3 Strings and Packages

字符串和包

本课内容包括：

- 更多关于 String 类的知识
- 字符在 Java 内部的表示
- 使用系统属性来保证平台无关的编码
- 使用 StringBuilder 来动态创建字符串
- 了解如何遍历集合，从而处理集合中的每一个对象
- 使用 System.out 来打印输出
- 利用包来组织类
- 加深对访问修饰符 `public` 和 `private` 的理解

字符和字符串

字符串，或者说一段文本，在执行一个典型的 Java 程序时，占据了已创建对象的百分之五十甚至更多。String 是对象，它们由一串单个的字符连接而成。Java 用基本类型 `char` 来表示字符。由于 `char` 是基本类型（就像 int），所以请记住您不能向 `char` 发送消息。

字符

Java 用 `char` 类型来表示字母、数字、标点符号、重音符号、以及其它特殊符号。Java 基于标准字符集 Unicode4.0 来表示每一个字符。Unicode 的设计目标是容纳世界上所有主要语言中的字符。可以从这个网址得到更多关于 Unicode 的资料：*http://www.unicode.org/*。

Java 使用两个字节来存储一个字符。两个字节是 16 位，这意味着 Java 可以表示 2^{16}，即 65 536 个字符。虽然看起来很多，但是 2^{16} 并不足以支持 Unicode 标准中的所有字符。您可能不需要支持超过双字节范围的字符，但是如果您需要，Java 允许使用 int 类型来处理字符。类型 int 字长为四个字节，所以可以支持数十亿的字符，甚至美国政府要求我们支持 Romulan 字母表，这都是

足够的。

在 Java 中，您可以用几种方式来表示字符。最简单的方式是用一对单引号来嵌入字符。

```
char capitalA = 'A';
```

语言测试

尽管敏捷 Java 一书中的大部分代码都是学生信息系统这个例子的一部分，不过我也用简短的代码段或者断言来展示 Java 语法的细节和变化。这一节中的单行断言提供了一个例子。我将其称之为语言测试——编写代码或者断言来学习这门语言。您可以保留它们，这样您在稍后需要用到它们时，可以帮助您理解。

您可以在任何喜欢的地方编写此类测试。我一般在当前测试类，单独的测试方法中编写它们，理解之后再删除此类测试方法。

您或许选择创建单独的类去包含此类测试。最终，您甚至为此类测试创建了包，测试套件，甚至测试项目。

您或许可以重用此类测试：一些语言测试最终变成了某些封装的实用方法的基础，简化了某些语言特性。

字符从本质上讲是数字。每一个字符映射到一个范围在 0 到 65 535 的正整数。下面有一段测试代码说明了：字符'A'有一个对应的数字值 65（字符'A'对应的 Unicode 值）。

```
assertEquals(65, capitalA);
```

并不是所有的字符都可以通过键盘输入到计算机。您可以用 Unicode 转义符（\u 或者\U，后面跟着四位的十六进制数字）来表示 Unicode 字符：

104

```
assertEquals('\u0041', capitalA);
```

此外，您还可以用三位八进制转义符：

```
assertEquals('\101', capitalA);
```

最大的八进制转义符是'\377'，该转义符等于 255。

多数比较老的语言（例如 C）用单字节来表示字符。最著名的单字节字符集（SBCS）标准是美国标准信息交换码（ASCII），ANSI X3.4[1]定义了该标准。

特殊字符

Java 定义了一些用来格式化输出的特殊字符。Java 使用转义符（escape sequence）来表示这些特殊字符，该转义符由一个反斜线（\）和一个随后的助记符组成。下面的表格总结了表示这

[1] 事实上，ASCII 是一个七位编码标准，只表示了从 0 到 127 的字符。但是，有几个相互竞争的标准，定义了从 128 到 255 的字符。

些特殊字符的字符序列：

回车	'\r'
换行	'\n'
Tab	'\t'
换页	'\f'
退格	'\b'

由于一对单引号和反斜线对于字符表示有特殊的意义，所以您必须用转义符来表示它们。您也许会转义（以转义符号\作为前缀）双引号字符，但是您并不必须这样做。

单引号	'\''
换码符	'\\'
双引号	'\"'

105

字符串

字符串对象表示固定长度的字符序列。Java 中的 String 类可能是任何 Java 程序中最常用的类。甚至在小型的程序中，数千的字符串对象也会被一再地创建。

String 类提供了很多方法。String 类有特殊的性能特性，使得 String 类和系统中的其它类不一样。最后，即使 String 和系统中的其它类相同，Java 语言也为和 String 对象协同工作提供了特殊的语法支持。

您可以用多种方式来创建字符串。每次创建字符串，Java 虚拟机都会在背后创建一个 String 对象。下面有两种创建字符串对象的方式，并且将字符串对象赋值给一个变量：

```
String a = "abc";
String b = new String("abc"); // DON'T DO THIS
```

避免使用第二种方式[2]。第二种方式创建了两个 String 对象，这样降低了性能：首先，Java 虚拟机创建了 String 对象“abc”。然后，java 虚拟机创建一个新的 String 对象，并把字符串“abc”传入构造函数。同样重要的是，这是一次不必要的构造，使得您的代码阅读起来更加困难。

由于字符串是一个字符序列，所以可以嵌入特殊字符。下面的字符序列中包含一个 tab 字符和一个换行符：

```
String z = "\t\n";
```

[2] [Bloch2001]。

字符连接

您可以将一个字符串和另一个字符串连接起来，从而生成第三个字符串。

```
assertEquals("abcd", "ab".concat("cd"));
```

在 Java 中，字符串连接是一个非常常用的操作。您可以用加号（+）作为字符串连接的捷径。事实上，多数 Java 连接操作用下面的方式：

106

```
assertEquals("abcdef", "abc" + "def");
```

由于连接两个字符串的结果是生成另一个字符串，所以您可以用多个加号（+）操作来将多个字符串连接成一个字符串：

```
assertEquals("123456", "12" + "3" + "456");
```

上一课中，您使用加号（+）来做整型数加法。Java 也允许使用加号操作符（+）来连接字符串。因为加号操作符根据不同的用途有着不同的意义，所以这种用法被称之为操作符重载。

字符串的不可改变性

浏览 Java API 文档关于 String 的部分，您会注意到，没有任何方法可以改变字符串。您不能改变字符串的长度，也不能改变字符串所包含的任何字符。字符串对象是不能改变的。如果您想对某个字符串做任何操作，您必须创建一个新字符串。举例来说，当您用加号连接了两个字符串，Java 虚拟机没有改变其中任何一个字符串，而是创建了一个新的 String 对象。

Sun 将 String 设计成不可改变的，这是为了让 String 的行为最优化。因为 String 在多数应用中都被大量使用，所以它的优化是非常关键的。

StringBuilder

有时您需要动态创建字符串。类 java.lang.StringBuilder 提供了这样的能力。新创建的 StringBuilder 表示空字符序列或者字符集合。您可以通过向 StringBuilder 对象发送 `append` 消息，来往该集合中增加字符。

就像 Java 重载加号操作符，从而支持整型数加法和字符串连接。类 StringBuilder 也重载了 `append` 方法来接受不同基本类型的参数。您可以传入字符、字符串、`int`、或者其它类型，来作为 `append` 的参数。参考 Java API 文档，可以看到重载方法的列表。

当针对 StringBuilder 完成所有 `append` 操作后，您可以通过向 StringBuilder 发送 `toString` 消息，从而得到一个连接起来的 String 对象。

学生信息系统的用户需要一个报表，来列出课程的清单。目前，一个简单的只包含学生姓名，而且没有任何排序的文本报表，已经够用了。

在 CourseSessionTest 中编写下面的测试。断言说明该报表需要一个简单的页眉和显示学生人数的页脚：

```
public void testRosterReport() {
   session.enroll(new Student("A"));
   session.enroll(new Student("B"));

   String rosterReport = session.getRosterReport();
   assertEquals(
      CourseSession.ROSTER_REPORT_HEADER +
      "A\nB\n" +
      CourseSession.ROSTER_REPORT_FOOTER + "2\n", rosterReport);
}
```

（请记住 testRosterReport 使用 CourseSessionTest 的 setUp 方法中创建的 CourseSession 对象。）用下面的代码来更新 CourseSession。

```
String getRosterReport() {
   StringBuilder buffer = new StringBuilder();

   buffer.append(ROSTER_REPORT_HEADER);

   Student student = students.get(0);
   buffer.append(student.getName());
   buffer.append('\n');

   student = students.get(1);
   buffer.append(student.getName());
   buffer.append('\n');

   buffer.append(ROSTER_REPORT_FOOTER + students.size() + '\n');

   return buffer.toString();
}
```

对每一个学生，传入一个 String（该学生的姓名）给 append 方法，接着传入一个字符（换行符）给 append 方法。您还需要将页眉和页尾信息附加到 buffer（StringBuilder 对象）中。名字 buffer 暗示 StringBuilder 保存了一个字符集合，该集合将来会用到。构建页脚的一行代码展示了如何将一个连接字符串作为参数，传入 append 方法。

您在类 CourseSession 中定义了 getRosterReport 方法。在同一个类中，可以直接调用静态变量。所以可以不用：

```
CourseSession.ROSTER_REPORT_HEADER
```

而是使用下面这种方式：

```
ROSTER_REPORT_HEADER
```


在第 4 课您将学到更多关于 static 关键字的知识。您将了解到，我们通过限定类名来调用静态变量和静态方法（就像 CourseSession.ROSTER_REPORT_HEADER），甚至在定义静态变量和静态方法的类中也可以用同样的方式调用。否则的话，就会使“你在使用静态元素”这个事实变得没有说明效果，从而导致一些麻烦的缺陷。然而，对于类常量而言，该命名规约

（UPPERCASE_WITH_UNDERSCORE，带下划线的大写字母）能够明晰“你在使用静态元素”这个情况。如此一来，第二种无限定类名的形式也就还可以接受（尽管有些厂商会禁止这么做），算是规则中的例外情况。

如果去阅读比较老的 Java 代码，您会看到使用类 java.lang.StringBuffer。与 StringBuffer 对象的交互和 StringBuilder 对象是一样的。两者之间的区别在于 StringBuilder 提供了更好的性能。不需要支持多线程应用，多线程应用中可能出现两段代码同时操作一个 StringBuffer 的情况。（另外，当两段代码同时操作一个 StringBuffer 时）请参考十三课关于多线程的讨论。

系统属性

getRosterReport 方法以及相应的测试，都在很多地方使用‘\n’表示换行符。这样做不仅有冗余，而且难以移植——不同平台使用不同的特殊字符序列来表示换行。类 java.lang.System 中可以找到这个问题的解决方案。和前面一样，参考 J2SE API 文档来获取对这个类的深入理解。

该类包含了方法 getProperty，此方法以一个系统属性的键值作为参数，并返回与该键值相关联的系统属性。Java 虚拟机在启动的时候，就设置好了若干个系统属性。多数属性返回与虚拟机以及当前执行环境相关的信息。API 文档中针对 getProperties 方法，给出了可用的属性列表。

其中一个属性是 line.separator。参考 Java API 文档，在 Unix 中该属性的值为‘\n’。然而，在 Windows 中，该属性的值是；‘\r\n’。在代码中，您应该使用 line.separator 来弥合不同平台之间的差异。

下面对测试和 CourseSession 所做的改动，展示了系统方法 getProperty 的用法。

测试代码：

```
public void testRosterReport()
{
   Student studentA = new Student("A");
   Student studentB = new Student("B");
   session.enroll(studentA);
   session.enroll(studentB);

   String rosterReport = session.getRosterReport();
   assertEquals(
      CourseSession.ROSTER_REPORT_HEADER +
      "A" + CourseSession.NEWLINE +
      "B" + CourseSession.NEWLINE +
      CourseSession.ROSTER_REPORT_FOOTER + "2" +
      CourseSession.NEWLINE, rosterReport);
}
```

109

生产代码：

```
class CourseSession {
   static final String NEWLINE =
      System.getProperty("line.separator");
   static final String ROSTER_REPORT_HEADER =
      "Student" + NEWLINE +
      "-----" + NEWLINE;
   static final String ROSTER_REPORT_FOOTER =
      NEWLINE + "# students = ";
   ...
   String getRosterReport() {
      StringBuilder buffer = new StringBuilder();

      buffer.append(ROSTER_REPORT_HEADER);

      Student student = students.get(0);
      buffer.append(student.getName());
      buffer.append(NEWLINE);

      student = students.get(1);
      buffer.append(student.getName());
      buffer.append(NEWLINE);

      buffer.append(ROSTER_REPORT_FOOTER + students.size() + NEWLINE);

      return buffer.toString();
   }
}
```

遍历所有的学生

测试方法 testRosterReport 展示了如何生成一个包含两个学生的报表。您知道，创建报表的代码基于这样的假设：只有两个学生。 110

您需要编写代码来支持数量不限的学生。为了实现这个目标，需要修改您的测试以招收更多的学生。然后，您意识到生产类包含了重复的代码——重复的数量只会变得更糟——每个学生都需要相同的三行代码，唯一不同的只是学生的序号。

您愿意做的是：针对数组中的每个学生，执行相同的三行代码，而不管数组中包含了多少个学生。Java 提供了几种方法来实现这个目标。在 J2SE5.0 中最简单的方法是使用 for-each 循环[3]。

For-each 循环有两种形式。第一种形式：在循环声明的后面是一对花括号，允许您在循环体中指定多行语句。

```
for (Student student: students) {
```

[3] 相对过去的循环，功能有所增强。

```
      // ... statements here ...
   }
```

第二种形式只允许您在循环体中指定一行语句，因此不需要花括号：

```
for (Student student: students)
  // ... single statements here;
```

在第七课，您将学习另一种允许循环确定次数的 `for` 循环，而不是在一个集合中循环遍历每一个元素。

Java 虚拟机针对集合 `students` 中的每一个学生，执行一次 `for` 循环体。

```
String getRosterReport() {
   StringBuilder buffer = new StringBuilder();

   buffer.append(ROSTER_REPORT_HEADER);

   for (Student student: students) {
      buffer.append(student.getName());
      buffer.append(NEWLINE);
   }


   buffer.append(ROSTER_REPORT_FOOTER + students.size() + NEWLINE);

   return buffer.toString();
}
```

上面的 `for-each` 循环读起来像一篇英语散文：将集合 `students` 中的每一个对象赋值给一个类型为 Student 的引用，该引用的名字是 `student`。然后在这个上下文中执行循环体。

111

单职责原则

 学生信息系统会不断需要新的报表。您现在或许已经被告知需要再生成三个报表。而且可以预计将来还会要求新的报表。可以预见，因为增加报表，需要不断地改变类 CourseSession。

面向对象有一个最基本的设计原则：一个类只做好一件事情。由于只做一件事情，所以改变类应该只有一个动机。这就是单职责原则[4]。

 类的改变应该只有一个动机。

CourseSession 应该做的唯一事情是跟踪与课程安排有关的所有信息。在 CourseSession 中增加存储教授信息的能力，这应该是符合类 CourseSession 主要实现目标的动机。生成报表，例如报名表，是改变类 CourseSession 的另一个动机，但是它违背了单职责原则。

创建一个测试类 RosterReporterTest，来展示如何用一个新的、单独的类 RosterReporter 来生

[4] [Martin2003]。

成报名表。

```
package studentinfo;

import junit.framework.TestCase;
import java.util.*;

public class RosterReporterTest extends TestCase {
   public void testRosterReport() {
      CourseSession session =
         new CourseSession("ENGL", "101", createDate(2003, 1, 6));

      session.enroll(new Student("A"));
      session.enroll(new Student("B"));

      String rosterReport = new RosterReporter(session).getReport();
      assertEquals(
         RosterReporter.ROSTER_REPORT_HEADER +
         "A" + RosterReporter.NEWLINE +
         "B" + RosterReporter.NEWLINE +
         RosterReporter.ROSTER_REPORT_FOOTER + "2" +
         RosterReporter.NEWLINE, rosterReport);
   }

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

112

方法 testRosterReport 和 CourseSessionTest 中的几乎一样。主要的区别是（如代码中黑体所示）：

- 以 CourseSession 对象为参数，创建了一个 RosterReporter 实例。
- 在 RosterReporter 而不是 CourseSession 中，使用声明的类常量。
- testReport 创建它自己的 CourseSession 对象。

您也应该注意到 CourseSessionTest 和 RosterReporterTest 中都有 createDate 方法，这造成了冗余。您将很快进行重构来消除冗余。

将新的测试添加到 AllTests 中：

```
package studentinfo;

import junit.framework.TestSuite;

public class AllTests {
   public static TestSuite suite() {
      TestSuite suite = new TestSuite();
      suite.addTestSuite(StudentTest.class);
```

```
        suite.addTestSuite(CourseSessionTest.class);
        suite.addTestSuite(RosterReporterTest.class);
        return suite;
    }
}
```

在测试通过的过程中，需要从 CourseSession 移走部分代码。以渐增的方式进行代码的移动——直到 RosterReporter 工作正常，才开始对 CourseSession 和 CourseSessionTest 进行改动。

113

```
package studentinfo;

import java.util.*;

class RosterReporter {
    static final String NEWLINE =
        System.getProperty("line.separator");
    static final String ROSTER_REPORT_HEADER =
        "Student" + NEWLINE +
        "-" + NEWLINE;
    static final String ROSTER_REPORT_FOOTER =
        NEWLINE + "# students = ";

    private CourseSession session;

    RosterReporter(CourseSession session) {
        this.session = session;
    }

    String getReport() {
        StringBuilder buffer = new StringBuilder();

        buffer.append(ROSTER_REPORT_HEADER);

        for (Student student: session.getAllStudents()) {
            buffer.append(student.getName());
            buffer.append(NEWLINE);
        }

        buffer.append(
            ROSTER_REPORT_FOOTER + session.getAllStudents().size() +
            NEWLINE);

        return buffer.toString();
    }
}
```

上面例子中的黑体部分显示了 RosterReporter 和 CourseSession 在相关代码上的主要不同。

为了完成上面的代码，应首先将 CourseSession 的 `getReport` 函数体直接粘贴到 RosterReporter 的同名方法中。然后，在 RosterReporter 中修改粘贴过来的请求 students 集合的代码，用 `getAllStudents` 消息替换直接访问（因为这个方法在 CourseSession 中不再执行）。由于您在前一课删除了 `getAllStudents` 方法，所以您不得不将其再次添加到 CourseSession。

```
class CourseSession {
```

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

114

而且，为了在 RosterReporter 中能够发送消息给 CourseSession 对象，必须在 RosterReporter 中存储一个 CourseSession 的引用。通过将一个 CourseSession 对象作为 RosterReporter 构造函数的参数来实现这一目标。

接着，从 CourseSessionTest 和 CourseSession 中删除和报表相关的代码。包括测试方法 `testRosterReport`，生产方法 `getRosterReport`，以及定义在 CourseSession 中的类常量。重新运行所有的测试。

当前的类结构如图 3.1。

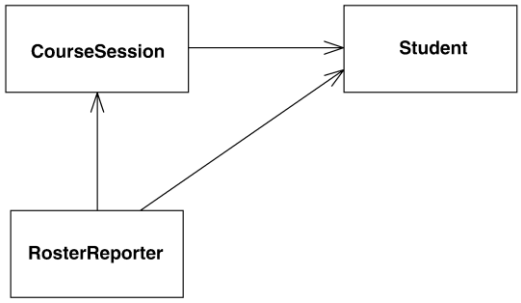

图 3.1　类图

重构

CourseSessionTest 和 RosterReporterTest 都需要 `createDate` 工具方法。`createDate` 中的代码没有针对课程安排和报名表做任何的处理，仅仅用来创建日期对象。类中包含少量有用的工具方法，是对单职责原则较轻微的违背。您可以忍受一些小规模的重复，但是这样做打开了大规模重复的大门，很快在您的系统中就会出现代价高昂的重复。在一个大型系统中，或许会有多个创建日期的方法，每个方法都有相似的代码。

此处的重复是显而易见的，因为您直接创建了它（希望能注意到）。有一种方法可以防止重复的发生：只要意识到可能正在引入重复的代码，马上进行必要的重构以避免潜在的重复。

115

您将创建一个新的测试类和生产类。您必须更新 AllTests 来引用这个新的测试类。三个类的代码如下。

```
// DateUtilTest.java
package studentinfo;

import java.util.*;
import junit.framework.*;

public class DateUtilTest extends TestCase {
   public void testCreateDate() {
      Date date = new DateUtil().createDate(2000, 1, 1);
      Calendar calendar = new GregorianCalendar();
      calendar.setTime(date);
      assertEquals(2000, calendar.get(Calendar.YEAR));
      assertEquals(Calendar.JANUARY, calendar.get(Calendar.MONTH));
      assertEquals(1, calendar.get(Calendar.DAY_OF_MONTH));
   }
}

// DateUtil.java
package studentinfo;

import java.util.*;

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

// AllTests.java
package studentinfo;

import junit.framework.TestSuite;

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

116

前面，createDate 方法没有相应的测试，因为它只是一个在测试类中才使用的工具方法。当从一个类中抽取代码去创建新类，您应该将存在的任何测试移动到相应的新测试类中。如果测试不存在，您应该花一些时间去创建相应的测试。这样做可以保持系统的可维护性。

既然您已经创建和测试了类 DateUtil，您应该更新代码去调用它。同时，您应该从 CourseSessionTest 和 RosterReporterTest 中删除 createDate 方法。一个稳妥的方法是从两个地

方删除 createDate 方法，然后编译。编译器会准确地告诉您调用了不存在的 createDate 方法的代码行。

 用编译器来帮助您重构代码。

改变代码行：

```
// CourseSessionTest
package studentinfo;

import junit.framework.TestCase;
import java.util.*;

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

// RosterReporterTest.java
package studentinfo;

import junit.framework.TestCase;

public class RosterReporterTest extends TestCase {
   public void testRosterReport() {
      CourseSession session =
         new CourseSession("ENGL", "101",
            new DateUtil().createDate(2003, 1, 6));
      ...
   }
}
```

为了使用工具方法 createDate，您每次都必须创建一个 DateUtil 对象。在 CourseSessionTest，您两次创建了 DateUtil 对象——这是重构的主要候选者。您可以创建一个实例变量来保存一个 DateUtil 实例。不过，一个更好的方案是将 DateUtil 转成静态方法——不用创建 DateUtil 实例就可以调用。在第 4 课您将学习如何使用静态方法。 117

System.out

getReport 方法返回了一个字符串，该字符串包含了报名学习某门课程的所有学生的汇

总。在实际的学生信息系统中，这样的字符串对任何人都不会有太大用处，除非您在某些地方打印或者显示它。Java 提供了允许将信息重定向到控制台、文件、或者其它目标的输出设施。在第 11 课您将深入学习这些输出设施。

在这个练习中，您将修改测试，以使报表显示在控制台上。这不是需求，但有些时候您需要能够为了不同的情况去显示信息。下一节会介绍某些情况。

在本书的“搭建环境”一节，您编写和运行了“Hello World”程序，该程序将信息打印到终端。将文本打印到终端的代码行是：

```
System.out.println("hello world");
```

浏览 J2SE API 文档关于类 System 的部分，该类在包 java.lang 中。您会看到 out 是一个类型为 PrintStream 的静态变量，该变量代表了标准输出流，也叫 stdout 或者简称 ““控制台”。使用下面的静态变量可以直接访问控制台对象：

```
System.out
```

一旦获取了控制台对象，您可能向其发送一些消息，包括 println 消息。println 方法接受一个字符串作为参数，然后将这个字符串写到底层的输出流。

给 RosterReporterTest 增加一行代码，使用 System.out 在终端上显示报表:

```
package studentinfo;

import junit.framework.TestCase;

public class RosterReporterTest extends TestCase {
   public void testRosterReport() {
      CourseSession session =
         new CourseSession("ENGL", "101",
            new DateUtil().createDate(2003, 1, 6));

      session.enroll(new Student("A"));
      session.enroll(new Student("B"));

      String rosterReport = new RosterReporter(session).getReport();
System.out.println(rosterReport);
      assertEquals(
         RosterReporter.ROSTER_REPORT_HEADER +
         "A" + RosterReporter.NEWLINE +
         "B" + RosterReporter.NEWLINE +
         RosterReporter.ROSTER_REPORT_FOOTER + "2" +
         RosterReporter.NEWLINE, rosterReport);
   }
}
```

118

重新运行测试。在屏幕上您会看到实际的输出。如果在 IDE 中运行测试，为了看到结果[5]，您需要抛弃 `System.out`，而是使用 `System.err`（标准错误输出，也叫 syserr）。

您应该注意到，我把新加的代码行（黑体）排版成和页面左边缘对齐。我用这种方式来提醒自己只是临时使用这行代码。这样的处理易于定位和删除临时用途的代码。

一旦您观察完输出，将代码恢复到原先的状态，然后重新运行测试。

[5] 结果应该显示在一个叫“console”的窗口中。

使用 System.out

在调试程序的时候，经常用 System.out 将消息输出到控制台。在代码中合适的地方插入 System.out.println 语句来显示有用的信息。执行程序的时候，这些跟踪语句的输出帮助您理解系统对象之间交互所产生的消息流和数据流。

调试器是复杂得多的工具，但可以完成更多的目标。不过，简单的跟踪语句有时候是非常快速和有效的方法。而且，在某些环境中，不可能使用调试器。

如果正确的使用 TDD，可以将代码调试的必要性、甚至插入跟踪语句的必要性减到最小。按照 TDD 的原则，以较小的步伐前进。这样，在发现某个问题之前，您只增加了非常少的代码。 119
相对于调试，更好的方法是放弃新增的少量代码，然后重新开始，使用更小的步伐。


以较小的步伐，增量构建系统的测试和代码。如果发现问题，放弃导致问题的增量代码，以更小的步伐重新开始。

多数程序员不编写基于控制台的程序，尽管您可能熟悉很多此类的程序。编译器 javac 本身就是个基于控制台的程序。服务器程序通常都是终端程序，所以程序员可以很容易监控它们的输出。

重构

在 CourseSessionTest 中删除 testReport 方法，并且从 CourseSession 中删除相应的生产代码。

writeReport 方法很短，但是从概念上讲，该方法做了三件事情。为了方便理解，您应该将 writeReport 方法分解成三个更小的方法，分别负责构建页眉、报表体、以及页脚：

```
String getReport() {
   StringBuilder buffer = new StringBuilder();
   writeHeader(buffer);
   writeBody(buffer);
   writeFooter(buffer);

   return buffer.toString();
}

void writeHeader(StringBuilder buffer) {
   buffer.append(ROSTER_REPORT_HEADER);
}

void writeBody(StringBuilder buffer) {
```

```
   for (Student student: session.getAllStudents()) {
      buffer.append(student.getName());
      buffer.append(NEWLINE);
   }
}

void writeFooter(StringBuilder buffer) {
   buffer.append(
      ROSTER_REPORT_FOOTER + session.getAllStudents().size() + NEWLINE);
}
```

120

包结构

您使用包来对类进行分组。类的分组，也叫包结构，会随着需求的改变而改变。开始的时候，您的关注点是开发的便利。随着类的数量不断增长，您应该创建另外的包以方便管理。一旦要部署应用，需求会有所变化：您应该组织包的结构，来满足潜在的不断增加的重用，从而减小维护工作给包的客户带来的影响。

迄今为止，您编写的类都在包 studentinfo 中。典型的组织包的方法是：分离用户接口类和表示业务逻辑的底层类。用户接口负责与最终用户的交互。前面例子中的类 RosterReporter 是用户接口的一部分，因为该类生成最终用户可以看见的输出。

下一个任务是将包 studentinfo 下降一个等级，从而包名变成 sis.stdentinfo。然后将类 RosterReporter 和 RosterReporterTest 分离出来，组成新包 report。

首先在 studentinfo 的同级目录中创建子目录 sis（代表“Student Information System”）。进入到 sis 目录，创建子目录 report。将目录 studentinfo 移动到 sis 目录。将类 RosterReporter 和 RosterreporterTest 移动到子目录 report。您的目录结构看起来应该像下面这样：

```
source
   |-sis
        |-studentinfo
        |-report
```

接着，在所有的类中改变包声明语句。对子目录 report 中的类，使用下面的包声明语句：

```
package sis.report;
```

对子目录 studentinfo 中的类，使用下面的包声明语句：

```
package sis.studentinfo;
```

就像您在第二课所做的，删除所有的 class 文件（*.class），然后重新编译所有的代码。您将会看到几个编译错误。原因是，现在类 RosterReporter、RosterReporterTest 和类 CourseSession、

121

Student 在不同的包中。它们不再能够正确访问其它包中的类。

访问修饰符

在 JUnit 类和方法中，您已经用过关键字 public。JUnit 要求测试类和方法必须声明成 public，不过或许您并不完全了解关键字 public 的意义。您也学习过将实例变量声明为 private，这样其它类的对象就不能访问这些实例变量。

关键字 public 和 private 都是访问修饰符。您可以使用访问修饰符来控制对 Java 元素的访问，例如成员变量、方法、类。访问修饰符对于类的意义不同于方法和成员变量。

通过声明一个类为 public，您允许其它包中的类可以用 import 语句来直接引用这个 public 的类。JUnit 框架类在不同的以 junit 为开头的包中。为了让 JUnit 类能够实例化您编写的测试类，您必须将测试声明为 public。

类 CourseSession 和 Student 都没有指定访问修饰符。如果某个类没有指定访问修饰符，那么该类拥有包访问级别，这也是默认的访问级别。意味着，同一个包中的其它类可以引用这个类。但是，不同包中的类不能访问这个类。

为了更安全地编程，推荐的顺序是：首先是最受限的访问，然后需要时打开相应的访问权限。暴露太多的类给客户，会导致客户对系统集成的细节产生不必要的依赖。如果您改变了某些细节，客户的代码就可能无法继续工作。而且，打开太多的访问权限，会使您的代码逐渐被破坏。


尽可能保护您的代码。只在必要的时候，放开访问控制。

目前，类 CourseSession 和 Student 有包级别的访问控制。您可以保留这种访问控制级别，直到其它包需要访问它们。

为了使代码编译通过，首先您不得不增加 import 语句，这样编译器就知道在包 studentinfo 中寻找类 Student 和 CourseSession。对 RosterReporterTest 的修改如下：

122

```
package sis.report;

import junit.framework.*;
import sis.studentinfo.*;

public class RosterReporterTest extends TestCase {
...
```

在 RosterReporter 中加入相同的 import 语句。

包 studentinfo 中的类依然只有包级别的访问控制，所以包 reports 中的类无法访问

它们。将 Student、CourseSession 和 DateUtil 的类声明改变为 public，就像下面的 Student 声明：

```
package sis.studentinfo;

public class Student {
...
```

您还会收到 AllTests.java 的编译错误。它无法识别类 RosterReporterTest，因为 RosterReporterTest 已经被移到不同的包。现在，注释 AllTests.java 的相应代码行：

```
package sis.studentinfo;

import junit.framework.TestSuite;

public class AllTests {
   public static TestSuite suite() {
      TestSuite suite = new TestSuite();
      suite.addTestSuite(StudentTest.class);
      suite.addTestSuite(CourseSessionTest.class);
//       suite.addTestSuite(RosterReporterTest.class);
      suite.addTestSuite(DateUtilTest.class);
      return suite;
   }
}
```

马上您将为包 report 创建一个新的 AllTests。小心地注释代码——以后很容易忘掉为什么当时注释了这些代码。

重新编译，您将会看到很多错误消息，这些错误消息都来自包 report 中的类 Student 和 CourseSession。像类一样，构造函数和方法的默认访问级别也是包。就像把类声明成 public 是为了从包的外部访问这些类，如果要在类的外部访问类的构造函数和方法，构造函数和方法也必须声明成 public。要合理地声明——不要把所有的方法都声明成 public。

123

考虑到风格与组织，在源代码中您应该把所有的 public 方法移动到非 public 方法的前面。这样做，对这个类感兴趣的客户程序员可以方便地找到 public 方法——这些方法应该是最受关注的。如果用的是 IDE，不必要做这样的代码组织工作，大多数 IDE 都提供了在源代码中组织和定位某个类的更好的方法。

完成上面的工作后，studentinfo 中的生产类看起来像下面这样。

Student.java:

```
package studentinfo;

public class Student {
   private String name;

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

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

CourseSession.java:

```
package studentinfo;

import java.util.*;

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

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

124

```
   String getDepartment() {
      return department;
   }

   String getNumber() {
      return number;
   }

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

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

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

   Date getStartDate() {
      return startDate;
   }
```

```
   public ArrayList<Student> getAllStudents() {
      return students;
   }

   /**
    * @return Date the last date of the course session
    */
   Date getEndDate() {
      GregorianCalendar calendar = new GregorianCalendar();
      calendar.setTime(startDate);
      final int sessionLength = 16;
      final int daysInWeek = 7;
      final int daysFromFridayToMonday = 3;
      int numberOfDays =
         sessionLength * daysInWeek - daysFromFridayToMonday;
      calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
      return calendar.getTime();
   }
}
```

DateUtil.java:

```
package studentinfo;

import java.util.*;

public class DateUtil {
   public Date createDate(int year, int month, int date) {
      GregorianCalendar calendar = new GregorianCalendar();
      calendar.clear();
      calendar.set(Calendar.YEAR, year - 1900);
      calendar.set(Calendar.MONTH, month - 1);
      calendar.set(Calendar.DAY_OF_MONTH, date);
      return calendar.getTime();
   }
}
```

125

测试在哪里运行

到目前为止，您的测试类和生产类在同一个包中。例如，StudentTest 和 Student 都在包 studentinfo 中。这是最简单的方法，但不是唯一的方法。另一种方法是为每一个生产包创建一个对应的测试包。例如，您可以用包 test.studentinfo 来包含 studentinfo 中的所有测试类。

将测试类和生产类放在同一个包中有这样一个好处：测试类可以获得被测试类在包级别的所有细节。但是，包级别的可视性也会带来负面影响：您应该尽量使用生产类的 public 接口来进行测试——证明 public 接口是可用的。如果测试需要的 private 信息越多，那么耦合和依赖就越紧密。紧密的耦合意味着很难在不影响测试类的情况下去修改生产类。

您依然希望有机会对没有设置 public 的类进行断言。因为这个原因，您可能需要将测试类

和生产类放在同一个包中。如果您发现这样做导致同一个目录中有太多的类，您可以利用 Java 的 classpath 提供的好处：在两个不同的子目录中创建相同的目录结构，并且让 classpath 指向这两个子目录。

例如，假设您编译生成的 class 文件存放在 c:\source\sis\bin。您可以创建第二个存放 class 文件的目录 c:\source\sis\test\bin。接下来，修改编译脚本(Ant 使这个步骤非常容易)，这样将编译后生成的测试 class 文件存放到 c:\source\sis\test\bin 目录。所有其它的 class 文件都存放到 c:\source\sis\bin。最后将 c:\source\sis\bin 和 c:\source\sis\test\bin 都放到 classpath 中。

使用这种方法，Student 的 class 文件会是：

c:\source\sis\bin\studentinfo\Student.class，StudentTest 的 class 文件会是：

c:\source\sis\test\bin\studentinfo\StudentTest.class。两个 class 文件都在包 studentinfo 中，但是每一个 class 文件都位于不同的目录。

这样，所有的文件都可以编译通过。测试也可以运行，但是不要忘了您注释掉了 RosterReporterTest。现在是去掉注释的时候了。 126

在包 sis.report 中创建一个新类 AllTests。总的来说，您应该为每一个包创建一个测试套件，以此保证包中的所有类都经过测试[6]。

```
package sis.report;

import junit.framework.TestSuite;

public class AllTests {
   public static TestSuite suite() {
      TestSuite suite = new TestSuite();
      suite.addTestSuite(RosterReporterTest.class);
      return suite;
   }
}
```

现在，您去掉了类 studentinfo.AllTests 中的注释。

将 AllTests.java 放置在目录 sis 中，这样就在包 sis 中新建了类 AllTests。该类将多个测试放在一起，以保证所有的类都可以被测试到。

```
package sis;

import junit.framework.TestSuite;

public class AllTests {
   public static TestSuite suite() {
      TestSuite suite = new TestSuite();
```

[6] 有其它的管理测试套件的方法，IDE 或许可以提供一些帮助。另外，请参考第 12 课有关动态收集测试的内容。

```
        suite.addTest(sis.report.AllTests.suite());
        suite.addTest(sis.studentinfo.AllTests.suite());
        return suite;
    }
}
```

向测试套件发送消息 `addTest`，而不是发送消息 `addTestSuite`。将消息 `suite` 的返回值作为参数传递给恰当的 AllTest 类。发送消息到某个类而不是某个对象，这将导致相应的静态方法被调用。下一课，我们将讨论静态方法。

127 为了运行整个测试套件，您应该将 `sis.AllTests` 传递给 JUnit。

使用 Ant

从这儿开始，我将一直使用 Ant 脚本来执行编译，我已经有了两个以上的目录需要编译。Ant 是一个平台无关的工具，您可以利用 Ant 来创建编译和部署项目的脚本。

如果使用 IDE，您应该可以使用 Ant 来容易地编译所有的代码。例如，在 Eclipse 中，每当保存对 Java 源码的修改之后，所有的源代码都会被自动编译。

不管是否使用 IDE，您应该使用 Ant 来获得 IDE 以及平台的无关性。另一种替代的方法是编写 shell 脚本或者批处理文件，就像我们第一课中所讲述的。有很多可用的 make 工具，您可以选择其中的一个。make 工具是一种类似 Ant 的编译工具，但是多数 make 工具都紧密地限制在某一个特定的操作系统上。几乎没有可以像 Ant 那样的，可以轻松编译 Java 应用程序的 make 工具。对 Java 而言，Ant 是最有效的编译工具。

我强烈推荐您学习如何使用 Ant。IDE 或许可以满足您的要求，但是对于团队开发可能是不够的。如果您工作在某个团队中，您会希望拥有一个编译和部署应用的标准流程。多数开发团队使用 Ant 作为标准，从而保证系统的编译与部署是一致和正确的。

看下面的“Ant 起步”，这部分内容针对如何使用 Ant 提供了简要的介绍。

Ant 起步

这部分内容，会帮助您对如何使用 Ant 有一个基本的了解。

多数 Java IDE 内建了对 Ant 的支持。如果您没有使用 IDE，按照下面的步骤来获取和使用 Ant。

- 从 *http://ant.apache.org/*下载最新的 Ant 版本这里。
- 参考 Ant 中的文档来安装 Ant。设置环境变量 `JAVA_HOME`，该环境变量的值是 J2SE 5.0 SDK 的安装路径。
- 更新系统的 PATH 环境变量，加入 Ant 的 `bin` 目录。

128

- 在项目的根目录中创建文件 build.xml。

无论您是否在使用 IDE，都需要提供 build.xml，该文件包含了如何编译、执行、和部署 Java 应用的指令。

下面是一个 build.xml 的例子，假设项目名称是 agileJava。

```
<?xml version="1.0"?>

<project name="agileJava" default="junitgui" basedir=".">
  <property name="junitJar" value="\junit3.8.1\junit.jar" />
  <property name="src.dir" value="${basedir}\source" />
  <property name="build.dir" value="${basedir}\classes" />

  <path id="classpath">
    <pathelement location="${junitJar}" />
    <pathelement location="${build.dir}" />
  </path>

  <target name="init">
    <mkdir dir="${build.dir}" />
  </target>

  <target name="build" depends="init" description="build all">
    <javac
        srcdir="${src.dir}" destdir="${build.dir}"
        source="1.5"
        deprecation="on" debug="on" optimize="off" includes="**">
      <classpath refid="classpath" />
    </javac>
  </target>

  <target name="junitgui" depends="build" description="run junitgui">
    <java classname="junit.awtui.TestRunner" fork="yes">
      <arg value="sis.AllTests" />
      <classpath refid="classpath" />
    </java>
  </target>

  <target name="clean">
    <delete dir="${build.dir}" />
  </target>

   <target name="rebuildAll" depends="clean,build"description="rebuild all"/>
</project>
```

理解上面的 Build 文件示例

Ant 允许您用 XML 来定义某个项目中的不同目标。目标有一个名字，并可能有一个或多个目标依赖：

```
<target name="rebuildAll" depends="clean,build" />
```

129

上面一行定义了一个叫 rebuildAll 的目标。当您执行该目标的时候，Ant 首先需要保证目标 clean 和 build 已经被执行过了。

目标包含一个需要执行的任务列表。Ant 提供一个手册，该手册描述了大量足以满足您的多数需求的任务。如果您不能找到某个相应的任务，可以自己创建一个。

目标 clean 包含了一个叫 delete 的任务。在这个例子中，delete 任务告诉 Ant 去删除引号中所包含的系统目录。

```
<target name="clean">
  <delete dir="${build.dir}" />
</target>
```

Ant 允许定义属性，属性类似 Java 中的常量。当执行任务 delete 时，Ant 将用属性 build.dir 的值来替换${build.dir}。属性 build.dir 在 agileJava 的 Ant 脚本中是这样定义的:

```
<property name="build.dir" value="${basedir}\classes" />
```

上面的声明设置 build.dir 值为${basedir}\classes。使用${basedir}来引用属性 basedir，该属性定义在 agileJava 相应的 Ant 脚本的 project 元素中:

```
<project name="agileJava" default="junitgui" basedir=".">
```

（意味着 basedir 代表当前目录——即执行 Ant 的目录。）

您可以通过指定一个目标来运行 Ant:

```
ant rebuildAll
```

如果您没有指定目标，那么就执行元素 project 中 default 所代表的目标。在这个例子中，junitgui 是默认的目标。如果您执行无参数的 ant，那么将会执行目标 junitgui:

```
ant
```

下面的命令可以列出所有的目标:

```
ant -projecthelp
```

该命令将列出主目标——指定了 description 属性的目标。

Ant 使用某些内建的功能，保证只执行必要的任务。例如，如果您执行 junitgui 目标，Ant 将运行 javac 来编译自上次执行 junitgui 以来有改动的源代码。Ant 使用 class 文件的时

间标签来判断是否有改动。

对 agileJava 项目进行总结，有三个主目标：build、junitgui 和 rebuildAll。有两个子目标：init 和 clean。

目标 build 依赖于目标 init，这种依赖保证存在编译的输出目录（/classes）。目标 build 使用内建的 javac 任务来编译目录（/source）中的源代码，将生成的 class 文件存放在编译输出目录（/classes）。任务 javac 定义了一组属性，包括 classpath。属性 classpath 引用 path 元素，path 元素包含了 JUnit 的 jar 文件和 classes 目录。

130

目标 junitgui 依赖于目标 build。如果目标 build 成功，那么目标 junitgui 将通过 Java 虚拟机来执行 JUnit GUI，而且以 AllTests 作为参数[7]。

目标 rebuildAll 依赖于目标 clean 和 build 的执行，目标 clean 删除编译输出目录。

参考 Ant 手册可以获得更多细节的信息。有几本关于 Ant 的书，其中非常全面的一本是《Java Development with Ant》。[8]

练习

1. 创建类 CharacterTest。不要忘了将其加入到类 AllSuites。观察到没有测试失败。然后，增加测试 testWhitespace。该方法验证：针对换行、tab、空格，Character.isWhitespace 都将返回 ture，针对其它的字符都将返回 false。您能发现其它可以返回 true 的字符吗？
2. Java 针对方法、类、变量以及其它元素的定义，都有命名限制。例如，命名时不能使用脱字符号 (^)。类 Character 包含了用以判断某个字符是否可以用于标识符的方法。参考 API 文档来理解这些方法。然后向类 CharacterTest 增加测试，用以发现 Java 命名的规则。
3. 断言黑卒的可打印形式是大写字母“P”，白卒的可打印形式是小写字母“p”。暂时将可打印形式作为 Pawn 构造函数的第二个参数。但是，请注意这样会产生冗余。后面您需要改进这种表现形式。
4. （练习 4 和练习 5 密切相关。您可以在完成练习 5 之后，再进行重构）。当客户创建了一个 Board 对象，客户会认为棋盘已经是初始化好的（棋子布置在正确的位置上）。您需要修改 Board 的测试以及相应的生产代码。在创建棋盘时，断言可用的棋子的数目：应该是 16。删除 testAddPawns：目前这个方法没有用。

131

5. 给 Board 增加一个 initialize 方法，这个方法为棋盘增加两行卒：一行是白色的卒(第二行)，

[7] 另一个可选任务是 junit，该任务将执行基于文本的 JUnit。
[8] [Hatcher2002]。

一行是黑色的卒(第七行)。使用 ArrayList 来存储一行卒对象。您可以这样声明：ArrayList<Pawn>。

在 testCreate 中增加断言，确保第二行是 "pppppppp"。另外，断言第七行是 "PPPPPPPP"。使用 StringBuilder 和 for 循环来收集每一行棋子的可打印形式。

确保您的解决方案是经过重构的。将卒添加到指定行，以及将卒添加到棋盘的其它区域，将会产生冗余的代码。在后面的课程中，您将学习如何消除这些冗余。

6. 断言棋盘可以正确地初始化，使用句点表示空的正方格子(行 8 是最上面一行，行 1 是最下面一行)：

```
........
PPPPPPPP
........
........
........
........
pppppppp
........
```

请记住在测试和棋盘打印方法中使用正确的系统属性，以确保不同操作系统间的可移植性。

7. 如果您实现了象棋棋盘的代码，并且测试中使用字符串连接的方式，那么改变代码，使用类 StringBuilder。如果您使用的是 StringBuilder 的方式，那么改变代码，使用字符串连接的方式。

8. 修改测试，在终端上显示棋盘。保证显示的结果和期望的一样。如果不一样，修改测试和代码。

9. 学习循环和其它 Java 构建方式以后，重新审视这里的代码，消除更多的冗余。

132

10. 创建 Ant 脚本，编译整个项目。然后，用命令运行所有的测试。