# 2

# Java Basics

# Java 基础

本课内存容包括：

- 使用整型 `int` 来统计学生人数
- 使用 Java 的集合类 java.util.ArrayList 存储多个学生对象
- 理解默认构造函数
- 学习如何利用 J2SE API 文档，理解如何使用 java.util.ArrayList
- 限定 java.util.ArrayList 只能包含 Student 对象
- 创建一个 TestSuite（测试套件）来测试一个以上的类
- 学习包和 `import` 语句
- 理解如何定义和使用类常量
- 使用系统库的 Date 和 Calendar 类（译注：原文为 date 和 calendar，疑为作者笔误）
- 学习不同类型的 Java 注释
- 使用 javadoc 为您的代码生成 API 文档

## 课程安排

在学校，有很多每个学期都开的课程，例如 Math 101 和 Engl 200。从一个学期到下一个学期，基本的课程信息都是一样的，例如名称、编号、学分、课程介绍。

CourseSession 表示课程安排，它存储上课时间和教师信息，同时保留一份这门课程的学生清单。



您需要定义 CourseSession 类，该类捕获基本的课程信息和上课学生的情况。只要您在仅考虑一个学期的情况下使用 CourseSession 对象，就不会有两个 CourseSession 对象指向同一门课程。一旦存在两个 CourseSession 对象必须指向同一门课程，那么基本课程信息存储在两个

CourseSession 对象就显得冗余了。目前，不考虑多个课程安排的情况，稍后您将整理现在的设计以支持“一门课程的多种课程安排”。

创建 CourseSessionTest.java，并在其内编写一个名为 `testCreate` 的测试。就像 StudentTest 中的 `testCreate`，这个测试方法将展示您如何创建 CourseSession 对象。创建测试是一个观察创建后的对象将会如何工作的好地方。

```
public class CourseSessionTest extends junit.framework.TestCase {
   public void testCreate() {
      CourseSession session = new CourseSession("ENGL", "101");
      assertEquals("ENGL", session.getDepartment());
      assertEquals("101", session.getNumber());
   }
}
```

该测试表明可以用课程名称和编号来创建一个 CourseSession。该测试也确保课程名称和编号可以被正确地存储在 CourseSession 对象中。

为了使这个测试得以通过，CourseSession 的代码如下：

```
class CourseSession {
   private String department;
   private String number;

   CourseSession(String department, String number) {
      this.department = department;
      this.number = number;
   }

   String getDepartment() {
      return department;
   }

   String getNumber() {
      return number;
   }
}
```

64

到目前为止，您已经创建了存储学生数据的 Student 类，以及存储课程信息的 CourseSession 类。两个类都提供了 getter 方法，来让其它对象获取数据。

但是，诸如 Student 和 CourseSession 的数据类并不怎么有趣。如果所有的面向对象开发都是关于数据的存储和获取，那么系统将不会特别有用，也不是真正的面向对象。请记住：面向对象系统是行为建模。行为通过向对象发送消息产生作用——让对象做某件事情或者从对象获取数据。

不过，您已经从某个地方开始了。而且，如果不能查询对象的话，您将不能在测试中编写断言。

# 学生注册

 除非有学生报名，否则课程不能为学校带来任何收入。多数情况下，学生信息系统要求能够同时处理多个学生。您应该能把很多学生存储到组或者集合里，并且在这些集合里对学生执行某些操作。

CourseSession 需要存储一个新的属性——Student 对象的集合。您应该增强 CourseSession 的创建测试，来支持这个新属性。如果您只是创建了一个新的 CourseSession 对象，还没有招收任何学生。那么，您能针对一个空的 CourseSession 对象做出什么断言呢？

修改 testCreate，使其包含黑体的断言：

```
public void testCreate() {
   CourseSession session = new CourseSession("ENGL", "101");
   assertEquals("ENGL", session.getDepartment());
   assertEquals("101", session.getNumber());
   assertEquals(0, session.getNumberOfStudents());
}
```

# int

新断言验证一门新课程的报名学生人数应该为 0。符号 0 是数字符号，代表整数零。特别地，它是一个整型数，在 Java 中称为 int。

65

将方法 getNumberOfStudents 添加到 CourseSession：

```
class CourseSession {
   ...
   int getNumberOfStudents() {
      return 0;
   }
}
```

（省略号代表实例变量，构造函数，以及您已经编写的 getter 方法。）指定 getNumberOfStudents 的返回值类型为 int。从方法返回的值必须与声明的返回值类型相匹配。在 getNumberOfStudents 方法中返回了一个整型数字。int 型允许的整型数字范围是－2 147 483 648 到 2 147 483 647。

不像字符串，Java 中的数字不是对象。尽管数字如同字符串一样，可以作为参数随着消息传递，但是您不能发送消息给数字。Java 在语法上提供了对基本算术运算的支持；对别的很多操作，由系统库提供支持。稍后您将学到类似的非对象类型。简而言之，这些非对象类型被称为基本类型。

您已经证实新 CourseSession 对象可以正确地初始化，但是您没有说明这个类可以正常地招收学生。创建第二个测试方法 testEnrollStudents 来招收两个学生。对每一个学生，创建一个 Student 对象，招收这个 student，然后确信 CourseSession 对象报告正确的学生数目。

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

   public void testEnrollStudents() {
      CourseSession session = new CourseSession("ENGL", "101");

      Student student1 = new Student("Cain DiVoe");
      session.enroll(student1);
      assertEquals(1, session.getNumberOfStudents());

      Student student2 = new Student("Coralee DeVaughn");
      session.enroll(student2);
      assertEquals(2, session.getNumberOfStudents());
   }
}
```

66

您如何知道需要 enroll 方法，并且该方法需要 Student 对象作为参数？在测试方法中您所做的部分工作是为类设计 public 接口——开发者如何与这个类交互。您的目标是设计一个类，并且让这个类以尽可能简单的方式来满足开发者的需要。

使第二个断言（学生人数为 2）能够通过的最简单的办法是让 getNumberOfStudents 方法返回值为 2。但是，这样会破坏第一个断言。所以您必须在 CourseSession 内部记录学生的人数。为了实现这个目的，您需要再引入一个成员变量。任何时候，如果需要存储信息，您可以使用一个表示对象状态的成员变量。像下面这样修改 CourseSession 类：

```
class CourseSession {
   ...
   private int numberOfStudents = 0;
   ...
   int getNumberOfStudents() {
      return numberOfStudents;
   }

   void enroll(Student student) {
      numberOfStudents = numberOfStudents + 1;
   }
}
```

将记录学生人数的成员变量 numberOfStudents 设为 private，这是一个好的实践。该成员变量的类型为 int，而且被初始化为 0。当一个 CourseSession 对象被初始化时，成员变量也会被初始化，例如 numberOfStudents。成员变量在构造函数执行之前被初始化。

getNumberOfStudents 方法返回成员变量 numberOfStudents，而不再返回整型数 0。

每次 enroll 方法被调用时，学生的数目都会增加 1。enroll 方法中的唯一一行代码实

现了这个功能：

```
numberOfStudents = numberOfStudents + 1;
```

加号和其它很多算术操作符，可以用来处理整型变量（也可以用来处理其它数字类型，稍后会讨论）。等号右边的表达式取出 numberOfStudents 当前的值，并将其加 1。由于成员变量 numberOfStudents 出现在等号的左边，所以右边表达式的结果赋值给 numberOfStudents。这个例子中，将变量加 1，是一种常用的变量自增的方法。后面会讨论其它增加变量值的方法。 67

numberOfStudents 同时出现在等号的两边，这让人看起来很奇怪。请记住：Java 虚拟机总是先执行某个赋值语句右边的代码。Java 虚拟机计算出等号右边表达式的值，然后将结果赋值给左边。

请注意 enroll 方法的返回值类型为 void，这意味着 enroll 方法针对消息发送者不返回任何东西。

图 2.1 显示了截至现在的系统结构。

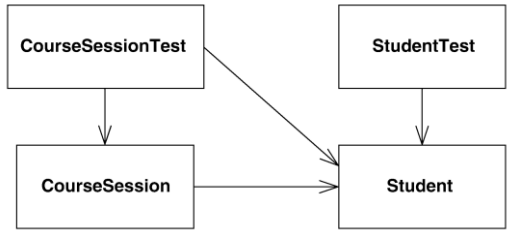



图 2.1 CourseSession 和 Student 类图

概念上，一个 CourseSession 可以包含多个学生。实际上——在代码中——CourseSession 没有包含学生对象的引用。目前 CourseSession 只包含学生的计数。稍后，当修改 CourseSession 类来实际存储 Student 引用的时候，您将修改 UML 类图来表示 CourseSession 和 Student 之间的一对多的关系。

因为 enroll 方法需要 Student 对象作为参数，所以 CourseSession 依赖于 Student 类。换句话说，如果 Student 类不存在，您将无法编译 CourseSession 类。

图 2.1 是最后一个显示每一个测试类的类图。由于您正在进行测试驱动的开发，所以如果没有特别说明，后面类图中每一个生产类都暗示存在对应的测试类。

## 初始化

在上一节，您引入了成员变量 numberOfStudents，并将其初始化为 0。从技术上，此类初始化是不需要的——int 成员变量默认被初始化为 0。按照这样，显式地初始化成员变量，有助于解释代码的意图。

68

目前，您有两种方法来初始化成员变量：可以在成员变量定义时初始化，或者在构造函数中初始化。您可以在 CourseSession 的构造函数中初始化 numberOfStudents：

```
class CourseSession {
  private String department;
  private String number;
  private int numberOfStudents;

  CourseSession(String department, String number) {
    this.department = department;
    this.number = number;
    numberOfStudents = 0;
  }
  ...
```

关于初始化没有什么必须遵循的规则。我倾向于尽可能在成员变量定义时初始化——在同一个地方完成声明和初始化，可以使代码变得更简单些。而且，稍后在本章中会学到，您可以拥有一个以上的构造函数，所以在成员变量定义时进行初始化，可以避免每个构造函数中都有重复的初始化代码。

您会遇到不能在成员变量定义时进行初始化的情况，所以在构造函数中初始化也许会成为您唯一的选择。

## 默认构造函数

您也许注意到，StudentTest 和 CouseSessionTest 都没有构造函数。您经常不需要显式地初始化任何代码，Java 编译器不要求您必须定义构造函数。如果在类中没有定义任何构造函数[1]，Java 会提供一个默认的、没有参数的构造函数。举例来说，对于 StudentTest，就像您编写了空的构造函数：

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

69

默认构造函数表明 Java 将构造函数视为类的必要元素。Java 需要构造函数来初始化一个类，即使构造函数中没有任何初始化代码。如果您没有提供构造函数，Java 编译器会替您生成一个。

[1] Java 编译器允许您在一个类中定义多个构造函数。后面会对此进行讨论。

# 测试套件

前面，您引入了第二个测试类 CourseSessionTest。以后，您会根据生产类的改变，来决定针对 CourseSessionTest 或者 StudentTest 来运行 JUnit。不幸的是，很有可能您在 Student 类中做出的改变，会使 CourseSessionTest 中的某个测试失败，然而 StudentTest 中的所有测试都运行成功。

您可以在 CouseSessionTest 上运行所有的测试，然后运行 StudentTest 中的所有测试。每次重启 JUnit，或者在 JUnit 中重新输入测试类的名称，甚至打开多个 JUnit 窗口。但是，没有一种方案是可伸缩的：当您增加更多的类，事情会很快变得无法管理。

相反，JUnit 允许您构建测试套件，也叫测试集合。套件还可以包含其它套件。就像运行测试一样，您可以在 JUnit test runner 中运行测试套件。

创建一个叫 AllTests 的新类，代码如下：

```
public class AllTests {
   public static junit.framework.TestSuite suite() {
      junit.framework.TestSuite suite =
         new junit.framework.TestSuite();
      suite.addTestSuite(StudentTest.class);
      suite.addTestSuite(CourseSessionTest.class);
      return suite;
   }
}
```

如果您用类名 AllTests 作为参数来启动 JUnit，将执行 CourseSessionTest 和 StudentTest 中的所有测试。从现在开始，运行 AllTests 而不是分别运行每个测试类。

在 AllTests 中，`suite` 方法的职责是创建一个包含测试类的套件，并返回这个套件。junit.framework.TestSuite 管理套件，您通过发送消息 `addTestSuite` 向套件中增加测试。消息 `addTestSuite` 的参数是一个类字面常量。类字面常量由类名加上 `.class` 组成。类字面常量唯一标示了一个类，可以使类定义本身能够像对象一样被处理。

每增加一个新的测试类，您都需要记得把它添加到测试套件。这是一种容易发生错误的技术，因为您很容易忘记更新测试套件。在第十二课，您会得到一个更好的解决方案：用工具来替您产生和执行测试套件。

70

在 AllTests 中也引入了静态方法的概念，在第四课您将深入学习这个概念。目前，只需要理解：为了让 JUnit 能够识别，您必须把 suite 方法定义为 static 类型。

# SDK 和 java.util.ArrayList

CourseSession 类很好地记录了学生的人数。但是，您和我都知道，CourseSession 仅仅维护了一个计数器，并没有保存招收的学生。对于将要完成的 testEnrollStudents 方法，您需要证明 CourseSession 对象保存了实际的 student 对象。

一个可能的方案是——要求 CourseSession 提供一个报名学生的列表，然后检查这个列表，确保该列表包含了期望的学生。在您的代码中加入下面黑体的代码行：

```
 public void testEnrollStudents() {
CourseSession session = new CourseSession("ENGL", "101");

Student student1 = new Student("Cain DiVoe");
session.enroll(student1);
assertEquals(1, session.getNumberOfStudents());
java.util.ArrayList<Student> allStudents = session.getAllStudents();
assertEquals(1, allStudents.size());
assertEquals(student1, allStudents.get(0));

Student student2 = new Student("Coralee DeVaughn");
session.enroll(student2);
assertEquals(2, session.getNumberOfStudents());
assertEquals(2, allStudents.size());
assertEquals(student1, allStudents.get(0));
assertEquals(student2, allStudents.get(1));
}
```

第一行新增加的代码：

```
java.util.ArrayList<Student> allStudents = session.getAllStudents();
```

表明 CourseSession 有一个 getAllStudents 方法。该方法返回 java.util.ArrayList<Student> 类型的对象，因为 getAllStudents 的返回值将赋值给该类型的变量。参数放在类名后面的一对尖括号里面，而且参数的值是某种类型，这样的类型叫做参数化类型。在这个例子中，java.util.ArrayList 的参数 Student，这表明 java.util.ArrayList 被限定只能包含 Student 对象。

71

类 java.util.ArrayList 是 Java SDK 类库提供的成千个类中的一个。您应该已经下载，并且在您的机器上安装了 SDK 文档。您也可以在 Sun 的 Java 网站上在线浏览这些文档。我倾向于在本地安装 SDK 文档，这样速度比较快而且可以方便地访问。

打开文档，定位到 Java 2 平台 API 规范。您可以看到类似图 2.2 的界面。

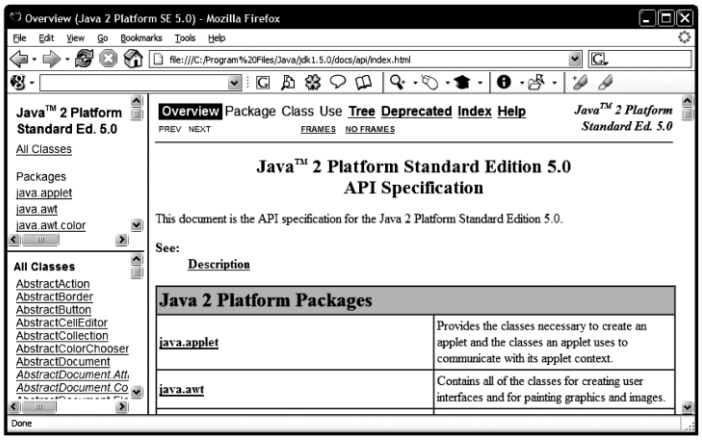



图 2.2　Java 2 平台 API 规范

除非结对编程[2]，否则 Java API 文档将是您最好的朋友。在结对编程的情况下，Java API 文档将是您第二要好的朋友。图 2.2 有三个部分，左上角列出了库中所有可用的包。包由一组相关的类构成。

左下角默认显示类库中所有的类。一旦选择了某个包，那么左下角只显示这个包中所有的类。右半部分显示当前选中的某个包或者类的详细信息。

滚动左上角，直到您看见名字叫 java.util 的包，选中这个包。左下角将会看到一个包含接口、类和异常的列表。稍后我会介绍接口和异常。在这里，请滚动左下角直到看见类 Arraylist，选中它。 72

包 java.util 含有几个工具类，在这本 Java 开发教程中您将经常使用这些工具类。这个包所含有的类支持一种集合框架（Collections Framework）。该集合框架支持标准数据结构，例如列表、链表、集合、哈希表。在 Java 中，您将经常使用集合来处理一组相关的对象。

右半部分显示了类 java.util.ArrayList 的详细信息。拖动滚动条直到您看见方法摘要。方法摘要列出了类 java.util.ArrayList 中实现的所有方法。花上几分钟浏览所有的方法。用鼠标点击每个方法的名字，可以看到方法的细节。

在这个练习中，您将使用 java.util.ArrayList 的三个方法：`add`、`get` 和 `size`。该测试已经用到了 `size` 方法。下面的代码断言 java.util.ArrayList 中有一个对象：

```
assertEquals(1, allStudents.size());
```

下面的新代码断言 `allStudents` 的第一个元素等同于招收的 student。

```
assertEquals(student, allStudents.get(0));
```

根据 API 文档，`get` 方法返回列表中某个绝对位置的元素。把位置当作索引，传入 `get` 方

[2] 开发团队中，程序员动态的两两组成一对，两个人一起编写代码。

法。索引从 0 开始，所以 `get(0)` 返回列表中的第一个元素。

## 增加对象

Java SDK API 文档定义 `add` 方法需要某个对象作为参数。如果您在 API 文档中点击参数 `Object`，您将看到它实际上是 java.lang.Object。

您或许听说过 Java 是一个纯面向对象语言，因为“所有东西都是对象”[3]。类 java.lang.Object 是 Java 系统类库中所有类的基类，也是任何自定义类的基类，包括 Student 和 StudentTest。每个类都直接或间接地继承自 java.lang.Object。StudentTest 继承自 junit.framework.TestCase，juni.framework.TestCase 继承自 java.lang.Object。

73

从 java.lang.Object 继承十分重要：您将学习几个依赖于 java.lang.Object 的核心语言特性。目前，您需要理解 String 继承自 java.lang.Object，就像 Student 以及其它您定义的类。继承意味着 String 对象和 Student 对象也是 java.lang.Object 对象。好处是，String 和 Student 对象可以作为参数传入以 java.lang.Object 作为参数的方法。就像上面提到的，`add` 方法接收 java.lang.Object 实例作为参数。

即使您可以通过 `add` 方法将任何类型的 object 传入 java.util.ArrayList，您也不能一直这样去做。在 CourseSession 对象中，您知道您只想招收学生，所以声明参数化类型。声明参数化类型的一个好处是：限制 java.util.ArrayList 只能包含 `Student` 对象，从而避免不小心把其它类型的对象添加到这个列表。

如果某个类型允许被加入，例如：加入一个 String 对象到学生列表。结果是，您的代码使用 `getAllStudents` 请求获得学生列表，并且使用 `get` 方法从该列表中获得 String 对象。当您试图将这个 String 对象赋值给 Student 引用时，Java 虚拟机会停下，报告一个错误消息。Java 是强类型的语言，不允许您将 String 赋值给 Student 引用。

针对 CourseSession，下面的改动（黑体）将使测试得以通过：

```
class CourseSession {
   ...
   private java.util.ArrayList<Student> students =
      new java.util.ArrayList<Student>();
   ...
   void enroll(Student student) {
      numberOfStudents = numberOfStudents + 1;
      students.add(student);
   }

   java.util.ArrayList<Student> getAllStudents() {
      return students;
   }
}
```

[3] Java 语言中的一个重要部分不是面向对象的，本书稍后会有介绍。

一个新成员变量：students，被用来存储学生列表。该变量被初始化为一个空的 java.util.ArrayList 对象，而且限定其只能包含 Student 对象[4]。enroll 方法把 student 添加到这个列表，getAllStudents 只是简单地返回这个列表。

74

图 2.3 显示 CourseSession 依赖于参数化类型 java.util.ArrayList<Student>。同时，CourseSession 依赖于 0 到多个 Student 对象（用 CourseSession 到 Student 连线末端的*表示）。

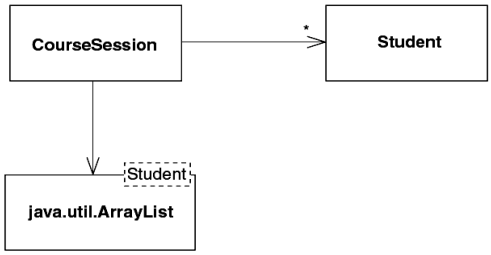



图 2.3　包含参数类型的类图

图 2.3 不是普通的类图——事实上用不同的方式将同一个信息描述了两次。参数化类型声明使得 CourseSession 与存储多个 Student 对象的 ArrayList 是一对一的关系。同时 CourseSession 到 Student 之间是一对多的关系。

# 渐增重构

既然 java.util.ArrayList 提供了 size 方法，那么您可以调用这个方法来获得 ArrayList 对象中的学生人数，而不必再去跟踪 numberOfStudents。

```
int getNumberOfStudents() {
   return students.size();
}
```

进行这样小型的重构，重新编译和运行测试。测试给了您不受惩罚地改变代码的信心——如果改变导致不能工作，就简单地恢复到原来的状态，再试试别的方法。

75

由于不再从 getNumberOfStudents 方法返回 numberOfStudents，所以您需要停止在 enroll 方法中增加这个变量。这也意味着您可以完全删掉成员变量 numberOfStudents。

搜索 CourseSession 类，删除每一个 numberOfStudents，您也可以使用编译器工具来做这件事情。删掉 numberOfStudents 的成员变量定义，然后重新编译。编译器在所有引用该成员变量的位置报告错误。您可以利用这些信息来直接定位您需要删除的代码。

类的最终版本如下：

[4] 由于宽度限制，我将 students 声明分割到了两行。您在代码中可以选择将其格式化为一行。

```
class CourseSession {
   private String department;
   private String number;
   private java.util.ArrayList<Student> students =
      new java.util.ArrayList<Student>();

   CourseSession(String department, String number) {
      this.department = department;
      this.number = number;
   }

   String getDepartment() {
      return department;
   }

   String getNumber() {
      return number;
   }

   int getNumberOfStudents() {
      return students.size();
   }

   void enroll(Student student) {
      students.add(student);
   }

   java.util.ArrayList<Student> getAllStudents() {
      return students;
   }
}
```

## 内存中的对象

在 testEnrollStudents，您发送 getAllStudents 消息给 session，并将结果存储到一个 java.util.ArrayList 引用，该引用被限定到 Student 类——只能包含 Student 对象。稍后在测试中，在招收了第二个学生后，allStudents 包含了两个学生——您不必再次向 session 对象请求获得它。

76

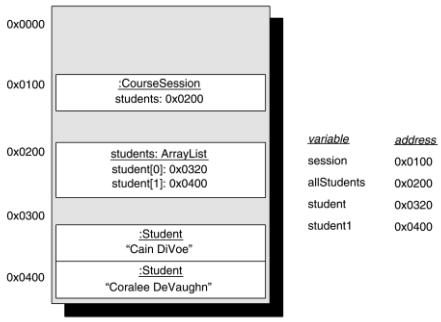
```
public void testEnrollStudents() {
   CourseSession session = new CourseSession("ENGL", "101");

   Student student1 = new Student("Cain DiVoe");
   session.enroll(student1);
   assertEquals(1, session.getNumberOfStudents());
   java.util.ArrayList<Student> allStudents = session.getAllStudents();
   assertEquals(1, allStudents.size());
   assertEquals(student1, allStudents.get(0));

   Student student2 = new Student("Coralee DeVaughn");
   session.enroll(student2);
   assertEquals(2, session.getNumberOfStudents());
```

```
    assertEquals(2, allStudents.size());
    assertEquals(student1, allStudents.get(0));
    assertEquals(student2, allStudents.get(1));
}
```

原因在图 2.4 中阐述。CourseSession 对象包含 `students` 域。这意味着 `students` 域在 CourseSession 对象的整个生命周期都是可用的。每次发送 `getNumberOfStudents` 消息给 `session`，同一个 `students` 的引用被返回。该引用是一个内存地址，意味着任何使用该引用的代码最终都指向存储 `students` 的同一个内存地址。




77

图 2.4　内存图示

# 包和 import 语句

到目前为止，您一直在使用类 java.util.ArrayList 的全名。类的全名包括包名（本例中是 java.util）和类名（ArrayList）。

包提供了一种将相关的类进行分组的机制。包有几个用途：首先，将一组类打成包，给开发者提供了相当的便利，避免开发者必须同一时刻在成打、成百、甚至数千的类中查找。第二，类被打成包也有发布的目的，可以方便地重用模块或者子系统。

第三，包在 Java 中提供了命名空间。假设您开发了 Student 类，而且您购买了第三方 API 来处理“学费账单”。假如第三方软件中也包含了名字叫 Student 的类，那么任何对 Student 的引用都会有歧义。包提供了赋予类一个唯一名字的机制，从而最小化类命名冲突。您的类也许有全称 com.mycompany.studentinfosystem.Student，第三方 API 也许会用 com.thirdpartyco.expensivepackage.Student。

使用 Java 系统类，我将不再加上包名，除非不清楚某个类所属的包。例如，我将用 ArrayList 来代替 java.util.ArrayList，用 Object 来代替 java.lang.Object。

在代码中输入 java.util.ArrayList 十分冗长乏味，而且使代码混乱。Java 提供了关键字——import——在源代码级指定类名和（或）包名。使用 import 语句允许您在其余的代码中只用简单地指定类名。

更新 CourseSessionTest，在源文件的第一行包含 import 语句。您可以将语句 extends junit.framework.TestCase 缩短成 extends TestCase。您可以将引用定义 java.util.ArrayList<Student>改变成 ArrayList<Student>。

```
import junit.framework.TestCase;
import java.util.ArrayList;

public class CourseSessionTest extends TestCase {
   ...
public void testEnrollStudents() {
     CourseSession session = new CourseSession("ENGL", "101");

     Student student1 = new Student("Cain DiVoe");
     session.enroll(student1);
     assertEquals(1, session.getNumberOfStudents());
     ArrayList<Student> allStudents = session.getAllStudents();
     ...
   }
}
```

78

（确保您的测试依然可以通过。我暂时提醒您一下。）

在 AllTests、StudentTest 和 CourseSession 中都使用 import 语句。您的代码将看起来更整洁。

# java.lang 包

String 类也是 java 类库的一部分，它属于包 java.lang。所以可以使用类的全名（java.lang.String）或者提供 `import` 语句。

Java 类库包含一些对于 Java 编程十分基础的类，所以它们被很多类所使用。String 和 Object 就是这样的两个类。这些类的特性广泛地被应用，所以 Java 的设计者希望避免处处都不得不加上 `import` 语句，否则会令人觉得很麻烦。

假如您调用了 String 类，那么下面的语句：

```
import java.lang.String;
```

会隐含加入到每一个 Java 源文件。

当学到第六课的继承，您会发现每个类都隐含地继承自 java.lang.Object。换句话说，即使一个类声明中没有包含 `extends` 关键字，也仿佛您已经编写了下面的代码：

```
class ClassName extends java.lang.Object
```

这是 Java 编译器提供的又一个便利。

# 默认包和 package 语句

您所编写的类 AllTests、StudentTest、Student、CourseSessionTest 和 CourseSession 中，都没有指定一个包。这意味着它们都在一个默认的包中。默认包对于例子或者非商业程序是允许的。

79

但是，对于任何真正的软件开发，所有类都应该属于某些包而不是默认包。事实上如果您把类放置在默认包，那么您将不能从其它包来使用这些类。

如果您用的是 IDE，将类名拖放到某个包名，就可以将该类移入到这个包中。如果您没有使用 IDE，那么设置包、并且理解相关的 classpath 问题可能有些复杂。即使您用的是 IDE，理解包与包之间的关系和隐含的文件系统目录结构也是十分重要的。

让我们试着将所有的类都移到一个叫 `studentinfo` 的包。注意：约定包名由小写字母组成。当您继续避免使用缩写，包名会很快变得难以处理。合理的使用缩写可以帮助您避免包名变得难以管理。

您不能以 `java` 或者 `javax` 作为包名的开始，因为 Sun 已经使用了它们。

如果在您的 IDE 中，把类移动到某个包是非常简单的操作，那么继续利用它。但是，要确信您理解类是如何关联到某个包结构的，因为这对您构建和部署库十分重要。无论如何，请将源文件复制到另一个目录，然后照下面的去做。

在 class 文件所在的目录，创建一个子目录 `studentinfo`。大小写十分重要——确保子目

录名称全是小写字母。目前源文件在 c:\source，所以您应该有一个目录 c:\source\studentinfo。如果是 Unix 目录/usr/src，那么您应该有一个目录 /usr/src/studentinfo。

首先，请小心地删除所有生成的 class 文件。记住 class 文件的扩展名是 class。如果对这一步骤不确信的话，请备份您的目录——不要遗失您的辛苦工作。在 Windows 平台下，使用命令：

```
del *.class
```

将完成任务。在 Unix，相应的命令是：

```
rm *.class
```

删除 class 文件后，将五个源文件（AllTests.java、StudentTest.java、Student.Java、CourseSession Test .java 和 CourseSession.java）移动到 studentinfo 目录。您需要将文件存储到和包名相对应的目录结构中。

80

逐个编辑五个 Java 源文件。在每个文件中加入 package 语句，以此标记每个类都属于 studentinfo 包。package 语句必须在文件的第一行。

```
package studentinfo;

class Student {
...
```

现在，您可以在 studentinfo 子目录中成功编译所有的类。

但是，如果使用 JUnit 来运行 AllTests，那么有两处需要有所改变。第一，类的全称必须传入 TestRunner，否则 JUnit 无法找到它。其次，如果当前目录是 studentinfo，TestRunner 将不能找到 studentinfo.AllTest。您必须回到父目录，或者最好是改变 classpath 来明确指定父目录。下面的命令显示了改变后的 classpath 以及测试类的全称。

```
java -cp c:\source;c:\junit3.8.1\junit.jar junit.awtui.TestRunner
  studentinfo.AllTests
```

事实上，如果您在 classpath 中明确指定了 c:\source，那么当执行 java 命令时您可以进入任何目录。同样，您也可以在 javac 编译时使用类似的 classpath：

```
javac -classpath c:\source;c:\junit3.8.1\junit.jar studentinfo\*.java
```

可以从另一个角度看这个问题：classpath 指定了起点，或者叫“根”目录，您可以从根目录出发来查找任何类。Class 文件必须在某一个根的子目录中，并且与包名相对应。例如，假如 classpath 是 c:\source，并且您打算包含类 com.mycompany.Bogus，那么文件 Bogus.class 必须在目录 c:\source\com\mycompany 中。

您的开发团队应该就包的命名规约达成一致。多数公司将他们的域名倒过来，作为包名的

开始。例如，一个名叫 Minderbinder Enterprises 的公司或许用 com.minderbinder 作为包名的开始。

## setUp 方法

CourseSessionTest 中的测试代码需要做一些清除工作。请注意：两个测试——testCreate 和 testEnrollStudents 方法，都初始化一个新的 CourseSession 对象，并存储该对象的引用到一个局部变量 session。

81

JUnit 提供了一种消除此类冗余的方法——setUp 方法。如果您在 setUp 方法中编写代码，JUnit 将在执行每个测试方法之前先执行 setUp 方法中的代码。您可以将公共的测试初始化代码放在 setUp 中。

```
public class CourseSessionTest extends TestCase {
   private CourseSession session;

   public void setUp() {
      session = new CourseSession("ENGL", "101");
   }

   public void testCreate() {
      assertEquals("ENGL", session.getDepartment());
      assertEquals("101", session.getNumber());
      assertEquals(0, session.getNumberOfStudents());
   }

   public void testEnrollStudents() {
      Student student1 = new Student("Cain DiVoe");
      session.enroll(student1);
      ...
   }
}
```

**注意**

编写 setUp 时，很容易犯这样的错误：将 session 声明为局部变量：

```
public void setUp() {
   CourseSession session =
      new CourseSession("ENGL", "101");
}
```

定义局部变量的名字和某个成员变量的名字相同，这是合法的。但是，这样意味着无法正确地初始化成员变量 session。结果会导致空指针异常（NullPointerException），在第四课您会了解什么是空指针异常。

在 CourseSessionTest 中，添加成员变量 session，并将 setUp 方法中新建的 CourseSession 赋值给该变量。这样，测试方法 testCreate 和 testEnrollStudents 就不再需要初始化代码。两个测试方法会得到各自独立的 CourseSession 实例。

尽管您可以创建构造函数，并在构造函数中编写公共的初始化代码，但这样做是相当差的

82

实践。最好是在 `setUp` 方法中完成测试初始化。

# 更多的重构

方法 testEnrollStudents 的代码没有必要这么长。该方法包含了太多的跟踪学生人数的断言。总的来说，这个方法稍微有点不好理解。

不要将整个 students 列表都暴露给客户代码（例如，其它处理 CourseSession 对象的代码），可以让 CourseSession 以指定索引的方式来返回 Student 对象，这样您就不需要整个 students 列表，从而可以删掉 getAllStudents 方法。这也意味着您不再需要去测试 getAllStudents 返回的 ArrayList 的尺寸。测试方法可以简化如下：

```
public void testEnrollStudents() {
   Student student1 = new Student("Cain DiVoe");
   session.enroll(student1);
   assertEquals(1, session.getNumberOfStudents());
   assertEquals(student1, session.get(0));

   Student student2 = new Student("Coralee DeVaughn");
   session.enroll(student2);
   assertEquals(2, session.getNumberOfStudents());
   assertEquals(student1, session.get(0));
   assertEquals(student2, session.get(1));
}
```

在 CourseSession 中增加 get 方法，并删除 getAllStudents 方法。此次重构显示了如何将公共代码从测试类直接移入到生产类，从而消除冗余。

```
class CourseSession {
   ...
   ArrayList<Student> getAllStudents() {
      return students;
   }

   Student get(int index) {
      return students.get(index);
   }
}
```

此次重构的另一个好处是隐藏了 CourseSession 中不必要暴露的细节。您封装了 students 集合，只可以用 get、add 和 size 方法来处理该集合。

封装提供了两个重要的优点：首先，目前您在 ArrayList 中保存 student 列表。ArrayList 是有着特定用法和性能指标的一种数据结构。如果您将该列表直接暴露给客户代码，客户代码将依赖于用 ArrayList 存储 students 的事实。这种依赖意味着您无法轻易地改变 students 的存储形式。第二，暴露完整的集合意味着其它类可以操作该集合——加入新的 Student 对象，删除 Student 对象，诸如此类——而且 CourseSession 类无法意识到这些改变。CourseSession 对象的完整性将遭到破坏。

# 类常量

像前面所提到的，直接把诸如字符串或者数字嵌在代码中不是个好主意。将局部变量声明为 final 是个好方法，这样可以防止其它代码修改该变量的值。final 关键字告诉其它程序员“我不打算让您修改这个变量的值”。

### 特质：final 关键字

您可以将局部对象和简单变量标记为 final:

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

您也可以将参数标记为 final:

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

这样提供了额外的保护措施，可以防止方法中的其它代码改变局部变量或者参数的值。很多程序员坚持这样做，而且这也不是一个坏的实践。

我没有选择这样做，但是推荐您试一试，看看它能怎样帮助您。对于我，将成员变量标记为 final，通常是为了可读性而不是为了保护。作为原则，您永远不要赋值给参数。而且，您很少应该为局部对象的引用重新赋值。我遵循这些原则，并且不觉得需要通过将成员变量标记为 final 来证明这些原则。相反，我使用 final 来强调某个局部声明是常量，而不止是一个初始化的变量。这是我的看法，您的看法可能和我的不一样。

84

您经常会在多个类中使用相同的字符串或者数字常量。事实上，如果您正在正确地进行测试驱动开发，任何时候在代码中创建一个字符串或者数字常量，都会在某个测试方法中利用断言来检查相同的字符串或者数字常量。某个测试可能会说：“断言在这些条件下会产生错误，并且错误输出是如此这般的一条消息”。一旦有了这样的测试，您编写出生产代码，在适当的条件下就会产生如此这般的错误输出。这样，您的代码就有“如此这般的错误输出”同时出现在测试代码和生产代码中。

尽管代码中出现一些重复的字符串并无大碍，不过将来您会看到消除此类重复是有价值的。一种可能的好处是有利于软件的成长。最初您可能只将系统部署给本国客户。慢慢的，您的软件取得了更大的成功，决定将软件部署到其它国家。这样您就需要国际化您的软件——支持其它语言并且考虑其它文化。

很多程序员遇到过这样的问题。他们已经开发某个软件数月甚至数年，代码中有几百甚至

几千个字符串常量。此时给软件加入国际化支持成了主要的障碍。

作为使用 Java 的软件人员，您的职责之一是——时刻警惕，一旦发现重复，立刻着手去消除它[5]。

 用类常量替换字符串或者数字。

用关键字 `static` 和 `final` 来声明类常量，类常量是成员变量。提醒一下，关键字 `final` 表明该成员变量的引用不能被改变，以指向不同的值。关键字 `static` 意味着在没有创建类实例的情况下就可以使用该成员变量。同时也意味着在内存中有且仅有一个成员变量，而不是每个创建的对象中都有成员变量。下面的例子声明了一个类常量：

```
class ChessBoard {
   static final int SQUARES_PER_SIDE = 8;
}
```

85

按照约定，用大写字母定义类常量。当所有的字母都是大写，用“驼峰模式”来标记就不大可能，所以标准方法是采用下划线来分割单词。

指定类名，类名后面是点操作符，再后面是常量的名字。用这样的顺序来使用类常量。

```
int numberOfSquares =
   ChessBoard.SQUARES_PER_SIDE * ChessBoard.SQUARES_PER_SIDE;
```

在下一节“Dates”中，您将使用 Java 类库中已经定义的类常量。在那以后，很快您将在 CourseSession 中定义自己的类常量。

## Dates

浏览 J2SE API 文档中关于包 java.util 的部分，您会看到几个和时间、日期相关的类，包括 Calendar、GregorianCalendar、Date、TimeZone 和 SimpleTimeZone。类 Date 提供简单的时间戳机制。其它相关的类与 Date 协作，针对国际化日期和时间戳处理，提供了完全的支持。

Java 的初始版本只提供了类 Date，来提供日期和时间的支持。类 Date 被设计来提供绝大多数必须的功能。类 Date 是一个简单的实现：在内部，时间被表示成自格林尼治标准时间（GMT）1970 年 1 月 1 日，00:00:00（也叫“epoch”） 以来的毫秒数。

## 重载构造函数

类 Date 提供了一组构造函数。提供给开发者以上构造新对象的方法，是可能的而且也是值

[5] 对于字符串重复问题，一个更好的方案是资源绑定。附加课Ⅲ中有资源绑定的简单讨论。

得的。本课将学到如何为您的类创建多个构造函数。

在类 Date 中，有三个构造函数允许您指定时间成分（年、月、日、小时、分钟或者秒），
86 以创建特定日期或者时间的 Date 对象。第四个构造函数允许您从输入的字符串构建 date 对象。第五个构造函数允许您用从 epoch 到现在的毫秒数来构建 Date 对象。最后一个构造函数没有参数，可以构建时间戳来表示当前时间，当前时间指的是该 Date 对象被创建时的时间。

同时 Date 也有很多用来获得/设置成员变量的方法，例如 `setHours`、`setMinutes`、`getDate` 和 `getSeconds`。

类 Date 不提供国际化时间的支持，但是，Java 设计者在 J2SE1.1 引入了类 Calendar。Calendar 用来作为类 Date 的补充。类 Calendar 提供设置时间成分的能力。这意味着，在 J2SE1.1 中类 Date 不再需要构造函数和 gettter/setters。Sun 用这种最清洁的方法清除了令人不愉快的构造函数和方法。

但是，假如 Sun 改变了 Date 类，大量已有的应用程序不得不重新编码、重新编译、重新测试、重新部署。Sun 为了避免出现这种不愉快的情况，在 Date 中不赞成使用这些构造函数和方法。也就是说，您现在依然可以使用这些构造函数和方法，但是 API 开发者警告您，他们将在 Java 的下一个主要版本中删除这些不被赞成的方法。如果您浏览 API 文档有关类 Date 的部分，您会看到 Sun 已经清楚地将相应的构造函数和方法标记为“不赞成”（deprecated）。

在这个练习中，您实际上还是在使用这些不被赞成的方法。您看到编译器会产生一些警告信息。警告很大程度是糟糕的——表明您的代码正在作一些不应该作的事情。总是存在更好的方法来避免编译器的警告信息。随着练习的深入，您将使用一个改进的方案来消除这些警告信息。

 在学生信息系统中，课程安排（CourseSession）需要开始时间和结束时间，用来标记课程的第一天和最后一天。您可以把开始时间和结束时间都提供给 CourseSession 的构造函数，但是您被告知总是 16 周（15 周课程，在第七周后会有一周的休息）。有了这些信息，您决定设计 CourseSession 类，让类的用户只需要提供开始时间——您的类将计算出结束时间。

下面是加入到 CourseSessionTest 中的测试：

```
public void testCourseDates() {
   int year = 103;
   int month = 0;
   int date = 6;
   Date startDate = new Date(year, month, date);

   CourseSession session =
87    new CourseSession("ABCD", "200", startDate);
year = 103;
   month = 3;
   date = 25;
   Date sixteenWeeksOut = new Date(year, month, date);
   assertEquals(sixteenWeeksOut, session.getEndDate());
```

```
}
```

您需要在 CourseSessionTest 的最上面加上 `import` 语句：

```
import java.util.Date;
```

这次，我们的代码使用 Date 的某一个过时的不被赞成的构造函数。同时也请注意传入到 Date 构造函数中的看起来很奇怪的参数。103 年？0 月？

API 文档应该是理解类库的第一手参考资料，它解释了传入的参数。特别地，文档中解释 Date 构造函数的第一个参数表示“年数减去 1900”，第二个参数表示“月份，在 0 到 11 之间”，第三个参数表示“一月中的天数，在 1 到 31 之间”。所以 `new Date(103,0,6)` 将创建一个表示 2003 年 1 月 6 号的 Date 对象。太有趣了。

由于开始时间对于定义 CourseSession 十分关键，所以您打算让构造函数接受一个开始日期参数。测试方法创建一个新的 CourseSession 对象，除了名称和课程编号，还传入最新创建的 Date 对象。您需要修改 `setUp` 方法中初始化 CourseSession 的代码，来使用这个修改后的构造函数。但是，作为一种过渡的、渐增的方法，您可以提供一个附加的、重载的构造函数。

该测试最后断言 `getEndDate` 返回的课程结束日期是 2003 年 4 月 25 号。

为了让测试通过，您应该在 CourseSession 中做出下列改变：

针对 java.util.Date、java.util.Calendar 和 java.util.GregorianCalendar，增加 `import` 语句。

增加 `getEndDate` 方法，计算和返回正确的课程结束日期。

增加一个新的构造函数，接受开始日期作为参数。

相应的生产代码为：

```
package studentinfo;

import java.util.ArrayList;
import java.util.Date;
import java.util.Calendar;
import java.util.GregorianCalendar;

class CourseSession {
   private String department;
   private String number;
   private ArrayList<Student> students = new ArrayList<Student>();
   private Date startDate;

   CourseSession(String department, String number) {
      this.department = department;
      this.number = number;
   }

   CourseSession(String department, String number, Date startDate) {
      this.department = department;
      this.number = number;
      this.startDate = startDate;
}
...
Date getEndDate() {
```

88

```
        GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(startDate);
        int numberOfDays = 16 * 7 - 3;
        calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
        Date endDate = calendar.getTime();
        return endDate;
    }
}
```

为了理解 getEndDate，请参考 J2SE API 文档中关于类 GregorianCalendar 和 Calendar 的部分。

在 getEndDate 中，先创建 GregorianCalendar 对象。然后使用 setTime[6]方法在该 calendar 中存储这个表示课程开始日期的对象。接着，创建一个局部变量 numberOfDays 来表示开始日期需要增加的天数，从而求出结束日期。正确的天数可以用 7 天（一周）乘以 16 周，然后再减去 3 天（因为课程的最后一天是第 16 周的星期五）。

下一行：

```
calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
```

发送 add 消息给 calendar 对象。GregorianCalendar 的 add 方法接受一个成员变量和一个数量作为参数。您将不得不阅读 J2SE API 文档有关 Calendar 的部分，以及有关 GregorianCalendar 的部分，来完全理解如何使用 add 方法。GregorianCalendar 是 Calendar 的子类，这意味着 GregorianCalendar 的工作方式与 Canlendar 紧密相关。第一个参数中的成员变量告诉 Canlendar 对象，什么是您想要增加的。在这个例子中，您打算把数字加到“一年中的第几天”。类 Calendar 定义了 DAY_OF_YEAR，以及其它几个类常量，来代表日期的某一部分，例如年份。

89

现在 canlendar 包含了代表课程结束日期的 date 对象。您使用 getTime 方法从 calendar 中得到日期，而且作为该方法的结果，最终返回 CourseSession 的结束日期。

您也许想知道，如果开始时间接近年终，getEndDate 方法能不能工作。如果这种情况可能发生，那么应该为它编写测试。但是，您正在开发的学生信息系统面向一个大约有 200 年历史的大学。没有任何学期从某一年开始，于下一年结束，而且这种情况将来也不会发生。简而言之，您不需要担心这样情况。

第二个构造函数是短暂的，但是它达到了让您快速通过测试这一目的。现在您应该删除这个旧的构造函数，因为它没有初始化课程的开始日期。为了初始化课程的开始日期，您需要修改 setUp 方法和 testCourseDates。同时您也需要修改创建测试，来验证可以正确存储开始日期。

```
package studentinfo;

import junit.framework.TestCase;
import java.util.ArrayList;
import java.util.Date;
```

[6] 不要将其当作好的方法命名的例子。

```
public class CourseSessionTest extends TestCase {
   private CourseSession session;
   private Date startDate;

   public void setUp() {
      int year = 103;
      int month = 0;
      int date = 6;
      startDate = new Date(year, month, date);
      session = new CourseSession("ENGL", "101", startDate);
   }
   public void testCreate() {
      assertEquals("ENGL", session.getDepartment());
      assertEquals("101", session.getNumber());
      assertEquals(0, session.getNumberOfStudents());
      assertEquals(startDate, session.getStartDate());
   }
   ...
```

90

```
public void testCourseDates() {
      int year = 103;
      int month = 3;
      int date = 25;
      Date sixteenWeeksOut = new Date(year, month, date);
      assertEquals(sixteenWeeksOut, session.getEndDate());
   }
}
```

现在您可以删除 CourseSession 的旧的构造函数。您也需要给 CourseSession 增加 getStartDate 方法：

```
class CourseSession {
   ...
   Date getStartDate() {
      return startDate;
   }
}
```

## 不赞成警告

前面的代码可以编译，也能够通过测试。但是，编译时输出了警告信息。如果您使用 IDE，或许您看不到这些信息。弄清楚如何在 IDE 中打开这些警告——您不应该隐藏它们。

消除所有的警告。

忽视编译警告就像忽视虫牙的危害——迟早您会为其付出代价，而且为您长期的忽视付出

昂贵的代价[7]。

**注意**：CourseSessionTest.java 使用或者重写了一个不被赞成的方法。

**注意**：为了获得更详细的信息，使用选项 -Xlint:deprecation 重新编译。

如果从命令行编译，您应该按照提示的去做：再次输入编译命令，并加上编译选项 -Xlint:deprecation。

```
javac -classpath c:\junit3.8.1\junit.jar -Xlint:deprecation *.java
```

新的编译输出应该像下面这样：

```
CourseSessionTest.java:15: warning: Date(int,int,int) in java.util.Date has been
deprecated
       startDate = new Date(year, month, date);
                   ^
CourseSessionTest.java:43: warning: Date(int,int,int) in java.util.Date has been
deprecated
       Date sixteenWeeksOut = new Date(year, month, date);
                              ^
2 warnings
```

91

如果是 IDE，应该可以修改某个设置来打开或者关闭警告。由于警告选项默认都是打开的，所以或许您已经看到了类似的不赞成信息。任何警告都应该触动您那软件工匠的敏感神经。也许这些警告会不停地来麻烦您。后面您将很快学到另一种创建 Date 对象的方法来避免此类警告信息。

测试会运行良好。但是您的代码中有很多应该消除的小瑕疵。

# 重构

有个可以立刻进行的改进是：删除不必要的局部变量 endData。该变量在 CourseSession 的方法 getEndDate 的末尾。声明这个临时变量有助于您对类 Calendar 的理解：

```
Date endDate = calendar.getTime();
return endDate;
```

可以用一种更简单的形式来返回这个调用 getTime 所得到的 Date 对象：

```
return calendar.getTime();
```

## Import 重构

类 CourseSession 从包 java.util 导入了四个不同的类：

```
import java.util.ArrayList;
import java.util.Date;
```

[7] 写这些文字，仿佛牙齿蛀烂了，坐在这儿。

```
import java.util.GregorianCalendar;
import java.util.Calendar
```

这样是合理的。但是，当更多的系统类被使用的时候，您会发现这个列表会长得很快失去控制。而且，每个 import 语句中都有重复的包名。请记住，重构的首要任务是尽可能消除重复。

92

Java 提供了一种使用 import 语句的捷径，来从指定的包中导入所有的类。

```
import java.util.*;
```

这种 import 的形式叫做包导入。星号起到了通配符的作用。

做出上面的改动之后，您可以使用包 java.util 中的任何其它类，而且不用修改或者增加 import 语句。请注意，使用包导入相对单个类，不会有任何运行时的损失。import 语句仅仅宣告某个类可以在这个 class 文件中使用。一个 import 语句不保证某个包中的所有类都可以在这个 class 文件中使用。

哪一种方式更正确，没有一致的意见。多数开发团队总是使用 * 形式的 import 语句，或者在 import 语句的数量变得难以控制时使用。一些开发团队坚持所有的类都必须使用 import 语句显示地声明，这样更容易知道每个类来自哪个包。现代 Java IDE 可以执行您的开发团队所制定的约定，甚至可以在不同的形式之间来回切换。IDE 使得我们采用何种形式不那么重要。

有可能导入了某个 class 或者包，但是您在源代码中没有使用这个 class，也没有使用包中的任何 class。Java 编译器对于此类不必要的 import 语句不会有任何警告。多数现代 IDE 提供了优化功能，帮助您删除掉不必要的 import 语句。

## 增进对工厂方法的理解

CourseSession 中的 getEndDate 是您所编写的所有方法中最复杂的。 您编写的多数方法的代码量在一行到六行之间。有些方法在六行到十二行左右。如果方法的代码行数超过这个长度或者更长一些，您就应该着手去重构它们。最主要的目标是保证方法能够被快速理解和维护。

如果方法足够短，我们就容易提供有意义的、简短的名字来命名这个方法。如果您发现为方法命名很困难，请考虑将其拆为几个更小的方法，每个方法只做一件简单的、可以命名的事情。

另一些不清晰的、让您不舒服的冗余出现在测试方法中。您暂时使用不被赞成的 Date 构造函数，这种技术比使用 Calendar 要简单一些。但是，这样的代码有些让人糊涂——因为要相对 1900 来指定年份，月份必须指定在 0 到 11 而不是 1 到 12 之间。

93

在 CourseSessionTest 中，创建一个新方法 createDate 来接受更容易理解的输入：

```
Date createDate(int year, int month, int date) {
  return new Date(year - 1900, month - 1, date);
}
```

这样，您可以使用四位的年份，和 1 到 12 之间的月份，来创建一个 Date 对象。

现在，您可以使用这个方法来重构 setUp 和 testCourseDates。根据该方法的说明，

定义局部变量 year、month 和 date，无助于增加对代码的理解，因为工厂方法[8]createDate 封装了混淆。您可以消除这些局部变量，将它们直接作为消息的参数传递给 createDate 方法：

```
public void setUp() {
   startDate = createDate(2003, 1, 6);
   session = new CourseSession("ENGL", "101", startDate);
}
...
public void testCourseDates() {
   Date sixteenWeeksOut = createDate(2003, 4, 25);
   assertEquals(sixteenWeeksOut, session.getEndDate());
}
```

或许有些人会对此摇头。已经做了很多工作，例如引入了局部变量。但是，很快又要取消这些工作。

雕琢代码的一部分工作是理解代码处于非常可锻造的形式。您能做的最好的事情是——时刻记住您是代码的雕刻师，不断地将代码塑造成更好的形式。偶尔，您对代码进行了某些修饰，后来却发现这些修饰事实上并不好，这时您可以向其他人请教更好的方案。您将学习到识别这

94

些代码中的麻烦点（就像 Martin Fowler 所说，“代码的臭味”）[9]。

您还将了解，等到代码与系统中的其它部分纠缠在一起时，再修改代码中的问题，比一开始就去修改，要付出更多的代价。不要等太久了！


时刻保持代码干净！

# 用 Calendar 创建日期

每次编译，您依然会收到不赞成信息。这太糟糕了。您应该在有人抱怨之前就清除这些警告。将创建日期的工作移到一个独立的方法 createDate 有这样的好处：为了消除警告，您将只需要改变代码中的一个地方，而不是两个地方。

用 GregorianCalendar 类替代不被赞成的构造函数，来创建 Date 对象。您可以使用 Calendar 中定义的 set 方法来创建日期或时间戳。Calendar 类的 API 文档列出了您可以设置的各种时间成分。createDate 方法通过提供年份、月份、月中的日期，来创建日期。

```
Date createDate(int year, int month, int date) {
   GregorianCalendar calendar = new GregorianCalendar();
   calendar.clear();
   calendar.set(Calendar.YEAR, year);
   calendar.set(Calendar.YEAR, year);
   calendar.set(Calendar.MONTH, month - 1);
   calendar.set(Calendar.DAY_OF_MONTH, date);
```

[8] 负责创建和返回对象的方法。另一种可能的更简练的术语是:”创建方法”。
[9] [Wiki2004]。

```
    return calendar.getTime();
}
```

GregorianCalendar 要比 Date 更有意义一些，在 GregorianCalendar 中年份就是年份。如果实际年份是 2005，那么您将 2005 作为参数传入 calendar 对象，而不是传入 105。

为了编译和测试这些改变，您需要修改 CourseSessionTest 中的 `import` 语句。最简单的方法是从包 java.util 中导入所有的类：

```
import java.util.*;
```

编译并且测试。恭喜您——不再有讨厌的不被赞成的警告信息了！

95

# 注释

`getEndDate` 中有一行代码用来计算天数与课程开始时间的和，这一行代码需要被澄清。

```
int numberOfDays = 16 * 7 - 3;
```

对于其它不得不维护这个方法的程序员而言，该数学表达式的意思并不显而易见。维护者可能要花上几分钟才能理解它，然而最初的开发者花上很小的代价就可以解释当时的想法。

Java 允许您用注释的形式，在源代码中自由地加上解释性文字。编译器处理源文件时，会忽略遇到的注释。您自己决定在什么地方、什么时候增加注释。

您可以给 `numberOfDays` 计算增加一个单行注释。单行注释以两个斜线(//)开头，一直持续到当前行的末尾。编译器会忽略从斜线到行尾的所有文字。

```
int numberOfDays = 16 * 7 - 3;  // weeks * days per week - 3 days
```

您也可以将单行注释放置在单独的行中：

```
// weeks * days per week - 3 days
int numberOfDays = 16 * 7 - 3;
```

但是，错误或者容易引起误解的注释是声名狼藉的。上面的注释是无效注释的经典例子。更好的方案是找到更清晰的表达代码的方法。

 用更有表现力的代码替代注释。

一个可能的方案：

```
final int sessionLength = 16;
final int daysInWeek = 7;
final int daysFromFridayToMonday = 3;
int numberOfDays =
```

```
sessionLength * daysInWeek - daysFromFridayToMonday;
```

哈，现在更有表现力了。但是，我不确定 daysFromFridayToMonday 是否给出了正确的解释。这表明永远没有完美的解决方案。重构不是一门精确的科学， 但是，不要停止努力。多数修改改进了我们的代码，某些人（或许就是您自己）总能在您之后提出更好的方法。现在就开始吧。 96

Java 还提供了多行注释。多行注释以两个字符 /* 开头，以两个字符 */.结束。编译器会忽略斜线之间的任何内容。

**注意：**多行注释中可以内嵌单行注释，但是多行注释不能内嵌其他多行注释。

例如，Java 编译器允许下面这样：

```
int a = 1;
/*  int b = 2;
//  int c = 3;
*/
```

但是，下面的代码编译无法通过：

```
int a = 1;
/*  int b = 2;
/*  int c = 3;  */
*/
```

您最好用单行注释为需要注解的代码增加注释。用多行注释来快速注释掉（关闭这些代码，使编译器不去处理）大块的代码。

# Javadoc 注释

多行注释的另一个用途是提供格式化的代码文档，代码文档可以用来自动生成具有精细格式的 API 文档。这样的注释也叫 javadoc 注释。利用 javadoc 工具扫描源代码文件，找到 javadoc 注释，从 javadoc 注释中提取必要的信息来生成网页格式的文档。Sun 的 Java API 文档就是用 javadoc 生成的。

javadoc 注释是多行注释。区别在于 javadoc 注释以/**开头而不是以/*开头。对于 java 编译器，这两种注释开头没有什么不同，因为都是以/*开头，以*/结尾。但是，javadoc 工具可以理解其中的不同。 97

javadoc 注释直接写在需要文档化的 Java 元素的前面。javadoc 注释可以在成员变量的前面，但是多数情况下，javadoc 注释被用来文档化类和方法。为了便于 javadoc 编译器正确分析，有一些格式化 javadoc 注释的规则。

制作 javadoc 网页的主要目的是为代码提供文档，以方便外部客户，例如其它项目团队或者

公众。尽管您可以为每一个 java 元素（成员变量、方法、类，等等）编写 javadoc 注释，但是通常只需要为您打算对外公布的 java 元素编写 javadoc 注释。javadoc 注释的作用是告诉程序员如何使用某个类。

如果某个团队正在进行测试驱动开发，那么 javadoc 注释的作用会很小。如果处理得当，您用测试驱动开发编写的测试就是最好的文档，测试最好地描述了某个类的能力。对于某些实践，例如结对编程和集体拥有代码，应用这些实践的程序员会工作在系统的每一个模块上，这也最小化了编写 javadoc 注释的需求。

如果方法的名字都十分简洁，并且命名得很好，有很好命名的参数，那么您需要编写 javadoc 注释的数量将是最小限度的。在缺乏文档的情况下，javadoc 做了很好的工作，javadoc 从代码中提取出您选择的需要解释的内容，并把它们很好的展现出来。

做个小练习：为 CourseSession 类提供 javadoc 注释——单参数的构造函数——并且为类的某个方法编写 javadoc 注释。

下面是我写的 javadoc 注释：

```
package studentinfo;

import java.util.*;

/**
 * Provides a representation of a single-semester
 * session of a specific university course.
 * @author Administrator
 */
class CourseSession {
   private ArrayList<Student> students = new ArrayList<Student>();
   private Date startDate;

   CourseSession() {
   }

   /**
    * Constructs a CourseSession starting on a specific date
    *
    * @param startDate the date on which the CourseSession begins
    */
   CourseSession(Date startDate) {
      this.startDate = startDate;
   }

   /**
    * @return Date the last date of the course session
    */
   Date getEndDate() {
   ...
}
```

98

请特别注意 javadoc 注释中的关键字@。当看到用 javadoc 命令生成的网页时，您就会很容

易理解 javadoc 编译器是如何处理注释的。您最经常用到的 javadoc 关键字是@param，该关键字用来描述参数，另外是用来描述方法返回值的关键字@return。

javadoc 中有很多规则和@关键字。参考 javadoc 文档可以得到进一步的信息。在 Java SDK 的 API 文档（在线浏览或者下载）的 tooldocs 子目录中可以找到有关 javadoc 的内容。

一旦您在代码中写完了注释，回到命令行（如果您使用 IDE，您也许能在 IDE 中生成文档）。您应该创建一个空目录来存放生成的文档，这样在重新生成文档时您可以很容易地删除它。进入到这个空目录[10]，然后输入下面的命令：

```
javadoc -package -classpath c:\source;c:\junit3.8.1\junit.jar studentinfo
```

javadoc 程序会生成一些.html 文件和一个样式表(.css)文件。在浏览器中打开 index.html，花几分钟看看这些简单的命令产生了什么样的效果（见图 2.5）。相当有表现力，不是吗？

噢，我认为这样既好又不好。对于添加到方法中的注释，我感到困窘。这些注释并没有增加代码中所没有描述的内容。关键字@param 只是简单的重新描述了本来可以从参数类型和名字得到的信息。关键字@return 重新描述了本来可以从方法名和返回类型得到的信息。如果您发现了需要@return 和@param 关键字的理由，那么请努力重新命名参数和方法来消除这个需要的理由。

99

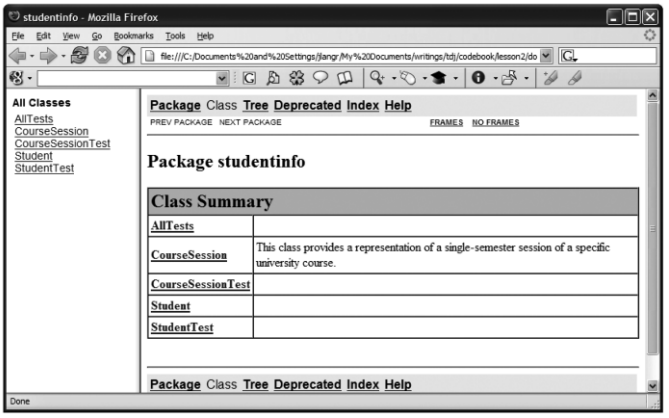



图 2.5　您的 API 页面

删掉所有构造函数和方法的注释，只留下类注释为读者提供一些方便。然后返回到 javadoc 命令，重新生成网页，看看是否丢失了什么有用的信息。您的看法也许和我不同。

[10] 也可以不进入该空目录，-d 开关可以重定向 javadoc 命令的输出。

# 练习

1. 为 TestPawn 增加一个测试，创建一个没有颜色的卒。为什么会有编译错误（提示：考虑默认构造函数）？通过增加第二个构造函数（默认创建一个白色的卒）来修改这个编译错误。
2. 设置这两种颜色为静态常量，并将它们添加到类 Pawn。
3. 没有棋盘的话，卒就起不了作用。用一个测试来定义类 Board 。断言棋盘开始时所有的格子都是空的。按照 TDD 的顺序：编写尽可能最小的测试，用红条或编译错误证明失败，然后渐增地添加代码以获得编译成功或者看到绿条。
4. 编写代码，允许卒可以被添加到棋盘上去。在某个测试中，各添加一个白色的卒和黑色的卒到棋盘上。每次添加一个卒，断言上面有棋子的格子的数目是正确的。而且，每增加一个卒，获得棋盘上有棋子的格子的列表，保证该列表包含预期的卒对象。

100

5. 为迄今为止的每一个生产类和方法编写 javadoc。小心地按照这样的原则去做：不要重复方法本身已经给出的信息！javadoc 只用来提供辅助信息。
6. 把您已经创建的四个测试和类移动到某个包中。将包命名为 `chess`。解决编译错误，并且再次得到绿条。利用 `import` 语句，替换 List 和 ArrayList 的全称类名。
7. 将 TestPawn 和 Pawn 移到包 `pieces` 中。解决发现的任何问题。
8. 保证除了卒，没有别的可以被添加到棋盘。试着将 `new Integer("7")` 添加到卒列表，看看由此导致的编译错误。
9. 创建测试套件，运行所有的测试。
10. 审核迄今为止您编写的所有代码。保证代码中没有任何冗余。记住测试代码也是代码。在合适的时候使用 `setUp` 方法。

101