - 人力资源上投资巨大：由于测试用例需要人工执行，所以在人工测试上需要更多的试验员。  - 可信度较低：人工测试可信度较低是可能由于人工错误导致测试运行时不够精确。  - 非程式化：编写复杂并可以获取隐藏的信息的测试的话，这样的程序无法编写。 | 借助工具支持并且利用自动工具执行用例被称为自动测试。  - 快速自动化运行测试用例时明显比人力资源快。  - 人力资源投资较少：测试用例由自动工具执行，所以在自动测试中需要较少的试验员。  - 可信度更高：自动化测试每次运行时精确地执行相同的操作。  - 程式化：试验员可以编写复杂的测试来显示隐藏信息。 |

* **什么是 JUnit**

JUnit 是一个 Java 编程语言的单元测试框架。JUnit 在测试驱动的开发方面有很重要的发展，是起源于 JUnit 的一个统称为 xUnit 的单元测试框架之一。

JUnit 促进了“先测试后编码”的理念，强调建立测试数据的一段代码，可以先测试，然后再应用。这个方法就好比“测试一点，编码一点，测试一点，编码一点……”，增加了程序员的产量和程序的稳定性，可以减少程序员的压力和花费在排错上的时间。

**特点：**

1. JUnit 是一个开放的资源框架，用于编写和运行测试。
2. 提供注释来识别测试方法。
3. 提供断言来测试预期结果。
4. 提供测试运行来运行测试。
5. JUnit 测试允许你编写代码更快，并能提高质量。
6. JUnit 优雅简洁。没那么复杂，花费时间较少。
7. JUnit 测试可以自动运行并且检查自身结果并提供即时反馈。所以也没有必要人工梳理测试结果的报告。
8. JUnit 测试可以被组织为测试套件，包含测试用例，甚至其他的测试套件。
9. JUnit 在一个条中显示进度。如果运行良好则是绿色；如果运行失败，则变成红色

* **什么是一个单元测试用例**

单元测试用例是一部分代码，可以确保另一端代码（方法）按预期工作。为了迅速达到预期的结果，就需要测试框架。JUnit 是 java 编程语言理想的单元测试框架。

一个正式的编写好的单元测试用例的特点是：已知输入和预期输出，即在测试执行前就已知。已知输入需要测试的先决条件，预期输出需要测试后置条件。

每一项需求至少需要两个单元测试用例：一个正检验，一个负检验。如果一个需求有子需求，每一个子需求必须至少有正检验和负检验两个测试用例。

# JUnit - 环境设置

JUnit 是 Java 的一个框架，所以最根本的需要是在你的机器里装有 JDK。

**步骤1：在你的机器里验证 Java 装置**

现在打开控制台，执行以下 java 要求

|  |  |  |
| --- | --- | --- |
| 操作系统 | 任务 | 命令 |
| Windows | 打开命令操作台 | c:>java –version |
| Linux | 打开命令终端 | $ java –version |
| Mac | 打开终端 | machine:~ joseph$ java -version |

我们来验证一下所有操作系统的输出：

|  |  |
| --- | --- |
| 操作系统 | 输出 |
| Windows | java 版本 “1.6.0\_21”  Java（TM）SE 运行环境（build 1.6.0\_21-b07）  Java 热点（TM）客户端虚拟机（build 17.0-b17，混合模式，共享） |
| Linux | java 版本“1.6.0\_21”  Java（TM）SE 运行环境（build 1.6.0\_21-b07）  Java 热点（TM）客户端虚拟机（build 17.0-b17，混合模式，共享） |
| Mac | java 版本“1.6.0\_21”  Java（TM）SE 运行环境（build 1.6.0\_21-b07）  Java 热点（TM）64-字节服务器虚拟机（build 17.0-b17，混合模式，共享） |

**步骤2：设置 JAVA 环境**

设置 JAVA\_HOME 环境变量，使之指向基本目录位置，即在你机器上安装 Java 的位置。

|  |  |
| --- | --- |
| 操作系统 | 输出 |
| Windows | 设置环境变量 JAVA\_HOME to C:\Program Files\Java\jdk1.6.0\_21 |
| Linux | 输出 JAVA\_HOME=/usr/local/java-current |
| Mac | 输出 JAVA\_HOME=/Library/Java/Home |

系统路径添加 Java 编译器位置。

|  |  |
| --- | --- |
| 操作系统 | 输出 |
| Windows | 在系统变量路径末端添加字符串  ;C:\Program Files\Java\jdk1.6.0\_21\bin |
| Linux | 输出 PATH=$PATH:$JAVA\_HOME/bin/ |
| Mac | 不需要 |

使用以上解释的 Java-version 命令验证 Java 安装。

**步骤3：下载 Junit jar包**

从 http://www.junit.org 下载 JUnit 最新版本的压缩文件。编写这个教程的同时，我已经下载了 Junit-4.10.jar 并且将它复制到 C:>JUnit 文件夹里了。

|  |  |
| --- | --- |
| 操作系统 | 档案名称 |
| Windows | junit4.10.jar |
| Linux | junit4.10.jar |
| Mac | junit4.10.jar |

**步骤4：设置 JUnit 环境**

设置 JAVA\_HOME 环境变量，使之指向基本目录位置，即在你机器上安装 JUNIT 压缩文件的位置。假设，我们已经在以下不同的操作系统的 JUNIT 文件夹里存储了 junit4.10.jar。

|  |  |
| --- | --- |
| 操作系统 | 档案名称 |
| Windows | 设置环境变量 CLASSPATH 到 %CLASSPATH%;%JUNIT\_HOME%\junit4.10.jar;.; |
| Linux | 输出 CLASSPATH=$CLASSPATH:$JUNIT\_HOME/junit4.10.jar:. |
| Mac | 输出 CLASSPATH=$CLASSPATH:$JUNIT\_HOME/junit4.10.jar:. |

**步骤6：测试 JUnit 建立**

在 C:\ > JUNIT\_WORKSPACE 中创建一个 java 类文件，名称为 TestJunit。

**import** org.junit.Test;

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

**public** **class** TestJunit {

@Test

**public** **void** testAdd() {

String str= "Junit is working fine";

assertEquals("Junit is working fine",str);

}

}

在 C:\ > JUNIT\_WORKSPACE 中创建一个 java 类文件，名称为TestRunner，来执行测试用例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

**步骤7：验证结果**

利用 javac 编译器按照以下方式编写类。

C:\JUNIT\_WORKSPACE>javac TestJunit.java TestRunner.java

现在运行 Test Runner 来看结果。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出。

ture

# JUnit - 测试框架

* **什么是 Junit 测试框架**

JUnit 是一个回归测试框架，被开发者用于实施对应用程序的单元测试，加快程序编制速度，同时提高编码的质量。JUnit 测试框架能够轻松完成以下任意两种结合：

1. Eclipse 集成开发环境
2. Ant 打包工具
3. Maven 项目构建管理

**特性**

JUnit 测试框架具有以下重要特性：

1. 测试工具
2. 测试套件
3. 测试运行器
4. 测试分类

**测试工具**

测试工具是一整套固定的工具用于基线测试。测试工具的目的是为了确保测试能够在共享且固定的环境中运行，因此保证测试结果的可重复性。它包括：

1. 在所有测试调用指令发起前的 setUp() 方法。
2. 在测试方法运行后的 tearDown() 方法。

让我们来看一个例子：

**import** junit.framework.\*;

**public** **class** JavaTest **extends** TestCase {

**protected** **int** value1, value2;

// assigning the values

**protected** **void** setUp(){

value1=3;

value2=3;

}

// test method to add two values

**public** **void** testAdd(){

**double** result= value1 + value2;

assertTrue(result == 6);

}

}

* **测试套件**

测试套件意味捆绑几个测试案例并且同时运行。在 JUnit 中，@RunWith 和 @Suite 都被用作运行测试套件。以下为使用 TestJunit1 和 TestJunit2 的测试分类：

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Suite;

//JUnit Suite Test

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

TestJunit1.**class** ,TestJunit2.**class**

})

**public** **class** JunitTestSuite {

}

**import** org.junit.Test;

**import** org.junit.Ignore;

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

**public** **class** TestJunit1 {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testPrintMessage() {

System.out.println("Inside testPrintMessage()");

assertEquals(message, messageUtil.printMessage());

}

}

**import** org.junit.Test;

**import** org.junit.Ignore;

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

**public** **class** TestJunit2 {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testSalutationMessage() {

System.***out***.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

* **测试运行器**

测试运行器 用于执行测试案例。以下为假定测试分类成立的情况下的例子：

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

* **JUnit 测试分类**

测试分类是在编写和测试 JUnit 的重要分类。几种重要的分类如下：

1. 包含一套断言方法的测试断言
2. 包含规定运行多重测试工具的测试用例
3. 包含收集执行测试用例结果的方法的测试结果

# JUnit - 基本用法

* **创建一个类**

在C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 MessageUtil.java 的类用来测试。

/\*

\* This class prints the given message on console.

\*/

**public** **class** MessageUtil {

**private** String message;

//Constructor

//@param message to be printed

**public** MessageUtil(String message){

**this**.message = message;

}

// prints the message

**public** String printMessage(){

System.***out***.println(message);

**return** message;

}

}

* **创建 Test Case 类**

1. 创建一个名为 TestJunit.java 的测试类。
2. 向测试类中添加名为 testPrintMessage() 的方法。
3. 向方法中添加 Annotaion @Test。
4. 执行测试条件并且应用 Junit 的 assertEquals API 来检查。

在C:\ > JUNIT\_WORKSPACE路径下创建一个文件名为 TestJunit.java 的类

**import** org.junit.Test;

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

**public** **class** TestJunit {

String message = "Hello World";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testPrintMessage() {

assertEquals(message,messageUtil.printMessage());

}

}

* **创建 Test Runner 类**

1. 创建一个 TestRunner 类
2. 运用 JUnit 的 JUnitCore 类的 runClasses 方法来运行上述测试类的测试案例
3. 获取在 Result Object 中运行的测试案例的结果
4. 获取 Result Object 的 getFailures() 方法中的失败结果
5. 获取 Result object 的 wasSuccessful() 方法中的成功结果

在 C:\ > JUNIT\_WORKSPACE 路径下创建一个文件名为 TestRunner.java 的类来执行测试案例

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 MessageUtil、Test case 和 Test Runner 类。

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java

现在运行 Test Runner,它可以运行在所提供的 Test Case 类中定义的测试案例。

C:\JUNIT\_WORKSPACE>java TestRunner

检查运行结果

Hello World

true

现在更新 C:\ > JUNIT\_WORKSPACE 路径下的 TestJunit，并且检测失败。改变消息字符串。

**import** org.junit.Test;

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

**public** **class** TestJunit {

String message = "Hello World";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testPrintMessage() {

message = "New Word";

assertEquals(message,messageUtil.printMessage());

}

}

让我们保持其他类不变，再次尝试运行相同的 Test Runner

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

现在运行在 Test Case 类中提供的即将运行测试案例的 Test Runner

C:\JUNIT\_WORKSPACE>java TestRunner

检查运行结果

Hello World

testPrintMessage(TestJunit): expected:<[New Wor]d> but was:<[Hello Worl]d>

false

# JUnit – API

JUnit 中的最重要的程序包是 junit.framework 它包含了所有的核心类。一些重要的类列示如下：

|  |  |  |
| --- | --- | --- |
| 序号 | 类的名称 | 类的功能 |
| 1 | Assert | assert 方法的集合 |
| 2 | TestCase | 一个定义了运行多重测试的固定装置 |
| 3 | TestResult | TestResult 集合了执行测试样例的所有结果 |
| 4 | TestSuite | TestSuite 是测试的集合 |

* **Assert 类**

**public** **class** Assert **extends** java.lang.Object

这个类提供了一系列的编写测试的有用的声明方法。只有失败的声明方法才会被记录。Assert 类的重要方法列式如下：

|  |  |
| --- | --- |
| 序号 | 方法和描述 |
| 1 | void assertEquals(boolean expected, boolean actual)  检查两个变量或者等式是否平衡 |
| 2 | void assertFalse(boolean condition)  检查条件是假的 |
| 3 | void assertNotNull(Object object)  检查对象不是空的 |
| 4 | void assertNull(Object object)  检查对象是空的 |
| 5 | void assertTrue(boolean condition)  检查条件为真 |
| 6 | void fail()  在没有报告的情况下使测试不通过 |

下面让我们在例子中来测试一下上面提到的一些方法。在 C:\ > JUNIT\_WORKSPACE 目录下创建一个名为 TestJunit1.java 的类。

**import** org.junit.Test;

**import** **static** org.junit.Assert.\*;

**public** **class** TestJunit1 {

@Test

**public** **void** testAdd() {

//test data

**int** num= 5;

String temp= **null**;

String str= "Junit is working fine";

//check for equality

assertEquals("Junit is working fine", str);

//check for false condition

assertFalse(num > 6);

//check for not null value

assertNotNull(str);

}

}

接下来，我们在 C:\ > JUNIT\_WORKSPACE 目录下创建一个文件名为 TestRunner1.java 的类来执行测试案例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner1 {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit1.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 Test case 和 Test Runner 类

C:\JUNIT\_WORKSPACE>javac TestJunit1.java TestRunner1.java

现在运行 Test Runner 它将运行在 Test Case 类中定义并提供的测试样例。

C:\JUNIT\_WORKSPACE>java TestRunner1

检查输出结果。

true

* **TestCase 类**

下面介绍的是 org.junit.TestCaset 类：

**public** **abstract** **class** TestCase **extends** Assert **implements** Test

测试样例定义了运行多重测试的固定格式。TestCase 类的一些重要方法列式如下：

|  |  |
| --- | --- |
| 序号 | 方法和描述 |
| 1 | int countTestCases()  为被run(TestResult result) 执行的测试案例计数 |
| 2 | TestResult createResult()  创建一个默认的 TestResult 对象 |
| 3 | String getName()  获取 TestCase 的名称 |
| 4 | TestResult run()  一个运行这个测试的方便的方法，收集由TestResult 对象产生的结果 |
| 5 | void run(TestResult result)  在 TestResult 中运行测试案例并收集结果 |
| 6 | void setName(String name)  设置 TestCase 的名称 |
| 7 | void setUp()  创建固定装置，例如，打开一个网络连接 |
| 8 | void tearDown()  拆除固定装置，例如，关闭一个网络连接 |
| 9 | String toString()  返回测试案例的一个字符串表示 |

我们在例子中尝试一下上文提到的方法。在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为TestJunit2.java 的类。

**import** junit.framework.TestCase;

**import** org.junit.Before;

**import** org.junit.Test;

**public** **class** TestJunit2 **extends** TestCase {

**protected** **double** fValue1;

**protected** **double** fValue2;

@Before

**public** **void** setUp() {

fValue1= 2.0;

fValue2= 3.0;

}

@Test

**public** **void** testAdd() {

//count the number of test cases

System.***out***.println("No of Test Case = "+ **this**.countTestCases());

//test getName

String name= **this**.getName();

System.***out***.println("Test Case Name = "+ name);

//test setName

**this**.setName("testNewAdd");

String newName= **this**.getName();

System.***out***.println("Updated Test Case Name = "+ newName);

}

//tearDown used to close the connection or clean up activities

**public** **void** tearDown( ) {

}

}

接下来，在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 TestRunner2.java 的类来执行测试案例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner2 {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit2.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 Test case 和 Test Runner 类

C:\JUNIT\_WORKSPACE>javac TestJunit2.java TestRunner2.java

现在运行 Test Runner 它将运行在 Test Case 类中定义并提供的测试样例。

C:\JUNIT\_WORKSPACE>java TestRunner2

检查输出结果

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

true

* **TestResult 类**

下面定义的是 org.junit.TestResult 类：

**public** **class** TestResult **extends** Object

TestResult 类收集所有执行测试案例的结果。它是收集参数层面的一个实例。这个实验框架区分失败和错误。失败是可以预料的并且可以通过假设来检查。错误是不可预料的问题就像 ArrayIndexOutOfBoundsException。TestResult 类的一些重要方法列式如下：

|  |  |
| --- | --- |
| 序号 | 方法和描述 |
| 1 | void addError(Test test, Throwable t)  在错误列表中加入一个错误 |
| 2 | void addFailure(Test test, AssertionFailedError t)  在失败列表中加入一个失败 |
| 3 | void endTest(Test test)  显示测试被编译的这个结果 |
| 4 | int errorCount()  获取被检测出错误的数量 |
| 5 | Enumeration errors()  返回错误的详细信息 |
| 6 | int failureCount()  获取被检测出的失败的数量 |
| 7 | void run(TestCase test) 运行 TestCase |
| 8 | int int runCount()  获得运行测试的数量 |
| 9 | void startTest(Test test)  声明一个测试即将开始 |
| 10 | void stop()  标明测试必须停止 |

在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 TestJunit3.java 的类。

**import** org.junit.Test;

**import** junit.framework.AssertionFailedError;

**import** junit.framework.TestResult;

**public** **class** TestJunit3 **extends** TestResult {

// add the error

**public** **synchronized** **void** addError(Test test, Throwable t) {

**super**.addError((junit.framework.Test) test, t);

}

// add the failure

**public** **synchronized** **void** addFailure(Test test, AssertionFailedError t) {

**super**.addFailure((junit.framework.Test) test, t);

}

@Test

**public** **void** testAdd() {

// add any test

}

// Marks that the test run should stop.

**public** **synchronized** **void** stop() {

//stop the test here

}

}

接下来，在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 TestRunner3.java 的类来执行测试案例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner3 {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit3.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 Test case 和 Test Runner 类

C:\JUNIT\_WORKSPACE>javac TestJunit3.java TestRunner3.java

现在运行 Test Runner 它将运行在 Test Case 类中定义并提供的测试样例。

C:\JUNIT\_WORKSPACE>java TestRunner3

检查输出结果。

true

* **TestSuite 类**

下面定义的是 org.junit.TestSuite 类：

**public** **class** TestSuite **extends** Object **implements** Test

TestSuite 类是测试的组成部分。它运行了很多的测试案例。TestSuite 类的一些重要方法列式如下：

|  |  |
| --- | --- |
| 序号 | 方法和描述 |
| 1 | void addTest(Test test)  在套中加入测试。 |
| 2 | void addTestSuite(Class<? extends TestCase> testClass)  将已经给定的类中的测试加到套中。 |
| 3 | int countTestCases()  对这个测试即将运行的测试案例进行计数。 |
| 4 | String getName()  返回套的名称。 |
| 5 | void run(TestResult result)  在 TestResult 中运行测试并收集结果。 |
| 6 | void setName(String name)  设置套的名称。 |
| 7 | Test testAt(int index)  在给定的目录中返回测试。 |
| 8 | int testCount()  返回套中测试的数量。 |
| 9 | static Test warning(String message)  返回会失败的测试并且记录警告信息。 |

在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 JunitTestSuite.java 的类。

**import** junit.framework.\*;

**public** **class** JunitTestSuite {

**public** **static** **void** main(String[] a) {

// add the test's in the suite

TestSuite suite = **new** TestSuite(TestJunit1.**class**, TestJunit2.**class**, TestJunit3.**class** );

TestResult result = **new** TestResult();

suite.run(result);

System.***out***.println("Number of test cases = " + result.runCount());

}

}

用 javac 编译 Test suit

C:\JUNIT\_WORKSPACE>javac JunitTestSuite.java

现在运行 Test Suit

C:\JUNIT\_WORKSPACE>java JunitTestSuite

检查输出结果。

No of Test Case = 1

Test Case Name = testAdd

Updated Test Case Name = testNewAdd

Number of test cases = 3

# JUnit - 编写测试

在这里你将会看到一个应用 POJO 类，Business logic 类和在 test runner 中运行的 test 类的 JUnit 测试的例子。

在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 EmployeeDetails.java 的 POJO 类。

**public** **class** EmployeeDetails {

**private** String name;

**private** **double** monthlySalary;

**private** **int** age;

/\*\*

\* **@return** the name

\*/

**public** String getName() {

**return** name;

}

/\*\*

\* **@param** name the name to set

\*/

**public** **void** setName(String name) {

**this**.name = name;

}

/\*\*

\* **@return** the monthlySalary

\*/

**public** **double** getMonthlySalary() {

**return** monthlySalary;

}

/\*\*

\* **@param** monthlySalary the monthlySalary to set

\*/

**public** **void** setMonthlySalary(**double** monthlySalary) {

**this**.monthlySalary = monthlySalary;

}

/\*\*

\* **@return** the age

\*/

**public** **int** getAge() {

**return** age;

}

/\*\*

\* **@param** age the age to set

\*/

**public** **void** setAge(**int** age) {

**this**.age = age;

}

}

EmployeeDetails 类被用于

1. 取得或者设置雇员的姓名的值
2. 取得或者设置雇员的每月薪水的值
3. 取得或者设置雇员的年龄的值

在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 EmpBusinessLogic.java 的 business logic 类

**public** **class** EmpBusinessLogic {

// Calculate the yearly salary of employee

**public** **double** calculateYearlySalary(EmployeeDetails employeeDetails){

**double** yearlySalary=0;

yearlySalary = employeeDetails.getMonthlySalary() \* 12;

**return** yearlySalary;

}

// Calculate the appraisal amount of employee

**public** **double** calculateAppraisal(EmployeeDetails employeeDetails){

**double** appraisal=0;

**if**(employeeDetails.getMonthlySalary() < 10000){

appraisal = 500;

}**else**{

appraisal = 1000;

}

**return** appraisal;

}

}

EmpBusinessLogic 类被用来计算

1. 雇员每年的薪水
2. 雇员的评估金额

在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 TestEmployeeDetails.java 的准备被测试的测试案例类

**import** org.junit.Test;

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

**public** **class** TestEmployeeDetails {

EmpBusinessLogic empBusinessLogic =**new** EmpBusinessLogic();

EmployeeDetails employee = **new** EmployeeDetails();

//test to check appraisal

@Test

**public** **void** testCalculateAppriasal() {

employee.setName("Rajeev");

employee.setAge(25);

employee.setMonthlySalary(8000);

**double** appraisal= empBusinessLogic.calculateAppraisal(employee);

assertEquals(500, appraisal, 0.0);

}

// test to check yearly salary

@Test

**public** **void** testCalculateYearlySalary() {

employee.setName("Rajeev");

employee.setAge(25);

employee.setMonthlySalary(8000);

**double** salary= empBusinessLogic.calculateYearlySalary(employee);

assertEquals(96000, salary, 0.0);

}

}

TestEmployeeDetails 是用来测试 EmpBusinessLogic 类的方法的，它

1. 测试雇员的每年的薪水
2. 测试雇员的评估金额

现在让我们在 C:\ > JUNIT\_WORKSPACE 路径下创建一个名为 TestRunner.java 的类来执行测试案例类

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestEmployeeDetails.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用javac编译 Test case 和 Test Runner 类

C:\JUNIT\_WORKSPACE>javac EmployeeDetails.java

EmpBusinessLogic.java TestEmployeeDetails.java TestRunner.java

现在运行将会运行 Test Case 类中定义和提供的测试案例的 Test Runner

C:\JUNIT\_WORKSPACE>java TestRunner

检查运行结果

true

# JUnit - 使用断言

* **断言**

所有的断言都包含在 Assert 类中

**public** **class** Assert **extends** java.lang.Object

这个类提供了很多有用的断言方法来编写测试用例。只有失败的断言才会被记录。Assert 类中的一些有用的方法列式如下：

|  |  |
| --- | --- |
| 序号 | 方法和描述 |
| 1 | void assertEquals(boolean expected, boolean actual)  检查两个变量或者等式是否平衡 |
| 2 | void assertTrue(boolean expected, boolean actual)  检查条件为真 |
| 3 | void assertFalse(boolean condition)  检查条件为假 |
| 4 | void assertNotNull(Object object)  检查对象不为空 |
| 5 | void assertNull(Object object)  检查对象为空 |
| 6 | void assertSame(boolean condition)  assertSame() 方法检查两个相关对象是否指向同一个对象 |
| 7 | void assertNotSame(boolean condition)  assertNotSame() 方法检查两个相关对象是否不指向同一个对象 |
| 8 | void assertArrayEquals(expectedArray, resultArray)  assertArrayEquals() 方法检查两个数组是否相等 |

下面我们在例子中试验一下上面提到的各种方法。在 C:\ > JUNIT\_WORKSPACE 路径下创建一个文件名为 TestAssertions.java 的类

**import** org.junit.Test;

**import** **static** org.junit.Assert.\*;

**public** **class** TestAssertions {

@Test

**public** **void** testAssertions() {

//test data

String str1 = **new** String ("abc");

String str2 = **new** String ("abc");

String str3 = **null**;

String str4 = "abc";

String str5 = "abc";

**int** val1 = 5;

**int** val2 = 6;

String[] expectedArray = {"one", "two", "three"};

String[] resultArray = {"one", "two", "three"};

//Check that two objects are equal

assertEquals(str1, str2);

//Check that a condition is true

assertTrue (val1 < val2);

//Check that a condition is false

assertFalse(val1 > val2);

//Check that an object isn't null

assertNotNull(str1);

//Check that an object is null

assertNull(str3);

//Check if two object references point to the same object

assertSame(str4,str5);

//Check if two object references not point to the same object

assertNotSame(str1,str3);

//Check whether two arrays are equal to each other.

assertArrayEquals(expectedArray, resultArray);

}

}

接下来，我们在 C:\ > JUNIT\_WORKSPACE 路径下创建一个文件名为 TestRunner.java 的类来执行测试用例

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner2 {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestAssertions.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 Test case 和 Test Runner 类

C:\JUNIT\_WORKSPACE>javac TestAssertions.java TestRunner.java

现在运行将会运行 Test Case 类中定义和提供的测试案例的 Test Runner

C:\JUNIT\_WORKSPACE>java TestRunner

检查运行结果

true

* **注释**

注释就好像你可以在你的代码中添加并且在方法或者类中应用的元标签。JUnit 中的这些注释为我们提供了测试方法的相关信息，哪些方法将会在测试方法前后应用，哪些方法将会在所有方法前后应用，哪些方法将会在执行中被忽略。

JUnit 中的注释的列表以及他们的含义：

|  |  |
| --- | --- |
| 序号 | 注释和描述 |
| 1 | @Test  这个注释说明依附在 JUnit 的 public void 方法可以作为一个测试案例。 |
| 2 | @Before  有些测试在运行前需要创造几个相似的对象。在 public void 方法加该注释是因为该方法需要在 test 方法前运行。 |
| 3 | @After  如果你将外部资源在 Before 方法中分配，那么你需要在测试运行后释放他们。在 public void 方法加该注释是因为该方法需要在 test 方法后运行。 |
| 4 | @BeforeClass  在 public void 方法加该注释是因为该方法需要在类中所有方法前运行。 |
| 5 | @AfterClass  它将会使方法在所有测试结束后执行。这个可以用来进行清理活动。 |
| 6 | @Ignore  这个注释是用来忽略有关不需要执行的测试的。 |

在 C:\ > JUNIT\_WORKSPACE 路径下创建一个文件名为 JunitAnnotation.java 的类来测试注释

**import** org.junit.After;

**import** org.junit.AfterClass;

**import** org.junit.Before;

**import** org.junit.BeforeClass;

**import** org.junit.Ignore;

**import** org.junit.Test;

**public** **class** JunitAnnotation {

//execute before class

@BeforeClass

**public** **static** **void** beforeClass() {

System.***out***.println("in before class");

}

//execute after class

@AfterClass

**public** **static** **void** afterClass() {

System.***out***.println("in after class");

}

//execute before test

@Before

**public** **void** before() {

System.***out***.println("in before");

}

//execute after test

@After

**public** **void** after() {

System.***out***.println("in after");

}

//test case

@Test

**public** **void** test() {

System.***out***.println("in test");

}

//test case ignore and will not execute

@Ignore

**public** **void** ignoreTest() {

System.***out***.println("in ignore test");

}

}

接下来，我们在 C:\ > JUNIT\_WORKSPACE 路径下创建一个文件名为 TestRunner.java 的类来执行注释

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(JunitAnnotation.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 Test case 和 Test Runner 类

C:\JUNIT\_WORKSPACE>javac TestAssertions.java TestRunner.java

现在运行将会运行 Test Case 类中定义和提供的测试案例的 Test Runner

C:\JUNIT\_WORKSPACE>java TestRunner

检查运行结果

in before class

in before

in test

in after

in after class

true

# JUnit - 执行过程

本教程阐明了 JUnit 中的方法执行过程，即哪一个方法首先被调用，哪一个方法在一个方法之后调用。以下为 JUnit 测试方法的 API，并且会用例子来说明。

在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 类文件命名为 JunitAnnotation.java 来测试注释程序。

**import** org.junit.After;

**import** org.junit.AfterClass;

**import** org.junit.Before;

**import** org.junit.BeforeClass;

**import** org.junit.Ignore;

**import** org.junit.Test;

**public** **class** ExecutionProcedureJunit {

//execute only once, in the starting

@BeforeClass

**public** **static** **void** beforeClass() {

System.***out***.println("in before class");

}

//execute only once, in the end

@AfterClass

**public** **static** **void** afterClass() {

System.***out***.println("in after class");

}

//execute for each test, before executing test

@Before

**public** **void** before() {

System.***out***.println("in before");

}

//execute for each test, after executing test

@After

**public** **void** after() {

System.***out***.println("in after");

}

//test case 1

@Test

**public** **void** testCase1() {

System.***out***.println("in test case 1");

}

//test case 2

@Test

**public** **void** testCase2() {

System.***out***.println("in test case 2");

}

}

接下来，让我们在目录 C:\ > JUNIT\_WORKSPACE 中创建一个 java 类文件 TestRunner.java 来执行注释程序。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(ExecutionProcedureJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

使用 javac 命令来编译 Test case 和 Test Runner 类。

C:\JUNIT\_WORKSPACE>javac ExecutionProcedureJunit.java TestRunner.java

现在运行 Test Runner 它会自动运行定义在 Test Case 类中的测试样例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出

in before class

in before

in test case 1

in after

in before

in test case 2

in after

in after class

观察以上的输出，这是 JUnite 执行过程：

1. beforeClass() 方法首先执行，并且只执行一次。
2. afterClass() 方法最后执行，并且只执行一次。
3. before() 方法针对每一个测试用例执行，但是是在执行测试用例之前。
4. after() 方法针对每一个测试用例执行，但是是在执行测试用例之后。
5. 在 before() 方法和 after() 方法之间，执行每一个测试用例。

# JUnit - 执行测试

测试用例是使用 JUnitCore 类来执行的。JUnitCore 是运行测试的外观类。它支持运行 JUnit 4 测试, JUnit 3.8.x 测试,或者他们的混合。 要从命令行运行测试，可以运行 java org.junit.runner.JUnitCore 。对于只有一次的测试运行，可以使用静态方法 runClasses(Class[])。

下面是 org.junit.runner.JUnitCore 类的声明：

**public** **class** JUnitCore **extends** java.lang.Object

* **创建一个类**

在目录 C:\ > JUNIT\_WORKSPACE 中创建一个被测试的 Java 类命名为 MessageUtil.java。

/\*

\* This class prints the given message on console.

\*/

**public** **class** MessageUtil {

**private** String message;

//Constructor

//@param message to be printed

**public** MessageUtil(String message){

**this**.message = message;

}

// prints the message

**public** String printMessage(){

System.***out***.println(message);

**return** message;

}

}

* **创建测试用例类**

1. 创建一个 java 测试类叫做 TestJunit.java。
2. 在类中加入一个测试方法 testPrintMessage()。
3. 在方法 testPrintMessage() 中加入注释 @Test。
4. 实现测试条件并且用 Junit 的 assertEquals API 检查测试条件。

在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 类文件命名为 TestJunit.java

**import** org.junit.Test;

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

**public** **class** TestJunit {

String message = "Hello World";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testPrintMessage() {

assertEquals(message,messageUtil.printMessage());

}

}

* **创建 TestRunner 类**

接下来，让我们在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 类文件命名为 TestRunner.java 来执行测试用例，导出 JUnitCore 类并且使用 runClasses() 方法，将测试类名称作为参数。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

使用 javac 命令来编译 Test case 和 Test Runner 类。

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java

现在运行 Test Runner 它会自动运行定义在 Test Case 类中的测试样例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出

Hello World

true

# JUnit - 套件测试

* **测试套件**

测试套件意味着捆绑几个单元测试用例并且一起执行他们。在 JUnit 中，@RunWith 和 @Suite 注释用来运行套件测试。这个教程将向您展示一个例子，其中含有两个测试样例 TestJunit1 & TestJunit2 类，我们将使用测试套件一起运行他们。

* **创建一个类**

在目录 C:\ > JUNIT\_WORKSPACE 中创建一个被测试的 java 类命名为 MessageUtil.java

/\*

\* This class prints the given message on console.

\*/

**public** **class** MessageUtil {

**private** String message;

//Constructor

//@param message to be printed

**public** MessageUtil(String message){

**this**.message = message;

}

// prints the message

**public** String printMessage(){

System.***out***.println(message);

**return** message;

}

// add "Hi!" to the message

**public** String salutationMessage(){

message = "Hi!" + message;

System.***out***.println(message);

**return** message;

}

}

* **创建 Test Case 类**

在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 测试类叫做 TestJunit1.java。

**import** org.junit.Test;

**import** org.junit.Ignore;

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

**public** **class** TestJunit1 {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testPrintMessage() {

System.***out***.println("Inside testPrintMessage()");

assertEquals(message, messageUtil.printMessage());

}

}

在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 测试类叫做 TestJunit2.java。

**import** org.junit.Test;

**import** org.junit.Ignore;

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

**public** **class** TestJunit2 {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testSalutationMessage() {

System.***out***.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

* **使用 Test Suite 类**

1. 创建一个 java 类。
2. 在类中附上 @RunWith(Suite.class) 注释。
3. 使用 @Suite.SuiteClasses 注释给 JUnit 测试类加上引用。

在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 类文件叫做 TestSuite.java 来执行测试用例。

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Suite;

@RunWith(Suite.**class**)

@Suite.SuiteClasses({

TestJunit1.**class**,

TestJunit2.**class**

})

**public** **class** JunitTestSuite {

}

* **创建 Test Runner 类**

在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 类文件叫做 TestRunner.java 来执行测试用例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(JunitTestSuite.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

使用 javac 命令编译所有的 java 类

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit1.java

TestJunit2.java JunitTestSuite.java TestRunner.java

现在运行 Test Runner，即运行所有的在之前 Test Case 类中定义的测试用例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出

Inside testPrintMessage()

Robert

Inside testSalutationMessage()

Hi Robert

true

# JUnit - 忽略测试

有时可能会发生我们的代码还没有准备好的情况，这时测试用例去测试这个方法或代码的时候会造成失败。@Ignore 注释会在这种情况时帮助我们。

1. 一个含有 @Ignore 注释的测试方法将不会被执行。
2. 如果一个测试类有 @Ignore 注释，则它的测试方法将不会执行。

现在我们用例子来学习 @Ignore。

* **创建一个类**

在目录 C:\ > JUNIT\_WORKSPACE 中创建一个将被测试的 java 类命名为 MessageUtil.java。

/\*

\* This class prints the given message on console.

\*/

**public** **class** MessageUtil {

**private** String message;

//Constructor

//@param message to be printed

**public** MessageUtil(String message){

**this**.message = message;

}

// prints the message

**public** String printMessage(){

System.***out***.println(message);

**return** message;

}

// add "Hi!" to the message

**public** String salutationMessage(){

message = "Hi!" + message;

System.***out***.println(message);

**return** message;

}

}

* **创建 Test Case 类**

1. 创建 java 测试类命名为 TestJunit.java。
2. 在类中加入测试方法 testPrintMessage() 和 testSalutationMessage()。
3. 在方法 testPrintMessage() 中加入 @Ignore 注释。

在目录 C:\ > JUNIT\_WORKSPACE 中创建一个 java 类文件命名为 TestJunit.java

**import** org.junit.Test;

**import** org.junit.Ignore;

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

**public** **class** TestJunit {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Ignore

@Test

**public** **void** testPrintMessage() {

System.***out***.println("Inside testPrintMessage()");

message = "Robert";

assertEquals(message,messageUtil.printMessage());

}

@Test

**public** **void** testSalutationMessage() {

System.***out***.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

* **创建 Test Runner 类**

在目录 C:\ > JUNIT\_WORKSPACE 创建一个 java 类文件叫做 TestRunner.java 来执行测试用例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

使用 javac 命令编译 MessageUtil, Test case 和 Test Runner 类。

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java

现在运行 Test Runner 类，即不会运行在 Test Case 类中定义的 testPrintMessage() 测试用例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出。testPrintMessage() 测试用例并没有被测试。

Inside testSalutationMessage()

Hi!Robert

true

现在更新在目录 C:\ > JUNIT\_WORKSPACE 中的 TestJunit 在类级别上使用 @Ignore 来忽略所有的测试用例

**import** org.junit.Test;

**import** org.junit.Ignore;

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

@Ignore

**public** **class** TestJunit {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

**public** **void** testPrintMessage() {

System.***out***.println("Inside testPrintMessage()");

message = "Robert";

assertEquals(message,messageUtil.printMessage());

}

@Test

**public** **void** testSalutationMessage() {

System.***out***.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

使用 javac 命令编译 Test case

C:\JUNIT\_WORKSPACE>javac TestJunit.java

保持你的 Test Runner 不被改变，如下：

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

现在运行 Test Runner 即不会运行在 Test Case 类中定义的任何一个测试样例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出。没有测试用例被测试。

true

# JUnit - 时间测试

Junit 提供了一个暂停的方便选项。如果一个测试用例比起指定的毫秒数花费了更多的时间，那么 Junit 将自动将它标记为失败。timeout 参数和 @Test 注释一起使用。现在让我们看看活动中的 @test(timeout)。

* **创建一个类**

创建一个在 C:\JUNIT\_WORKSPACE 中叫做 MessageUtil.java 的 java 类来测试。

在 printMessage() 方法内添加一个无限 while 循环。

/\*

\* This class prints the given message on console.

\*/

**public** **class** MessageUtil {

**private** String message;

//Constructor

//@param message to be printed

**public** MessageUtil(String message){

**this**.message = message;

}

// prints the message

**public** **void** printMessage(){

System.***out***.println(message);

**while**(**true**);

}

// add "Hi!" to the message

**public** String salutationMessage(){

message = "Hi!" + message;

System.***out***.println(message);

**return** message;

}

}

* **创建 Test Case 类**

1. 创建一个叫做 TestJunit.java 的 java 测试类。
2. 给 testPrintMessage() 测试用例添加 1000 的暂停时间。

在 C:\JUNIT\_WORKSPACE 中创建一个文件名为 TestJunit.java 的 java 类。

**import** org.junit.Test;

**import** org.junit.Ignore;

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

**public** **class** TestJunit {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test(timeout=1000)

**public** **void** testPrintMessage() {

System.***out***.println("Inside testPrintMessage()");

messageUtil.printMessage();

}

@Test

**public** **void** testSalutationMessage() {

System.***out***.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

* **创建 Test Runner 类**

在 C:\JUNIT\_WORKSPACE 中创建一个文件名为 TestRunner.java 的 java 类来执行测试样例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 MessageUtil，Test case 和 Test Runner 类。

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java

现在运行 Test Runner，它将运行由提供的 Test Case 类中所定义的测试用例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出。testPrintMessage() 测试用例将标记单元测试失败。

Inside testPrintMessage()

Robert

Inside testSalutationMessage()

Hi!Robert

testPrintMessage(TestJunit): test timed out after 1000 milliseconds

false

# JUnit - 异常测试

Junit 用代码处理提供了一个追踪异常的选项。你可以测试代码是否它抛出了想要得到的异常。expected 参数和 @Test 注释一起使用。现在让我们看看活动中的 @Test(expected)。

* **创建一个类**

在 C:\ > JUNIT\_WORKSPACE 中创建一个叫做 MessageUtil.java 的 java 类来测试。

在 printMessage()方法中添加一个错误条件。

/\*

\* This class prints the given message on console.

\*/

**public** **class** MessageUtil {

**private** String message;

//Constructor

//@param message to be printed

**public** MessageUtil(String message){

**this**.message = message;

}

// prints the message

**public** **void** printMessage(){

System.***out***.println(message);

**int** a =0;

**int** b = 1/a;

}

// add "Hi!" to the message

**public** String salutationMessage(){

message = "Hi!" + message;

System.***out***.println(message);

**return** message;

}

}

* **创建 Test Case 类**

1. 创建一个叫做 TestJunit.java 的 java 测试类。
2. 给 testPrintMessage() 测试用例添加需要的异常 ArithmeticException。

在 C:> JUNIT\_WORKSPACE 中创建一个文件名为 TestJunit.java 的 java 类

**import** org.junit.Test;

**import** org.junit.Ignore;

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

**public** **class** TestJunit {

String message = "Robert";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test(expected = ArithmeticException.**class**)

**public** **void** testPrintMessage() {

System.***out***.println("Inside testPrintMessage()");

messageUtil.printMessage();

}

@Test

**public** **void** testSalutationMessage() {

System.***out***.println("Inside testSalutationMessage()");

message = "Hi!" + "Robert";

assertEquals(message,messageUtil.salutationMessage());

}

}

* **创建 TestRunner 类**

在 C:> JUNIT\_WORKSPACE 中创建一个文件名为 TestJunit.java 的 java 类来执行测试用例。

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(TestJunit.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 MessageUtil，Test case 和 Test Runner 类。

C:\JUNIT\_WORKSPACE>javac MessageUtil.java TestJunit.java TestRunner.java

现在运行 Test Runner，它将运行由提供的 Test Case 类中所定义的测试用例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出。testPrintMessage() 测试用例将通过。

Inside testPrintMessage()

Robert

Inside testSalutationMessage()

Hi!Robert

true

# JUnit - 参数化测试

Junit 4 引入了一个新的功能参数化测试。参数化测试允许开发人员使用不同的值反复运行同一个测试。你将遵循 5 个步骤来创建参数化测试。

1. 用 @RunWith(Parameterized.class) 来注释 test 类。
2. 创建一个由 @Parameters 注释的公共的静态方法，它返回一个对象的集合(数组)来作为测试数据集合。
3. 创建一个公共的构造函数，它接受和一行测试数据相等同的东西。
4. 为每一列测试数据创建一个实例变量。
5. 用实例变量作为测试数据的来源来创建你的测试用例。

一旦每一行数据出现测试用例将被调用。让我们看看活动中的参数化测试。

* **创建一个类**

在 C:\ > JUNIT\_WORKSPACE 创建一个叫做 PrimeNumberChecker.java 的 java 类来测试。

**public** **class** PrimeNumberChecker {

**public** Boolean validate(**final** Integer primeNumber) {

**for** (**int** i = 2; i < (primeNumber / 2); i++) {

**if** (primeNumber % i == 0) {

**return** **false**;

}

}

**return** **true**;

}

}

* **创建 Parameterized Test Case 类**

1. 创建一个叫做 PrimeNumberCheckerTest.java 的 java 类。

在 C:> JUNIT\_WORKSPACE 中创建一个文件名为 PrimeNumberCheckerTest.java 的 java 类。

**import** java.util.Arrays;

**import** java.util.Collection;

**import** org.junit.Test;

**import** org.junit.Before;

**import** org.junit.runners.Parameterized;

**import** org.junit.runners.Parameterized.Parameters;

**import** org.junit.runner.RunWith;

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

@RunWith(Parameterized.**class**)

**public** **class** PrimeNumberCheckerTest {

**private** Integer inputNumber;

**private** Boolean expectedResult;

**private** PrimeNumberChecker primeNumberChecker;

@Before

**public** **void** initialize() {

primeNumberChecker = **new** PrimeNumberChecker();

}

// Each parameter should be placed as an argument here

// Every time runner triggers, it will pass the arguments

// from parameters we defined in primeNumbers() method

**public** PrimeNumberCheckerTest(Integer inputNumber,

Boolean expectedResult) {

**this**.inputNumber = inputNumber;

**this**.expectedResult = expectedResult;

}

@Parameterized.Parameters

**public** **static** Collection primeNumbers() {

**return** Arrays.*asList*(**new** Object[][] {

{ 2, **true** },

{ 6, **false** },

{ 19, **true** },

{ 22, **false** },

{ 23, **true** }

});

}

// This test will run 4 times since we have 5 parameters defined

@Test

**public** **void** testPrimeNumberChecker() {

System.***out***.println("Parameterized Number is : " + inputNumber);

assertEquals(expectedResult,

primeNumberChecker.validate(inputNumber));

}

}

* **创建 TestRunner 类**

在 C:> JUNIT\_WORKSPACE 中创建一个文件名为 TestRunner.java 的 java 类来执行测试用例

**import** org.junit.runner.JUnitCore;

**import** org.junit.runner.Result;

**import** org.junit.runner.notification.Failure;

**public** **class** TestRunner {

**public** **static** **void** main(String[] args) {

Result result = JUnitCore.runClasses(PrimeNumberCheckerTest.**class**);

**for** (Failure failure : result.getFailures()) {

System.***out***.println(failure.toString());

}

System.***out***.println(result.wasSuccessful());

}

}

用 javac 编译 PrimeNumberChecker，PrimeNumberCheckerTest 和 TestRunner 类。

C:\JUNIT\_WORKSPACE>javac PrimeNumberChecker.java PrimeNumberCheckerTest.java

TestRunner.java

现在运行 TestRunner，它将运行由提供的 Test Case 类中所定义的测试用例。

C:\JUNIT\_WORKSPACE>java TestRunner

验证输出。

Parameterized Number is : 2

Parameterized Number is : 6

Parameterized Number is : 19

Parameterized Number is : 22

Parameterized Number is : 23

true

# JUnit - Eclipse 插件

为了设置带有 eclipse 的 JUnit，需要遵循以下步骤。

**步骤 1：下载 Junit archive**

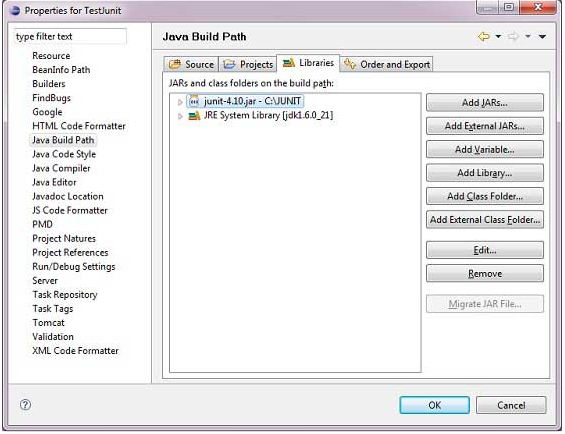
下载 JUnit

|  |  |
| --- | --- |
| 操作系统 | 文件名 |
| Windows | junit4.10.jar |
| Linux | junit4.10.jar |
| Mac | junit4.10.jar |

假设你在 C:>JUnit 文件夹中复制了以上 JAR 文件。

**步骤 2：设置 Eclipse 环境**

1. 打开 eclipse -> 右击 project 并 点击 property > Build Path > Configure Build Path，然后使用 Add External Jar 按钮在函数库中添加 junit-4.10.jar。



1. 我们假设你的 eclipse 已经内置了 junit 插件并且它在 C:>eclipse/plugins 目录下，如不能获得，那么你可以从 JUnit Plugin 上下载。在 eclipse 的插件文件夹中解压下载的 zip 文件。最后重启 eclipse。

现在你的 eclipse 已经准备好 JUnit 测试用例的开发了。

**步骤 3：核实 Eclipse 中的 Junit 安装**

1. 在 eclipse 的任何位置上创建一个 TestJunit 项目。
2. 创建一个 MessageUtil 类来在项目中测试。

/\*

\* This class prints the given message on console.

\*/

**public** **class** MessageUtil {

**private** String message;

//Constructor

//@param message to be printed

**public** MessageUtil(String message){

**this**.message = message;

}

// prints the message

**public** String printMessage(){

System.***out***.println(message);

**return** message;

}

}

1. 在项目中创建一个 test 类 TestJunit

**import** org.junit.Test;

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

**public** **class** TestJunit {

String message = "Hello World";

MessageUtil messageUtil = **new** MessageUtil(message);

@Test

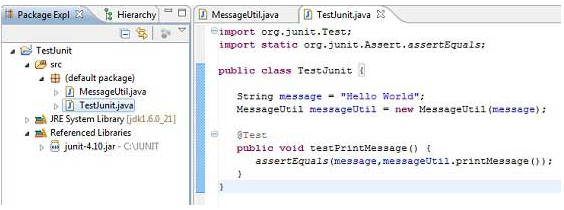
**public** **void** testPrintMessage() {

assertEquals(message,messageUtil.printMessage());

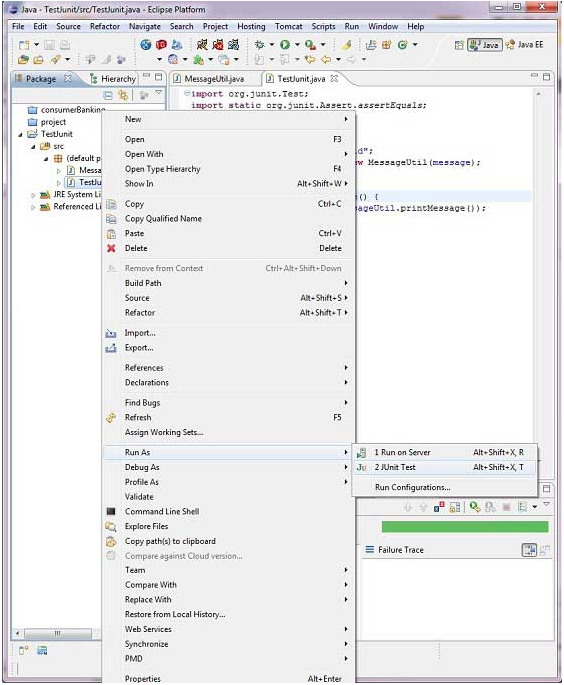
}

}

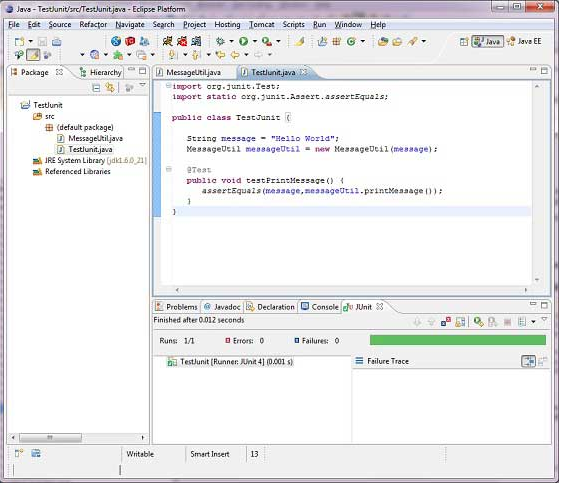
下面是项目结构



最后，通过右击程序和 run as junit 验证程序的输出。



验证结果



# JUnit - 框架扩展

以下是 JUnit 扩展

1. Cactus
2. JWebUnit
3. XMLUnit
4. MockObject

* **Cactus**

Cactus 是一个简单框架用来测试服务器端的 Java 代码（Servlets, EJBs, Tag Libs, Filters）。Cactus 的设计意图是用来减小为服务器端代码写测试样例的成本。它使用 JUnit 并且在此基础上进行扩展。Cactus 实现了 in-container 的策略，意味着可以在容器内部执行测试。

Cactus 系统由以下几个部分组成：

1. Cactus Framework（Cactus 框架） 是 Cactus 的核心。它是提供 API 写 Cactus 测试代码的引擎。
2. Cactus Integration Modules（Cactus 集成模块） 它是提供使用 Cactus Framework（Ant scripts, Eclipse plugin, Maven plugin）的前端和框架。

这是使用 cactus 的样例代码。

**import** org.apache.cactus.\*;

**import** junit.framework.\*;

**public** **class** TestSampleServlet **extends** ServletTestCase {

@Test

**public** **void** testServlet() {

// Initialize class to test

SampleServlet servlet = **new** SampleServlet();

// Set a variable in session as the doSomething()

// method that we are testing

session.setAttribute("name", "value");

// Call the method to test, passing an

// HttpServletRequest object (for example)

String result = servlet.doSomething(request);

// Perform verification that test was successful

assertEquals("something", result);

assertEquals("otherValue", session.getAttribute("otherName"));

}

}

* **JWebUnit**

JWebUnit 是一个基于 Java 的用于 web 应用的测试框架。它以一种统一、简单测试接口的方式包装了如 HtmlUnit 和 Selenium 这些已经存在的框架来允许你快速地测试 web 应用程序的正确性。

JWebUnit 提供了一种高级别的 Java API 用来处理结合了一系列验证程序正确性的断言的 web 应用程序。这包括通过链接，表单的填写和提交，表格内容的验证和其他 web 应用程序典型的业务特征。

这个简单的导航方法和随时可用的断言允许建立更多的快速测试而不是仅仅使用 JUnit 和 HtmlUnit。另外如果你想从 HtmlUnit 切换到其它的插件，例如 Selenium(很快可以使用)，那么不用重写你的测试样例代码。

以下是样例代码。

**import** junit.framework.TestCase;

**import** net.sourceforge.jwebunit.WebTester;

**public** **class** ExampleWebTestCase **extends** TestCase {

**private** WebTester tester;

**public** ExampleWebTestCase(String name) {

**super**(name);

tester = **new** WebTester();

}

//set base url

**public** **void** setUp() **throws** Exception {

getTestContext().setBaseUrl("http://myserver:8080/myapp");

}

// test base info

@Test

**public** **void** testInfoPage() {

beginAt("/info.html");

}

}

* **XMLUnit**

XMLUnit 提供了一个单一的 JUnit 扩展类，即 XMLTestCase，还有一些允许断言的支持类：

1. 比较两个 XML 文件的不同（通过使用 Diff 和 DetailedDiff 类）
2. 一个 XML 文件的验证（通过使用 Validator 类）
3. 使用 XSLT 转换一个 XML 文件的结果（通过使用 Transform 类）
4. 对一个 XML 文件 XPath 表达式的评估（通过实现 XpathEngine 接口）
5. 一个 XML 文件进行 DOM Traversal 后的独立结点（通过使用 NodeTest 类）

我们假设有两个我们想要比较和断言它们相同的 XML 文件，我们可以写一个如下的简单测试类：

**import** org.custommonkey.xmlunit.XMLTestCase;

**public** **class** MyXMLTestCase **extends** XMLTestCase {

// this test method compare two pieces of the XML

@Test

**public** **void** testForXMLEquality() **throws** Exception {

String myControlXML = "<msg><uuid>0x00435A8C</uuid></msg>";

String myTestXML = "<msg><localId>2376</localId></msg>";

assertXMLEqual("Comparing test xml to control xml",

myControlXML, myTestXML);

}

}

* **MockObject**

在一个单元测试中，虚拟对象可以模拟复杂的，真实的（非虚拟）对象的行为，因此当一个真实对象不现实或不可能包含进一个单元测试的时候非常有用。

用虚拟对象进行测试时一般的编程风格包括：

1. 创建虚拟对象的实例
2. 在虚拟对象中设置状态和描述
3. 结合虚拟对象调用域代码作为参数
4. 在虚拟对象中验证一致性

以下是使用 Jmock 的 MockObject 例子。

**import** org.jmock.Mockery;

**import** org.jmock.Expectations;

**class** PubTest **extends** TestCase {

Mockery context = **new** Mockery();

**public** **void** testSubReceivesMessage() {

// set up

**final** Sub sub = context.mock(Sub.**class**);

Pub pub = **new** Pub();

pub.add(sub);

**final** String message = "message";

// expectations

context.checking(**new** Expectations() {

oneOf (sub).receive(message);

});

// execute

pub.publish(message);

// verify

context.assertIsSatisfied();

}

}