

单元测试一条龙

唐光杰、周定鹏、胡景林、云层（排名不分先后）

2011-8-15

为什么会有这个文档？

带学生做单元测试专题项目的时候发现很多问题行业上没有比较通俗易懂的文档和整体思路，于是发动学生们做了这个。

这个文档有啥用？

该文档包含了 JAVA 基本概念，如何使用 Junit 进行单元测试，如何使用 Feed4Junit 进行参数化扩展，如何使用 junitperf 进行单元性能测试，如何使用 selenium junit 完成自动化测试。算是功能，自动化，性能一条龙。

哪些学生这么给力？

上海 68 期最后阶段留下了 3 名同学唐光杰、周定鹏、胡景林，他们在 JAVA 语言上有一定的基础共同解决了很多技术上的问题。

这个文档谁写的？

主要的编写工作是云层负责的，代码的调试和评审及部分内容的添加是由学生们完成的。这里还要感谢 51testing 的强哥和德宝，中间参考了他们留下的部分文档。

文档中的相关代码哪里可以找到？

这个可以访问 51testing 的云层 blog，里面会有相关的内容，附属的一些 jar 包由于体积问题，最终可能从代码包中删除，请自行搜索下载。

还有什么要说的？

可以用自己的语言口头话写那么一个东西真是感觉太好了，如果能够用这样的方式写《性能测试进阶指南 2》就好了，写成官方用语确实很让我纠结。

最后我还可以感谢党，感谢国家，感谢领导，感谢 51testing，感谢老爸老妈，感谢老婆，感谢同学，感谢。。。。。。。（甲某：这个人废话太多了，拉出去让他去竹林坐滑道再多撒点水吧）

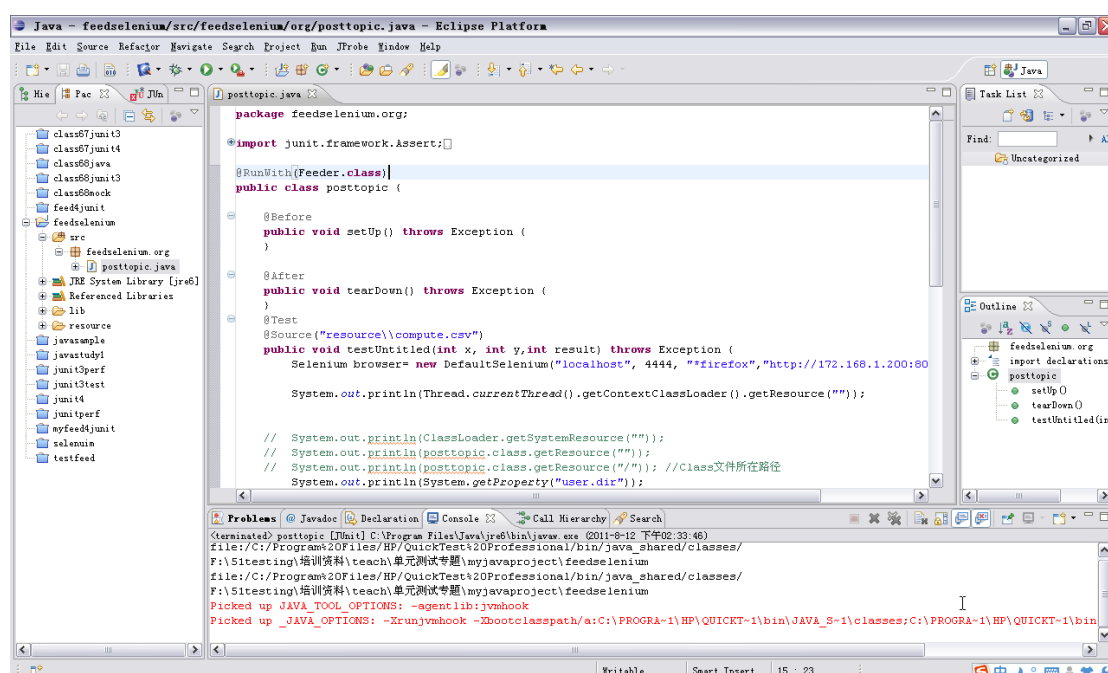
1 什么是面向对象.....	3
1.1 基本概念.....	3
1.2 类, 方法, 属性.....	6
1.3 接口	15
1.4 继承	18
1.4.1 父类.....	19
1.4.2 子类.....	19
1.4.3 复写.....	19
1.4.4 扩展.....	20
1.5 多态	21
1.6 构造函数.....	22
2 单元测试.....	23
2.1 Junit.....	23
2.1.1 junit3	24
2.1.2 junit4	30
2.2 feed4junit	37
2.3 Mock.....	39
2.4 Junitperf	41
3 自动化测试.....	45
3.1 Selenium.....	45
3.2 seleniumide 插件	45
3.3 selenium rc.....	46
3.4 使用 Junit 调用 selenium rc 实现自动化	47
3.5 加载 feed4junit 完成数据驱动	48
3.6 加入页面内容断言.....	48

1 什么是面向对象

面向对象是一种开发模式，相对面向过程更加方便扩展。

1.1 基本概念

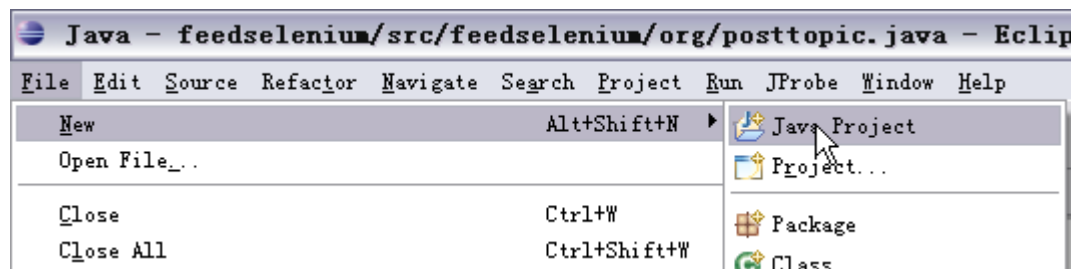
在使用 java 进行单元测试前，首先我们需要安装 jdk 和开发工具 eclipse.



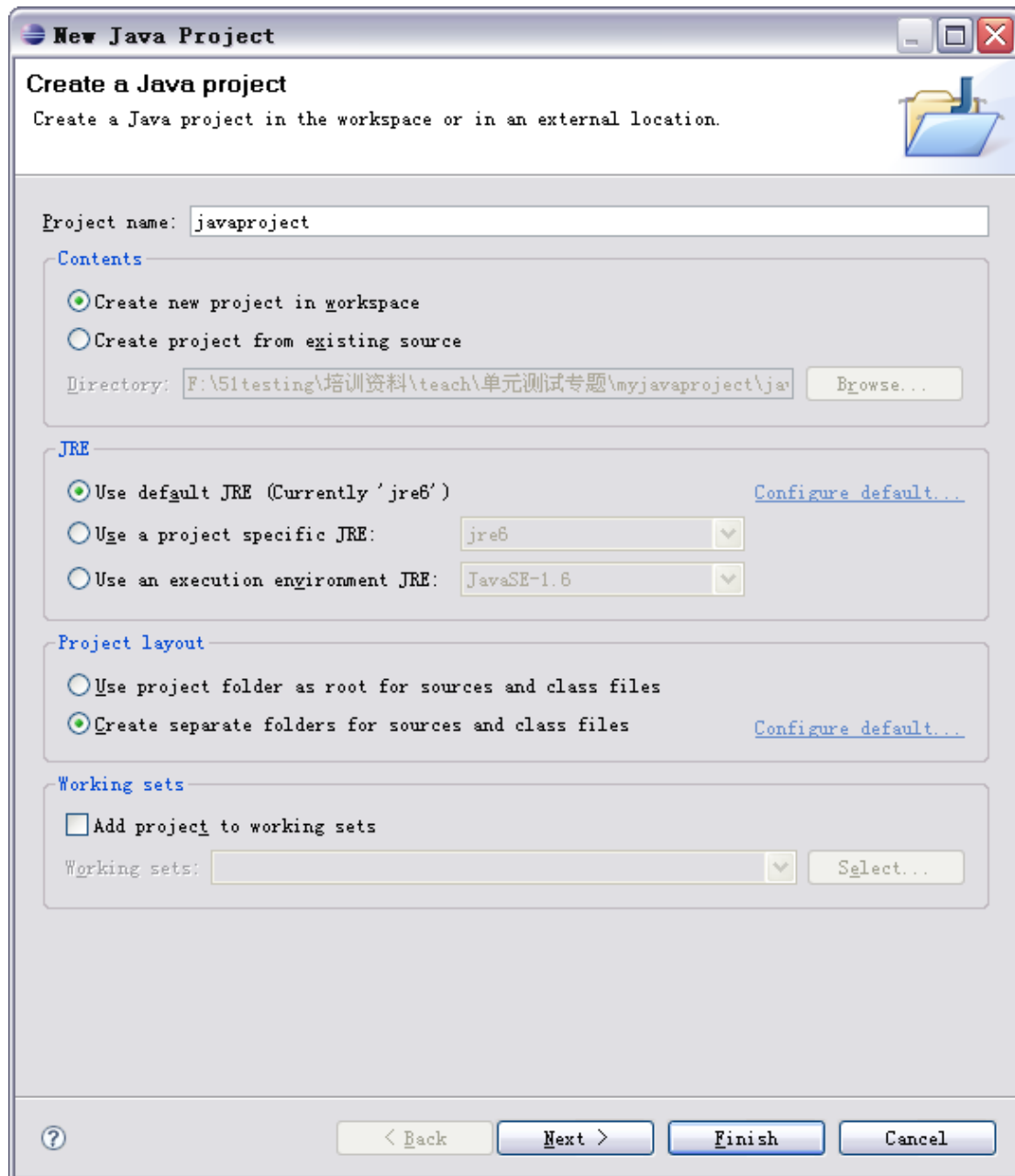
Eclipse 的基本操作

新建项目：

打开 file 菜单下的 new 下的 java project

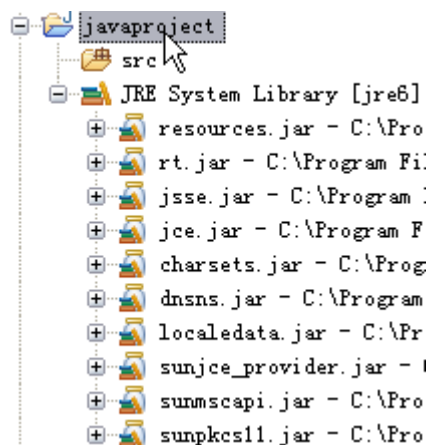


在弹出的项目信息中我们可以填写 project name

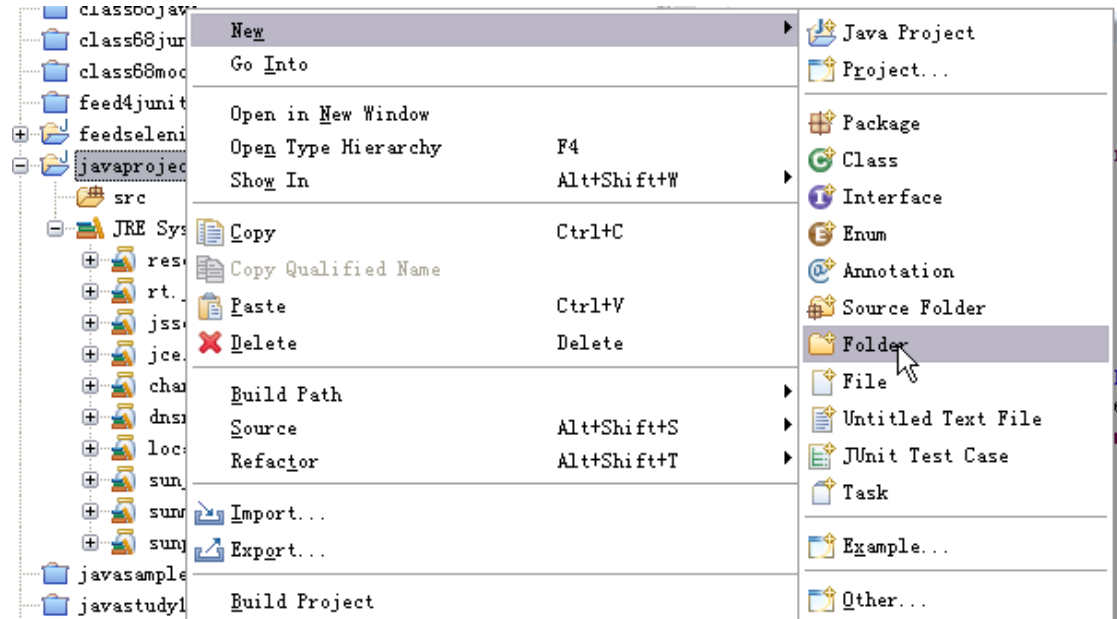


点击 finish 完成项目新建。

在左侧的 package explore 中可以看到新建出来的项目结构



在对应的代码目录里面会出现 src 的目录。这里建议在项目下另外新建一个 lib 和一个 source 目录用来存放使用的 jar 包及资源文件，方便以后调用。



在项目上右键，新建一个 folder 即可。

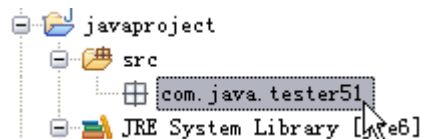
什么是包？

包是一种命名方式，方便通过域名类似格式来区分不同的代码，一般我们会为不同项目的代码设置不同的包名，编译后的代码会存放在 bin 目录下的改包结构的目录下。

这里我们在 src 下新建一个 com.java.tester51 的包。



确认后左侧会出现对应的包名



基础准备就这点，接着来说面向对象的内容。

1.2 类，方法，属性

万事万物都是按照某些类别来的，例如人就是一个类别，而人具备身高，体重这样的属性；人还具备一些行为，这些行为就是方法，例如跑跳；我们在 `com.java.teste51` 上新建一个类

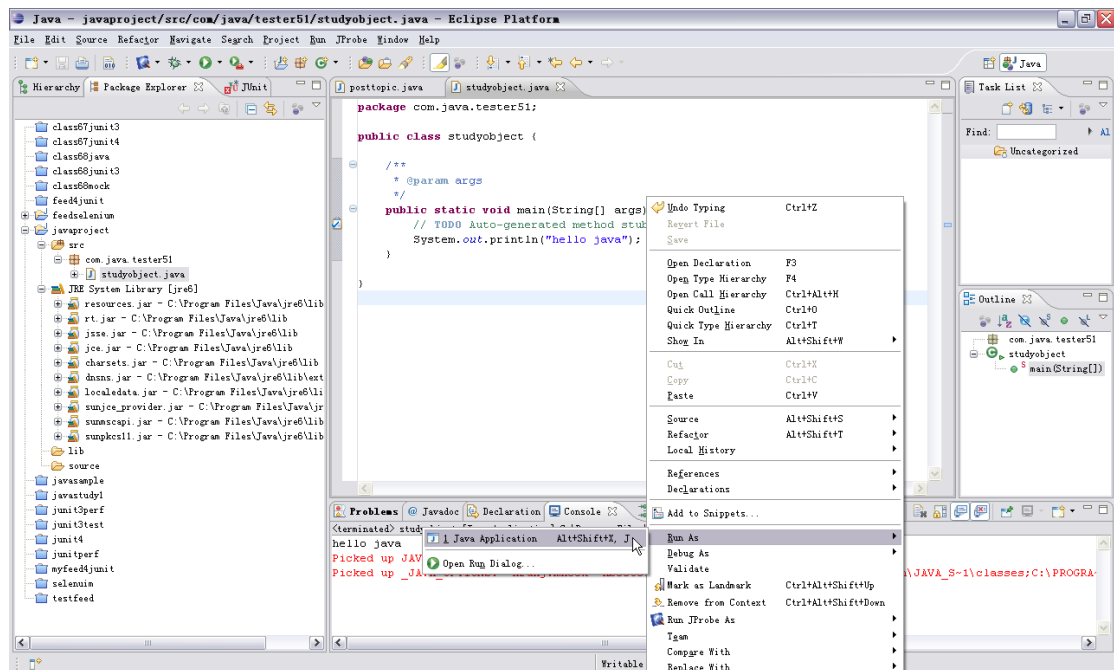


类名为 studyobject，另外这里选中了 public static void main，这个就是 Java 的主方法，代码都是从这里开始运行的，这里添加的目的就是为了后面测试代码方便。

```
package com.java.teste51;
```

```
public class studyobject {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
    }  
}
```

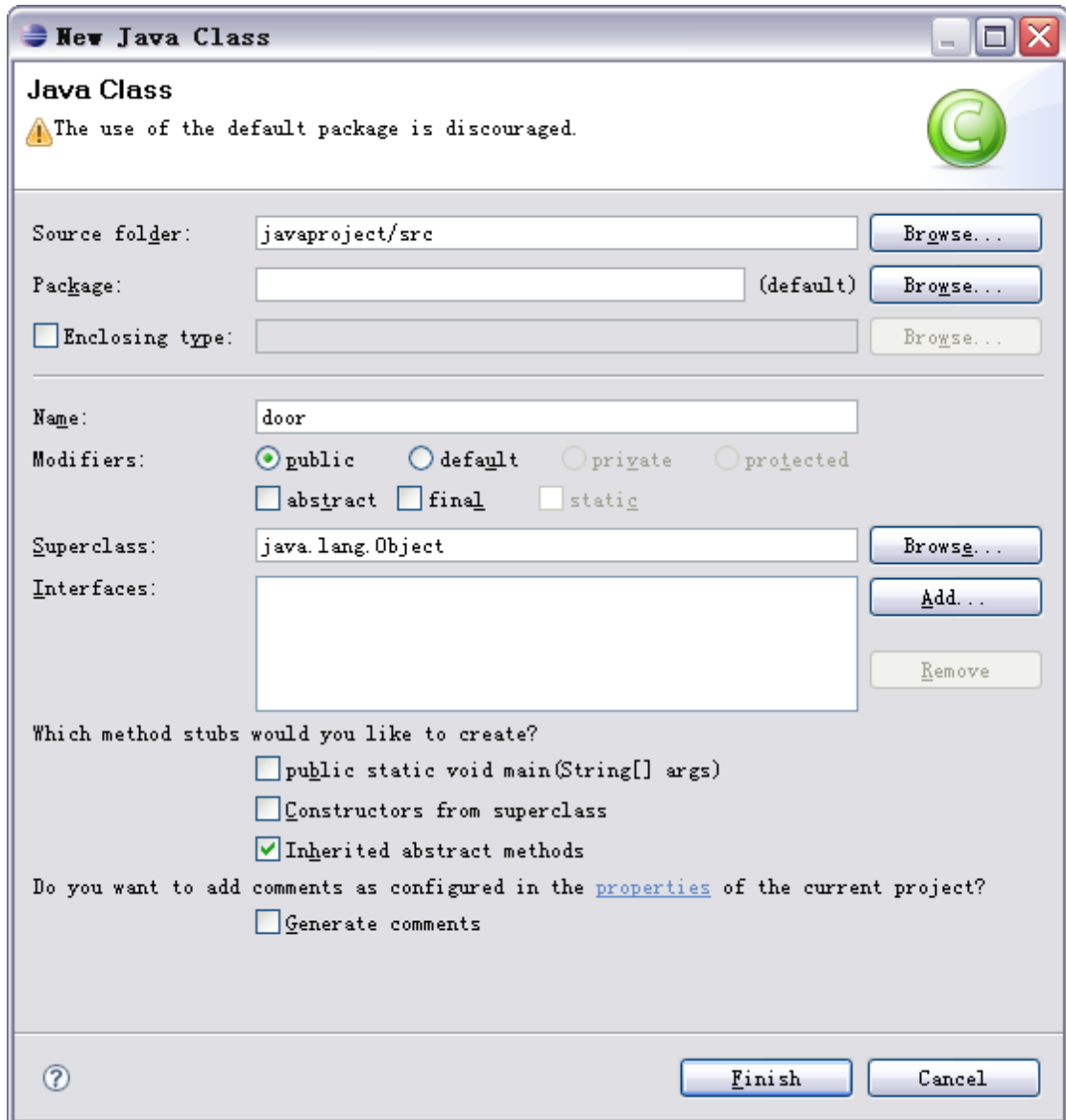
现在我们生成的了这样的代码，首先先添加这样的一个语句
`System.out.println("hello java");`
该代码的作用是在命令行输出一个 Hello java 的字符串。



在代码的 main 方法内，右键选择 Run As /Java Application 即可运行，在下方的 Console 里面可以看到运行后的结果。

新建我们自己的类:

在代码中我们可以新建自己的类，例如我们建立一个 door 门的类，新建一个类



Finish完成后将会生成下面这样的代码

```
public class door {  
  
}
```

这里我们新建了一个（）没有输入参数的 door 类，它是一个 public 公共的类，可以被别人方便的访问。

然后为这个类附加属性和方法。

```
package com.java.tester51;  
  
public class door {  
    private int height;  
    public void opendoor()  
}
```

```
{  
  
}  
}
```

这里我们添加了一个私有的 `height` 属性和一个 `opendoor` 的无返回方法。

方法和属性有静态和普通之分，静态/属性方法可以在类不被实例化的情况下直接使用，而普通方法只有类实例化之后才能使用。静态方法用 `static` 修饰。

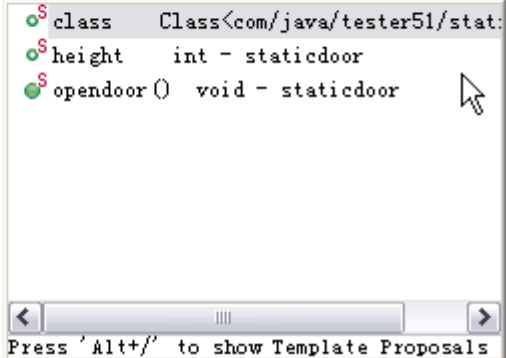
然后我们创建一个普通类内含有静态方法和静态属性的代码。

```
package com.java.tester51;  
  
public class staticdoor {  
    private static int height;  
    public static void opendoor()  
    {  
  
    }  
}
```

这里我们创建了一个私有的 `Height` 属性和一个静态的 `opendoor` 无返回方法。

对象的实例化是通过 `new` 关键字来实现的。首先我们在 `studyobject` 的 `main` 代码里面感受一下静态方法的好处。

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    System.out.println("hello java");  
    staticdoor.  
}
```



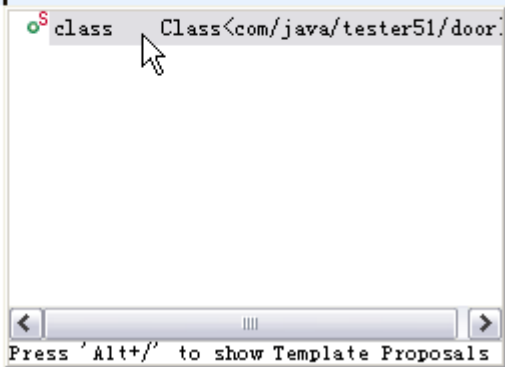
Press 'Alt+/' to show Template Proposals

当我们打入 `staticdoor.` 的时候，`eclipse` 给了我们该类的语法提示，其中就可以看到其中的 `opendoor` 方法和静态的 `height` 属性。而我们直接打 `door` 是没有办法访问到属性和方法的。

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    System.out.println("hello java");
    door.
}

```



为了访问到该类的内容，我们需要通过实例化来实现。

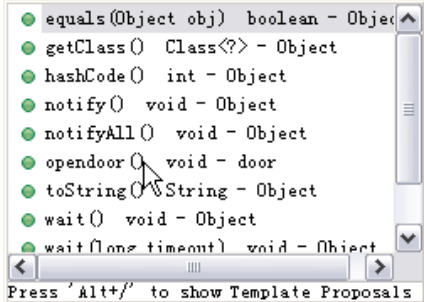
```

package com.java.testers51;

public class studyobject {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("hello java");
        door mydoor= new door();
        mydoor.
    }
}

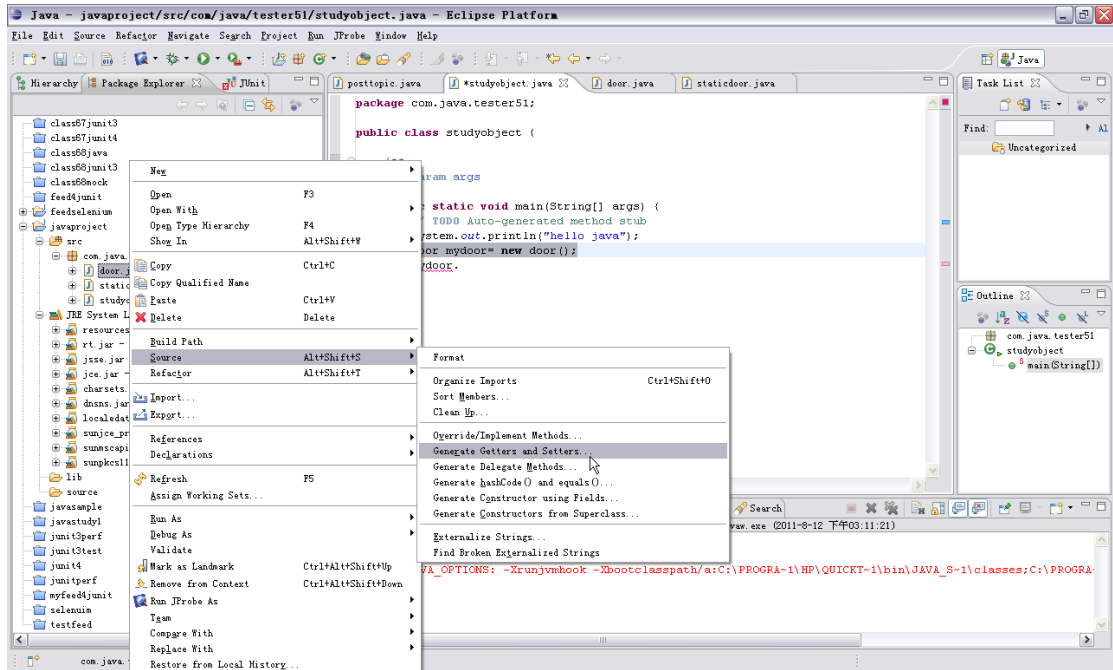
```



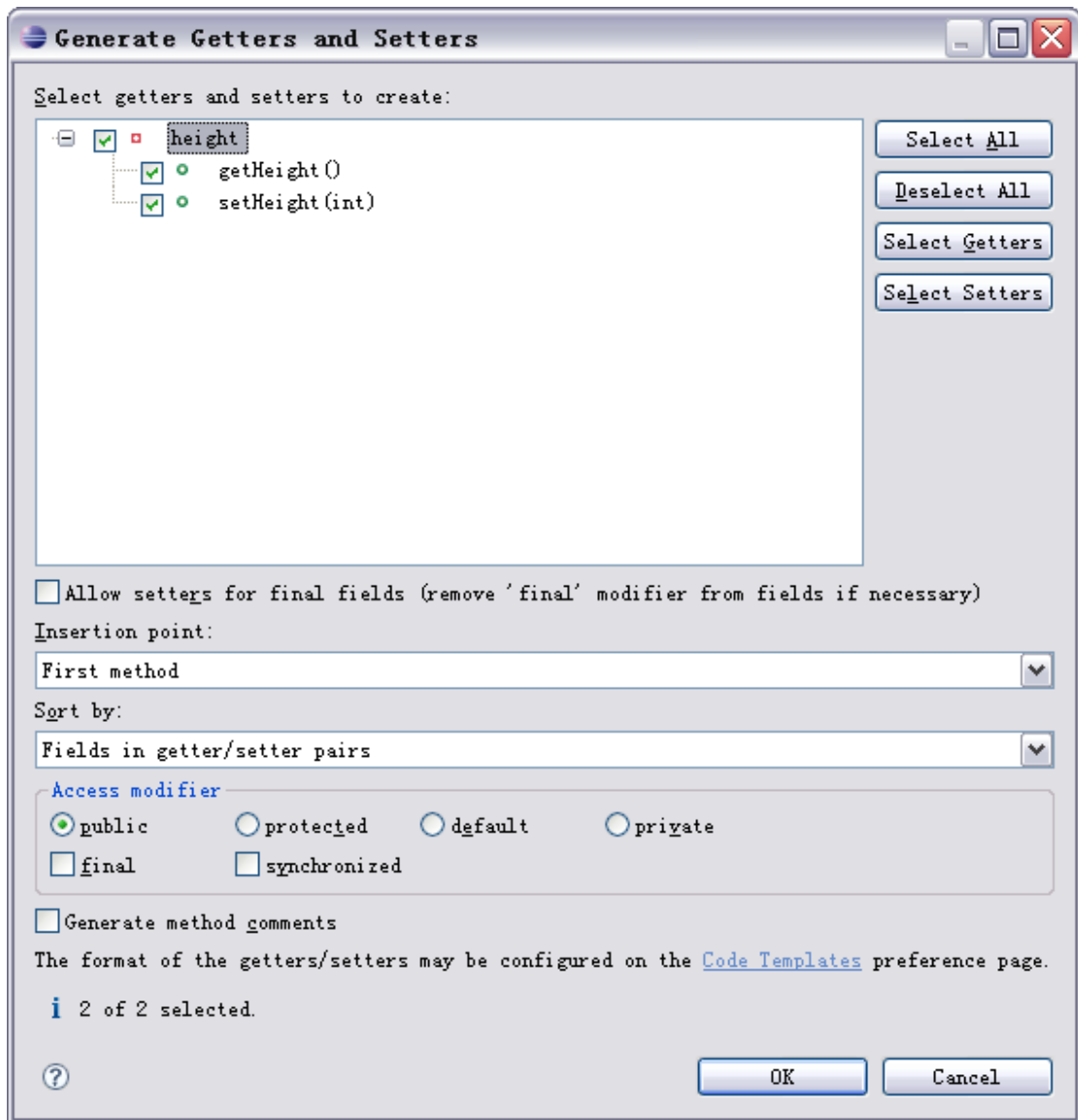
实例化方法是通过这样的代码实现的

```
door mydoor= new door();
```

我们将 door 实例化了一个 mydoor 出来，通过访问 mydoor 即可访问到我们想要访问的内容了，这里我们会发现 door 的 Height 属性是看不到的，这是因为我们使用了 private 的方式来修饰属性，一般对于属性我们推荐使用方法来设置。



在 door.java 上选择右键，source 下的生成 get 和 set，然后选择为 Height 生成对应的赋值和得值的方法



这样代码就会变成

```
package com.java.test51;
```

```
public class door {
    private int height;
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
    public void opendoor()
    {
    }
}
```

通过方法 `setHeight` 我们可以将传入的值赋值给 `this.height`, 这里的 `this` 是指本类里面的 `height` 属性, 还有一个叫做 `super` 是用来说明父类属性的。

从 `xx` 书引用:

Super关键字

子类在隐藏了父类的成员变量或重写了父类的方法后, 有时还要访问父类的成员变量或调用父类的方法, java中通过`super`来实现对父类成员的访问。

接着我们可以为 `opendoor` 里面编写一个简单的输出语句来确认该方法被执行了。

```
public void opendoor()
{
    System.out.println("the door be open");
}
```

回到 `studyobject` 我们来使用一下刚才的几个方法。

```
package com.java.tester51;
```

```
public class studyobject {
```

```
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("hello java");
        door mydoor= new door();//实例化mydoor
        mydoor.opendoor();//调用mydoor对象的opendoor方法
        mydoor.setHeight(10);//为mydoor对象设置高度属性为10
        System.out.println(mydoor.getHeight());//获得mydoor对象的高度属性
    }
}
```

返回的代码结果为

```
hello java
the door be open
10
```

当我们使用 `public` 属性的时候要注意, 属性默认是没有值的如果直接访问返回为 `0`。

例如我们新建一个 `door2` 的类, 其中属性为 `public`

```
package com.java.tester51;
```

```
public class door2 {
    public int height;
    public void opendoor()
```

```
    {  
    }  
}
```

将该类实例化

```
door2 myolddoor=new door2();  
System.out.println(myolddoor.height);
```

这个时候我们可以直接在类下访问到该属性了，但是输出值为 0，因为该数据还没有被初始赋值，如果需要赋值可以考虑使用 set,get 的方式来做，或者使用构造函数来初始化，所以一般属性都是 private 的，使用 public 就需要有构造函数来初始化了。而 private 最好先 set 一个值再来 get 或者也是用构造函数来做。

1.3 接口

从 xx 书引用：

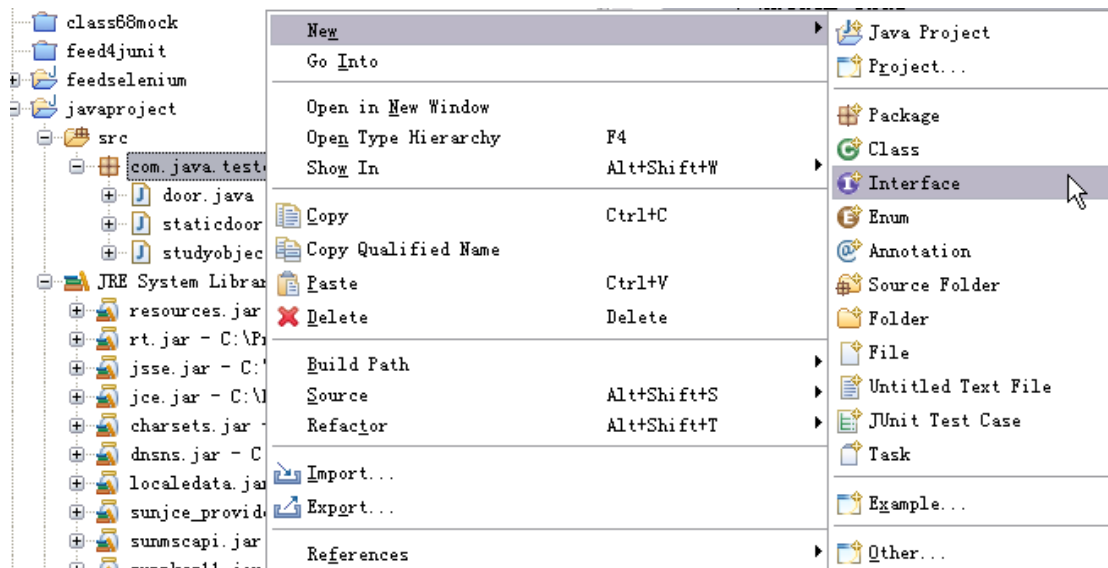
接口是一组对类的需求描述，这些类要遵从接口描绘的统一格式进行定义。接口绝不能含有实例域，也不能在接口中实现方法，提供实例域和方法的任务应由实现接口的那个类来完成。接口中所有方法默认都是 public，因此接口中声明方法时不必提供 public 关键字，但是在实现接口时必须将方法声明成 public。

1. 接口不是类，不能用 new 运算符实例化一个接口
2. 尽管不能构造接口对象，但能声明接口变量
3. 接口变量必须引用实现了接口的类对象
4. 与可以建立类的继承关系一样，接口也是可扩展的
5. 接口中可以包含常量

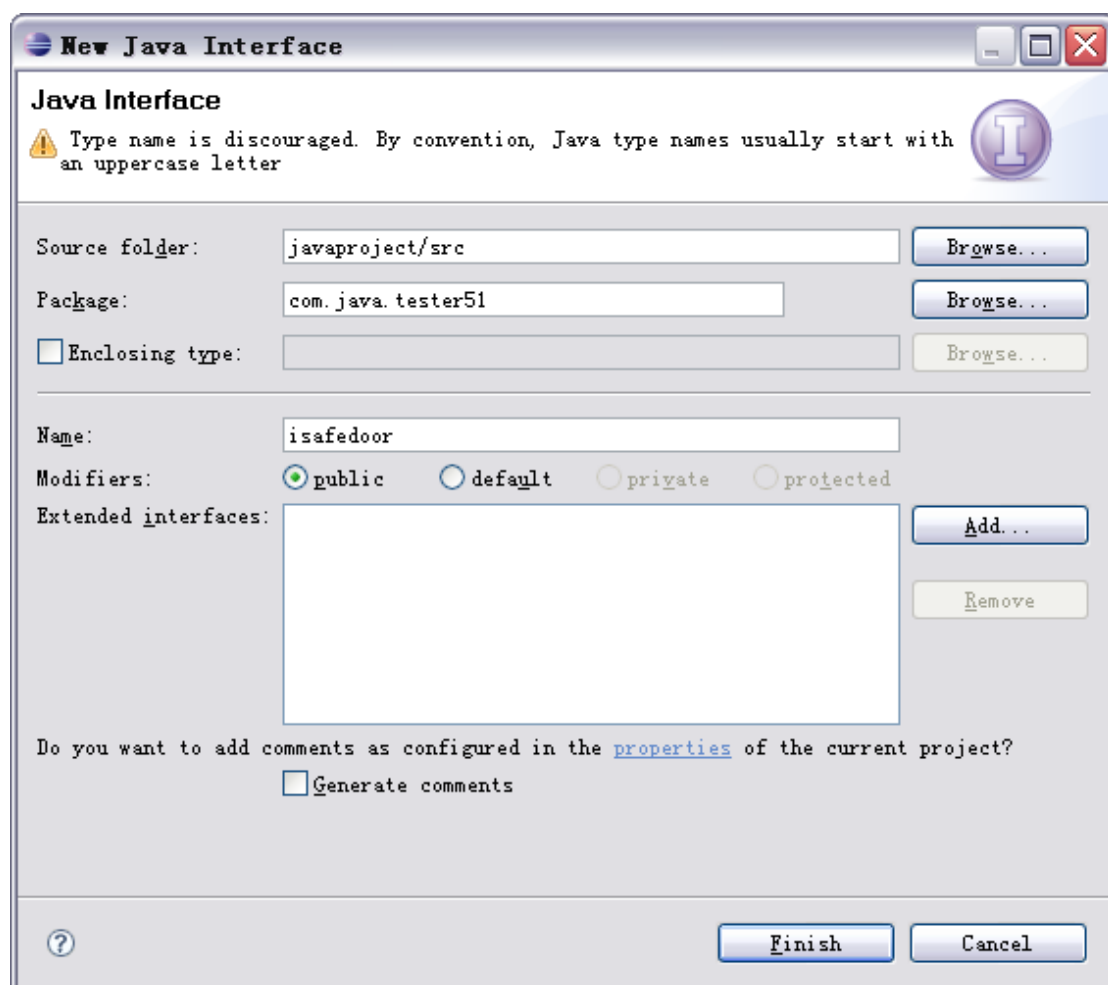
接口好比一种规范，接口自身不能使用，但是可以通过被实现而发扬，与接口对应的还有一个叫做抽象类的东西。

抽象类本身不能被实例化必须要通过继承来实现。抽象类可以实现属性，方法。而接口只能描述方法。

在包中我们新建一个 safedoor 的接口。



接口可以继承接口，到继承的时候再介绍这个概念



Finish 确定后生成接口 isafedoor，生成的代码如下

```
package com.java.tester51;

public interface isafedoor {
```



```
}
```

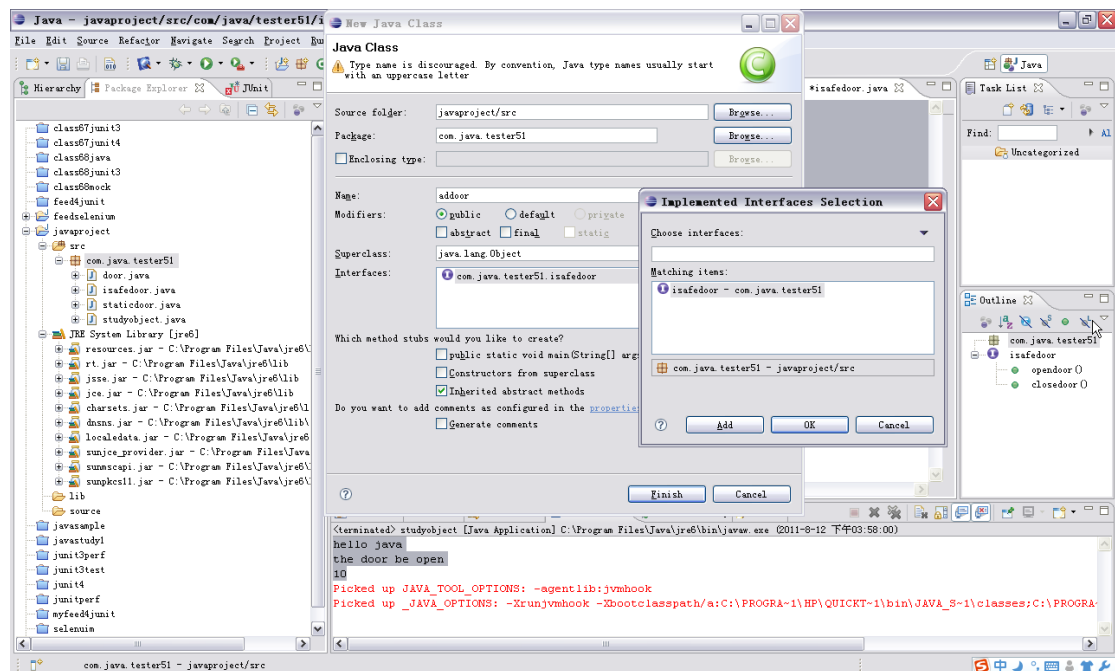
接着我们要为这个接口添加点方法描述,一个门应该有开门和关门两个最基本的功能(方法)

```
package com.java.testers1;
```

```
public interface isafedoor {  
    public void opendoor();  
    public void closedoor();  
}
```

所有的类(门)如果使用该接口都必须要实现这两个 void 的方法。

接着我们可以新建一个类,实现该接口



在包上新建一个类,为该类添加一个接口,结构选择 isafedoor 确定。

然后我们会发现新建出来的 addoor 会自然包含对应的两个必须实现的方法。

```
package com.java.testers1;
```

```
public class addoor implements isafedoor {  
  
    @Override  
    public void closedoor() {  
        // TODO Auto-generated method stub  
    }  
  
    @Override  
    public void opendoor() {  
        // TODO Auto-generated method stub  
    }  
}
```

```
}  
  
}
```

这两个方法必须实现否则就违反了接口的规定。Implements isafedoor 就是说明 addoor 实现了 isafedoor 这个接口。

1.4 继承

何为继承，简单来说就是子承父业，一个门会有第二代，第三代，那么后一代会吸取前一代的一些内容，所以我们说后一代继承前一代。继承和接口的实现有点类似，使用 extends 关键字来说明。例如我们新建一个 door1 是继承于 door 的，那么 door1 就拥有 door 自身所拥有的所有属性和方法。



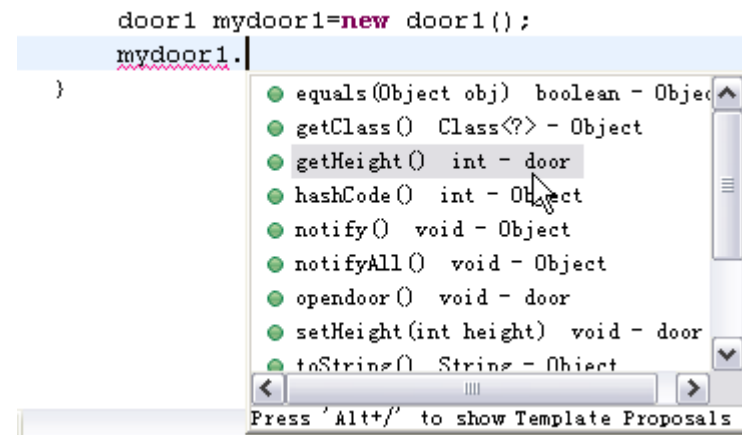
代码如下

```
package com.java.tester51;

public class door1 extends door {

}
```

虽然 door1 里面什么都没有，但是当我们把 door1 实例化的时候会发现一样拥有 opendoor() 和 getheight()/setheight()方法



这就是继承的作用。

1.4.1 父类

被继承的类相对于继承类来说就是父类，父类可以通过 super 关键字来访问。

1.4.2 子类

继承的类叫做子类。

1.4.3 复写

从 xx 书引用：

重写 (override)，在类层级结构中，当子类的成员变量与父类的成员变量重名时，子类的成员变量会隐藏父类的成员变量；当子类的方法与父类的方法具有相同的名字、参数列表、返回值类型时，子类的方法就叫重写了父类的方法，当重写的方法在子类的对象被调用时，它总是参考在子类中定义的版本，在父类定义的方法就被隐藏。

个人理解：

复写是指当某些父类的东西写得不够好得时候我们可以修改掉它。比如我们 door 父类的 opendoor()方法操作的结果是

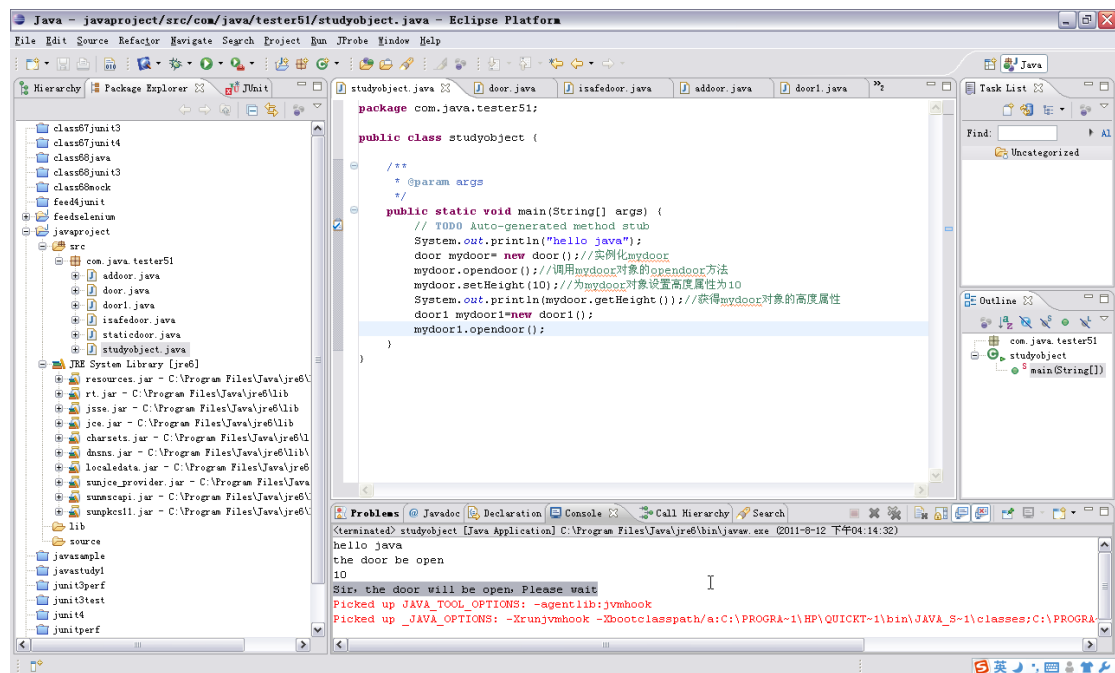
```
System.out.println("the door be open");
```

我们会觉得这个太老土了是不是要专业点，那么在 door1 里面我们可以复写这个方法

```
package com.java.testers51;
```

```
public class door1 extends door {  
    public void opendoor()  
    {  
        System.out.println("Sir, the door will be open, Please wait");  
    }  
}
```

这里在 door1 内我们重写了 opendoor 方法,并且把输出的信息做了个修改。回到 studyobject 内,我们可以将 door1 实例化再访问 opendoor 方法,即可发现复写有效。



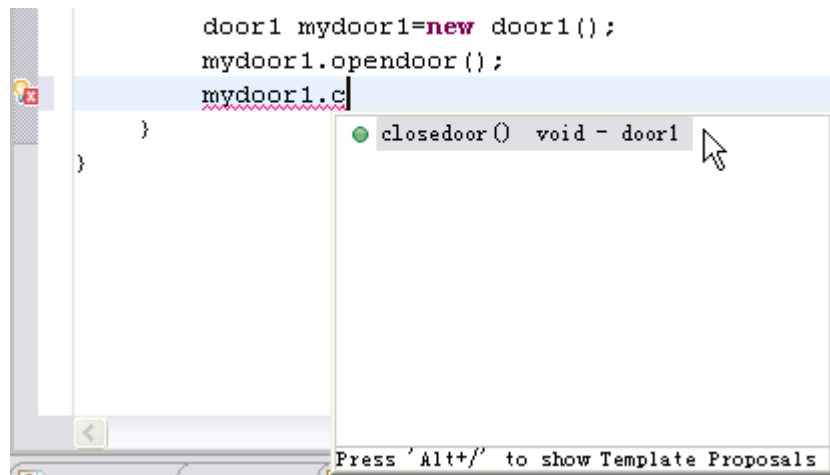
1.4.4 扩展

扩展和复写类似,有些新功能父类没有,那么我需要写个新的,那么就叫做在原来父类的基础上扩展一个功能。直接在子类上编写一个 closedoor()方法即可。

```
package com.java.testers51;
```

```
public class door1 extends door {  
    public void opendoor()  
    {  
        System.out.println("Sir, the door will be open, Please wait");  
    }  
    public void closedoor()  
    {  
        System.out.println("Sir, the door will be close, Please wait");  
    }  
}
```

这里的 `closedoor()` 是父类没有的，所以这是一个扩展方法。当实例化 `door1` 的时候我们可以看到该方法可以被访问。



1.5 多态

多态是一种表现形式，可以将一个对象处理不同内容的不同行为表现出来。例如不同的人问你借钱你回答的内容都不会相同，这种情况就是多态。

在编写多态的时候我们需要使用接口变量。我们现在先新建两个不同的 `door`，一个是 `addoor` 一个是 `badoor` 都实现了 `isafedor` 接口，分别能够完成开门和关门两个方法，并且输出的内容不同。

```
package com.java.testers51;
```

```
public class badoor implements isafedor {
```

```
    @Override
```

```
    public void closedoor() {
```

```
        // TODO Auto-generated method stub
```

```
        System.out.println("this is badoor use hand close door");
```

```
    }
```

```
    @Override
```

```
    public void opendoor() {
```

```
        // TODO Auto-generated method stub
```

```
        System.out.println("this is badoor use hand open door");
```

```
    }
```

```
}
```

```
package com.java.testers51;
```

```
public class addoor implements isafedor {
```

```

@Override
public void closedoor() {
    // TODO Auto-generated method stub
    System.out.println("this is addoor auto close door");
}

@Override
public void opendoor() {
    // TODO Auto-generated method stub
    System.out.println("this is addoor auto open door");
}
}

```

这里实现了两个同一接口实现的不同类后我们编写多态调用的代码。

```

isafedoor myi;
myi=new addoor();
myi.closedoor();
myi.opendoor();

```

这个代码说明我们新建了一个 myi 的 isafedoor 接口类型变量，我可以为这个 myi 接口变量赋值为一个 addoor() 的类，然后调用 myi 下的方法，代码执行结果返回

```

this is addoor auto close door
this is addoor auto open door

```

当我们把代码修改为

```

isafedoor myi;
myi=new badoor();
myi.closedoor();
myi.opendoor();

```

这次把 badoor() 这个类赋值给 myi 这个接口变量的时候，结果变为

```

this is badoor use hand close door
this is badoor use hand open door

```

这里我们可以发现只要给这个接口变量不同的实例化类，就能实现不同的功能，这就是多态。

1.6 构造函数

构造函数用来在类被执行前做初始化操作的，与之对应的还有一个析构函数。

构造函数的写法如下

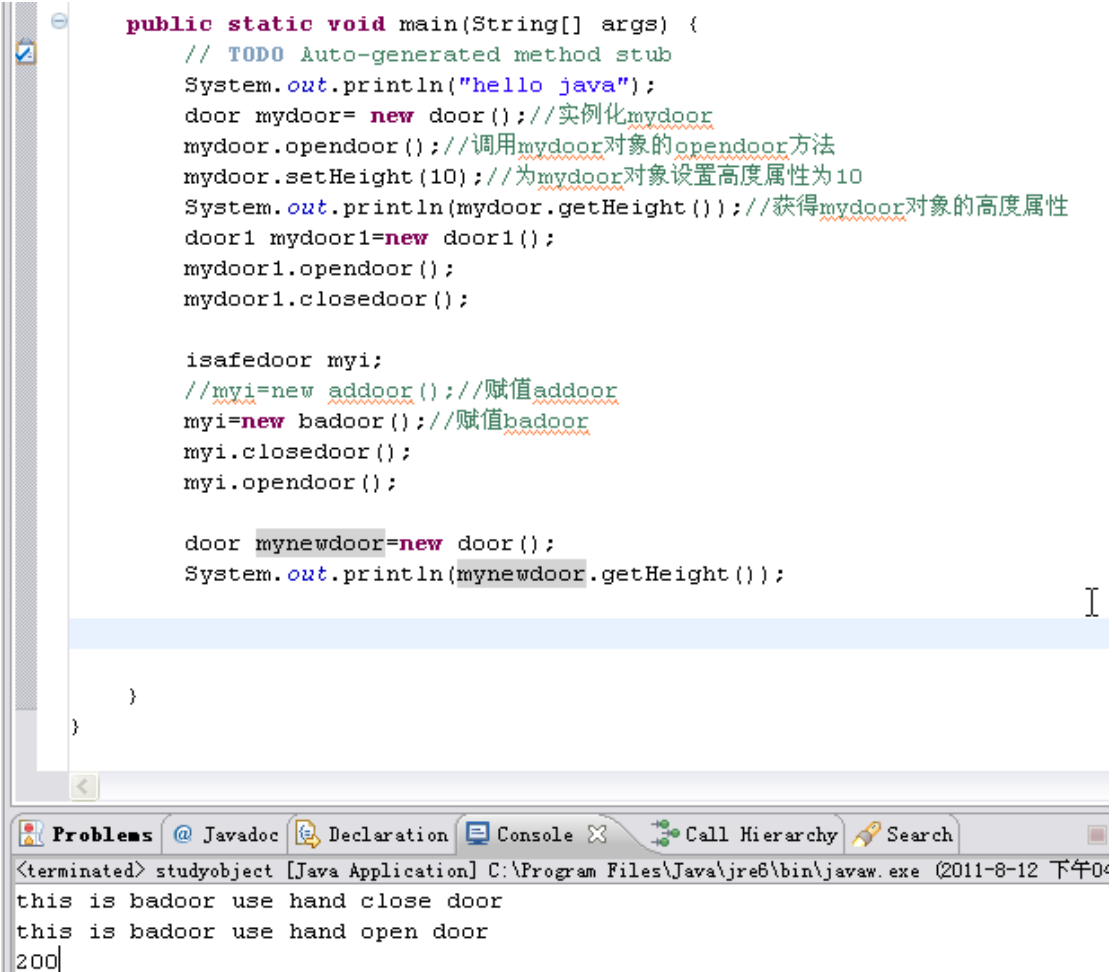
```

public door() {
    this.height = 200;
}

```

构造函数名必须与类名相同，而且是没有返回类型的。

当我们再次访问 door 这个类的 getHeight 方法时，就不用像以前一样需要先 set 一下才有值了，相当于初始化。



```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    System.out.println("hello java");
    door mydoor= new door();//实例化mydoor
    mydoor.opendoor();//调用mydoor对象的opendoor方法
    mydoor.setHeight(10);//为mydoor对象设置高度属性为10
    System.out.println(mydoor.getHeight());//获得mydoor对象的高度属性
    door1 mydoor1=new door1();
    mydoor1.opendoor();
    mydoor1.closedoor();

    isafedoor myi;
    //myi=new addoor();//赋值addoor
    myi=new badoor();//赋值badoor
    myi.closedoor();
    myi.opendoor();

    door mynewdoor=new door();
    System.out.println(mynewdoor.getHeight());
}
}
```

Problems @ Javadoc Declaration Console Call Hierarchy Search

```
<terminated> studyobject [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (2011-8-12 下午04:00)
this is badoor use hand close door
this is badoor use hand open door
200|
```

现在我们可以看到 mynewdoor 可以直接访问 getHeight 并且返回 200。析构函数的作用就是在实例化对象被回收的时候可以规定一下做什么，Java 是自动回收机制的。

代码见 javaproject.rar

2 单元测试

单元测试就是针对代码的逻辑进行测试的一种白盒测试方法，主要是通过桩和驱动来完成对代码的路径、条件分支、代码覆盖率测试。

2.1 Junit

Junit 是 xunit 系列中的针对 java 语言的单元测试框架。

我们首先新建一个简单的类 compute 在里面实现一个 add 方法。

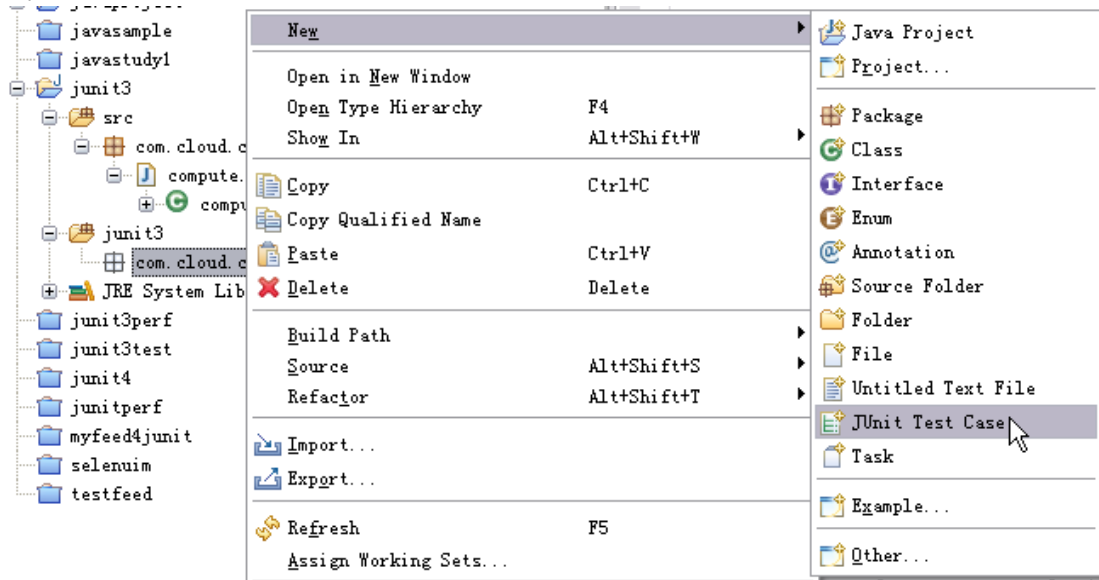
```
package com.cloud.chen;
```

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

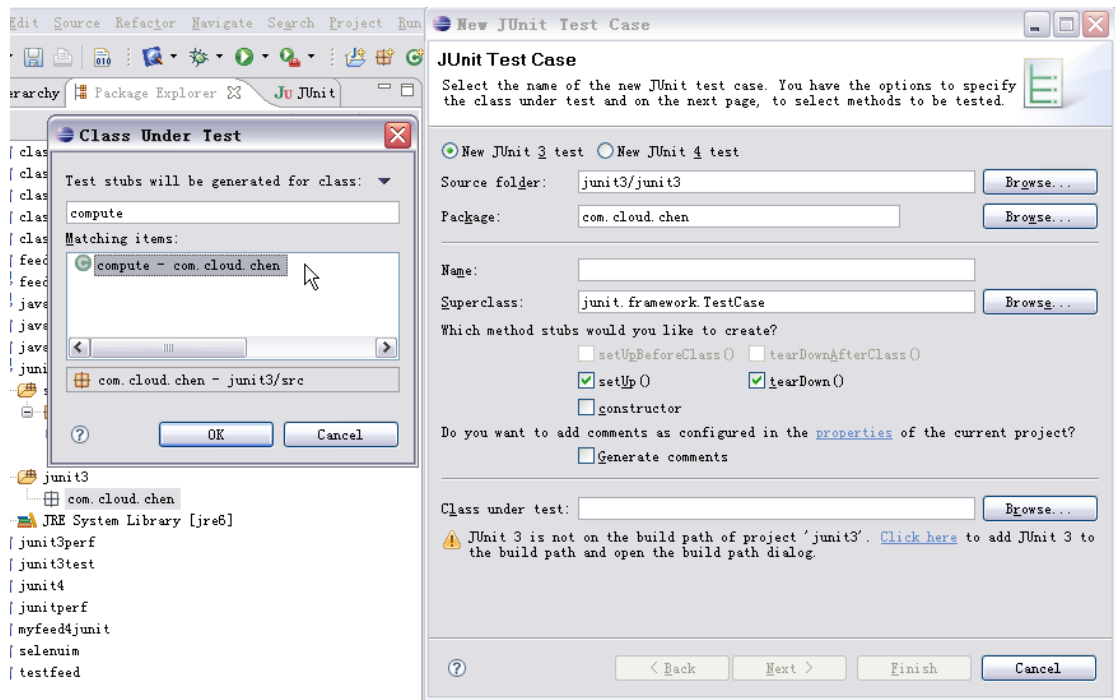
2.1.1 junit3

这里我们使用 junit3 来对这个 add 方法进行单元测试。

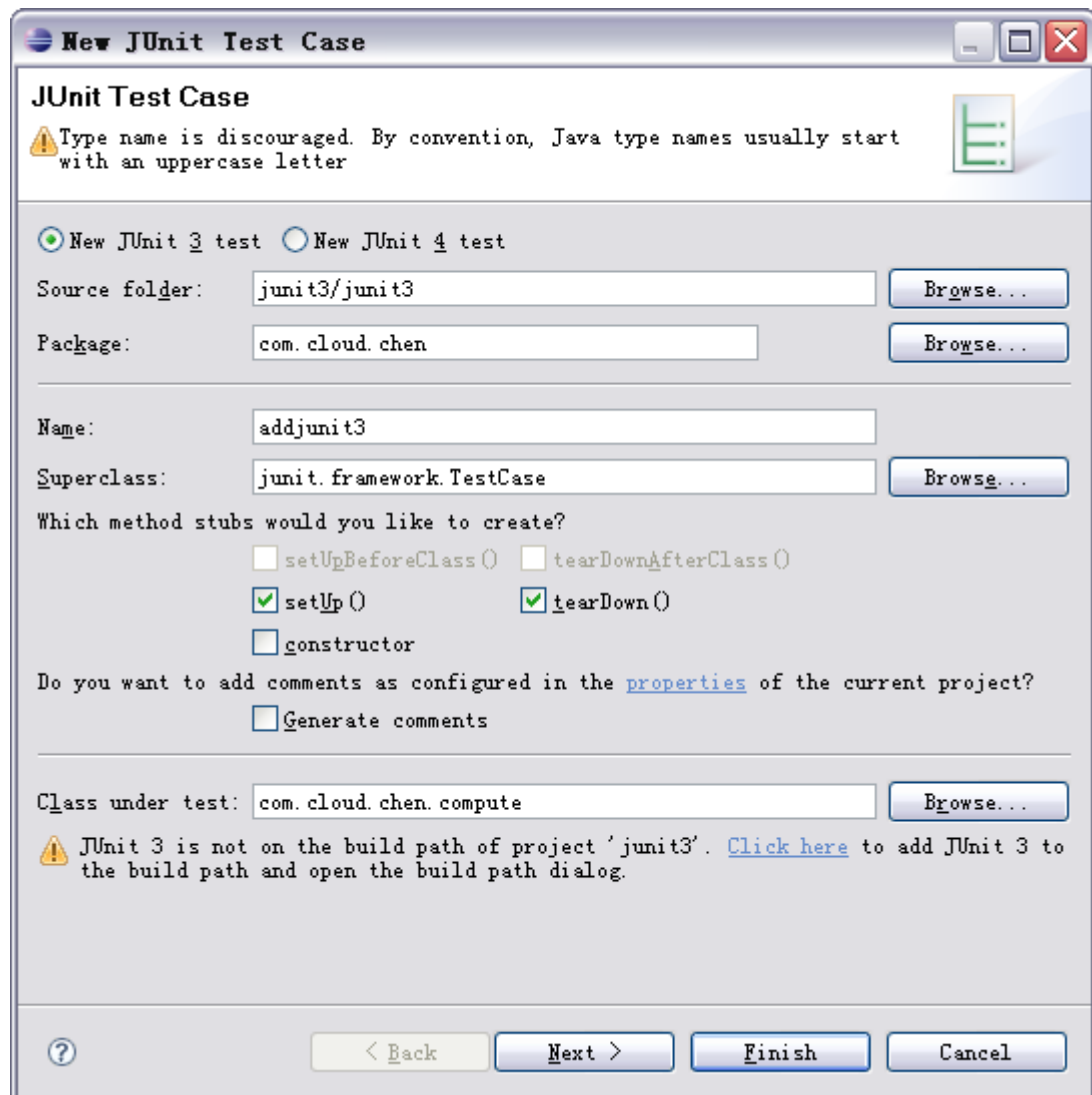
首先建立一个新的代码目录，这样方便把测试代码和开发代码分离，而新建的测试代码目录里面的包名最好和开发代码的包名相同，这样编译出来的东西会在同一个 bin 目录下，避免 Import 包名的问题。



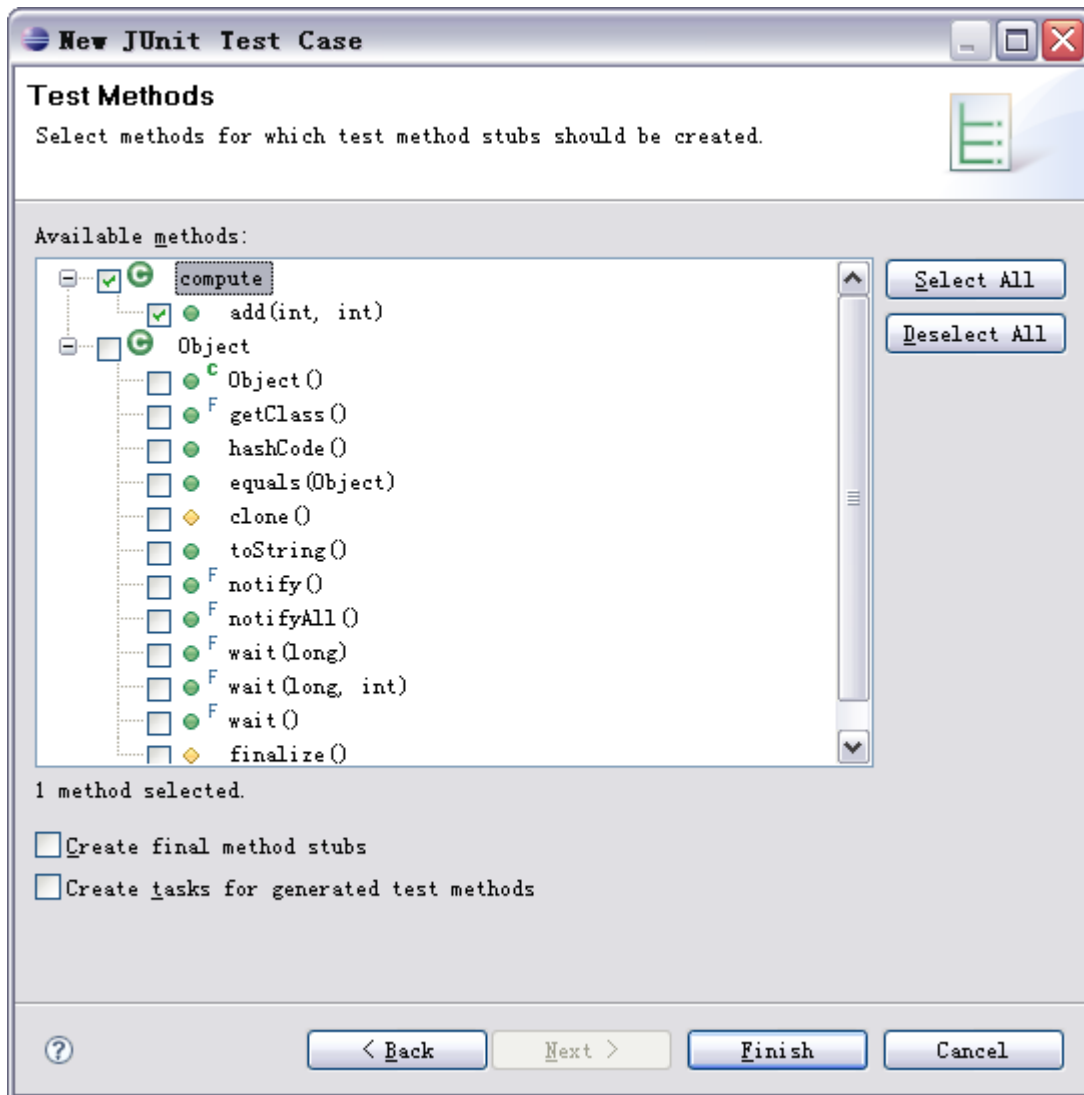
这里我们在测试代码的包上新建一个 Junit Test Case，在弹出的界面里面选择添加 Class under test，将前面新建的 compute 类加入



然后编写测试用例的名称叫做 addjunit3



这里确保我们选择的是 Junit3 并且默认的 setUp 和 tearDown 是选中的，点击 next



这里把我们要测试的 add 方法选中点击 Finish，即可将该方法的 junit3 测试代码框架生成。代码生成后我们会看到在测试代码的包下多了个 addjunit3.java 文件，代码如下

```
package com.cloud.chen;
```

```
import junit.framework.TestCase;
```

```
public class addjunit3 extends TestCase {
```

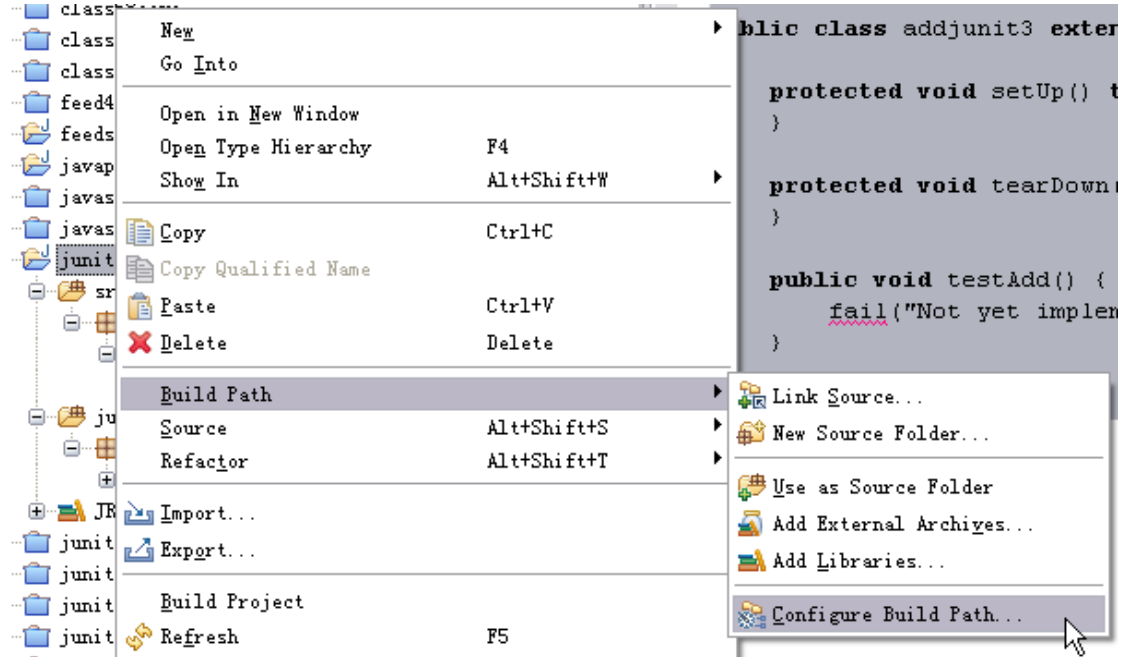
```
    protected void setUp() throws Exception {  
    }
```

```
    protected void tearDown() throws Exception {  
    }
```

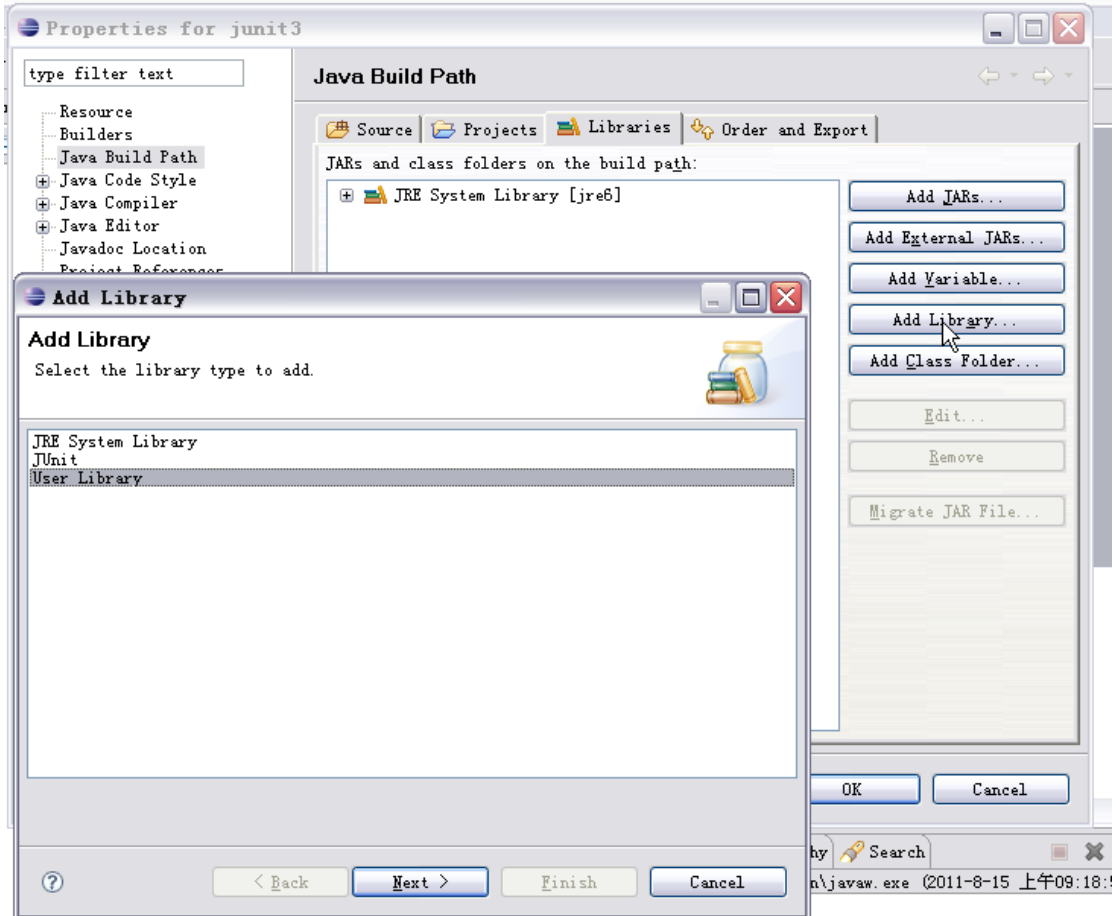
```
    public void testAdd() {  
        fail("Not yet implemented");  
    }
```

```
}  
  
}
```

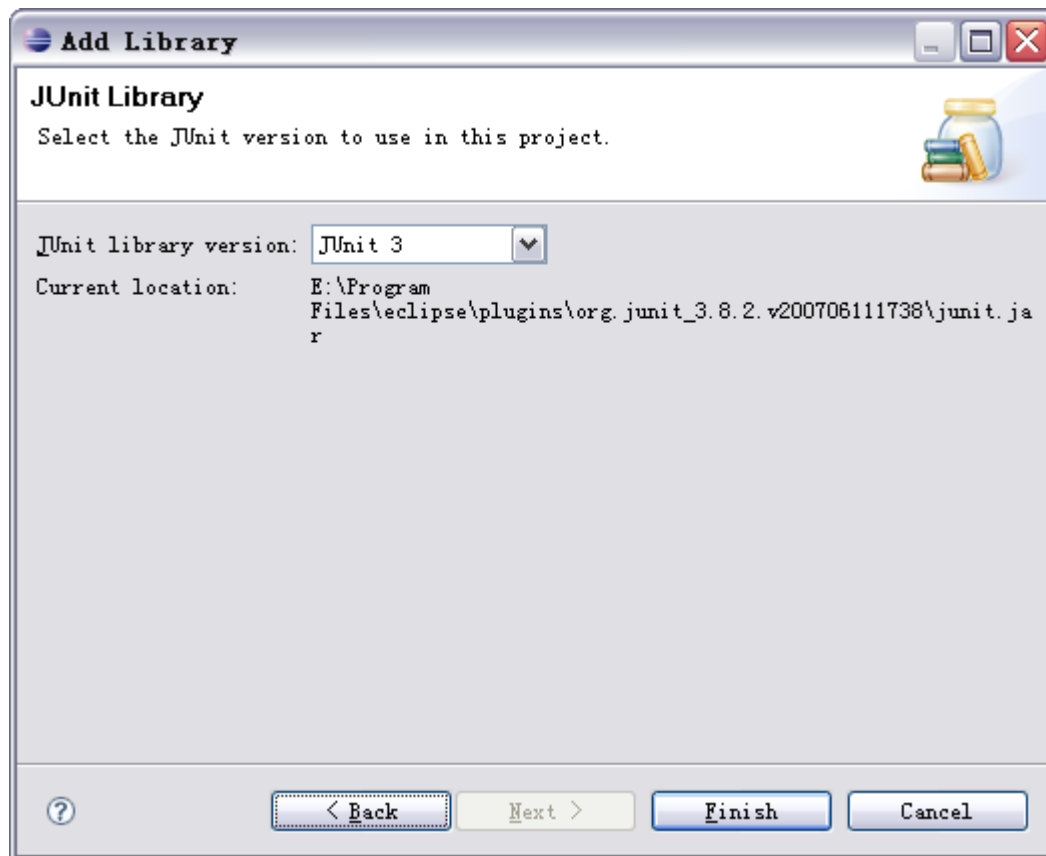
当代码添加后会发现代码左侧会有错误提示,这是因为我们还没加入 junit3 的 JAR 包导致的,接着我们添加对应的 JAR 包。



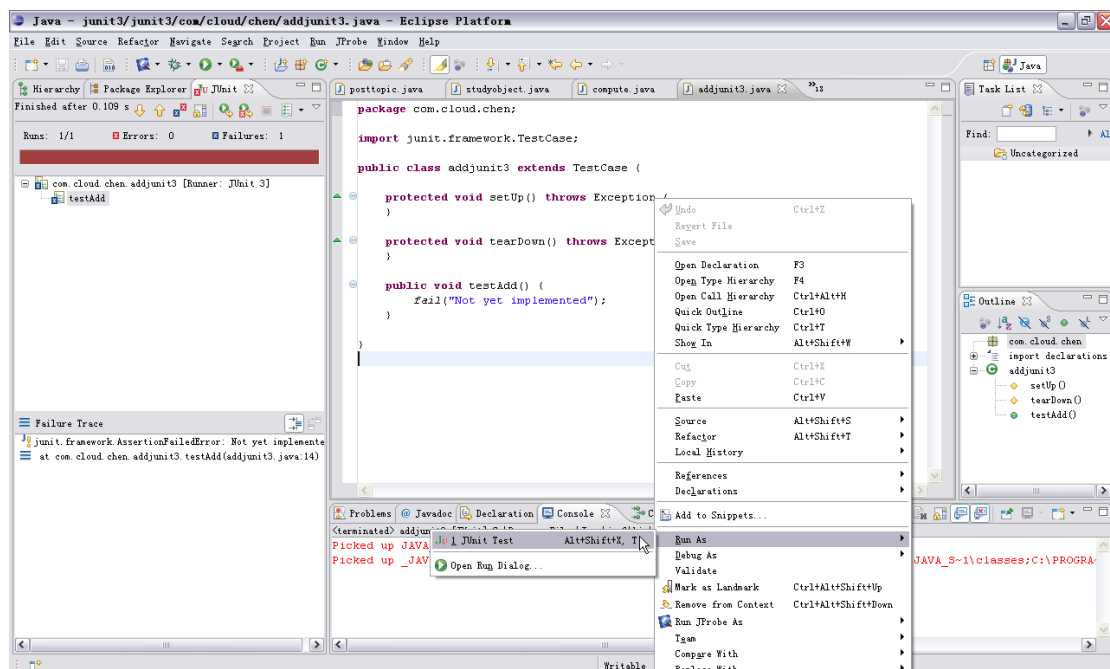
在项目上选择右键 Build Path 里面的 Configure Build Path



弹出的编译窗口中点击 Add Library，然后我们选择 Junit 库需要添加



这里选择 Junit 3 的版本 Finish 完成添加，再回到代码相关的错误提示就消失了。接着我们可以直接在代码里面右键选择运行为 junit 看看效果。



当我们运行 Junit Test 后会在左侧出现 Junit 测试进度条，得到一个失败的结果，这是因为在我们的测试代码里面是设置了 fail 的结论的。

接着我们开始编写我们的测试代码，testAdd 是我们的一个测试方法，在 Junit3 中规定如果

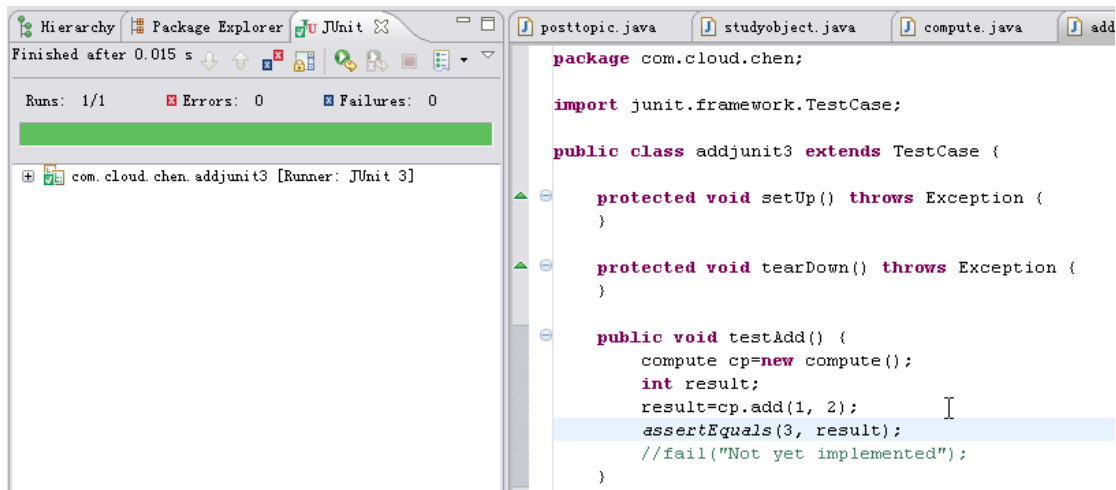
是一个测试方法/用例那么必须要遵守以下 4 点

1. 方法 `void` 无返回
2. `test` 开头的的方法名
3. 方法没有输入参数
4. 测试的类必须继承于 `TestCase`

我们接着在 `testadd` 中完成我们对 `add` 方法的调用

```
public void testAdd() {
    compute cp=new compute();
    int result;
    result=cp.add(1, 2);
    assertEquals(3, result);
    //fail("Not yet implemented");
}
```

这里我编写了实例化 `compute` 类的代码,写了数据驱动(输入 1,2),写了获得返回值 `result`。这里多了一个 `assertEquals` 的东西,这个东西叫做断言(断言有很多种这里就只用最简单的相等断言),也就是说判断期望结果和实际结果是不是相同我们可以通过这个方法来实现,我期望的结果是 3,实际结果是 `result` 变量的值。接着我们可以运行这个测试用例。

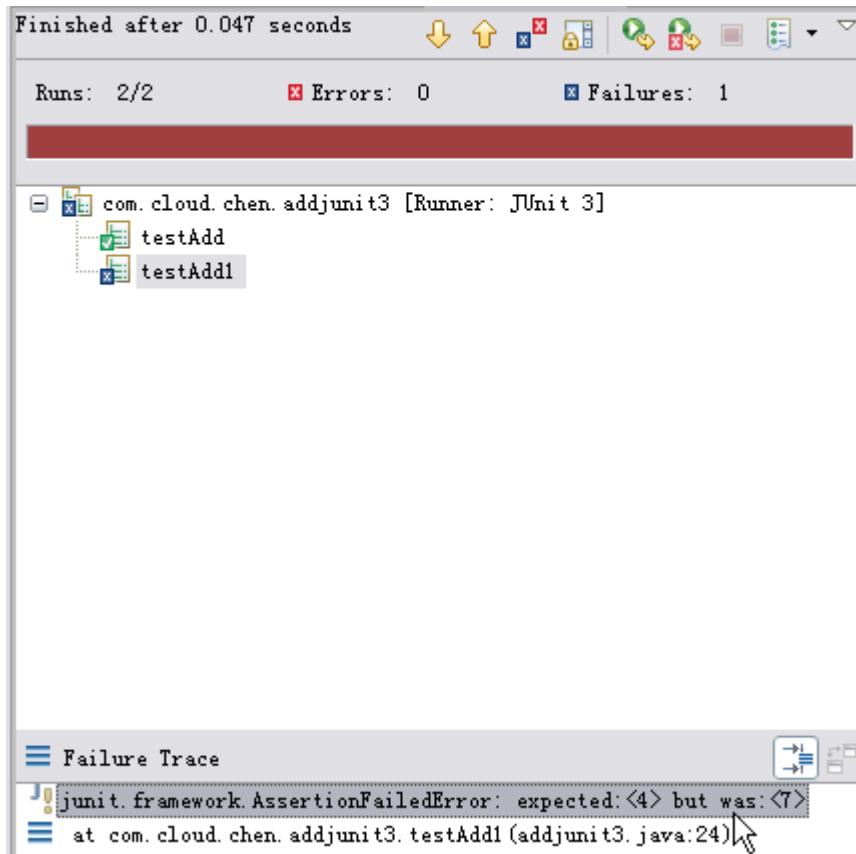


现在的运行结果就是 `PASS` 了,说明该用例的驱动执行后,被测对象的返回和我的期望值相同。

接着我们再添加一个用例,复制上面的 `testAdd` 方法,略作修改

```
public void testAdd1() {
    compute cp=new compute();
    int result;
    result=cp.add(5, 2);
    assertEquals(4, result);
    //fail("Not yet implemented");
}
```

我们叫做 `testAdd1`,这里我故意把期望结果给写错,看看当用例执行失败的时候会有什么效果。



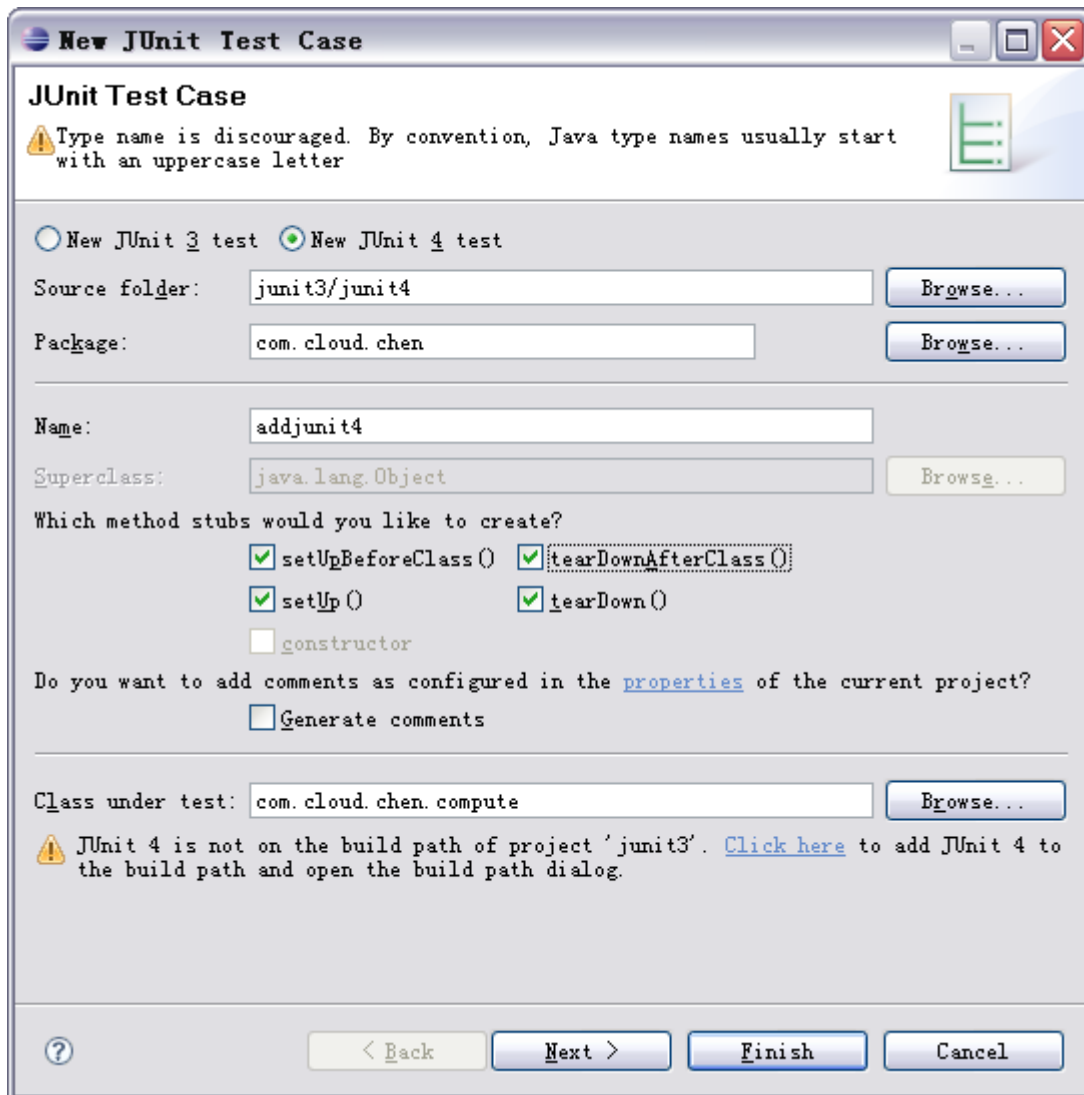
然后我们会看到 `testAdd1` 的执行是失败的，而且错误的原因是我的期望值是 4，但是实际结果却是 7。

`setUp()` 和 `tearDown()` 这两个东西是用来在每个用例执行的前后初始化操作的。`setUp` 当每个用例方法执行前都会被执行一次，`tearDown` 在每个用例方法执行后也会被执行一次，在 Junit3 中是无法为整个测试方法设定一个整体初始化一次和整体释放一次的操作的，到了 Junit4 里面才有了类似的方法。

更多的内容在 Junit4 里面实现。

2.1.2 junit4

Junit4 作为新版本的 Junit 在很多地方做了调整，我们按照前面的方式新建一个 `addjunit4` 的测试方法，添加的时候注意把测试的环境设置为 Junit4



当我们选择 junit4 的时候会多出来 setUpBeforeClass()和 tearDownAfterClass()两个新的方法，这就是更大一级的全体初始化一次和释放一次的新功能。

最后生成的代码如下

```
package com.cloud.chen;
```

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

```
import org.junit.After;
```

```
import org.junit.AfterClass;
```

```
import org.junit.Before;
```

```
import org.junit.BeforeClass;
```

```
import org.junit.Test;
```

```
public class addjunit4 {
```

```
    @BeforeClass
```

```
    public static void setUpBeforeClass() throws Exception {
```

```

}

@AfterClass
public static void tearDownAfterClass() throws Exception {
}

@Before
public void setUp() throws Exception {
}

@After
public void tearDown() throws Exception {
}

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

```

在 Junit4 中不再规定测试方法名一定要通过 test 开头了，而是通过 @Test 修饰符来实现了。我们同样按照 Junit3 中的例子完成两个测试用例。代码如下

```

@Test
public void testAdd() {
    compute cp=new compute();
    int result;
    result=cp.add(1, 2);
    assertEquals(3, result);
    //fail("Not yet implemented");
}

@Test
public void testAdd1() {
    compute cp=new compute();
    int result;
    result=cp.add(5, 2);
    assertEquals(4, result);
    //fail("Not yet implemented");
}

```

测试执行，测试的结果和 Junit3 没有区别。

基于 Junit 的参数化

在 Junit 中参数化也是默认支持的，使用方式如下，这里需要使用构造函数和参数化生成器。首先我们需要在类外加载参数化生成器

```

@RunWith(Parameterized.class) // 使用参数化运行器

```


然后为测试方法添加属性及构造函数

```
public int x;
    public int y;
    public int result;

    // 构造函数
    public junit4(int x,int y, int result){
        this.x = x;
        this.y = y;
        this.result=result;
    }
```

这里的构造函数需要和测试类名相同。

接着编写数据初始化的方法

```
@Parameters    // 指定该方法为参数生成器
@SuppressWarnings("unchecked") // 忽略警告信息
public static Collection getParamters() {
    // 输入值与结果必须与构造函数定义一一对应
    Object[][] object = {{2,1,1}, {2,3,-1},{3,7,-4}};
    return Arrays.asList(object);
}
```

在这里我们声明了一个对象类型的二维数组,里面存放了测试用例中的两个输入和一个期望结果。好了这样我们就完成参数化的准备工作了,使用方法就是调用属性。

```
@Test
public void Getresult() {
    //fail("Not yet implemented");
    bcl compute=new bcl();
    compute.sub(this.x, this.y);
    int aresult=compute.getresult();
    assertEquals(this.result, aresult);
}
```

在测试方法中使用 this 关键字来读取属性, Junit 会自动完成对象数组的依次遍历最后完成测试。

```
package com.cloud.chen;

import static org.junit.Assert.*;

import java.util.Arrays;
import java.util.Collection;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
```

```

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

@RunWith(Parameterized.class) // 使用参数化运行器
public class addjunit4 {

    public int x;
    public int y;
    public int result;

    // 构造函数
    public addjunit4(int x,int y, int result){
        this.x = x;
        this.y = y;
        this.result=result;
    }

    @Parameters // 指定该方法为参数生成器
    @SuppressWarnings("unchecked") // 忽略警告信息
    public static Collection getParamters() {
        // 输入值与结果必须与构造函数定义一一对应
        Object[][] object = {{2,1,1}, {2,3,-1},{3,7,-4}};
        return Arrays.asList(object);
    }

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test

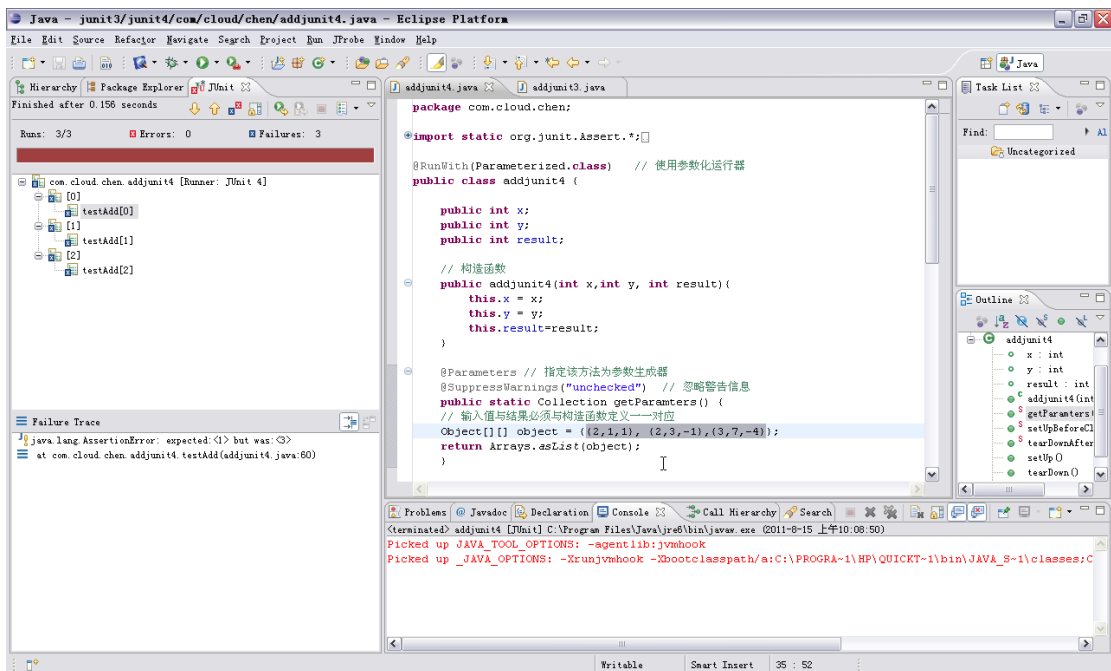
```

```

public void testAdd() {
    compute cp=new compute();
    int cpreresult;
    cpreresult=cp.add(this.x, this.y);
    assertEquals(this.result, cpreresult);
    //fail("Not yet implemented");
}
// @Test
// public void testAdd1() {
//     compute cp=new compute();
//     int cpreresult;
//     cpreresult=cp.add(5, 2);
//     assertEquals(4, cpreresult);
//     //fail("Not yet implemented");
// }
}

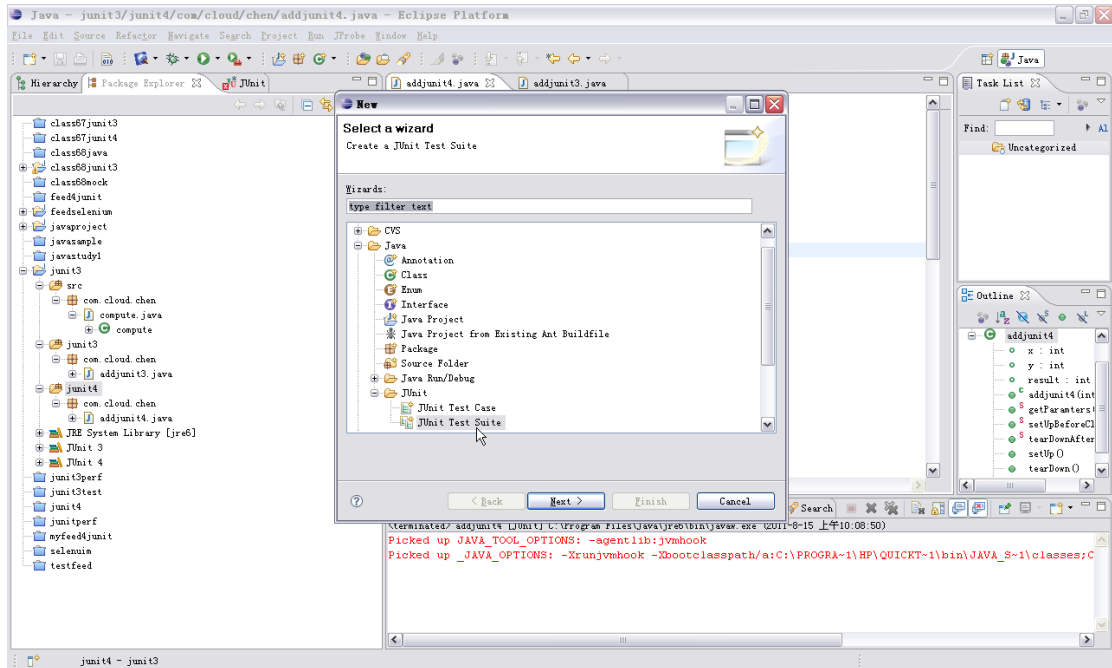
```

过去的 testAdd1 方法已经没有必要存在了，因为通过参数化我们可以在一个测试方法里面完成多个不同数据的操作了。执行用例，所有用例都是失败结果，这是因为我们在用例里面的数据本来就是错误的导致的。



测试 Suite

虽然有了参数化，但是我们有很多方法需要测试，这样会带来大量的测试类，如果我每次都需要手工一个个执行太麻烦了，所以 Junit 提供了 Suite 的概念，可以把多个测试类放在一起共同执行。



我们在 junit4 这个代码目录上选择新建一个 Junit Test Suit，点击 Finish 后会生成一个 Alltest 的类。

```
package com.cloud.chen;
```

```
import junit.framework.Test;
```

```
import junit.framework.TestSuite;
```

```
public class AllTests {
```

```
    public static Test suite() {
```

```
        TestSuite suite = new TestSuite("Test for com.cloud.chen");
```

```
        //$JUnit-BEGIN$
```

```
        //$JUnit-END$
```

```
        return suite;
```

```
    }
```

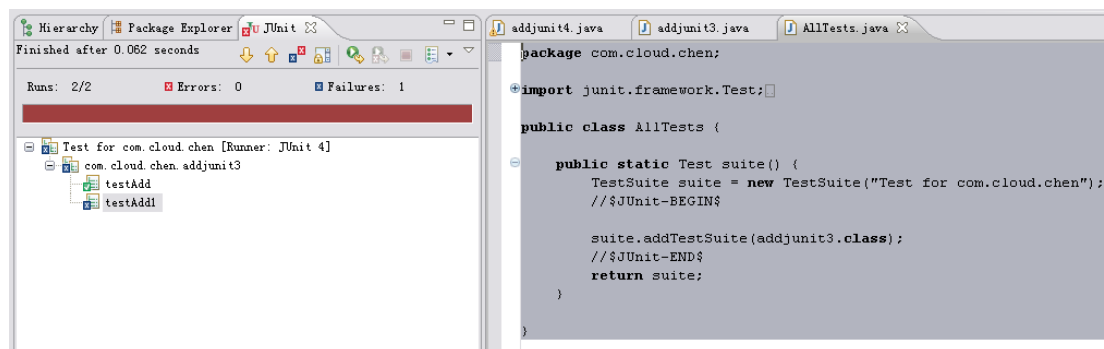
```
}
```

在这里需要添加我们要调用的测试类了。

使用 suite 类下的 addTestSuite 方法，将我们需要加载的测试类编译后的 class 加入

```
suite.addTestSuite(addjunit3.class);
```

现在我们在 Alltest 里面执行用例 Suite 就可以完成对应用例集的执行操作了。



在 suite 里面加载 junit4.class 遇到了点问题，错误暂时还没解决，错误如下：
junit.framework.AssertionFailedError: Class com.cloud.chen.addjunit4 has no public constructor
TestCase(String name) or TestCase()

```
at junit.framework.Assert.fail(Assert.java:47)
at junit.framework.TestSuite$1.runTest(TestSuite.java:97)
at junit.framework.TestCase.runBare(TestCase.java:134)
at junit.framework.TestResult$1.protect(TestResult.java:110)
at junit.framework.TestResult.runProtected(TestResult.java:128)
at junit.framework.TestResult.run(TestResult.java:113)
at junit.framework.TestCase.run(TestCase.java:124)
at junit.framework.TestSuite.runTest(TestSuite.java:232)
at junit.framework.TestSuite.run(TestSuite.java:227)
at junit.framework.TestSuite.runTest(TestSuite.java:232)
at junit.framework.TestSuite.run(TestSuite.java:227)
at org.junit.internal.runners.OldTestClassRunner.run(OldTestClassRunner.java:76)
at
org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestReference.java:38)
at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)
at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:460)
at
org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:673)
at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:386)
at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:196)
```

上面所用到的所有代码见 junit.rar

2.2 feed4junit

Feed4junit 是一个第三方为 junit 提供数据驱动的工具，在前面的例子中使用 Junit 自己的数据驱动略微复杂，而希望扩展单元测试所使用的数据例如（文本文件，数据库），就要靠 feed4junit 了。

我们先按照前面的方式新建一个被测方法及类，然后我们需要把 feed4junit 的 JAR 包添加了，这里推荐在代码目录下新建一个 lib 目录用来存放 JAR 包，另外新建一个 source 目录用来存

云层 <http://www.51testing.com/?104>

放过会需要使用到的参数文件。接着将所有的 feed4junit 的 JAR 包导入。然后对被测方法生成 junit4 的测试用例。

使用 Feed4junit 是比较简单的，首先在 Junit4 的测试类前添加

`@RunWith(Feeder.class)`

用来引入 feeder.class

接着在测试用例的 `@test` 后加载测试用例数据

`@Source("source\\compute.csv")`

这里是指加载一个 source 目录下的 compute.csv 文件，文件的格式如下

x,y,result

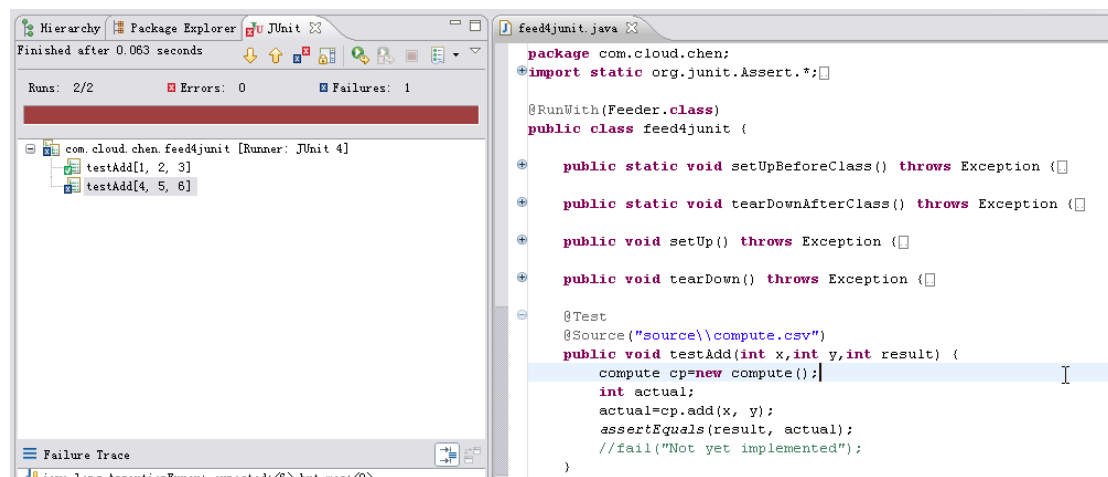
1,2,3

4,5,6

接着需要将测试方法的没有输入参数改为输入 x,y,result (这里的名字和被读文件中的属性需要保持一致)

```
public void testAdd(int x,int y,int result) {
    compute cp=new compute();
    int actual;
    actual=cp.add(x, y);
    assertEquals(result, actual);
    //fail("Not yet implemented");
}
```

将对应的包 Import 后即可运行



可以看到 csv 文件中的两个记录都被运行了，第一个用例是执行成功的，而第二个执行是执行错误的。

Feed4junit 除了可以读取文本文件以外还能支持数据库等。

该案例代码参考

feed4junit-sample.rar

2.3 Mock

Mock 也叫做桩，我们在做单元测试的时候经常会遇到被测对象可以被调用但是它的某个方法或者它调用的别的方法还没有实现，导致我们无法有效断言，从而导致单元测试的无法执行。这个时候我们就需要写一个桩来替代哪些没完成的访问对象了。

JMock 这样的扩展可以帮助我们完成桩的设计，这里我们没有采用这种方式而是使用更本质的基于多态的方式来解决桩的问题，因为就算是 Jmock 也是基于多态的方式来完成桩的实现。

@这里需要回顾一下前面多态的概念

如果需要完成桩，那么对被测对象是有一定的要求的，就是被测对象要使用多态的方式编写，通过接口变量的 this.来完成具体的逻辑。

我们先新建一个接口并且完成这个接口的实现。

接口设计为：

```
package com.cloud.chen;

public interface Icompute {
    public int add(int x,int y);
}
}
```

实现为 Simplecompute 类：

```
package com.cloud.chen;

public class SimpleCompute implements Icompute {

    @Override
    public int add(int x, int y) {
        // TODO Auto-generated method stub
        return 0;
    }
}
}
```

这里我们会发现 SimpleCompute 这个类下的 add 方法还没有被实现。

编写使用 SimpleCompute 这个类下 add 方法的代码。

```
package com.cloud.chen;

public class Analysis {
    public Icompute icp;
    public int analysissum(int x,int y)
    {
```

```

        int sum;
        sum=2*icp.add(x,y);
        return (sum);
    }
}

```

这里在编写的时候我们使用了多态的写法，调用的是 icp 接口变量下的 add 方法。

接着我们为 Analysis 下的 analysissum 方法编写我们的单元测试

```

@Test
public void testAnalysissum() {
    Analysis as=new Analysis();
    SimpleCompute sc=new SimpleCompute();
    as.icp=sc;
    int actual;
    int expected=7;
    actual=as.analysissum(3, 4);
    assertEquals(expected, actual);

    //fail("Not yet implemented");
}

```

执行以后肯定是失败的因为 add 方法还没被实现。接着我们仍然从接口上实现一个新的东西，这个东西就是我们的 Mock，在该 Mock 中我们完成具体的处理或者是定死的返回（这个需要根据你的测试用例或者返回业务来决定的）

```

package com.cloud.chen;

```

```

public class SCmock implements Icompute {

    @Override
    public int add(int x, int y) {
        // TODO Auto-generated method stub
        return (x+y);
        //return 0;
    }

}

```

接着我们就可以利用多态来完成对 Analysis 下的 analysissum 方法的测试了。
将测试代码修改为

```

@Test
public void testAnalysissum() {
    Analysis as=new Analysis();
    //SimpleCompute sc=new SimpleCompute();
    SCmock sc=new SCmock();//开发好的桩
    as.icp=sc;
}

```



```

    int actual;
    int expected=14;
    actual=as.analysisissum(3, 4);
    assertEquals(expected, actual);

    //fail("Not yet implemented");
}

```

这样我们利用多态将原本调用没有开发完的 SimpleCompute 下的 add 变为了 SCMock 下的 add，从而实现了测试桩的替换。

参考代码见 mock.rar

2.4 Junitperf

最后我们来看一下如何做白盒级别的性能测试，我们希望知道一个方法在被多线程调用多次的情况下时间的响应情况，这个时候我们就可以使用 junitperf 来解决这个问题。

首先我们还是来新建一个被测的类和方法。

```

package com.cloud.chen;

public class compute {
    public int add(int x,int y)
    {
        return (x+y);
    }
    public double sum(int x)
    {
        double result=0;
        int i;
        // try {
        //     Thread.sleep(2000);
        // } catch (InterruptedException e) {
        //     // TODO Auto-generated catch block
        //     e.printStackTrace();
        // }
        for (i=0;i<=x;i++)
        {
            result=result+i*i;
            for (int t=0;t<=999999;t++)
            {
                result=result+t;
            }
        }
    }
}

```

```

        return(result);
    }
}

```

这里我们新建了两个方法一个是简单的加法一个是复杂点得循环累加,在代码中有一段注释的内容是一个浪费时间的代码 `sleep(2000)`让线程等 2 秒钟,可以对后面的性能测试结果提供验证支持。`Perf` 是要在 `Junit` 的基础上运行的,所以我们要先给这两个方法写 2 个用例。

```

@Test
public void testAdd() {
    compute cp=new compute();
    int expected=10;
    int actual=cp.add(3, 7);
    assertEquals(expected, actual);
    //fail("Not yet implemented");
}

@Test
public void testSum() {
    compute cp=new compute();
    double expected=35499964616795.0;
    double actual=cp.sum(70);
    assertEquals(expected, actual);
    //fail("Not yet implemented");
}

```

然后要做点特别的事情,先给整个 `computetest` 方法加个继承与 `TestCase`,然后为测试方法添加一个构造函数

```

public class computetest extends TestCase{

    public computetest(String string) {
        // TODO Auto-generated constructor stub
        super(string);
    }
}

```

接着我们新建 `lib` 目录并且把 `perf` 的 `JAR` 包都放进去,然后添加到项目中。新建一个类并且带主函数,然后写一个静态的方法来实现调用测试用例并且统计时间。

```

public static Test suite() {

    long maxElapsedTime = 2000;

    Test testCase = new computetest("testAdd");
    Test timedTest = new TimedTest(testCase, maxElapsedTime);
}

```

```
        return timedTest;
    }
}
```

这里我们制订了一个最长时间为 2 秒的变量，希望运行的时候该用例（testAdd）的执行时间不要超过 2 秒，超过了就不执行了，然后将测试结果返回。

调用 perf 查看执行时间

在主函数内添加命令行启动 Junit 的代码

```
public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
}
```

然后作为 java application 执行，最后我们就能在 console 内看到执行的结果

```
.TimedTest (WAITING): testAdd(com.cloud.chen.computetest): 15 ms
```

```
Time: 0.015
```

```
OK (1 test)
```

然后我们切换到复杂的 testSum 上再看看执行时间

```
.TimedTest (WAITING): testSum(com.cloud.chen.computetest): 110 ms
```

```
Time: 0.11
```

```
OK (1 test)
```

负载测试

这里我们要调用新的 Loadtest，新建 Loadsutie 静态类

```
public static Test Loadsuite(){
    int users = 10; //模拟负载用户数
    int iterations = 10; //每个用户数的调用次数
    long maxElapsedTime = 20000; //超时上限
    //Timer timer = new ConstantTimer(1000);
    Timer timer = new ConstantTimer(100); //线程启动间隔时间100毫秒
    Test testCase = new computetest("testSum");
    Test loadTest = new LoadTest(testCase, users, iterations, timer);
    Test timedTest = new TimedTest(loadTest, maxElapsedTime);
    return timedTest;
}
```

在主函数中添加执行

```
junit.textui.TestRunner.run(Loadsuite());
```

执行后的结果为

```
.....
```

```
.....  
.....TimedTest (WAITING): LoadTest (NON-ATOMIC):  
ThreadedTest: testSum(com.cloud.chen.computetest) (repeated): 5219 ms
```

Time: 5.219

OK (100 tests)

最终代码

```
package com.cloud.chen;  
  
import junit.framework.Test;  
  
import com.clarkware.junitperf.ConstantTimer;  
import com.clarkware.junitperf.LoadTest;  
import com.clarkware.junitperf.TimedTest;  
import com.clarkware.junitperf.Timer;  
  
public class perfrun {  
    public static Test suite() {  
  
        long maxElapsedTime = 2000;  
        // Test testCase = new computetest("testAdd");  
        Test testCase = new computetest("testSum");  
        Test timedTest = new TimedTest(testCase, maxElapsedTime);  
  
        return timedTest;  
    }  
  
    public static Test Loadsuite(){  
        int users = 10;//模拟负载用户数  
        int iterations = 10;//每个用户数的调用次数  
        long maxElapsedTime = 20000; //超时上限  
        //Timer timer = new ConstantTimer(1000);  
        Timer timer = new ConstantTimer(100);//线程启动间隔时间 100 毫秒  
        Test testCase = new computetest("testSum");  
        Test loadTest = new LoadTest(testCase, users, iterations, timer);  
        Test timedTest = new TimedTest(loadTest, maxElapsedTime);  
        return timedTest;  
    }  
  
    public static void main(String[] args) {  
        junit.textui.TestRunner.run(suite());  
    }  
}
```

```
        junit.textui.TestRunner.run(Loadsuite());
    }
}
```

上述代码访问 junitperfsample.rar

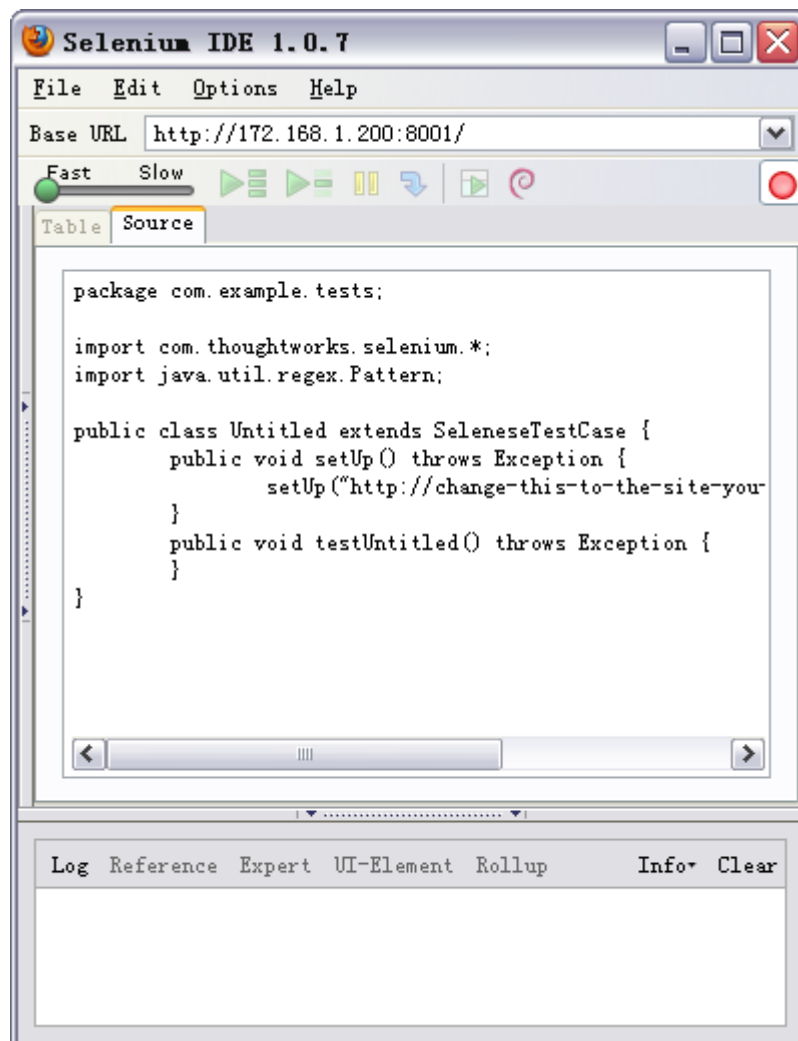
3 自动化测试

3.1 Selenium

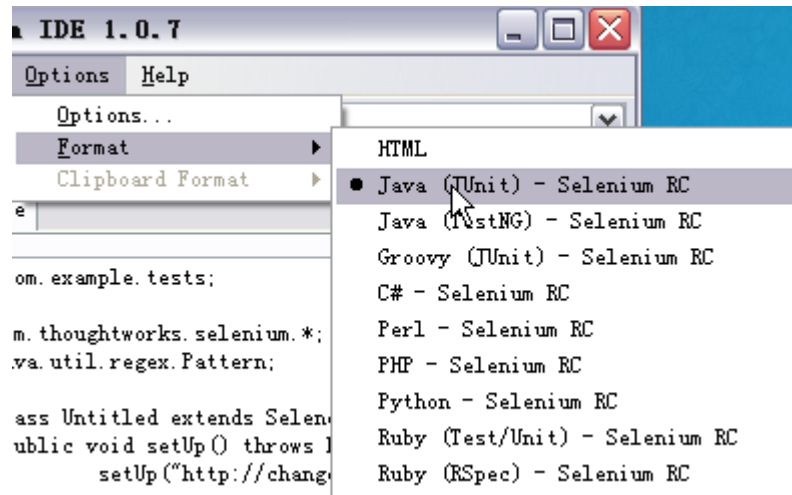
Selenium 是一个开源的浏览器自动化测试框架

3.2 seleniumide 插件

我们安装 firefox3.6 并在上面安装 seleniumIDE1.0.7 插件，将插件设置为录制状态，并且设置 BaseURL 为你要录制系统的根地址



并在 Option 内把录制出来的代码格式设置为 JAVA junit



然后使用 IDE 插件录制用户登录发帖的操作,得到对应的代码
package com.example.tests;

```
import com.thoughtworks.selenium.*;  
import java.util.regex.Pattern;
```

```
public class Untitled extends SeleniumTestCase {  
    public void setUp() throws Exception {  
        setUp("http://change-this-to-the-site-you-are-testing/", "*chrome");  
    }  
    public void testUntitled() throws Exception {  
        selenium.open("/index.aspx");  
        selenium.click("link=[登录]");  
        selenium.type("username", "admin");  
        selenium.type("password3", "51testing");  
        selenium.click("login");  
        selenium.click("link=默认版块");  
        selenium.waitForPageToLoad("30000");  
        selenium.click("//img[@alt='发新话题']");  
        selenium.type("title", "selenium 发帖");  
        selenium.type("e_textarea", "单元测试一条龙之 selenium 自动化测试框架 by 云层");  
        selenium.click("postsubmit");  
    }  
}
```

3.3 selenium rc

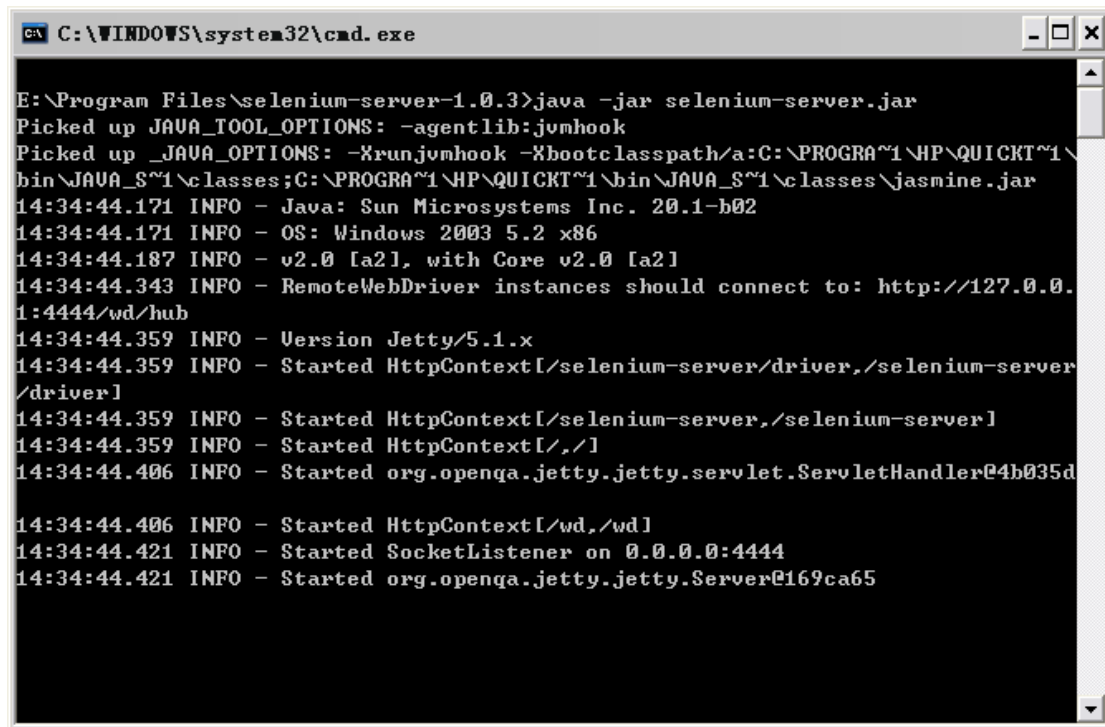
如果我们要使用 junit 来运行前面的代码,那么我们需要先部署一个 selenium remote core 远程核心。

将 selenium-remote-control-1.0.3.zip 解压, 并且编写批处理

云层 <http://www.51testing.com/?104>

java -jar selenium-server.jar

使用这样的方式将 selenium-server-1.0.3 目录下的 selenium rc 启动



```
C:\WINDOWS\system32\cmd.exe
E:\Program Files\selenium-server-1.0.3>java -jar selenium-server.jar
Picked up JAVA_TOOL_OPTIONS: -agentlib:jvmti
Picked up _JAVA_OPTIONS: -Xrunjvmti -Xbootclasspath/a:C:\PROGRA~1\HP\QUICKT~1\bin\JAVA_S~1\classes;C:\PROGRA~1\HP\QUICKT~1\bin\JAVA_S~1\classes\jasmine.jar
14:34:44.171 INFO - Java: Sun Microsystems Inc. 20.1-b02
14:34:44.171 INFO - OS: Windows 2003 5.2 x86
14:34:44.187 INFO - v2.0 [a2], with Core v2.0 [a2]
14:34:44.343 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
14:34:44.359 INFO - Version Jetty/5.1.x
14:34:44.359 INFO - Started HttpContext[/selenium-server/driver, /selenium-server/driver]
14:34:44.359 INFO - Started HttpContext[/selenium-server, /selenium-server]
14:34:44.359 INFO - Started HttpContext[/, /]
14:34:44.406 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@4b035d
14:34:44.406 INFO - Started HttpContext[/wd, /wd]
14:34:44.421 INFO - Started SocketListener on 0.0.0.0:4444
14:34:44.421 INFO - Started org.openqa.jetty.jetty.Server@169ca65
```

启动后会看到 4444 端口正在被监听

3.4 使用 Junit 调用 selenium rc 实现自动化

这里没有什么特别的地方，将 selenium-remote-control-1.0.3.zip 文件下的 selenium-java-client-driver-1.0.1 目录中的相关 Jar 都添加到项目中，然后就可以直接使用访问 selenium 的对象及方法了。

在测试方法中我们可以先实例化一个 selenium 对象

```
Selenium browser= new DefaultSelenium("localhost", 4444,
"*firefox", "http://172.168.1.200:8001/index.aspx");
```

这里说明了启动的浏览器类型，BaseURL 的路径，以及 SeleniumRC 的地址及端口

接着可以把 IDE 录制的代码略作调整添加到测试用例中去

```
browser.start();
browser.open("/index.aspx");
browser.click("link=[登录]");
Thread.sleep(2000);
browser.type("username", "admin1");
browser.type("password3", "51testing");
browser.click("login");
Thread.sleep(5000);
```

```

        browser.click("link=默认版块");
        browser.waitForPageToLoad("30000");
        browser.click("//img[@alt='发新话题']");
        Thread.sleep(2000);
        browser.type("title", "selenium发帖");
        browser.type("e_textarea", "单元测试一条龙之selenium自动化测试框架
by 云层");
        browser.click("postsubmit");
        browser.waitForPageToLoad("30000");
        System.out.println(mystr);

        browser.close();

```

这里我添加了一些等待的时间，为了避免页面刷新慢，导致对象无法找到并操作。执行用例，发现自动出现 Firefox 浏览器并且完成相关操作

3.5 加载 feed4junit 完成数据驱动

按照 feed4junit 的特点在测试用例上略作调整。

```

@RunWith(Feeder.class)
public class posttopic {

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    @Source("resource\\compute.csv")
    public void testUntitled(int x, int y, int result) throws Exception
    {
        //selenium 代码 略
    }
}
即可

```

3.6 加入页面内容断言

Selenium 有好几种断言，这边用最简单的两个

```
String mystr=browser.getTitle();
```

通过 getTitle()可以获得页面的标题

```
browser.isTextPresent("退出")
```


通过检查页面内是否存在“退出”这个元素，返回 Bool

我们在代码中将断言和 Junit 断言结合即可完成自动化验证
package feedselenium.org;

```
import junit.framework.Assert;
```

```
import org.databene.benerator.anno.Source;
```

```
import org.databene.feed4junit.Feeder;
```

```
import org.junit.After;
```

```
import org.junit.Before;
```

```
import org.junit.Test;
```

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

```
import com.thoughtworks.selenium.DefaultSelenium;
```

```
import com.thoughtworks.selenium.Selenium;
```

```
@RunWith(Feeder.class)
```

```
public class posttopic {
```

```
    @Before
```

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

```
    }
```

```
    @After
```

```
    public void tearDown() throws Exception {
```

```
    }
```

```
    @Test
```

```
    @Source("resource\\compute.csv")
```

```
public void testUntitled(int x, int y, int result) throws Exception {  
    Selenium browser= new DefaultSelenium("localhost", 4444,  
    "*firefox", "http://172.168.1.200:8001/index.aspx");
```

```
    browser.start();
```

```
    browser.open("/index.aspx");
```

```
    browser.click("link=[登录]");
```

```
    Thread.sleep(2000);
```

```
    browser.type("username", "admin1");
```

```
    browser.type("password3", "51testing");
```

```
    browser.click("login");
```

```
    Thread.sleep(5000);
```

```
    Assert.assertTrue(browser.isTextPresent("退出"));
```

```
    browser.click("//tbody[@id='forum2']/tr/th/h2/a");
```

```
    browser.waitForPageToLoad("30000");
```

```
browser.click("//img[@alt='发新话题']");
Thread.sleep(2000);
browser.type("title", "selenium发帖");
browser.type("e_textarea", "测试发帖"+x+y+result);
browser.click("postsubmit");
browser.waitForPageToLoad("30000");
String mystr=browser.getTitle();
// Assert.assertTrue(browser.isTextPresent("admin234234"));
System.out.println(mystr);

browser.close();
browser.stop();
}
}
```

Selenium 这段内容由于时间关系写的很简洁，这块东西需要读者对 selenium 有点基础才能明白。

Selenium 部分代码见 feedselenium.rar，不包含服务器 selenium RC。