# 1 Getting Started
# 起步

敏捷 Java 前半部分的课程围绕着一个学生信息系统的开发展开。您将不会去构建一个完整的系统，但是您会去完成多个子系统，这些子系统是完整系统的一部分。

学生信息系统涵盖了学校运营的多个方面，包括注册、年级、课程表、收费、记录等。

本课内容包括：

- 创建一个简单的 Java 类
- 创建一个测试类来执行这个 Java 类
- 使用 JUnit 框架
- 学习构造函数
- 重构您所写的代码

这一节我会讲得非常细，尽可能清晰地讲述 TDD 的每一个步骤。假定以后的课程您会按照 TDD 的流程来编写正确的测试和代码。

## 测试

TDD 意味着您不仅需要为每一段代码编写测试用例，而且意味着测试优先。测试用例用来定义代码需要做什么。在完成相应的代码之后，运行测试用例来保证代码确实符合测试用例的规定。



如图 1.1 StudentTest 将创建 Student 类的对象，发送消息给这些对象，并且证明一旦所有的消息被发送出去，一切都能像预期的那样。因而 StudentTest 类依赖于 Student 类，如图中的有向关联所表达的意思。相反，Student 不依赖于 StudentTest：生产类对为它编写的测试一无所知。

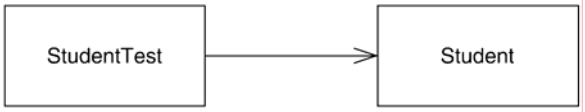



图 1.1 测试类和生产类

## 设计

您基于客户的需求来设计和构建系统。设计过程的一部分是把客户需求转换成为有关该系统将被如何使用的粗略想法或者框架。对于基于 Web 的系统，这意味着设计网页，使其提供应用程序的功能。假如您正在开发中间件，您的中间件将被其它客户端软件调用，并且中间件还会和别的服务器软件交互。这样的话，您就会开始定义其他系统与中间件之间通信的接口。

一开始只是概要的设计，不会涉及到特别多的细节。随着对客户需求的更多了解，您将持续优化和提炼您的设计。同样，当在您的 Java 代码中发现优点和缺点的时候，您将更新您的设计。面向对象开发的强大之处在于它提供了灵活性，这种灵活性使您可以快速地改变设计来拥抱变化。

在对您将要使用的 Java 语言没有完整理解的情况下，去设计上面所说的系统是一件很困难的任务。为了起步，您将构建系统的某些内部组件，这样会使您掌握语言的基础。

学生信息系统主要关于学生，所以您的首要任务是把客观世界的概念抽象成为面向对象的表示。一个候选类也许是 Student。Student 对象包含一些基本的信息，比如学生的姓名，ID 号，年级。您甚至可以先集中在一个更小的概念上：创建一个唯一的学生对象来存储所有学生的姓名。

32

前一段文字左边的书本形状的图标会在本书中反复出现。我将用这个图标来表示需求，或者 *story*，您将在学生信息系统中实现这些需求。这些 story 是对添加到系统中用以满足用户的所有功能的简要描述。您将把这些 story 翻译成详细的以测试的形式实行的规格说明。

## 一个简单的测试

获取学生信息是初始的需求。为了表达这个需求，我们开始创建一个用作测试用例的类。首先，在您的机器上新建一个目录[1]，然后在这个目录里面创建一个文件 StudentTest.java。您可以暂时在这个目录以外保存、编译、执行代码。在编辑器中输入以下代码：

```
public class StudentTest extends junit.framework.TestCase {
}
```

[1] 这一节课程针对 Java 的命令行用法。如果使用 IDE，您需要在“default package”中创建 StudentTest 类。如果提示输入 package 名字，您什么也不要输入。

保存成文件 StudentTest.java。

StudentTest.java 中的两行代码定义了名字叫 StudentTest 的类。在花括号（{和}）之间的所有代码都是 StudentTest 类定义的一部分。

您必须指定该类为 `public` 类型，这样 JUnit 测试框架才能识别它。后面的章节我将对 `public` 进行更深入的讲解。目前，您只需要知道 `public` 关键字可以使您编写的代码与 JUnit 框架协同工作。

代码段 `extends junit.framework.TestCase` 声明 StudentTest 类是另一个名为 junit.framework.TestCase 类的子类。这意味着 StudentTest 将从 junit.framework.TestCase 获得或是继承所有的能力（行为）和数据（属性）。StudentTest 也可以添加自己的行为或者属性。`extends` 子句也使 JUnit 用户接口将 StudentTest 视为含有测试方法的类。

33

图 1.2 中的 UML 类图展示了 StudentTest 和 junit.framework.TestCase 之间的继承关系。现在 StudentTest 同时依赖于 junit.framework.TestCase 和 Student。请记住表示不同依赖关系的箭头有所区别：封闭的箭头表示继承关系。

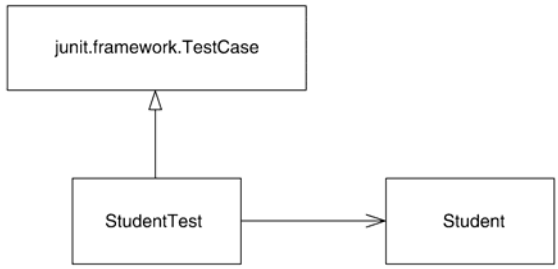



图 1.2 StudentTest 继承自 junit.framework.TestCase

下一步是编译 StudentTest 类。为了编译正确，您必须告诉 Java 在什么地方可以找到 StudentTest 所引用的类。目前只需要能够找到 junit.framework.TestCase。这个类存在于一个 JAR（Java 存档文件）格式的包文件中。这个 JAR 文件中还包含其他很多类，这些类构成了 JUnit 框架。

在扩展课程三中，我会进一步讨论 JAR 文件。在这里，您只需理解怎样告诉 Java 到某个地方找到包含 JUnit 的 JAR 文件。您可以通过设置 *classpath* 实现这一点。

**更多关于 Classpath**

classpath 对于 Java 新手是容易混淆的概念之一。在这里，您只需对它有基本的理解。

classpath 是路径列表，在 Windows 中用分号来分割列表中的不同路径名，在 Unix 中用冒号来分割列表中的不同路径名。编译器和 Java 虚拟机都需要您提供 classpath。路径名可以是 JAR 文件（包含了 class 文件），或者是包含 class 文件的目录。

通过 classpath，您提供给 Java 一个路径列表，当需要加载某个类，Java 就会搜索这些列表。Java 依靠这种机制，从而可以在编译和执行的时候，实现类的动态加载。

34 假如您的 classpath 中有空格，您也许要根据操作系统，给 classpath 加上合适的引号对。

从命令行，您可以在编译源文件的同时，指定 classpath。

```
javac -classpath c:\junit3.8.1\junit.jar StudentTest.java
```

您必须指定 JUnit.jar 文件的绝对路径或者相对路径[2]。您可以在 JUnit 的安装目录里发现这个文件。上面的例子指定了 JUnit.jar 文件的绝对路径。

您需要保证和 StudentTest.java 在同一个目录。

假如您忽略了 classpath, Java 编译器会报告下面的出错信息：

```
StudentTest.java:1: package junit.framework does not exist
public class StudentTest extends junit.framework.TestCase {
                                 ^
1 error
```

IDE 可以让您在当前项目的属性设置中指定 classpath。比如在 Eclipse 中，打开当前项目的属性对话框，在 Java 编译路径下的库 Tab 页中进行设置。

## JUnit

当成功编译了 StudentTest, 您可以在 JUnit 中执行它。JUnit 提供了两个 GUI 界面和一个文本界面。请参考 JUnit 文档来获取详细信息。下面的命令将使用 JUnit 的 junit.awtui.TestRunner 类，在 AWT 界面[3]中执行 StudentTest.class。

35
```
java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest
```

您再次指定了 classpath，这次用的是关键字缩写`-cp`。不止 Java 编译器需要知道 JUnit 类在什么地方，Java 虚拟机也需要找到这些类，这样 Java 虚拟机就可以在运行时随需加载。此外，classpath 含有一个用以表示当前目录的“.”。这样 Java[4]就能够定位 StudentTest.class：如果指定一个目录而不是 JAR 文件，Java 会扫描这个目录并查找必要的 class 文件。

上面的命令也包含了唯一的参数：`StudentTest`。通过传递这个参数给 junit.awtui.TestRunner 类，junit.awtui.TestRunner 就知道要测试的类的名字了。

执行 TestRunner，您会看到如图 1.3 的窗口：

[2] 绝对路径是文件的的完整路径，以驱动器名称或者文件系统的根作为开始。相对路径是某一文件相对当前位置的路径。例如，假如 StudentTest 位于/usr/src/student，JUnit.jar 在/usr/src/JUnit.3.8.1，您可以把相对路径指定为../JUnit/3.8.1/JUnit.jar。

[3] 相对 Swing，AWT 是 Java 提供的更底层的用户界面开发包，AWT 提供了更多的控制和功能。JUnit 中的 AWT 版本十分简单和容易理解。使用 junit.swingui.TestRunner 替代 junit.awtui.TestRunner，可以使用 Swing 版本。使用类 junit.textui.TestRunner，可以使用 text 版本。

[4] 您注意到我经常使用诸如“Java does this”的短语。这是用口语（比如：偷懒）的方式说“Java 虚拟机 does this”，或者”Java 编译器 does this”。您应该能够从上下文判断我是在说 Java 虚拟机还是编译器。

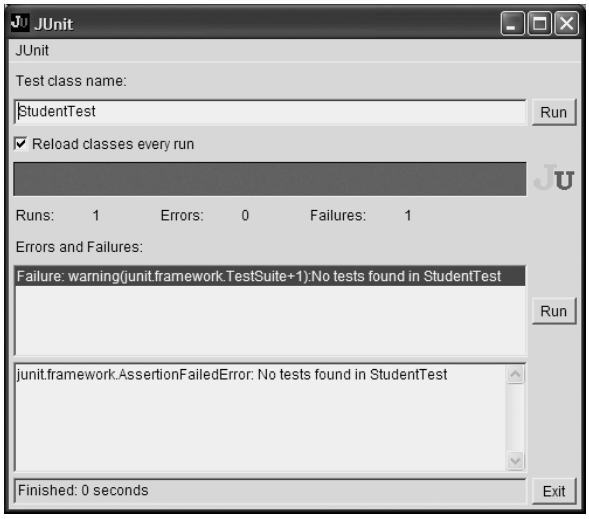



图 1.3 JUnit 执行出错（显示了一根红条）

36

这里，我并不想深入介绍 JUnit 接口。暂时只讨论一部分有关的内容，在适当的时候我会介绍剩下的内容。被测试类的名字，StudentTest，显示在顶部的文本框中。点击右边的 Run 按钮可以执行这个测试。上面的界面显示已经执行了一次测试。如果您点击 Run 按钮，您会看到一根红条[5]从窗口的一边快速到达另一边。

实际上，JUnit 的红条表示发生了错误。红条下方的摘要说明有一个失败。“Erros and Failures”列表解释了发生的错误：在这个例子中，JUnit 抱怨“在 StudentTest 中没有任何测试”。所以作为 TDD 程序员，您的首要任务是检查 JUnit 中的错误，然后快速更正它们。

## 增加一个测试

编辑 StudentTest 类，像下面这样：

```
public class StudentTest extends junit.framework.TestCase {
   public void testCreate() {
   }
}
```

新增的第二行和第三行在 StudentTest 类中定义了一个*方法*：

```
public void testCreate() {
}
```

方法是一个可以包含任意行语句的代码块。就像类声明，Java 也用括号对来表示方法的开始和结束。所有括号对之间的代码都属于这个方法。

[5] 如果您是色盲，红条下面的统计为您提供了所需的信息。

这个例子中的方法叫 `testCreate`。JUnit 测试框架要求把该方法指定为 `public` 类型。

方法一般有两个作用。首先，当 Java 执行一个方法的时候，会逐行执行括号对之间的代码，其间可以调用其他方法，也可以修改对象的属性。另外，方法可以返回值给调用它的代码。

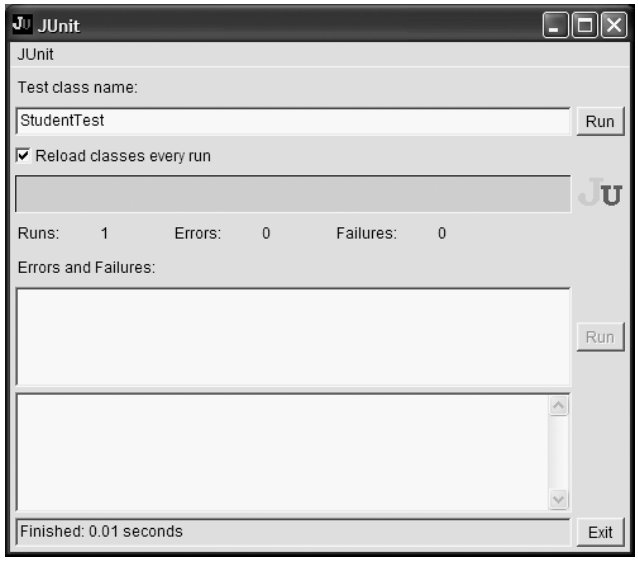
方法 `testCreate` 没有返回任何值给 JUnit，当然 JUnit 也不需要这样的信息。一个方法不返回任何信息，那么它的返回值为空类型。稍后您将学到如何从方法返回信息（请看本章的:
37 从方法中返回值）。

左右圆括号中间为空，表明 `testCreate` 不接受任何参数。该方法不需要传入任何信息就可以完成工作。

方法名 `testCreate`，暗示这是一个测试方法。对 Java 而言，这不过是个方法名。但是 JUnit 根据名称来识别一个测试方法，所以测试方法的命名要遵从下面的标准：

- 方法必须声明为 `public`
- 方法的返回值必须为 `void`
- 方法的名字必须以小写 `test` 为前缀
- 方法不能接受任何参数

编译您的代码，并且重新运行 JUnit 的 TestRunner，图 1.4 表明结果看起来还不错。




38 图 1.4 JUnit 执行成功（显示一根绿条）

您一直为之奋斗的就是那根绿条。对测试类 StudentTest，JUnit 显示成功地执行了一个测试方法（Runs:1），没有任何错误和失败。

请记住在 `testCreate` 中没有任何代码。JUnit 执行成功表明空的测试方法一定可以通过。

## 创建 Student 对象

在 testCreate 方法中增加一行代码：

```
public class StudentTest extends junit.framework.TestCase {
   public void testCreate() {
      new Student("Jane Doe");
   }
}
```

每行代码以分号结束。

当测试框架调用 testCreate 方法时，Java 会执行增加的这行语句。一旦 Java 执行完这行语句，就把控制权交回给调用 testCreate 的测试框架。

testCreate 方法中新增的这行语句告诉 Java 去创建 Student 类的对象。

```
new Student("Jane Doe");
```

把关键字 new 放在类名的前面，类名的后面是参数列表。参数列表包含了为了实例化一个 Student 对象所要的信息。不同的类需要不同的信息。有些类根本就不需要信息。一切都由类的设计者（在这里，就是您）来决定需要提供什么信息。

该例中的唯一参数表示学生的姓名，Jane Doe。“Jane Doe”是一个字符串。字符串是预先定义的 Java 类 java.lang.String 的实例。Java 用字符串来表示一段文本。

不久您将编写 Student 类，那时您可以指定如何来处理这个字符串参数。对参数您可以作如下处理：可以将其作为其它操作的输入数据；可以把它保存起来，以后再使用或取出；您可以忽略它；可以把它传递给其它对象。

当 Java 虚拟机执行到 new 操作符时，Java 虚拟机分配一块内存来存储这个 Student 对象。 39
Java 虚拟机根据 Student 类的定义来决定内存分配的大小。

## 创建 Student 类

编译这个测试。因为您只编写了测试类 StudentTest，所以会发现有错误[6]。StudentTest 引用了 Student 类，然而您还没有创建后者。

```
StudentTest.java:3: cannot find symbol
symbol  : class Student
location: class StudentTest
     new Student("Jane Doe");
         ^
1 error
```

注意脱字符号（^）的位置在错误的下方。表明 Java 编译器不知道 Student 代表着什么。

[6] 您可能看到不同的错误，这取决于您的 Java 编译器或者 IDE。

我们期望编译错误。编译时发现错误可以在整个开发过程中向我们提供反馈，这是好事情。您可以把编译错误看作编写测试后，得到的第一个反馈：您编写代码是否用了正确的 Java 语法，使得测试可以被正确执行？

**简化编译和执行**

为了减轻重复执行每条命令的沉闷，您也许应该编写一个批处理文件或者脚本。一个 Windows 批处理文件的例子：

```
@echo off
javac -cp c:\junit3.8.1\junit.jar *.java
if not errorlevel 1 java -cp .;c:\junit3.8.1\junit.jar junitawtui.TestRunner
StudentTest
```

假如编译器报告任何编译错误，该批处理文件就不会执行 JUnit 测试。下面是 Unix 中一个功能类似的脚本：

```
#!/bin/sh
javac -classpath "/junit3.8.1/junit.jar" *.java
if [ $? -eq 0 ]; then
   java -cp ".:/junit3.8.1/junit.jar" junit.awtui.TestRunnerStudentTest
fi
```

40

在 Unix 中另一个选择是使用 make，make 是一个在多数系统上得以应用的编译工具。然而，更好的方案是 Ant 工具。在第三课，您将学到如何使用 Ant，作为跨平台的方案来编译和运行您的测试。

为了消除目前的错误。我们创建一个新类 Student.java。输入下面的代码：

```
class Student {
}
```

再次运行 javac，这次我们使用通配符来编译所有的源文件：

```
javac -cp c:\junit3.8.1\junit.jar *.java
```

您会看到一个新的类似的报错。编译器再一次没能找到一个符号，但这一次指到了关键字 new。同时，编译器指出该符号寻找一个以 String 为参数的构造函数。编译器找到了 Student 类，但是现在需要知道如何针对字符串“Jane Doe”进行处理。

```
StudentTest.java:3: cannot find symbol
symbol  : constructor Student(java.lang.String)
location: class Student
     new Student("Jane Doe");
     ^
1 error
```

## 构造函数

编译器抱怨不能找到一个合适的 Student 类的构造函数。构造函数看起来非常像是一个方法，可以包含任意行的代码，可以接受任意数目的参数。但是，您必须把类名作为构造函数的

名字。同时，您不能从构造函数返回值，甚至不能返回空值。您使用构造函数来初始化一个对象，经常使用其他对象作为构造函数的参数。

本例中，在初始化一个 Student 对象时，您应该传入学生姓名作为参数。代码如下：

```
new Student("Jane Doe");
```

41

表明在 Student 类中必须定义一个拥有唯一 String 类型参数的构造函数。您可以编辑 Student.java，定义构造函数如下：

```
class Student {
   Student(String name) {
   }
}
```

再次编译，并且运行您的 JUnit 测试：

```
javac -classpath c:\junit3.8.1\junit.jar *.java
java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest
```

您会看到绿条。

现在构造函数对传入的姓名字符串不作任何事情。传入的字符串好像消失在了大气中。很快，您将修改 Student 的类定义来处理姓名。

## 局部变量

目前为止，当 JUnit 执行 `testCreate` 时，Java 执行了一行语句来创建一个新的 Student 对象。一旦语句执行完毕，控制返回给 JUnit 框架。此时，在 `testCreate` 中创建的对象消失了：该对象的生命周期和 `testCreate` 的执行周期相同。也就是说 Student 对象的范围局限在 `testCreate` 方法内。

稍后在测试中您应该能够引用 Student 对象。您必须保存 Java 虚拟机存储 Student 对象的内存地址。操作符 `new` 返回对象在内存的地址的引用，您可以通过赋值操作符（=）来存储如下引用。修改 StudentTest：

```
public class StudentTest extends junit.framework.TestCase {
   public void testCreate() {
      Student student = new Student("Jane Doe");
   }
}
```

原来的语句现在变成了赋值语句:赋值操作符右边的对象或值，被存储为操作符左边的引用。

42

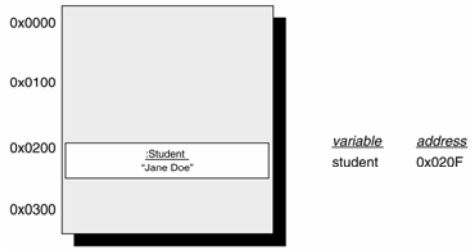
当 Java 虚拟机执行到该语句时，首先执行赋值语句右边的代码，在内存创建一个 Student 对象。虚拟机记住它放置新 Student 对象的实际内存地址。然后，虚拟机把地址赋值给左边的引用。

语句的左边创建了一个名字叫 `student` 的 Student 类型的引用。该引用包含了 Student 对象的内存地址。因为这个引用只在 test 方法的执行期内生存，所以这个引用是局部的。局部变量也被叫做临时变量。

您可以把变量命名为 `someStudent` 或者 `janeDoe`。但在这个例子里，普通名字 `student` 刚刚好。关于如何给变量命名，请参阅本章末尾的“命名约定”。

图 1.5[7] 展示了 `student` 引用和 Student 对象。该图只是帮助您理解幕后发生了什么，您不必了解如何创建这样的示意图。

在幕后，Java 维护了一个列表，包括您定义的所有变量和每个变量的内存地址。Java 的美妙之一就在于您不必自己编写代码来申请和释放内存。而使用 C 或 C++的程序员在内存管理上需要花费相当多的精力。




43

图 1.5 对象的引用

使用 Java，尽管您不必考虑内存管理，但是仍然有导致内存泄漏的可能。内存泄漏是指应用程序持续使用越来越多的内存直到内存耗尽。因为不甚了解 Java 如何替您管理内存，所以也存在程序执行不正确的可能。为了掌握这门语言，您必须理解幕后到底在发生什么。

本章中，我将使用概念上的内存图示，来帮助您理解当操作对象时，Java 做了什么。这里没有一个标准的图表工具。一旦您对内存分配有基本的理解，在用 Java 编程的时候，您就可以很大程度上不用再考虑内存问题了。

重新编译源文件，重新运行测试。到目前为止，您还没有编写任何做具体测试的代码，所以您的测试还会通过。

## 从方法返回一个值

下一步，您向测试中创建的 Student 对象请求学生的姓名。

```
public class StudentTest extends junit.framework.TestCase {
   public void testCreate() {
      Student student = new Student("Jane Doe");
      String studentName = student.getName();
   }
}
```

现在您在 `testCreate` 中有了两行代码。每行代码都以分号结尾。当 Java 虚拟机执行

[7] 内存地址用十六进制表示，数字以”0x”开头表示是十六进制数。第十课有关于十六进制数的讨论。

testCreate，虚拟机将控制返回给调用类之前，会自上到下，顺序执行每行代码。

第二行代码类似第一行代码，是另一行赋值语句。但是在这行代码中，您没有实例化一个新的对象。而是用上一行代码的 student 引用，发送消息给 Student 对象。

第二行代码的右边是一个消息发送，向 Student 对象请求姓名。您，作为程序员和 Student 类的设计者，将决定消息名称和它的参数。在这里，您决定消息的名字是 getName，并且这个消息不需要任何额外信息（参数）。

```
student.getName();
```

您也必须指定消息的接收者——您打算向其发送消息的对象。为了达到目的，首先指定对象引用 student，后面跟上一个句点（.），再后面是消息 getName()。圆括号表明没有参数随着消息传递。为了让第二行代码工作，您需要定义相应的方法 getName()。

44

第二行代码的左边，将返回的 String 对象的内存地址赋值给局部变量 studentName。为了让该赋值可以正常工作，您需要在 Student 类中定义返回 String 对象的 getName()方法。

马上您就会看到如何编写 getName()。

编译所有的源文件。编译错误表明编译器无法找到 Student 类中的 getName 方法。

```
StudentTest.java:4: cannot find symbol
symbol  : method getName()
location: class Student
     String studentName = student.getName();
                          ^
1 error
```

往 Student 类定义中添加 getName 方法，就可以消除这个错误。

```
class Student {
   Student(String name) {
   }

   String getName() {
   }
}
```

前面您了解过，如何使用 void 关键字来指定一个方法没有返回值。getName 方法定义了 String 类型的返回值。如果您现在就编译 Student.java，会得到一个报错：

```
Student.java:5: missing return statement
   }
   ^
1 error
```

因为 getName 方法要求 String 类型的返回值，所以需要 return 语句来提供一个 Srting 对象，并将该 String 对象返回给发送 getName 消息的代码。

```
class Student {
   Student(String name) {
   }

   String getName() {
```

```
        return "";
    }
}
```

45

`return` 语句返回了一个空 String 对象。再次编译，就不会有任何编译错误了。您可以再次用 JUnit 运行这个测试。

## 断言

现在，`testCreate` 有了完整的内容：第一行测试语句用指定的姓名创建了一个 Student 对象，第二行语句从这个 Student 对象请求获得姓名。现在您需要做的就是证明学生姓名和预期的一样，也就是说学生姓名和通过构造函数传递给 Student 对象的姓名一样。

```
public class StudentTest extends junit.framework.TestCase {
  public void testCreate() {
    Student student = new Student("Jane Doe");
    String studentName = student.getName();
    assertEquals("Jane Doe", studentName);
  }
}
```

第三行语句用来证明、或者断言前两行语句的执行结果是正确的。第三行语句是这个测试方法中的测试部分。下面来解释第三行语句的意图：您希望确保学生姓名是字符串“`Jane Doe`”。笼统地说，第三行语句是一个断言，断言的第一个参数要和第二个参数相同。

第三行语句也是一个消息发送，就如同第二行语句右边的代码。但是，这里没有消息接收者——`assertEquals` 消息发送给了谁呢？假如您没有指定接收者，Java 就把该方法所在的对象作为当前接收者。

Junit.framework.TestCase 类包含了 `assertEquals` 方法的定义。回忆一下您对 StudentTest 类的定义，您这样声明：

```
public class StudentTest extends junit.framework.TestCase {
```

声明表示 StudentTest 类从 junit.framework.TestCase 继承而来。当您发送消息 `assertEquals` 给当前 StudentTest 对象，Java 虚拟机尝试在 StudentTest 中找到 `assertEquals` 的定义。Java 虚拟机无法找到这样的定义，然后将使用在 junit.framework.TestCase 找到的定义。一旦 Java 虚拟机找到了 `assertEquals` 方法，就会像执行任何其他方法一样来执行该方法。

46

重要的是，要记住尽管 `assertEquals` 方法定义在 StudentTest 的父类中，该方法还是操作当前的 StudentTest 对象。在第六课您将再次接触继承的概念。

第三行语句也向您展示：通过用逗号分割参数，我们可以传递一个以上的参数。`AssertEquals` 方法有两个参数：字符串“`Jane Doe`”和您在第二行语句创建的 `studentName` 引用。这两个参数代表您要在 `assertEquals` 中进行比较的对象。JUnit 利用比较结果来决定 `testCreate` 方法是通过还是失败。如果 `studentName` 指向的字符串也是“`Jane Doe`”，那

么测试通过。

重新编译并运行测试。JUnit 看起来会像图 1.6。


图 1.6 测试尚未全部完成

47

JUnit 显示一根红条，表示有一个错误。并且在第一个列表框中告诉您出错原因：

```
Failure: testCreate(StudentTest): expected:<Jane Doe> but was:<>
```

出错的方法是 StudentTest 类的 `testCreate`。原因是我们期望得到字符串“`Jane Doe`”，然而收到的是空字符串。谁期望、在哪里期望得到字符串“`Jane Doe`”？JUnit 在第二个列表框显示了导致错误的代码回溯（也叫栈跟踪）。栈跟踪的第一行指出这里有一个失败的比较，第二行告诉您失败发生在 Student.java 的第五行。

用编辑器打开 StudentTest 的源代码，定位到第五行。可以看出 `assertEquals` 方法是导致比较失败的根源。

```
assertEquals("Jane Doe", studentName);
```

就像前面提到的，`assertEquals` 方法比较两个对象[8]，如果不相同就返回失败。JUnit 将第一个参数“`Jane Doe`”视为预期值。第二个参数 `studentName` 变量，作为实际值。您期望实际值也是“`Jane Doe`”，然而 `studentName` 是空字符串，因为您从 `getName` 方法中返回一个空字符串。

修改代码非常简单，改变 `getName` 方法，返回字符串“`Jane Doe`”：

```
class Student {
   Student(String name) {
   }

   String getName() {
```

[8] 特别地，该例子比较一个字符串对象和一个字符串变量或引用。Java 获取变量对应的值，再拿来比较。

```
        return "Jane Doe";
    }
}
```

重新编译和运行 JUnit。成功！（图 1.7）看到绿条使人感到些满足。再次按下按钮“Run”。条还是绿色。一切都好，但并不十分好。现在，所有的学生都被命名为“Jane Doe”。这样是不正确的，除非是女子学校而且只招收名字叫“Jane Doe”的学生。

48

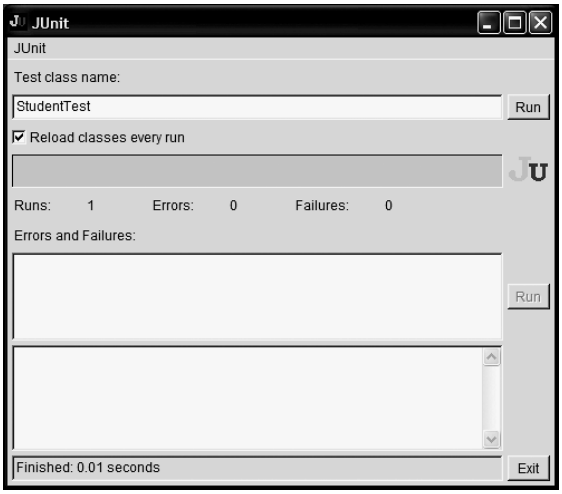



图 1.7 运行成功

## 实例变量

您已经编写了自己的第一个测试和相应的类。使用 StudentTest 来帮助您以一种渐增的方式来构建 Student 类。该测试也用来保证未来的任何改变不会影响已经完成的代码。

不幸的是，前面的代码并不十分完善。假如您创建了更多的学生对象，所有的学生对象都会说自己名字叫“Jane Doe”，来响应 `getName` 消息。在这一小节，通过解决这个问题，您将使 StudentTest 类和 Student 类趋于成熟。

您可以证明：每个 Student 对象会通过 `testCreate` 方法，并返回自己的名字——“Jane Doe”。添加代码创建第二个 student 对象：

49

```
public void testCreate() {
  Student student = new Student("Jane Doe");
  String studentName = student.getName();
  assertEquals("Jane Doe", studentName);

  Student secondStudent = new Student("Joe Blow");
  String secondStudentName = secondStudent.getName();
  assertEquals("Joe Blow", secondStudentName);
  }
```

图 1.8 显示了第二个 student 对象在内存中的逻辑视图。Java 为新的 Student 对象寻找空间并填充它。您不想知道也不必关心每个对象在何处终止。跳出内存视图，重要的是您知道有引用来表示内存中两个离散的对象。

再次运行 JUnit。假如您从命令行运行，以后台进程启动（Unix）[9]，或者用 `start` 命令启动（Windows）[10]。这样可以将控制权交给命令行，让 JUnit 在一个单独的窗口运行。不管您使用 IDE 还是从命令行运行，您可以使 JUnit 窗口保持打开状态。由于 JUnit 会重新加载有变化的 class 文件，所以您不必在每次代码变更之后重新启动 JUnit 窗口 [11]。

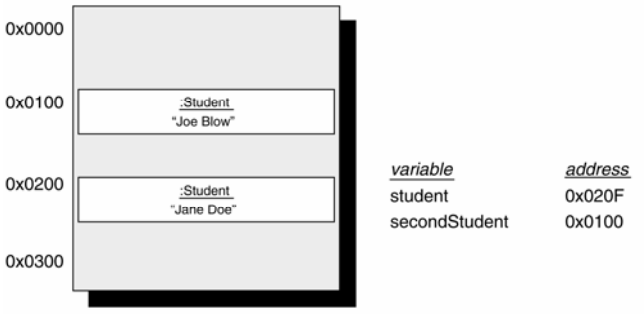



图 1.8　对象在内存中

50

本次测试失败：

```
junit.framework.ComparisonFailure: expected:<...oe Blow> but was:<...ane Doe>
```

很明显，由于 `getName` 方法总是返回“`Jane Doe`”，所以第二个 assertEquals 语句会使测试失败。

问题是您把学生姓名传给 Student 类的构造函数，但是构造函数对姓名不作任何处理。如果后面打算使用姓名，Student 类要负责存储姓名。

您需要把学生姓名作为 student 的一个属性，这是一个被 student 对象保存的信息。Java 中表现属性最直接的方式是将其定义为成员变量，也叫实例变量。在表示类的开始和结束的花括号对之间来定义成员变量。成员变量可以出现在方法以外的类定义的任何地方。不过根据约定，您最好把成员变量放在类的开始或者结束的地方。

就像局部变量，成员变量也有类型。这里定义了 String 类型的成员变量 `myName`:

```
Class student {
   String myName;

   Student(String name) {
   }

   String getName() {
```

[9] 多数系统中，在命令的后面附加上&。
[10] 在命令的前面加上 start，例如：`start java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest`。
[11] 如果能生效，请保证选中 JUnit 界面上的复选框“Reload classes every run”。

```
      return "Jane Doe";
   }
}
```

在构造函数中，把参数 name 赋值给 myName。

```
Student(String name) {
   myName = name;
}
```

最后，从 getName 方法中返回 myName，而不是返回字符串“Jane Doe”。

```
String getName() {
   return myName;
}
```

JUnit 应该仍然打开着，显示着比较错误和红条。通过点击“Run”按钮，重新运行测试，

51 将会显示绿条。

再看一下这个测试方法：

```
public void testCreate() {
   Student student = new Student("Jane Doe");
   String studentName = student.getName();
   assertEquals("Jane Doe", studentName);

   Student secondStudent = new Student("Joe Blow");
   String secondStudentName = secondStudent.getName();
   assertEquals("Joe Blow", secondStudentName);
}
```

该测试显示了如何用不同的姓名来创建 Student 实例。为了进一步支持断言，在测试方法的最后增加一行代码，从而确保您可以正确获取第一个 student 对象的姓名：

```
public void testCreate() {
   Student student = new Student("Jane Doe");
   String studentName = student.getName();
   assertEquals("Jane Doe", studentName);

   Student secondStudent = new Student("Joe Blow");
   String secondStudentName = secondStudent.getName();
   assertEquals("Joe Blow", secondStudentName);

   assertEquals("Jane Doe", student.getName());
}
```

注意，这里没有把 student.getName()的返回值赋值给一个局部变量，而是直接把 student.getName()作为了 assertEquals 的第二个参数。

编译后重新运行这个测试。测试成功表明 student 和 secondStudent 指向两个不同的对象。

## 总结这个测试

让我们逐行总结这个测试可预期的动作：

| | |
|---|---|
| `Student student = new Student ("Jane Doe");` | 创建名字为“Jane Doe”的学生对象，并保存到局部变量 |
| `String studentName= student.getName();` | 请求 student 的姓名，保存到局部变量 |
| `assertEquals(“Jane Doe”, studentName);` | 验证 student 的姓名是“Jane Doe” |

为了理解上面短短的几行代码，我们需要了解很多知识。不过，目前您已经掌握了 Java 编程的基础。在良好设计的面向对象 Java 编码中，大多数语句是创建新的对象、发送消息给其它对象、或者将对象地址赋值（用 new 创建对象，或者从消息发送返回对象）给对象引用。 52

## 重构

软件开发中的一个主要问题是代码维护的高成本。原因之一是匆忙行动或者纯粹疏忽导致的代码混乱。软件开发的主要任务是让软件可以工作，可以通过在编码之前先编写测试代码来应对这个挑战。其次，您的工作要确保代码是干净的。可以通过两种机制来实现：

1. 保证在系统中没有重复的代码。
2. 保证代码是干净的，并且富有表现力，可以清晰地体现程序员的意图。

贯穿敏捷 Java 的进程，您将经常停下来反思刚刚写下的代码。任何不符合这两条简单准则的代码都需要立刻重新处理，或者重构。即使设计非常完美，糟糕的代码实现同样会给修改它带来非常头痛的体验。

在您前进的时候，越是持续雕琢改进您的代码，您遇到需要付出高昂代价才能解决代码错误的可能性就越小。原则是永远不能让代码比开始时的状况要差。

即使在刚才的小例子中，也有一些不太理想的代码。看一下这个测试，我们开始整理代码：

```
public void testCreate() {
   Student student = new Student("Jane Doe");
   String studentName = student.getName();
   assertEquals("Jane Doe", studentName);

   Student secondStudent = new Student("Joe Blow");
   String secondStudentName = secondStudent.getName();
   assertEquals("Joe Blow", secondStudentName);

   assertEquals("Jane Doe", student.getName());
}
```

第一步要清除不必要的局部变量：studentName 和 secondStudentName。它们丝毫无助于对方法的理解，它们可以被 student 对象的查询所替代，就像最后一个 assertEquals。 53

当您完成这样的修改后，应重新编译和在 JUnit 中运行测试，以确保没有不好的影响。您的

代码看起来像这样：

```
public void testCreate() {
   Student student = new Student("Jane Doe");
   assertEquals("Jane Doe", student.getName());

   Student secondStudent = new Student("Joe Blow");
   assertEquals("Joe Blow", secondStudent.getName());

   assertEquals("Jane Doe", student.getName());
}
```

第二步：代码中到处嵌入字符串被视作不良的编程习惯。一个原因是，如果每个字符串所代表的意义不清晰的话，将很难理解这样的代码。

在这个例子中，您违背了不能有重复代码的准则。每个字符串都出现了两次。如果您不得不改变其中之一的话，您将不得不改变另一个。这样工作量就更多了。而且意味这样的可能性：改变了一个，没有改变另一个，从而代码中引入了缺陷。

消除此类冗余的方法（代码中增加一点表现力）是用字符串常量来替代一个字符串。

```
final String firstStudentName = "Jane Doe";
```

这条语句创建了 String 类型的引用 `firstStudentName`，并赋给其初始值“Jane Doe”。

语句开头的关键字 `final`，表明这个字符串引用是不可修改的，其它对象不可赋值给这个引用。您从来没有被要求指定 `final`，但这被认为是一种好的形式，可以帮助记住 `firstStudentName` 将像常量一样去工作。往后您会学到更多 `final` 的用法。

在下面您已经定义了常量来替代字符串：

```
final String firstStudentName = "Jane Doe";
Student student = new Student(firstStudentName);
assertEquals(firstStudentName, student.getName());
...
assertEquals(firstStudentName, student.getName());
```

54

编译并重新运行测试，以确保没有因不小心而破坏了什么。

同样，对另一个字符串进行重构。为了和变量名 `secondStudent` 保持协调，将局部变量 `student` 的名字改为 `firstStudent`。每一次小的改动之后，重新编译并且用 JUnit 确保您没有破坏已有的代码。所有改动完成之后，您的代码看起来像下面这样：

```
public void testCreate() {
   final String firstStudentName = "Jane Doe";
   Student firstStudent = new Student(firstStudentName);
   assertEquals(firstStudentName, firstStudent.getName());

   final String secondStudentName = "Joe Blow";
   Student secondStudent = new Student(secondStudentName);
   assertEquals(secondStudentName, secondStudent.getName());
```

```
   assertEquals(firstStudentName, firstStudent.getName());
}
```

最后一个 assertEquals 检验您对 Java 工作方式的理解，而不是代码功能的一部分。在实际系统中，您不大可能保留这个断言。您可以选择保留或者删除这个断言。如果删除它，当然您还需要编译和运行测试。

您的开发循环是：

- 编写一个小的测试，来断言某些功能正确与否。
- 运行测试，如果结果是失败。
- 编写代码，使测试通过。
- 重构测试和代码，清除重复的概念，确保代码富于表现力。

这样的循环，会很快成为一种根深蒂固、自然的开发流程。

## this

审视 Student 类的代码，看看是否有可以改进的地方。

```
class Student {
  String myName;

   Student(String name) {
      myName = name;
   }

   String getName() {
      return myName;
   }
}
```

55

代码看起来干净，但是成员变量 myName 的命名过于学生气。用一个更好的名字来体现您的专业素养。第一个想法可能是把成员变量命名为 studentName。但是，这样将导致命名重复，因为该成员变量定义在 Student 类中，所以很清楚这个成员变量表示的就是 student 的姓名。

而且，您还需要重新命名针对这个成员变量的 get 方法，因此冗余将变得十分明显，例如下面的代码：

```
student.getStudentName();
```

简单的用 name 命名如何？

用 name 作为该成员变量的名字也有问题，因为 Student 构造函数的参数是 name，这样两者就有了冲突。即使有冲突，我们试一下，看看会发生什么：

```
class Student {
   String name;

   Student(String name) {
```

```
      name = name;
   }

   String getName() {
      return name;
   }
}
```

编译会通过（也许您会看到一条警告）。然而，运行测试，却会失败：

```
junit.framework.ComparisonFailure: expected:<Jane Doe> but was:<null>
```

为什么？问题的部分原因是 Java 编译器允许成员变量的名字和参数的名字相同，甚至可以和局部变量的名字相同。编译代码的时候，Java 试着弄明白 name 究竟代表什么。编译器的解决方案是就近原则，使用最近定义的 name。这里就把 name 视作形式参数的 name。语句：

```
name = name;
```

导致存储在形式参数中的对象自己赋值给自己。这意味着没有赋值给成员变量 name。成员变量没有赋值，就会被指定为 null。所以就有了 JUnit 消息：

```
expected:<Jane Doe> but was:<null>
```

56

有两种方法可以保证形式参数的值被正确地传递给成员变量：参数和变量用不同的名字，或者使用关键字 this 来区分它们。使用 this 是最通用的方法。

第一种方法意味着您必须重新命名参数或者成员变量。Java 程序员使用很多不统一的约定来解决命名问题。一种约定是重新命名形式参数，参数使用单个字母或者参数前面加上前缀。例如，name 可以被命名为 n 或者 aName。另一个通常的选择是在成员变量的前面加上下划线作为前缀：_name。这样可以使成员变量比较突出，在某种程度上对理解代码是有价值的。也还有其他方法，比如给参数名称加上 a 或者 an 的前缀（如 aName）。

第二种消除歧义的方法是：成员变量和参数使用相同的名字，但是在必要的地方用 Java 关键字 this 来调用成员变量。

```
class Student {
   String name;

   Student(String name) {
      this.name = name;
   }

   String getName() {
      return name;
   }
}
```

关键字 this 指向当前对象的引用，当前对象是指正在运行的代码所属的对象。上面的例子中将形式参数 name 赋值给成员变量 name。

确信修改后测试可以通过。从现在开始，记住在修改代码之后和运行测试之前，要重新编译。本书后面的内容中将不会再有这样的提示。

# private

Java 允许访问对象的成员变量，就像您可以调用一个方法：

```
public void testCreate() {
   final String firstStudentName = "Jane Doe";
   Student firstStudent = new Student(firstStudentName);
   assertEquals(firstStudentName, firstStudent.getName());

   final String secondStudentName = "Joe Blow";
   Student secondStudent = new Student(secondStudentName);
   assertEquals(secondStudentName, secondStudent.getName());

   assertEquals(firstStudentName, firstStudent.name);
}
```

57

运行这个测试，会通过。但是，它展现了一种特别不好的面向对象编码风格。

不要把成员变量直接暴露给其他对象。

假设您打算设计 Student 类，让学生姓名不可改变，即一旦您创建了 Student 对象就不能改变该学生的姓名。下面的测试代码展示了，允许其它对象访问成员变量是一个坏主意：

```
final String firstStudentName = "Jane Doe";
Student firstStudent = new Student(firstStudentName);
firstStudent.name = "June Crow";
assertEquals(firstStudentName, firstStudent.getName());
```

测试说明了 Student 的客户代码——与 Student 对象交互的代码，能直接修改存储在 `name` 中的字符串。尽管这看起来不像是特别可怕的问题，但是客户修改对象的数据使您失去了所有的控制。如果您打算允许客户代码修改学生姓名，您可以创建一个方法让客户使用。例如，可以创建一个 `setName` 方法，该方法以字符串作为参数。在 `setName` 方法中，您可以增加任何必要的控制。

对 StudentTest 做出上面的修改。运行测试，测试显示失败。

为了保护您的成员变量，请隐藏它们，将它们指定为 `private`。修改 Student 类以隐藏 `name`。

```
class Student {
   private String name;
   ...
```

做完上面的修改之后，任何存取 name 成员变量的行为甚至连编译都无法通过。由于 StudentTest 中的代码直接调用 name 成员变量：

```
assertEquals(firstStudentName, firstStudent.name);
```

您将得到一个编译报错：

```
name has private access in Student
```

```
    assertEquals(firstStudentName, firstStudent.name);
                                   ^
58  1 error
```

删掉有问题的代码，重新编译和测试。

成员变量私有化的另一个好处在于可以强制加强面向对象和封装的观念：一个面向对象的系统更关注行为，而不是数据。您应该通过发送消息来获得数据，也应该封装实现细节。后面您也许打算修改 Student 类来分别存储姓和名，修改 `getName` 方法以返回姓名。如果那样，直接访问 `name` 成员变量将不再有效。

任何规则都有例外。至少有两个合法的理由，让我们不把某个成员变量指定为 private。（后面您会了解到这些。）

## 命名约定

迄今为止，您应该已经注意到 Java 代码中的一个命名模式。大多数已经学到的 Java 元素，例如成员变量、形式参数、方法、局部变量，都用一种相似的方法来命名。这种命名规约有时被称为“驼峰模式”[12]。按照驼峰模式，您可以把多个单词直接连接起来组成一个名字或者标识符。除了第一个单词，标识符中的每一个单词都以大写字母开头。

您应该用名词来为成员变量命名。名字要能够描述该成员变量被用作什么或者它表示什么，而不是如何实现。代表成员变量类型的前缀或者后缀是不必要的，应该避免。应该避免的例子有 `firstNameString`、`trim` 以及 `sDescription`。

好的命名成员变量的例子有 `firstName`、`trimmer`、`description`、`name`、`mediaController` 以及 `lastOrderPlaced`。

方法通常是动作或查询：您发送消息告诉对象做某件事情，或者您向对象请求获取某些信息。您应该使用动词来命名动作型方法。同样也应该使用动词来命名查询型方法。对于请求获取属性，通常的 Java 规约是在名称前面加上前缀 get，就像 `getNumberOfStudents` 和
59 `getStudent`。后面我会讨论关于此规则某些例外的情况。

好的方法名字有 `sell`、`cancelOrder` 以及 `isDoorClosed`。

类命名使用“大写驼峰模式”——标识符的第一个字母是大写字母的驼峰模式[13]。几乎总是应该用名词作为类名——对象是事物的抽象。不要使用复数名词作为类名。类在某一时刻被用来创建一个单一对象。例如，Student 类可以创建一个 Student 对象。后面会提到，创建对象集合时依然使用非复数名字：用 StudentDirectory 代替 Students。从 StudentDirectory 类，您可以创建一个 StudentDirectory 对象。从 Students 类，您可以创建一个 Students 对象，但是这听起来很别扭并且导致同样别扭的代码。

[12] 想象一下：字符串展现了从侧面所观察到的骆驼的形象，大写字母表示驼峰。关于这个术语有一个有趣的讨论，请看 *http://c2.com/cgi/wiki?CamelCase*。您也许还听说过其他的名字，比如混合大小写。
[13] 驼峰模式有时也被叫做小写驼峰模式，以此来和大写驼峰模式有所区分。

好的类名有 Rectangle、CompactDisc、LaundryList 以及 HourlyPayStrategy。

对于好的面向对象设计，您会发现设计影响命名的能力。一个设计良好的类处理一件重要的事情，并且仅仅处理这一件事情[14]。类通常不用来处理多个事情。例如，如果有一个类被用来切割支票、打印支票报表、计算不同部门的退款，那么提出一个名字来简洁地描述这个类是非常困难的。相反，应该把这个类分开成三个独立的类：CheckWriter、PayrollSummaryReport 和 ChargebackCalculator。

您可以在标志符中使用数字，但是您不能用数字作为开头的第一个字母。避免特殊符号——Java 编译器不允许很多特殊符号。避免下划线（_），特别是不要用下划线作为单词间的分隔符。有两个例外：前面提到过，一些程序员喜欢用下划线作为成员变量的前缀；另外，类常量（后面会介绍）通常含有下划线。

避免缩写。软件中，清晰是很有价值的。花时间多输入几个字符吧。多输入几个字符所花时间，较之将来的阅读者为理解您的代码所花的时间要少得多。例如用更有表现力的 `custard` 和 `numerology` 替代 `cust` 和 `num`[15]。现代 IDE 提供诸如快速重命名和自动补齐代码的功能，所以没有理由使用含义模糊的名字。通用的缩写是可以接受的，例如 `id` 或者 `tvAdapter`。

60

请记住 Java 是大小写敏感的。这意味着 `stuDent` 和 `student` 是不同的名字。尽管这样，这是不好的形式，会在命名上引起混淆。

这些命名规约被广泛接受而且是不被编译器控制的编码规范。编译器允许您使用甚至会激怒同事的可怕的名字。多数开发团队明智地采用了某种通用编码规范，被所有程序员所遵循。较之 C++社区，Java 社区也在此类规范上达成了更高层次的约定。Sun 的 Java 编码规范在 *http://java.sun.com/docs/codeconv* 可以找到，非常好，虽然不完整而且过时，但是可以作为创建自己编码规范的起点。有一些关于风格/规范的书，例如 Essential Java Style[16] 和 Elements of Java Style[17]。

## 空白区域

代码版面设计是另一个您和您的团队应该有共同规范的领域。空白区域包括空格，tab 键，换页符，换行符（通过按下回车键产生）。某些元素之间要求空格，某些元素之间是可选的。例如，在关键字 class 和类的名称之间，至少需要一个空格：

```
class Student
```

下面的空格是允许的（但是要避免）：

```
String studentName = student   .   getName();
```

Java 编译器会忽略额外的空白空间。您应该使用空格、tabs、空行来明智地组织您的代码。

[14] 单职责原则[Martin2003]。
[15] 我打赌您会错误地以为是 `customer` 和 `number`。看看这多么容易让人误入歧途。
[16] [Langr2000]。
[17] [Vermeulen2000]。

为了有利于将来可以轻松地理解代码，您应该坚持这样。

本书的例子提供了一种一致的、可靠的、广泛被接受的方法，来格式化 Java 代码。如果您的代码看起来像这些例子，就可以进行编译。本书中的例子同样符合多数 Java 开发团队的规范。您需要自己决定某些方面，例如使用 tabs 还是空格来缩进，以及用多少个字符来缩进（通常是
61 3 或 4 个）。

## 练习

练习出现在课程的最后。多数练习是让您完成国际象棋程序的一部分。如果您对国际象棋的规则不熟悉，您可以从这里了解：*http://www.chessvariants.com/d.chess/chess.html*。

1. 就像处理 Student 一样，您从创建一个表示 pawn（象棋中的卒）的类开始。首先，创建一个空的测试类 PawnTest。针对 PawnTest 运行 JUnit，因为您还没有编写任何测试方法，您将观察到失败。
2. 创建测试方法 `testCreate`。确信您遵循了正确的声明测试方法的语法。
3. 在 `testCreate` 中添加实例化一个 Pawn 对象的代码。确信您收到一个编译错误，因为不存在相应的类。创建 Pawn 类，展现一个正确的编译。
4. 将一个实例化的 Pawn 对象赋值给一个局部对象。向 pawn 请求获得它的颜色。增加一个断言，要求 pawn 的默认颜色是字符串“`white`”。观察到测试失败，然后在 Pawn 中增加代码以使测试通过。
5. 在 `testCreate` 中，创建第二个 pawn，传递颜色“`black`”到它的构造函数。断言第二个 pawn 的颜色为“`black`”。观察到测试失败，然后增加代码使测试通过。请注意：消除默认的构造函数——要求客户代码在创建 Pawn 对象时传入颜色。修改将影响到您在练习 4 中编写的代码。
62 6. 在 `testCreate` 中，为字符串“`white`”和“`black`”创建常量。确保重新运行测试。