4 Class Methods and Fields

类方法和类变量

本课的内容包括：

- 重构，将实例方法转变成类方法
- 学习类变量和类方法
- 使用静态导入
- 使用复杂的赋值和增量操作
- 理解什么是简单设计
- 创建工具方法
- 学习如何合理地使用类方法
- 理解将测试作为文档为什么重要
- 暴露异常和堆栈跟踪
- 学习更多关于初始化的内容

类方法

对象是行为（Java 中用方法实现）和属性（Java 中用成员变量实现）的组合。属性和对象本身有着相同的生命周期。在任何给定的时间点，对象有着特定的状态，状态是类的全部实例变量所组合而成的快照。因为这个原因，有时候实例变量也被称之为状态变量。

行为方法操作或者改变对象的属性。换句话说，行为方法可以改变对象的状态。查询方法返回对象状态的某个片断。

把方法设计成：要么改变对象的状态，要么返回信息。不要两件事情都做。

有时候，您发现某个方法接受参数，只对这些参数进行处理，然后返回一个值。该方法不

需要操作对象的状态。这样的方法叫做工具方法。有时候，工具方法在别的语言中被称之为函数。工具方法是全局的：任何客户代码都可以访问它们。

有时候，为了使用某个工具方法而不得不创建对象，但是这样是没有意义的。例如，第 3 课中 DateUtil 的方法 createDate，该方法以月份、天、和年份作为参数，最后返回 Date 对象。方法 createDate 不改变其它数据。如果不去创建 DateUtil 对象，还会稍稍简化您的代码。最后，因为 createDate 是 DateUtil 中唯一的方法，所以没有必要创建 DateUtil 实例。

因为这些原因，createDate 是类方法的候选者。在这个练习中，您将重构 createDate，将其变为类方法。首先改变测试，使其进行类方法调用：

```
package sis.studentinfo;

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

public class DateUtilTest extends TestCase {
   public void testCreateDate() {
      Date date = 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));
   }
}
```

不再使用操作符 new 来创建 DateUtil 实例。为了调用类方法，您指定定义类方法的类（DateUtil），后面跟着点操作符（.），再跟着方法名和参数（createDate(2000, 1, 1)）。

对 DateUtil 的改动同样很小：

```
package sis.studentinfo;

import java.util.*;

public class DateUtil {
    private DateUtil() {}
   public static 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();
   }
}
```

134

一般将方法声明为常规方法、或者叫实例方法，除非您在声明中加上关键字 static。

除了把方法 createDate 设置成 static，把 DateUtil 的构造函数设置为 private 也是

个好主意。只有类 DateUtil 中的代码可以新建 DateUtil 实例，没有别的任何代码可以这样做。尽管允许创建 DateUtil 对象是无害的，但是避免客户代码做一些无意义或者无用的事情，是一个好主意。

将构造函数设置为私有，也可以帮助您发现 createDate 非静态的调用。当编译代码的时候，创建 DateUtil 对象的方法将会产生编译错误。例如，CourseSessionTest 中的 setUp 方法就会有编译错误：

```
public void setUp() {
   startDate = new DateUtil().createDate(2003, 1, 6);
   session = new CourseSession("ENGL", "101", startDate);
}
```

修改代码，静态调用 createDate：

```
public void setUp() {
   startDate = DateUtil.createDate(2003, 1, 6);
   session = new CourseSession("ENGL", "101", startDate);
}
```

修改其它编译错误，然后重新运行测试。现在，您拥有了一个通用的、可能在系统中频繁使用的工具[1]。

J2SE 5.0 中的类 java.lang.Math 提供了非常多的数学函数。例如，Math.sin 返回一个双精度的正弦，Math.toRadians 将一个双精度值从度转换成弧度。该数学类也提供了两个标准数学常量： Math.PI 和 Math.E。由于 java.lang.Math 中所有的方法都是类方法（工具方法），所以这个类也被称为工具类。 135

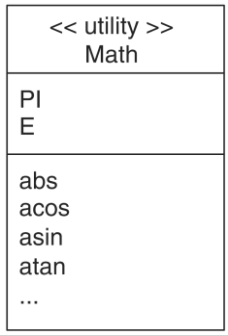

图 4.1 工具类 Math

[1] 该工具的性能不是最好的。没有必要每次调用 createDate 时，都创建一个 GregorianCalendar 对象。如果只是零星的使用，这样可能没有问题。如果频繁使用——通过读入某个输入文件，创建 10 000 个日期对象——您应该考虑使用一个类变量来缓存该日历对象（请看下一节）。

在 UML（图 4.1），使用 stereotype <<utility>>来表明一个工具类。在 UML 中，使用 stereotype 来超越 UML 的限制，从而提供自定义的语义。工具 stereotype 表明所有类行为和属性都是全局可访问的。

通常，在 UML 图中需要对类的行为和属性加上下划线。由于<<utility>> stereotype 声明某个类中所有的方法和属性都是全局的，所以您不用给它们加上下划线。

静态初始化代码块

创建类的实例的时候，构造函数会执行。您可以使用构造函数来进行复杂的实例初始化。

有时候，您需要在类级别进行复杂的初始化。您可以使用静态初始化代码块来实现这个目标。当 Java 虚拟机第一次加载类的时候，执行静态初始化代码块。

```
import java.util.Date;
public class St {
   static {
      long now =
         System.currentTimeMillis();
      then = new Date(now + 86400000);
   }
   public static Date then;
}
```

为了定义静态初始化代码块，您在一个代码块（{…}）之前加上关键字 static。将代码块放在类定义的内部，同时在任何方法和构造函数的外部。事实上，任何代码都可以出现在静态初始化代码块中，但是该代码块不能抛出任何异常（请看第 8 课）。

136

类变量

有时候，您需要跟踪某个类的所有实例，或者在没有创建实例的情况下执行某些操作。举一个简单的例子，您也许打算跟踪 CourseSession 的总数。每创建一个 CourseSession 对象，将计数器加一。问题是，在什么地方放置计数器？您可以为 CourseSession 设置一个实例变量来进行计数，但是这样有问题：难道所有的 CourseSession 实例都不得不进行计数吗？某个 CourseSession 实例怎样才能知道其它实例的创建，从而更新计数呢？

您可以提供另外一个类 CourseSessionCounter，该类的唯一职责是跟踪 CourseSession 对象的创建。但是，用一个新类来完成这样的目标，似乎有点过了。

在 Java 中，您可以使用类变量。相对于实例变量，类变量是另一种解决方案。客户代码在无须创建类实例的情况下，就可以访问类变量。类变量有静态的作用范围：只要类存在，类变量就存在，类变量的生命周期是从类的第一次加载直到应用程序的结束。

您应该已经看到了类常量的使用。类常量是指定了关键字 final 的类变量。

下面的测试代码（在 CourseSessionTest）对 CourseSession 实例的创建进行计数：

```
public void testCount() {
   CourseSession.count = 0;
   createCourseSession();
   assertEquals(1, CourseSession.count);
   createCourseSession();
   assertEquals(2, CourseSession.count);
}

private CourseSession createCourseSession() {
   return new CourseSession("ENGL", "101", startDate);
}
```

（为了使用 createCourseSession，不要忘了更新 setUp 方法。）

为了支持该测试，在类 CourseSession 中创建一个类变量 count。使用关键字 static 来指定某个变量拥有静态的作用范围。同时，在 CourseSession 中增加更新 count 的代码（在创建 CourseSession 实例的时候）。

```
public class CourseSession {
   // ...
   static int count;
public CourseSession(
        String department, String number, Date startDate) {
     this.department = department;
     this.number = number;
     this.startDate = startDate;
     CourseSession.count = CourseSession.count + 1;
   }
   // ...
```

137

访问类变量 count 的方法类似于类方法的调用：首先指定类名（CourseSession），后面跟着点操作符（.），再跟着变量的名称（count）。访问类变量的时候，Java 虚拟机不会创建一个 CourseSession 实例。

就像我在前面提到的，类变量和实例变量有着不同的生命周期：实例变量和包含它的对象有着相同的生命周期。每一个 Java 虚拟机创建的 CourseSession 对象，管理它自己的实例变量的集合。虚拟机创建一个 CourseSession 对象，同时也会初始化它所有的实例变量。

但是，类变量出现在 Java 虚拟机第一次加载引用该类变量的类的时候。内存中有一个类变量的拷贝。当 Java 虚拟机第一次加载某个类，同时也会初始化它的类变量。如果您需要在稍后的时候，将某个类变量的值重置为初始的状态，那么您必须亲自对其进行显式的初始化。

做一个试验，注释掉 testCount 中的第一行代码（该行代码是 CourseSession.count=0）。然后在 JUnit 中运行测试。关掉 JUnit 的复选框“Reload classes every run[2]”。如果运行该测试两

[2] 如果打开这个 JUnit 选项，那么每次在 JUnit 中运行测试，都会导致从磁盘物理加载测试类，并且重新初始化测试类。

次（通过点击“Run”按钮），测试会失败，并且您可以看到每次测试的实际计数结果。您甚至会发现第一次测试也是失败的：CourseSessionTest 中的其它测试方法创建了 CourseSession 对象，该方法增加了变量 count 的值。

使用类方法操作类变量

把实例变量直接暴露给客户是一种不好的方式。和实例变量一样，公开类变量也是一种不好的方式。值得注意的例外是类常量——但是，第 5 课您将了解到，存在避免使用类常量的更好的理由。

138

类方法除了用来作为工具函数，您还可以使用类方法操作静态数据。

CourseSession 中的方法 testCount 直接访问类变量 count。改变测试代码，通过类方法调用来获得计数。

```
public void testCount() {
   CourseSession.count = 0;
   createCourseSession();
   assertEquals(1, CourseSession.getCount());
   createCourseSession();
   assertEquals(2, CourseSession.getCount());
}
```

然后在 CourseSession 中增加返回类变量 count 的类方法。

```
static int getCount() {
   return count;
}
```

类方法可以直接访问类变量。在类方法中访问类变量，不用指定类名。

调用类方法时，Java 虚拟机不会创建 CourseSession 的实例。这意味着 CourseSession 的类方法不能访问 CourseSession 中定义的任何实例变量，例如 department 或者 students。

测试方法依然直接调用类变量 count，但是，您需要每次测试时都初始化它：

```
public void testCount() {
   CourseSession.count = 0;
   ...
```

改变 CourseSessionTest 中的代码，发送静态消息来重置计数：

```
public void testCount() {
```

如果使用 IDE，例如 Eclipse，您也许无法设置这个 JUnit 特性。

```
   CourseSession.resetCount();
   createCourseSession();
   assertEquals(1, CourseSession.getCount());
   createCourseSession();
   assertEquals(2, CourseSession.getCount());
}
```

在 CourseSession 中增加方法 resetCount，并且将类变量 count 设置为 private：

```
public class CourseSession {
   // ...
   private static int count;
   // ...
   static void resetCount() {
      count = 0;
   }
    static int getCount() {
     return count;
   }
   // ...
```

139

将变量 count 设置为 private，当重新编译的时候，会暴露出客户代码中所有直接访问它的代码行。

方法 testCount 文档化了某个客户将如何使用类 CourseSession，现在这个方法是完整和干净的。但是，类 CourseSession 依然在构造函数中直接访问类变量。除了从成员（构造函数，方法）中直接访问静态数据，一个更好的方案是创建类方法。这样的封装可以使您对类变量有更好的控制。

改变 CourseSession 的构造函数，用发送消息 incrementCount 替代直接访问类变量：

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

然后，在 CourseSession 中编写增加计数的类方法。将该方法声明为 private，这样可以防止其它类改变这个计数器，因为这样的改变会破坏计数的完整性。

```
private static void incrementCount() {
   count = count + 1;
}
```

在没有指定类名的情况下，从类的实例访问类方法或者类变量，也是可能的。例如，您可以像下面这样，改变构造函数中的代码：

```
public CourseSession(String department, String number, Date startDate) {
   this.department = department;
   this.number = number;
   this.startDate = startDate;
   incrementCount(); // don't do this!
}
```

即使这样可以正常工作，也要避免。调用类方法时不指定类名，会引入不必要的混淆，这被认为是一种糟糕的方法。incrementCount 是类方法还是实例方法呢？从 CourseSession 构造函数中不能得到答案，您的意图是不清晰的。将某个类方法误以为是实例方法，会导致一些有趣的问题。 140

 除了从同一个类中的其它类方法中调用某个类方法，从任何其它地方调用这个类方法，都必须在类方法的前面加上类名作为限定。

静态导入

我只是告诉您不要从类的实例调用类方法，除非您提供了类名。这样做，使得类方法在何处定义显得比较模糊。这同样也适用于对类变量的访问（不包括类常量）。

Java 甚至允许您将事情变得更容易混淆。在类中使用静态导入，可以让您使用其它类中定义的类方法和类变量，而且这些类方法和类变量就像在本地定义的一样。换句话说，静态导入允许您在调用其它类中定义的静态成员时，可以忽略类名。

这里有一些关于静态导入正确的和错误的用法。首先，我演示一个错误的用法。修改 CourseSessionTest：

```
// avoid doing this
package sis.studentinfo;

import junit.framework.TestCase;
import java.util.*;
import static sis.studentinfo.DateUtil.*; // poor use of static import

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

   public void setUp() {
      startDate = createDate(2003, 1, 6); // poor use of static import
      session = CourseSession.create("ENGL", "101", startDate);
   }
   ...
   public void testCourseDates() {
      // poor use of static import:
      Date sixteenWeeksOut = createDate(2003, 4, 25);
```

```
      assertEquals(sixteenWeeksOut, session.getEndDate());
   }
   ...
}
```

141

静态导入语句看起来和普通的 import 语句非常相似。但是，普通 import 语句从某个包中导入了一个或所有的类，而静态 import 语句从某个类中导入一个或所有的类方法以及类变量。上面的例子，从类 DateUtil 中导入了所有的类方法和类变量。由于 DateUtil 中只有一个类方法，所以您可以显式地导入：

```
import static sis.studentinfo.DateUtil.createDate;
```

如果 DateUtil 包含了多个名字为 createDate 的类方法（但是有不同的参数），或者只是包含了一个名字叫 createDate 的类变量，那么它们都会被静态导入。

静态导入使您不用提供类名，这样做可以偷点儿懒，但也引入了不必要的混淆。正如 createDate 在何处定义这个问题。如果您正在编写的某个类，需要调用多个外部类方法（可能几十个甚至更多），这样您或许有了使用静态导入的借口。但是，更好的方法是，弄明白为什么需要这么多的静态调用，或许您需要重新审视类的设计。

使用静态导入的可能理由之一是：简化对多个类常量的调用，而且这些类常量定义在一个地方。假设您创建了多个报表类，而且每个报表类都需要将换行符追加到输出，所以每个报表类都需要用到常量 NEWLINE，就像 RosterReporter 中的定义：

```
static final String NEWLINE = System.getProperty("line.separator");
```

您不会希望在每个报表类中都重复定义这个常量。所以您可以创建一个新类，这个类的职责就是为了持有常量。稍后，它也可以持有别的常量，例如对任何报表类都适用的页面宽度。

```
package sis.report;

public class ReportConstant {
   public static final String NEWLINE =
      System.getProperty("line.separator");
}
```

在典型的报表类中，由于很多地方都需要使用常量 NEWLINE，所以您可以增加一个静态导入，这样可以使您的代码看起来干净一些[3]：

142

```
package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

public class RosterReporterTest extends TestCase {
   public void testRosterReport() {
      CourseSession session =
```

[3] 你可以用其他方法完全消除对常量 NEWLINE 的需要。在第 8 课中，您将了解此类技术——Java 的格式化类。

```
        CourseSession.create(
           "ENGL", "101", DateUtil.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" + NEWLINE +
        "B" + NEWLINE +
        RosterReporter.ROSTER_REPORT_FOOTER + "2" +
        NEWLINE, rosterReport);
  }
}
```

您可以针对类 RosterReporter 进行类似的改动。

将一组常量放到一个没有行为（方法）的类中，这是一种受到置疑的面向对象设计。类不可以存在于真空。最好取消类 ReportConstants 中的常量，将这些常量作为其它普通 Java 类的一部分，例如类 Report。

关于静态导入，还有一些需要注意的：

- 针对一个给定的包，不可能用一行语句静态地导入所有类的所有类方法和类变量。也就是说，您不能这样编写代码：

```
import static java.lang.*;  // this does not compile!
```

- 如果一个本地方法，和一个静态导入的方法有着相同的名字，那么本地方法被调用。

谨慎地使用静态导入。因为静态导入使得类方法和类变量的定义位置变得模糊，所以加大了理解代码的难度。使用静态导入的原则是：限制静态导入的使用，不要在应用程序中普遍使用静态导入。

143

增量

在方法 incrementCount 中，有这样的代码：

```
count  =  count + 1;
```

等式右边的表达式，将 count 的值与 1 相加。然后，Java 把相加的和赋值给变量 count。

增加变量的值是一个常用的操作，所以 Java 提供一种便捷的方式。下面两行语句的结果是相同的：

```
count = count + 1;
count += 1;
```

第二行语句使用了复合赋值。第二行语句对复合赋值运算符右边的 1 和左边的 count 进行加法运算，然后将算术和赋值给左边的变量 count。复合赋值适用于任何数学操作符。例如，

```
rate *= 2;
```

等同于：

```
rate = rate * 2;
```

将整型变量的值加 1，这样的操作十分常用，以至于 Java 提供了更便捷的方法。下面的代码使用自增操作符，将 count 的值加 1：

```
++count;
```

下面的代码使用自减操作符，将 count 的值减 1：

```
--count;
```

您也可以把加号和减号放到变量的后面：

```
count++;
```

```
count--;
```

结果是一样的。但是，当在一个更大的表达式中使用它们的时候，前缀操作符（加号或者减号在变量的前面）和后缀操作符（加号或者减号在变量的后面）有一个重要的区别。 144

Java 虚拟机遇到前缀操作符，首先增加变量的值，然后将结果应用在更大的表达式中。

```
int i = 5;
assertEquals(12, ++i * 2);
assertEquals(6, i);
```

Java 虚拟机遇到后缀操作符，首先把当前值应用在更大的表达式中，然后再增加变量的值。

```
int j = 5;
assertEquals(10, j++ * 2);
assertEquals(6, j);
```

修改 CourseSession，使用自增操作符。由于只是增加 count 的值，并没有将其作为某个更大表达式的一部分，所以使用先增操作符还是后增操作符，都是一样的。

```
private static void incrementCount() {
   ++count;
}
```

重新编译，并且重新测试（您一直都这么做）。

工厂方法

修改 CourseSession，提供一个静态工厂方法来创建 CourseSession 对象。这样做，您可以对 CourseSession 实例创建过程中所发生的一切进行控制。

修改 CourseSessionTest 的方法 `createCourseSession`，来演示您将如何使用这个工厂方法。

```
private CourseSession createCourseSession() {
  return CourseSession.create("ENGL", "101", startDate);
}
```

在 CourseSession 中，增加一个静态的工厂方法，该方法返回一个新建的 CourseSession 对象：

```
public static CourseSession create(
     String department,
     String number,
     Date startDate) {
   return new CourseSession(department, number, startDate);
}
```

145

找出所有使用 `new CourseSession()` 来创建 CourseSession 对象的代码。给 CourseSession 的构造函数加上 `private` 关键字，然后利用编译器来帮助你：

```
private CourseSession(
     String department, String number, Date startDate) {
   // ...
```

用静态工厂方法替代您找到的 CourseSession 构造代码（在 RosterReporterTest 和 CourseSessionTest 中各有一个）。因为 CourseSession 的构造函数被声明为 `private`，所以客户代码（包括测试代码）不能直接利用构造函数来创建 CourseSession 实例。客户必须使用静态工厂方法。

Joshua Kerievsky 将上面的重构称之为“用创建方法替换多个[4]构造函数”[5]。这样的创建方法是类方法。创建方法最重要的好处在于——您可以提供有意义的名字。由于构造函数的名字必须和类名相同，所以无法向程序员传递更多的关于如何使用构造函数的信息。

既然有了工厂方法创建 CourseSession 对象，所以可以用它来进行计数。面向对象设计很大程度上是将代码放到它所属的地方。这并不意味着您必须一开始就把代码放到正确的地方，但是一旦发现了更好的位置，就应该把代码移到那里。在工厂方法中发送消息 `incrementCount`，应该更有意义：

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

public static CourseSession create(
     String department,
     String number,
     Date startDate) {
```

[4] Java 允许您在一个类中编写多个构造函数。由于创建方法的名字提供了更多的信息，可以帮助客户程序员决定如何选择，所以在这种情况下，创建方法更有价值。

[5] [Kerievsky2004]。

```
    incrementCount();
    return new CourseSession(department, number, startDate);
}
```

146

简单设计

正统的软件开发者会告诉您——在开始阶段思考一个完整的设计将节省您大量的时间。经过充分的考虑，您发现静态创建方法是一个好主意，所以一开始就把它们放在代码里。是的，在有了相当多的面向对象开发经验后，您应该学习如何在开始阶段就拥有更好的设计。

但是，设计的效果常常在编码开始以后才能体现。不通过代码来验证自己设计的设计者，经常创造出失败的系统，比如在不需要的地方使用静态创建方法，诸如此类。他们在设计时也经常会遗漏某些重要的方面。

最好的策略是尽可能地保持代码的干净。保持干净的设计也是很重要的：

- 确保测试是完备的，而且总是运行成功。
- 消除重复。
- 保证代码是干净和富有表现力的。
- 将类和方法的数量减到最小。

代码也不应该存在过度设计，不过支持当前的功能是必要的。这样的规则被称之为简单设计[6]。

简单设计带来了可伸缩性，随着需求的改变，您可以更新和改进设计。就像您所看到的，创建一个静态工厂方法不是那么困难。遵循简单设计，工厂方法是简单和安全的。

静态的危险

错误的使用静态方法或者静态变量，会造成严重的而且难以解决的软件缺陷。对于初学者，一个典型的错误是将本来应该是实例变量的属性，声明成了类变量。

类 Student 定义了一个实例变量 name。每个 Student 对象都应该有自己的 name 拷贝。如果将 name 声明为静态，那么所有 Student 对象将会使用同一个 name 拷贝：

147

```
package sis.studentinfo;

public class Student {
   private static String name;
```

[6] [Wiki2004b]。

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

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

下面的测试可以展现该错误所带来的破坏性影响：

```
package sis.studentinfo;

import junit.framework.*;

public class StudentTest extends TestCase {
   ...
   public void testBadStatic() {
      Student studentA = new Student("a");
      assertEquals("a", studentA.getName());
      Student studentB = new Student("b");
      assertEquals("b", studentB.getName());
      assertEquals("a", studentA.getName());
   }
}
```

因为 studentA 和 studentB 共享同一个类变量 name，所以最后一个 assertEquals 语句会失败。所有类似的情况，都会导致测试方法失败。

这样的错误可能会浪费您大量的时间，特别是如果您没有好的单元测试。程序员经常会认为，像变量声明这么简单的事情不会带来问题，所以他们经常直到最后才到声明变量的地方去查找问题的原因。

去掉关键字 static，恢复类 Student。重新编译，然后重新测试。

使用静态所需要注意的

- 避免仅仅为了使用静态，就将实例方法改变为类方法。不仅要保证语义上是有意义的，而且至少有一个类需要访问这个类方法。

148

146

垃圾回收

Java 会努力管理应用程序对内存的使用。在多数计算机系统中，内存都是珍贵的、有限的资源。每当代码需要创建对象，Java 必须找到可以存储该对象的内存空间。如果 Java 对内存管理不做任何事情，那么内存中的对象会永远呆在那里，而且您将很快消耗完所有可用的内存。

Java 使用一种叫垃圾回收的技术来管理应用程序对内存的使用。Java 虚拟机跟踪所有的对象，不时地在后台运行垃圾回收器。垃圾回收器收回您不再使用的对象。

当没有其他对象引用某个对象时，该对象就不再需要。假设您在某个方法中创建了一个对象，并且将这个对象赋值给一个局部变量（但是，没有别的处理）。当虚拟机执行完这个方法，该对象仍然在内存中，但是没有任何东西指向它——局部变量只在方法的作用范围内有效。这样的对象符合垃圾回收的条件，会在下次运行垃圾回收器时消失（不能保证垃圾回收器将会运行）。

如果某个实例变量指向一个对象，那么您可以将实例变量设置成 `null`，从而将对象释放给潜在的垃圾回收器。或者您可以等待，直到该对象不被引用。一旦该对象不被引用，那么它将符合垃圾回收的条件。

如果您把对象存储在标准的集合中，例如 ArrayList。这样集合就持有了该对象的一个引用。只要集合包含着它，就不能对该对象进行垃圾回收。

- 静态集合（例如，用类变量存储 ArrayList 对象）通常是个坏主意。集合持有某个对象的引用。添加到类集合中的任何对象，都会一直存在，直到从集合中删除它或者应用程序终止。实例集合不存在这个问题，请参考上面有关垃圾回收的介绍。

Jeff 静态规则

最后是不谦虚地被我命名的“Jeff 静态规则”：


直到确信需要使用静态，才使用静态。

这个简单的规则来源于对 Java 开发的观察。走了很长的路才得到这个小小的知识。对静态缺乏认识，常常导致程序员滥用它。

149

我的哲学是反对过度使用静态，因为它们是非面向对象的。系统中使用的静态方法越多，则越表明系统是过程式的——本质上是大量的全局函数操作全局数据。我的实践表明，不正确或者粗心地使用静态，会导致各种各样的问题，包括设计局限的、诱人的、怪异的缺陷、以及内存泄漏。

您应该知道什么时候使用静态是正确的，什么时候是错误的。保证不要使用静态，除非您理解为什么您要这么做。

布尔型


下一步，学生信息系统需要得到某个学期的学生清单。目前，学生清单基于以下三点：

是否是本州学生，是否是全日制学生，学生获得了多少学分。为了支持学生清单，您不得不修改类 Student 来容纳这些信息。

学生要么全日制，要么就是在职的。换句话说，学生要么全职，要么就不是全职的。任何时候，在 Java 中，如果您需要表示两种状态中的一个——是或者不是——您可以使用 boolean 类型。一个 boolean 变量，有两个可能的布尔值，即 true（是）或者 false（不是）。像 int 一样，boolean 类型也是基本类型，您不能向 boolean 变量发送消息。

在类 StudentTest 中创建方法 testFullTime。该方法初始化一个 Student 对象，然后测试该学生不是全职。全职学生必须至少有 12 个学分，一个新创建的 student 没有学分。

```
public void testFullTime() {
   Student student = new Student("a");
   assertFalse(student.isFullTime());
}
```

方法 assertFalse 也是 StudentTest 从 junit.framework.TestCase 继承而来的。该方法接受单个布尔表达式作为参数。如果表达式的值为 false，那么测试通过；否则，测试失败。在 testFullTime 中，如果学生不是全职，那么测试通过；也就是说，如果 isFullTime 返回 false，那么测试通过。

150

在类 Student 中增加方法 isFullTime：

```
boolean isFullTime() {
   return true;
}
```

方法的返回值是 boolean 类型。因为该方法返回 true，所以测试会失败——因为测试断言 isFullTime 将返回 false。观察到测试失败，修改该方法，使其返回 false，然后观察到测试通过。

脱产还是在职，取决于该学生选了多少学分的课程。至少选 12 学分的课程，才能被认为是脱产。学生通过报名参加课程的学习，从而获得学分。

现在的需求是：学生报名学习一门课程，就能增加该学生的学分。简单地说：学生需要能够记录自己的学分，新生没有学分。

```
public void testCredits() {
   Student student = new Student("a");
   assertEquals(0, student.getCredits());
   student.addCredits(3);
   assertEquals(3, student.getCredits());
   student.addCredits(4);
   assertEquals(7, student.getCredits());
}
```

在 Student 中：

```
package sis.studentinfo;

public class Student {
   private String name;
   private int credits;

   public Student(String name) {
      this.name = name;
      credits = 0;
   }

   public String getName() {
      return name;
   }

   boolean isFullTime() {
      return false;
   }

   int getCredits() {
   return credits;
   }

   void addCredits(int credits) {
   this.credits += credits;
   }
}
```

151

为了满足新生学分为 0 的需求，Student 构造函数将 credits 初始化为 0。在第 2 课中学到过，既可以自己初始化成员变量，也可以不管它，因为 Java 默认将 int 变量初始化为 0。

到这里，学生依然是兼职的。因为学分的数量直接关联到学生的状态，也许您会合并两个测试方法（但是，这是一种存在争议的做法）。将测试方法 testCredits 和 testFullTime，合并成一个测试方法 testStudentStatus。

```
public void testStudentStatus() {
   Student student = new Student("a");
   assertEquals(0, student.getCredits());
   assertFalse(student.isFullTime());

   student.addCredits(3);
   assertEquals(3, student.getCredits());
   assertFalse(student.isFullTime());

   student.addCredits(4);
   assertEquals(7, student.getCredits());
   assertFalse(student.isFullTime());
}
```

断言失败信息

JUnit 断言可以定制错误信息。如果断言失败，JUnit 会显示该错误信息。尽管编码良好的测试可以帮助其它程序员理解测试失败的含义，但是增加一条消息有助于更迅速地理解。下面

是一个例子：

```
assertTrue(
  "not enough credits for FT status",
  student.isFullTime());
```

对于 assertEquals，默认信息通常是足够的。无论如何，请阅读默认生成的信息，看它是否可以向其它程序员传递充分的信息。

测试应该通过。现在修改测试，为了达到 12 个学分，让学生选修一门 5 个学分的课程。您
152 可以使用方法 assertTrue 来测试该学生是否为脱产。如果传入 assertTrue 的参数为 true，那么测试将通过，否则测试失败。

```
public void testStudentStatus() {
   Student student = new Student("a");
   assertEquals(0, student.getCredits());
   assertFalse(student.isFullTime());

   student.addCredits(3);
   assertEquals(3, student.getCredits());
   assertFalse(student.isFullTime());

   student.addCredits(4);
   assertEquals(7, student.getCredits());
   assertFalse(student.isFullTime());

   student.addCredits(5);
   assertEquals(12, student.getCredits());
   assertTrue(student.isFullTime());
}
```

测试失败。为了通过测试，您必须修改方法 isFullTime，使得学分为 12 或者更多的情况下返回 true。这需要您编写一个条件。在 Java 中，条件表达式返回一个布尔值。改变 Student 中的方法 isFullTime，加上一个合适的表达式：

```
boolean isFullTime() {
   return credits >= 12;
}
```

上面代码的意思是：“如果学分的数目大于等于 12，就返回 true，否则返回 false”。

重构 isFullTime，在 Student 中引入表示 12 学分的类常量。

```
static final int CREDITS_REQUIRED_FOR_FULL_TIME = 12;
...
boolean isFullTime() {
   return credits >= CREDITS_REQUIRED_FOR_FULL_TIME;
}
```

既然学生可以增加学分，那么修改 CourseSessionTest，确保 Student 对象能够正确地增加学分。这样开始测试：

```
public class CourseSessionTest extends TestCase {
   // ...
   private static final int CREDITS = 3;

   public void setUp() {
      startDate = createDate(2003, 1, 6);
      session = createCourseSession();
   }
   // ...
   public void testEnrollStudents() {
      Student student1 = new Student("Cain DiVoe");
      session.enroll(student1);
      assertEquals(CREDITS, student1.getCredits());
      assertEquals(1, session.getNumberOfStudents());
      assertEquals(student1, session.get(0));

     Student student2 = new Student("Coralee DeVaughn");
      session.enroll(student2);
      assertEquals(CREDITS, student2.getCredits());
      assertEquals(2, session.getNumberOfStudents());
      assertEquals(student1, session.get(0));
      assertEquals(student2, session.get(1));
   }
   // ...
    private CourseSession createCourseSession() {
       CourseSession session =
          CourseSession.create("ENGL", "101", startDate);
       session.setNumberOfCredits(CourseSessionTest.CREDITS);
       return session;
   }
}
```

153

修改 CourseSession，使上面失败的测试得以通过：

```
public class CourseSession {
      ...
      private int numberOfCredits;
      ...
      void setNumberOfCredits(int numberOfCredits) {
         this.numberOfCredits = numberOfCredits;
   }

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

测试就是文档

测试方法 testStudentStatus 用来保证学生拥有正确的状态：全职或者兼职。同时，该方法也保证类 Student 可以正确地增加学分。

测试无法穷尽所有的可能性。测试的通常策略是考虑 0、1、很多、所有边界条件、所有异常条件。至于学生的学分，测试需要保证学分为 0、1 或者 11 的学生的状态为在职，学分为 12 或者 13 的学生的状态为脱产。同时，也要测试意外的情况，例如增加的学分为负数，或者增加的学分数目特别大。

154

测试驱动开发采取略微不同的方法。策略是相同的，但是目标并不十分一样。测试并不只是保证代码正确的手段。而且，测试驱动开发提供了一种持续渐增开发的技术。您学习如何渐增地编写代码，每分每秒都得到反馈以确认您行进在正确的方向上。测试和信心相关。

测试驱动开发也会影响设计。这是深思熟虑的结论：测试驱动开发教会您如何构建易于测试的系统。很少生产系统具有易测试的特点。使用测试驱动开发，你将学会在与系统中其它类隔离的情况下，如何测试某个类。这样将产生一个类与类之间松耦合的系统。松耦合是判断面向对象系统设计是否优秀的主要指标。

最后，测试可以看作是类的功能文档。完成编码后，您应该能够通过阅读测试，来理解某个类是什么，以及这个类是怎样工作的。第一，您应该能够通过查看测试的名字，来理解某个类所支持的所有功能。第二，每个测试都应该像文档一样，具有良好的可读性，可以帮助理解如何使用某项功能。

 测试代码是其他人可以理解的、非常全面的文档。

在 testStudentStatus 这个例子中，作为程序员，您对生产代码 isFullTime 具有高度的自信。它只有一行代码，而且您确切知道该行代码的意思：

```
return credits >= Student.CREDITS_REQUIRED_FOR_FULL_TIME
```

您可能希望将这个测试考虑得更充分些，从而选择继续前进，况且这样做不会带来不协调。再强调一次，测试和信心高度相关。信心越小或者代码越复杂，您就应该编写越多的测试。

什么是意外的操作？如果某些人用负数作为学分，会不会破坏 Student 对象的完整性？请记住，您是这个系统的开发者，是对类 Student 的访问进行控制的人。您有两个选择：测试并且预防任何可能的异常情况，或者根据系统设计做出一些假设。

155

在学生信息系统中，类 CourseSession 是唯一增加学分的地方，这是由设计决定的。如果 CourseSession 编码正确，那么它将存储合理的学分数目。由于存储合理的学分数目，所以理论上不会有负数传入 Student。因为使用 TDD，所以您知道自己编写了正确的 CourseSession。

当然，某些地方某些人不得不为了某个课程安排，把学分数目输入到系统中去。在用户界面级别——您必须预防任何可能性。某些人可能什么也不输入，可能输入一个字母，一个负数，

或者一个字符“$”。测试需要保证——只有合理的整数被输入到系统中去。

一旦设置了针对无效数据的屏障，您可以认为系统中的其它部分都在控制之中——当然是理论上。有了这个假设，对其它类可不用屏蔽无效数据。

实际上，总会存在一些“洞”，使您在代码里无意中引入缺陷。但是，成功使用测试驱动开发的关键在于——理解反馈的重要性。缺陷暗示您的单元测试并不完整：您忽略了某项测试。退回去编写这个遗漏的测试。通过测试失败，然后修改它。慢慢地，您会了解什么对测试是重要的，什么是不重要的。您会了解，针对测试，什么地方花的时间多，什么地方花的时间少。

对于 testStudentStatus 和相应的实现，我有信心。测试缺少作为文档的能力。问题的部分原因在于，作为开发者您非常了解自己如何编码实现了某个功能。这些背景知识可以帮助其它程序员去正确阅读测试中的业务逻辑以及限制条件。如果其它程序员不了解这些背景知识，那么回过头看这些测试，就仿佛您在这里编写了隐藏的代码。测试是否告诉您如何使用某个类？测试是否展示了不同的场景？测试是否指出了约束和限制——或许被忽略了？

在这个例子中，测试应该和 Student 使用相同的常量。这样做，测试会变得更有表现力。现在，可以很好地了解全职的边界条件。

```
public void testStudentStatus() {
   Student student = new Student("a");
   assertEquals(0, student.getCredits());
   assertFalse(student.isFullTime());
   student.addCredits(3);
   assertEquals(3, student.getCredits());
   assertFalse(student.isFullTime());

   student.addCredits(4);
   assertEquals(7, student.getCredits());
   assertFalse(student.isFullTime());

   student.addCredits(5);
   assertEquals(Student.CREDITS_REQUIRED_FOR_FULL_TIME,
      student.getCredits());
   assertTrue(student.isFullTime());
}
```

156

关于初始化的更多内容

为了支持本州学生和外州学生的区别，Student 对象需要存储学生居住所在州的信息。学校位于科罗拉多州（缩写是：CO）。如果学生居住在任何其它州，或者没有指定学生居住所在的州（学生是国际学生，或者没有把表格填写完整），那么该学生是外州学生。

下面是测试：

```
public void testInState() {
   Student student = new Student("a");
   assertFalse(student.isInState());
   student.setState(Student.IN_STATE);
   assertTrue(student.isInState());
   student.setState("MD");
   assertFalse(student.isInState());
}
```

判断学生是否为本州学生的逻辑，必须要对代表学生所在州的字符串和"CO"进行比较。当然，这意味着您需要创建一个成员变量 state。

```
boolean isInState() {
   return state.equals(Student.IN_STATE);
}
```

使用方法 equals 来比较两个字符串。以另一个字符串作为参数，发送消息 equals 给 String 对象。如果两个字符串有相同的长度，并且两个字符串的字符都一一对应相同，那么 equals 方法返回 true。所以"CO".equals("CO")将返回 ture，"Aa".equals("AA")将返回 false。

157

为了让测试 testInState 得以通过，赋给成员变量 state 正确的初始化值是非常重要的。除了"CO "以外的字符串都可以，但是最好是空字符串。

```
package sis.studentinfo;

public class Student {
   static final String IN_STATE = "CO";
   ...
   private String state = "";
   ...
   void setState(String state) {
      this.state = state;
   }
   boolean isInState() {
      return state.equals(Student.IN_STATE);
   }
}
```

或许您考虑编写一个测试，用以展示：当传入小写的州名缩写时，会发生什么。现在，如果客户代码传入"Co"，因为"Co"和 "CO"不相同，所以将不会把该学生的状态设置为本州学生。尽管这是可接受的行为，但是更好的方案是：在和"CO"比较之前，一律将州名缩写转成对应的大写形式。您可以使用方法 toUpperCase 实现大写转换。

异常

如果不给成员变量 state 赋初值，会发生什么？在这个测试中，创建一个 Student 对象，

并且马上发送消息 isInState。isInState 消息导致 equals 消息被发送到成员变量 state。将消息发送给未初始化的对象，会发生什么。改变成员变量 state 的声明，并且注释掉初始化：

```
private String state; // = "";
```

然后重新运行测试。JUnit 将报告一个错误，而不是测试失败。生产代码或者测试代码遇到问题时，错误会发生。在这个例子中，问题在于您将消息发送给了一个未初始化的引用。JUnit 的第二个面板显示了这个问题。

```
java.lang.NullPointerException
  at studentinfo.Student.isInState(Student.java:37)
  at studentinfo.StudentTest.testInState(StudentTest.java:39)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
...
```

158

这叫堆栈回溯，或者堆栈跟踪。堆栈跟踪提供了发生错误的信息，但是理解它还需要一点侦探的工作。堆栈跟踪的第一行告诉您问题是什么。这个例子中，问题是 NullPointerException。NullPointerException 被有问题的代码抛出，实际上是一个错误对象，或者叫异常。

堆栈跟踪中的其余行回溯了消息的发送，直到当前错误。一些行引用了您编写的类和方法，其它行引用了 Java 系统库的代码和第三方代码。解密堆栈跟踪最简单的方法是——向下阅读，直到第一行您自己编写的代码。然后，继续阅读，直到最后一行您自己编写的代码。最后一行代码是入口点，您应该从这里开始研究。

上面的例子中，最后一行执行的代码是 Student.java 的第 37 行（与代码里的行号可能不一样）。该行代码被 StudentTest 第 39 行的消息激活。所以，循着 StudentTest 第 39 行代码的线索，直到解决问题。接着，您应该来到下面的代码行：

```
assertFalse(student.isInState());
```

看一下 Student.java 的第 37 行，在这里产生了 NullPointerException：

```
return state.equals(Student.IN_STATE);
```

如果某个引用没有被显式的初始化，那么该引用的值为 null。null 代表名为 null 的对象的唯一实例。如果发送消息给 null 对象，您将收到 NullPointerException。该行代码中，您发送 equals 消息给未初始化的引用 state，所以就产生了 NullPointerException。

在第 6 课您将看到，打算发送消息之前如何判断某个成员变量为 null。现在，保证 String 类型的成员变量被初始化为空字符串（""）。

恢复您的代码，正确初始化成员变量 state。重新运行测试。

再看基本类型的初始化

只有引用类型的成员变量——可以指向内存中的对象——可以被初始化为 null 或者被赋值为 null。您不能将基本类型的变量初始化为 null，而且这样的变量也永远不能为 null。 159

这包括了布尔变量和数字类型的变量（char 和 int，以及第十课会学到的：byte、short、long、float 和 double）。布尔类型的成员变量有初值 false。数字类型的成员变量有初值 0。

对于数字类型，即使 0 通常是有用的初值，但是如果 0 对于成员变量是一个有意义的值，那么您应该显式地将该成员变量初始化为 0。例如，如果 Java 虚拟机初始化对象时，0 表示计数器将从 0 开始计数，那么就应该显式地将计数器初始化为 0。如果您打算在稍后的代码中，将有意义的值显式地赋给某个成员变量，那么您不必将这个成员变量显式地初始化为 0。

只在必要的时候，才提供显式的初始化。这将有助于澄清开发者的意图。

练习

1. 在字符串后面连接换行符是重复性的操作，这导致了重复代码。把这个功能提取出来，在类 util.StringUtil 中增加一个工具方法。通过把构造函数设置成 private，从而将类 StringUtil 变成工具类。您需要把类常量 NEWLINE 移动到这个类。然后使用编译器找出所有受影响的代码。确保对这个工具方法进行了测试。
2. 将类 Pawn 转变成更通用的类 Piece。一个 Piece（格子）由颜色和名字（卒、马、车、象、王后、国王）组成。Piece 应该是值对象：拥有私有构造函数，而且在创建之后不能被修改。创建工厂方法，返回基于颜色和名字的 Piece 对象。消除创建默认格子的能力。
3. 改变 BoardTest，从而反映完整的棋盘：

```
package chess;

import junit.framework.TestCase;
import util.StringUtil;

public class BoardTest extends TestCase {
   private Board board;

   protected void setUp() {
      board = new Board();
   }
   public void testCreate() {
      board.initialize();
      assertEquals(32, board.pieceCount());
      String blankRank = StringUtil.appendNewLine(".......");
      assertEquals(
```

160

```
                StringUtil.appendNewLine("RNBQKBNR") +
                StringUtil.appendNewLine("PPPPPPPP") +
                blankRank + blankRank + blankRank + blankRank +
                StringUtil.appendNewLine("pppppppp") +
                StringUtil.appendNewLine("rnbqkbnr"),
            board.print());
    }
}
```

4. 保证 Board 实例包含 16 个黑色的格子和 16 个白色的格子。使用类计数器来记录 Piece 对象的数目。确保两种情况下运行测试都没有问题（第二次：点击“Run”之前，取消 JUnit 的复选框“Reload Classes Every Run”）。
5. 在类 Piece 中创建方法 `isBlack` 和 `isWhite`（当然，要测试优先）。
6. 收集每一个测试方法的名字。将类名放在每一个测试方法名字的前面。让朋友看这个列表，并且请朋友回答每个类中的方法都是什么意思。
7. 再次阅读“简单设计”。当前的象棋设计是否符合简单设计这个模式？ 161