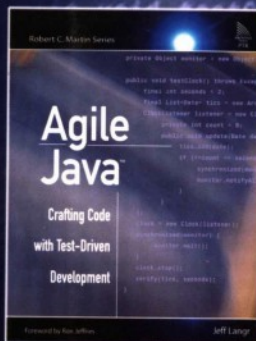


Robert C. Martin Series



Agile Java

中文
版

[美] Jeff Langr 著

涂波
孙勇

译

测试驱动开发的编程技术

Crafting Code
with Test-Driven
Development

Foreword by Ron Jeffries

Agile Java中文版

测试驱动开发的编程技术

掌握Java 5.0和TDD: 构建更健壮、更专业的软件

掌握Java 5.0, 面向对象设计和测试驱动开发。本书将三者编织在一起, 形成一种旨在构建专业、健壮的软件系统的统一连贯的方法。Jeff Langr向你展示如何把Java和TDD结合在一起, 并且贯穿整个开发周期: 帮助你从一开始就借助当前最快最有效的开发技术。

Langr的这本书是写给每一位程序员, 甚至包括对Java、面向对象开发或敏捷方法几乎没有经验的程序员的。他向我们展示如何把口头需求转变成实际的, 然后如何利用这些测试创建可靠的、高性能的Java代码, 从而解决实际问题。本书不仅是教授Java语言的核心特性, 而且提供这些特性的测试代码例子。以TDD为中心的方法不仅带来更好的代码, 而且提供有效的反馈, 可以帮助你更快地掌握Java。较之传统的教学技巧, TDD方法是一个划时代的开始。

- ▶ 从Java程序员的角度, 展现对TDD和敏捷编程技术的专家看法
- ▶ 带给你融合Java、TDD和面向对象设计的最佳实践
- ▶ 设置Java 5.0的开发环境, 编写你的第一个程序
- ▶ 覆盖所有基础内容, 包括字符串、包, 等等
- ▶ 介绍面向对象概念, 包括类、接口、多态和继承
- ▶ 一些章节包含了异常、日志、数学、I/O、反射、多线程和Swing的细节
- ▶ 对Java 5.0的关键创新, 提供了无缝的解释, 从generics到annotations
- ▶ TDD如何影响系统设计, 以及系统设计如何影响TDD
- ▶ 补充敏捷或者传统方法, 包括极限编程 (XP)

Jeff Langr有着超过二十年的开发经验。现在, 他通过自己的公司—Langr Software Solution (www.LangrSoft.com) 提供软件开发、设计, 以及敏捷过程方面的咨询。Langr在Object Mento为Uncle Bob Martin工作过两年。Langr是*Essential Java Style* (Prentice Hall PTR, 1999) 的作者, 并且在*Software Development*、*C/C++ Users Journal*, 以及其他各种在线杂志和门户上发表了很多关于Java和TDD的文章。

图书分类: 程序设计

ISBN 7-121-02704-6



9 787121 027048 >



网上订购: www.dearbook.com.cn
第二书店 第一服务



责任编辑: 周 筠

本书贴有激光防伪标志, 凡没有防伪标志者, 属盗版图书。

ISBN 7-121-02704-6

定价: 69.00 元

Agile Java 中文版

——测试驱动开发的编程技术

Agile Java Crafting Code with Test Driven Development

[美] Jeff Langr 著

涂 波 孙 勇 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING



内容简介

本书将当前流行的敏捷开发方法融入到了 Java 语言的实践中, 给了您学习并掌握 Java5.0、面向对象方法的机会, 同时您还将学习测试驱动开发方法。作者很好地将三者融合在一起, 全力教读者如何使用 Java5.0 开发专业的软件。

本书语言浅显易懂, 颇有趣味, 覆盖了上述三方面的内容, 准确地展示了如何将 Java 和 TDD 进行有效的整合; 帮助开发者在整个软件生命周期中使用这些方法, 以适应现代软件行业对高开发速度和高准确性的要求。对于希望使用 Java 5.0 作为开发工具的开发人员而言, 本书是一本很好的入门书籍。

从市场来看, 现在的软件业竞争非常激烈, 软件的业务需求变化快, 经常会出现变更, 传统的软件开发方法已经很难适应这种快速变化, 在这种高速变化的情况下, 显得捉襟见肘; 而融合了敏捷方法的 Java 可以从某些方面应对这种变化, 这使得本书成为 Java5.0 学习者一本优秀的参考书。

Authorized translation from the English language edition, entitled Agile Java™: Crafting Code with Test-Driven Development, 1st Edition, 0131482394 by Langr, Jeff, published by Pearson Education, Inc, publishing as Prentice Hall PTR, Copyright©2005 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright ©2006.

本书简体中文版由电子工业出版社和 Pearson Education 培生教育出版亚洲有限公司合作出版。未经出版者预先书面许可, 不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育集团激光防伪标签, 无标签者不得销售。

版权贸易合同登记号: 图字: 01-2005-0910

图书在版编目 (CIP) 数据

Agile Java 中文版: 测试驱动开发的编程技术 / (美) 兰格 (Jeff.L.) 著; 涂波, 孙勇译. —北京: 电子工业出版社, 2006.9

书名原文: Agile Java: Crafting Code with Test Driven Development

ISBN 7-121-02704-6

I. A... II. ①兰... ②涂... ③孙... III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2006) 第 054418 号

责任编辑: 周 筠

印 刷: 北京智力达印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 42 字数: 680 千字

印 次: 2006 年 9 月第 1 次印刷

定 价: 69.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系电话: (010) 68279077;

邮购电话: (010) 88254888。

质量投诉请发邮件至 zhs@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

电子工业出版社
PDG

对 Agile Java 的赞誉

“这是我阅读过的最实用、最高效、并且令人愉悦的编程书籍。此书具有结果导向的特点。”

Steve Bartolin, President and CEO, The Broadmoor

“终于有了这样一本书——书中提供了把 Java 和测试驱动开发整合在一起的、作为扎实起点的课程！我知道，如果在最初学习 Java 的时候，就能读到 Jeff 的这本书，那么大多数的软件开发人员都能避免某些给您带来多年痛苦的编程习惯。”

Steven A. Gordon, Ph.D., Manager, Arizona State University Software Factory (<http://sf.asu.edu/>)

“Jeff 利用他在 Java 和敏捷过程方面的特别深入的专业知识，向我们展示了一种强大的方法。利用这种方法，我们能够构建干净的、结构良好的、并且经过彻底测试的 Java 程序。强烈向您推荐这本书。”

Paul Hodgetts, Founder and CEO, Agile Logic, Inc.

“这是一种优秀的学习 Java 的方式。Agile Java 不止讲述 Java 语言的基础、配置、以及工具，而且还针对测试驱动开发的概念、重构、以及面向对象编程，提供了卓越的指导。我们整个工程师团队都要求阅读这本书。”

Andrew Masters, President and CIO, Stwo8, Inc.

“任何开发 Java 应用的程序员都必须阅读这本书。书中包含了关于如何比传统方法更快、更有效率地编写高质量、可扩展的 Java 程序的思想，这些思想是创新的、富有洞察力的。初学者和有经验的程序员都能从本书中汲取大量的营养。”

Bret McInnis, Vice President eBusiness Technologies, Corporate Express

Robert C. Martin 系列

本系列的目的是提高软件开发的技术水平。系列中的所有书籍都是关于技术的，并且注重实践、内容充实。所有作者都是经验丰富的软件大师，他们都专注于实践，而不是理论。通过阅读，您可以知道他们都做过什么，而不是他们想了什么。如果系列中的某本书是关于编程，那么书中会有很多代码。如果系列中的某本书是关于管理，那么会有大量来自于实际项目的案例学习。

任何严肃的从业者都会把下面这些书籍摆放在自己的书架上。这些书籍与众不同，并且可以指导从业者成长为软件大师。

Agile Java™: Crafting Code with Test-Driven Development

Jeff Langr

Working Effectively with Legacy Code

Michael Feathers

UML For Java Programmers

Robert C. Martin

Agile Software Development: Principles, Patterns, and Practices

Robert C. Martin

Agile Software Development with SCRUM

Ken Schwaber and Mike Beedle

Extreme Software Engineering: A Hands on Approach

Daniel H. Steinberg and Daniel W. Palmer



译者序

测试驱动开发是极限编程 (Extreme Programming) 的重要特点。它以持续测试来推动代码的开发, 既简化了代码, 又保证了软件质量。测试驱动开发起源于 XP 方法中提倡的测试优先实践。测试优先实践重视单元测试, 强调程序员除了编写代码, 还应该编写单元测试代码。在开发的顺序上, 它改变了以往先编写代码, 后编写测试的过程; 而是采用先编写测试, 后编写代码来满足测试的方法。这种方法在实践中能够起到非常好的效果, 使得测试工作不仅仅是单纯的测试, 而且成为设计的一部分。

测试驱动开发是近年来“敏捷革命”中最热的话题之一。但是, 市面上鲜见与此相关的有深度的、和 Java 开发紧密结合的作品。本书的出版填补了这一空白, 能够满足众多 Java 程序员的需要。本书的特点是:

- 一本结合 Java、描述测试驱动开发方法的非常实用的指南: 真实的问题, 真实的解决方案, 真实的代码。
- 本书包含了一个完整的用 Java 编写并使用 JUnit、Ant 等工具的软件项目。
- 本书适合每一位对测试驱动开发方法感兴趣的 Java 开发人员和项目经理。

快乐工作的基础就是对自己有信心, 对自己的工作成果有信心。测试驱动开发提供的测试集可以作为您信心的源泉。相信您在阅读本书后, 也一定会为这种敏捷的开发过程所感染, 喜欢上 Java, 喜欢上测试驱动开发的技术。

非常荣幸能有机会和孙勇先生合作, 翻译这本书。由于本人才疏学浅, 翻译中难免存在错误, 衷心的欢迎读者指正。特别要感谢聂涛先生和魏泉先生。他们对本书的初译进行了非常细致的技术审阅, 我从中学到了很多。没有聂涛先生和魏泉先生的辛苦工作, 就不会有本书现在的翻译质量。另外, 要感谢我的父母, 还有我的妻子吕玫, 他(她)们的支持永远是我继续向前的动力。

为了保持原书的风采, 本书的封面、版式、字体等各方面均维持原书的风格。关于本书中文版的文字或技术问题, 您可以发送 email 到: javatdd@gmail.com。

涂波

2006 年 4 月于北京

Agile Java 中文版

关于作者

Jeff Langr 是一名有着多年开发经验的独立软件咨询师。他通过自己的公司 Langr Software Solutions (<http://www.LangrSoft.com>)，向客户提供关于软件开发、软件设计，以及敏捷过程的专业咨询。

Langr 在极受尊敬的 Object Mentor 工作了两年，这是第一家针对 XP 编程的咨询公司，公司的老板是 Bob Martin。之后，Langr 又在多家世界 500 强公司任职。当然，也在倒闭了的互联网公司工作过。

Langr 有在大学里教授 Java 课程的背景。他成功地培养了数百个在 Java、TDD、XP，以及面向对象开发等方面都非常专业的学生。Langr 也多次在国际会议和地区社团会议上，发表关于软件开发的演讲。

Langr 编写了入门书籍 Essential Java Style (Langr2000)，这是一本关于如何构建高质量 Java 代码的指南。五年以后，您依然能在 blog 中发现，某些人发誓说他们已经把这本书看得折了角。在 Software Development、C/C++ Users Journal，以及包括 Developer.com 在内的多个在线杂志中，都能找到 Langr 的关于 Java 和 TDD 的文章。您可以从这里找到关于这些文章的链接：
<http://www.langrsoft.com/resources.html>。

他想补充说：他生活在软件开发的世界里，希望他对代码进行精雕细刻的热爱，能够贯穿在本书之中。Langr 居住在 Colorado Springs，和他的妻子，以及三个孩子 Katie、Time、Anna 生活在一起。

前言

Jeff Langr 从 agile (敏捷) Java 的角度, 写了一本非常有趣的 Java 书籍: 利用测试驱动开发的技术来雕琢代码。本书的目的是教会初学者如何使用 Java 语言, 以及他和我都知道的最佳的开发方法——测试驱动开发 (Test-Driven Development, 简称 TDD)。TDD 承诺可以带来巨大的潜在价值, Jeff 已经证明了这一点。我非常荣幸为这本书作序, 也非常荣幸向您推荐这本书。

本书不仅对初学者很有用, 对于有经验的程序员而言, 这也是一本能够带给您新的内容, 帮助提升 Java 语言水平的好书。我不打算说这是一本认证指南, 或者和其他很多“Java 大全”相类似的书籍。这不是本书的要点。本书的要点在于帮助您熟练掌握如何使用 TDD。TDD 对于您将来的学习和日常工作, 都会有很大帮助。

本书以面向对象的概念和思想开始阐述。如果您了解对象的概念, 那将有助于阅读这本书。如果您不了解对象的概念, 那就要在继续之前, 去熟悉面向对象。接着, 每前进一步, 您都会使用测试驱动开发技术。如果您没有用过 TDD, 可能在一开始的时候会觉得有些不适应。但是, 如果您象我们一样, 尝试了 TDD, 那 TDD 就会成为您的开发工具箱中常用的工具。

如果您已经安装了 Java 和 JUnit, 请继续。如果没有, 请一定阅读“搭建环境”, 在开始真正的例子之前, 正确地配置您的环境。一旦能够正确编译和运行一个简单的 Java 程序, 就意味着准备就绪了。

Jeff 要求您敲入测试和例子代码, 我也赞同 Jeff 的要求。TDD 的原则是通过实践来学习, 而不只是阅读。您需要形成自己的关于开发节奏的看法。此外, 敲入编程书籍中的例子是最好的学习方法。

在本书中, Jeff 帮助您构建两个应用。一个是学生信息系统, 另一个是国际象棋。在阅读所有章节的过程中, Jeff 向您介绍了 Java 的基础。或许更重要的是, 您会接触到某些最重要的深入的知识, 包括接口、多态、模拟对象、反射、多线程, 以及泛型。

第 10 课是关于 Java 的数学特性的, 在这节课中, 我发现了利用测试来学习语言和库的新特性的方法。很容易理解类似 `BigDecimal` 的内容, 您可能会想: “我已经明白了。”也许, 暂时您的确是明白了。但是, 当把学到的知识变成测试的时候, 两件事情发生了: 第一, 阅读会丢失一些信息, 通过编写测试可以掌握这些遗漏的知识。第二, 测试记录了我们的学习过程, 以

及在学习中的思考过程。因为我已经养成了保留测试用例的习惯，所以我可以参考它们，唤起我的记忆。我甚至常常把书籍的页码索引到某些测试，当我回过头来希望进一步深入的时候，这些测试提供了注解。

第 11 课关于 I/O，提供了一个我不太熟悉但是非常好的例子。因为我在 Java 方面从事的工作比较少，而且我常用的语言中没有和嵌套类相对应的概念，但是 Jeff 提供了一个非常好的例子，帮助我很好的使用和测试嵌套类。

当我写这份前言的时候，我真的进入了这本书，因为 Jeff 把我带到了我从没有到过的地方。我喜欢那样。第 12 课关于模拟对象，第一个例子是敏捷软件开发者经常会遇到的问题：针对一个定义良好，但是还没有实现的外部 API，我们如何实现增量开发？Jeff 向我们展示如何通过接口定义来实现增量开发——必要的时候，接口定义来自文档——然后构建一个表示您对 API 的理解的模拟对象。编写针对模拟对象的测试，通过这些测试我们可以确信：API 和我们所期望的、最终得到的实际代码一样，可以正常工作。这又是一个优秀的方法。

Jeff 是一个相当好的老师。Jeff 要求我们思考和做练习。他知道如果您和我打算开始学习，那么就必须实践：我们必须做练习。书中每一章都有练习。他建议我们思考这些练习，并且完成和我们不熟悉的主题相关的练习。在键盘上敲入代码，让这些练习深入您的大脑，并且在完成这些有趣的例子的时候，参照 Jeff 的指导。您会非常满意所做的工作。

本书至少可以带给您三个好处：学会可能此前并不了解的 Java 知识，即使您不是一个 Java 初学者。学会在多种情况下，如何应用测试驱动开发（某些情况下，您可能发现自己很难找到合适的方法）。最后，在掌握本书的技巧之后，您可以把这种颇有价值的技术放进自己的专业工具箱里。

我很喜欢这本书，而且发现它非常值得一读。希望您也会喜欢这本书！

Ron Jeffries

www.XProgramming.com

Pinckney, Michigan

November 2, 2004

致 谢

非常感谢所有想象力丰富的程序员，以及正在学习中的程序员，给予了本书很多富有洞察力的意见。

感谢我的前任雇主——Object Mentor 的 Bob Martin，Bob 是一个伟大的导师，同时也是该丛书的编辑。Bob 早期的复审，使得本书更加精确。非常荣幸我的这本书能够成为 Robert Martin 系列的一部分。也非常感谢 Ron Jeffries 为拙作说了很多赞扬的话。

ThoughtWorks 的 Jeff Bay 做了非常重要的工作，他给了我想要的东西——残酷但是真实的反馈。Jeff 负责提供每节后面的练习。

非常感谢 Steve Arneil、Dave Astels、Tim Camper、Dan D'Eramo、Michael Feathers、Paul Holser、Jerry Jackson、Ron Jeffries、Bob Koss、Torri Lopez、Andrew Masters、Chris Mathews、Jim Newkirk、Wendy、David Peterson、Michael Rachow、Jason Rohman、Tito Velez 和 Jeroen Wenting。他们校对这本书，提出了大量的修改意见。

特别感谢 Paul Petralia 以及他在 Prentice Hall 的全体员工，是他们使我尽可能顺利地完此书。

再次感谢我的姐姐 Christine Langr，她提供了本书封面的图片，并且帮助我解决了很多设计问题。

同时，感谢 San Antonio 的 Jim Condit 为我提供了 Nerf play 和住所。并且，感谢我的妻子 Kathy Langr 提供了例子（而且陪伴我度过整个艰苦的写作过程）。

引言

我是一名软件工匠¹。我的开发生涯中，大部分时间都在致力于快速构建解决问题的方案。与此同时，我也竭尽全力确保我的代码经过了细心的雕琢。我一直为完美的代码而努力，但是我知道这个目标是无法达到的，特别是为了发布产品，公司持续给你压力的时候。面对自己每天写的代码，我会有一丝自豪，然而当我回过头看昨天的代码，我经常 would 感到困惑：“昨天我到底是怎么想的？”这种挑战使我不断改进自己的代码——永远热切的盼望着下次做得更好一点，比上次痛苦更少一些。

本书为您描绘出一条成功学习和掌握 Java 开发的捷径。本书基于我在教授编程和自学一门新的编程语言中逐渐总结出来的方法：测试驱动开发（TDD: test-driven development²），这是一种引入大量底层反馈的技术。这些反馈使您很快看到行动的结果。使用 TDD，您将学会如何雕琢您的 Java 代码，从而得到稳定的面向对象设计和高可维护、高质量的系统。

我已经使用 TDD 的方法开发软件超过四年，然而时至今日我仍然惊讶于这种方法带给我的好处。TDD 提高了我编写代码的质量，而且每周都教给我新的东西，使我变得更富有效率。我也曾经使用 TDD 的方法在我的公司和 Object Mentor（该公司一直用这种方法教授课程）编写和教授编程语言。

在学习 TDD 之前，我花费了超过十五年的时间采用“传统”方法来学习、使用和教授开发语言，没有使用测试驱动的开发。学生编写并执行一个例子程序，通过程序的输出获得反馈。尽管这是一种有效的方法，但是我过去的经验表明这种方法会导致学生对语言细节不能很好的理解。◀ 1

与之相对照的是，TDD 所带来的大量迅速的反馈会持续地强迫您编写正确的代码，并且很快指出有问题的代码。经典的“编码—执行—观察”的方法可以提供反馈，但是反馈的速度要慢得多。不幸的是，当前的教学将其作为一种占主导地位的方法。

已经有人尝试了革新方法来教学。上个世纪九十年代，Adele Goldberg 开发了一种用来教授年轻学生的软件——LearningWorks。这种软件通过让用户动态执行小段代码来直接操作可视化对象，用户可以立刻看到执行的结果。最近有一个 Java 培训工具采用了类似的方法，该工

¹ See [McBreen2000]。

² 译注：后面的内容中多数地方直接使用 TDD 表示测试驱动开发。

具可以让学生执行小段代码，在活动对象上产生可视的结果。

这些方法的问题是局限在了学习环境中。一旦您完成了培训，您依然必须去了解在不使用这些受限工具的情况下，如何从头构建您自己的系统。通过使用 TDD，您被教会了一种不受限制的技术，您可以继续将其应用在您的职业软件开发生涯中。

本书尽最大可能采用了面向对象的方法，学习 Java 的困难部分在于需要“循序渐进”。为了写出一组有实际意义的类，您必须掌握的 Java 语言的最小集合是什么呢？

大多数书籍用“Hello world”作为第一个 Java 例子程序，不过这个程序似乎并不适合作为最初的例子：`class Hello { public static void main (String[] args) { System.out.println ("hello world"); } }`。这个简短的程序至少包含了一打您必须要学习的概念。更糟的是，除了这一打概念，至少还有三个您最好马上掌握的非面向对象的概念。

在这本书中，您将学到正确的从头开始编写代码的方法，之后您将会回过头来完全理解“Hello world”³。使用 TDD，您可以很快编写出出色的面向对象的代码。在学习之初您仍然会有很大的障碍需要去克服，但是 TDD 使您免于在一开始就必须去理解一些非面向对象的概念，例如静态方法和数组。您可以适时的掌握核心的 Java 概念，但是您最初的重点应该放在对象上面。

本书向您呈现出一种与以往截然不同的解决问题的方法。它允许您假定从来没有一种叫 C 的语言，C 是 Java 语言的基础，存在了有大约三十年的时间。C 语言给 Java 语言打上了自己的烙印，妨碍您构建出色的面向对象的系统。使用本书，您可以在理解 Java 语言中这些来自 C 的遗产之前，学到正确的构建面向对象系统的方法。

预期的读者

我为想把 Java 作为第一门语言的编程初学者设计了本书；这本书同样会对那些熟悉 TDD 但对 Java 不了解的程序员有所帮助。有经验的 Java 开发者会发现，本书向您展现了一种崭新的，更好的实现目标的方法。

本书涵盖了 Java 2 标准版 (J2SE) 5.0。

Sun 提供了许多类库和 API (应用程序编程接口)，这些类库或者 API 增强了核心 Java 语言的功能。例如：JMS (Java 消息服务) 提供了标准的基于消息解决方案的定义。EJBs (企业 Java Beans) 提供了一种针对大型企业应用、构建基于组件的软件的方法。JDBC (Java 数据库连接) 提供了与关系型数据库交互的标准接口。许多这样的高级 APIs 集合在一起，被称为 J2EE (Java

³ 不要着急，尽管如此，为了保证您可以编译和执行 Java 代码，您实际上还是以“hello world”程序开始，但是您不必马上完全理解它。

2 企业版)。很多 APIs 需要一整本书来阐述。现在已经出版了许多有关 J2EE 的书籍。

本书只针对一小部分扩充 APIs 进行了入门介绍。本书用敏捷 Java 向您展现大多数企业应用都会广泛使用的技术,比如日志、JDBC,以及 Swing。某些内容(例如日志)会带给您大多数应用都需要掌握的知识。还有些内容(例如 Swing 和 JDBC)会让您对该技术有基本的理解。所有的内容会提供给您起步所必须的知识,并且会告诉您何处去寻找更多的信息。

假如您正在开发移动应用,您或许正在使用 J2ME (Java 2 微型版)。J2ME 是一个面向资源限制型环境的 Java 版本,例如手机。相对 J2SE, J2ME 有很多显著的限制。本书不针对 J2ME 作专门的讨论,但是,本书讨论的大多数 Java 核心技术和概念同样适用于 J2ME 环境。

在使用任何 Java 技术扩展之前,您必须理解 J2SE 中提供的核心语言和开发库。本书将帮助您掌握这些知识。

3

本书所不能提供的

本书不能对 Java 语言的每个方面都进行详尽地论述。它提供了一种敏捷学习 Java 语言的方法。不是授之以鱼,而是教您如何钓鱼,以及在什么地方发现鱼。本书将教会您绝大多数的 Java 语言的核心概念。的确,当您完成本书的 15 节课程,您将能够编写出有质量的 Java 代码。但是,这一目标要求您了解本书所没有提及的一些语言功能和细微之处。

熟悉所有 Java 语言细节的方法是仔细阅读 Java 语言规格说明(JLS)。您可以从这里得到 JLS 的第二版:<http://java.sun.com/docs/books/jls>。第二版的 JLS 涵盖了 J2SE5.0 之前的版本,但不包括 J2SE5.0。我写这本书的时候, JLS 第三版正在编写之中,您可以从这里得到它的维护检查版本:http://java.sun.com/docs/books/jls/java_language-3_0-mr-spec.zip。

如果您要深入了解扩展 Java 类库, Java API 文档和源代码是最好的材料。

本书不是一本认证指南,它不能帮助您通过认证考试。一本好的认证教材会教您如何应付考试。本书会帮助您破译代码质量低劣的原因,教会您如何编写专业的 Java 代码。

本书不会试图去溺爱你。学习编程是一种很大的挑战。编程包括思考和解决问题——不是容易到连白痴和傀儡都能够胜任的。我努力避免在本书中侮辱您的智慧,也就是说,我努力使全书更有趣和易于阅读。它是您和我之间的对话,更是在您将来职业生涯中和我的对话,同时它也是您和您的计算机之间的对话。

TDD 不承诺

对 TDD 有经验的开发者也许注意到他们的方法和我在本书中的方法有风格上的不同。有很多种方法来实践 TDD。没有一种技术是完美的,或者绝对正确的。按照对您最有利的方式去做,

4

按照最有意义的方式去做，只要不违背本书中的基本原则。

读者一定会发现需要改进代码的地方⁴；即使是初学者，也请不要犹豫，告诉我您看到的您不喜欢的代码。把您的建议发送给我，我会在下一版中修正。您能改进几乎所有的代码或者技术。

在您学完第1课后，会发现大量的测试，好像这些代码是一气呵成，可事实上并不这样。每个测试都是由逐个断言构建而成，采用了比这本书所能忍受的还要小的增量一点一点完成的。请您一定要记住，使用TDD编写代码的时候，最重要的是采用尽量小的步伐并得到经常的反馈。当我说“小”的时候，我的意思是非常小。如果您认为您正在采用小的步伐，请尝试采用更小的步伐。

如何使用这本书

本书的核心包括15节课程，每一课大约30页。您将像幼儿蹒跚学步一样，开始您的Java、TDD、面向对象的旅程，结束的时候您将拥有扎实的基础以从事专业Java开发。

15节的核心课程是循序渐进的。您从第1课开始，完成每一课之后再再进行后面的学习。一旦您学完了所有核心课程，您将对如何编写健壮的Java代码有深刻的理解。

假如您没有完成15节核心课程，就不能假设您知道如何编写优秀的Java代码！（即使您完成了这些课程，您依然不能算是一个专家）。每一节课程都基于前面的课程。假如您在完成所有课程之前就停止您的脚步，您将缺乏完整的理解，这也许会导致您写出拙劣的代码。

每一节课程的开始，都有本节所要讨论主题的一个概要。后面紧接着详细的论述。我用文字介绍每一个语言特性，用测试代码详细说明它。我用相应的代码实现来展现每一个语言特性。点缀其间的是关于TDD技术、面向对象、优秀开发实践的讨论。

5

此外，我提供了3节附加课，以覆盖更多的Java主题。其中的两节课程介绍Swing，这是一个专注于用户界面开发的Java工具。这两节课程给您提供充分的信息，足以帮助您开始用Java构建健壮的用户界面程序。但是我最大的愿望是给您一些怎样用TDD构建它们的思路。第三节课简要介绍了一些大多数Java程序员都想了解的Java主题。

为了最有效的学习，您应该输入并执行本书中的每一段测试和实现代码。您可以下载这些代码（请看下一段），但是我强烈建议您亲自输入每一段代码。正确运用TDD依赖于您清晰地理解测试和编码之间来回转换所包含的韵律。假如您只是下载代码然后去执行它，您将几乎不会学到太多。键盘的触觉似乎传递了大量的知识。

但是，我又算什么，能要求您按照指定的方式去学习？您可以选择从这个地址下载代码：

⁴ 稳定的成时持续的结对编程组合会有帮助。

<http://www.langrsoft.com/agileJava/code>。我按照课程章节组织好了这些代码，也提供了代码的执行结果，也就是课程结束的时候它的状态。这易于您在任何一点拾起课程，特别是很多例子一直沿用到后续章节。

练习

15 节核心课程的每一节要求您一点一点构建一个大学学员信息系统。我选择这个普通的主题有助您增量开发和扩展现有代码。每一节课程的最后都有若干练习。不同于学员信息系统，这些练习由 Jeff Bay 提供，要求您一步一步构建一个象棋软件。

其中的一些练习非常有挑战性，但是我强烈建议您完成每一个练习。通过这些练习，您可以在没有我的帮助下，实战性的学习如何用 Java 解决问题。所有的练习给您又一次深入理解每一节课程的机会。

本书中的约定

代码紧随文字之后，目的是使其成为交流的一部分。假如我说“以下代码”，那就意味您接下来会读到一段代码；而我说“以上代码”，那就意味着代码会出现在相邻的文字上方。

代码（下面）用等宽字体显示：

```
this.isCode();
```

大段代码中较少的部分用黑体显示，黑体表明这是当前例子中新增加的代码，或者是需要特别的关注。

```
class B {
    public void thisIsANewMethod() {
    }
}
```

直接出现在文字中的代码，例如 `this.someCode()`，同样用等宽字体。但是类的名字，例如 `CourseSession`，和其余的文字使用相同的字体。

我经常在例子中使用省略号。表明这部分代码存在，但是与当前的讨论或者例子无关。

```
class C {
    private String interestingVariable;
    ...
}
```

```
private void someInterestingMethod() {
}
...
```

交替使用文字和代码来表达思想。例如，也许我会说需要取消工资支票，也可能会说您应该发送一个 cancel 消息给 PayrollCheck 对象。这种方法有助于您在理解需求和用代码实现需求二者之间建立必要的联系。

类名约定使用单个名词，例如 Customer：“您使用 Customer 类创建多个 Customer 对象”。为了增加可读性，我有时用类名的复数形式代指这些 Customer 对象：“这个集合包含了所有从文件系统加载的 Customers”。

新术语第一次出现用斜体。绝大多数的术语在术语表中（附录 A）。

贯穿全书，您都将根据规格说明并把它们转换成 Java 代码。按照敏捷过程的传统，我会用非正式的语言来描述这些规格说明。这些规格说明是用户需求，也叫 stories：一个 story 是对多次对话的约定。您经常为了获取进一步的需求细节，需要和 story 的讲述者（有时候称其为客户）进行对话。在本书中，如果一个 story 没有理解，请试着往下读，读相应的测试，看其如何解释这个 story。



我在本书中用一个“讲故事”图标来强调 stories。该图标强调口头的非正式的 stories。

学习开发艺术的最好途径是和一位有实际经验的人一起工作。您将会发现很多编程的秘密。一些秘密是您为了掌握开发所必须知道的基本概念。其他的秘密向您展现需要密切注意的易犯的错误。您必须赢得这些挑战，记住您所学习的，努力成为一名成功的 Java 程序员。



我用桥梁图标来标记这样的关键点（桥梁跨越了易犯的错误）。很多关键点在任何语言中都是有效的，不只是 Java。

“敏捷”综述

这一节简要介绍本书中的一些核心概念，包括 Java，面向对象编程，以及测试驱动开发（TDD）。您将会得到下面问题的答案：

- 什么是“敏捷”？
- 什么是 Java？
- 什么是面向对象编程？
- 为什么要面向对象？
- 什么是对象？
- 什么是类？
- 为什么要采用 UML？
- 什么是继承？
- 为什么要测试驱动开发（TDD）？

什么是“敏捷”

这本书的名字是 *Agile Java*。“敏捷（agile）”是描述一些软件开发方法学的新的标志性词汇。如果给一个宽松的定义，一个方法就是构建软件的一种过程。目前有一系列方法可以用来构建我们的软件，其中有很多已经被规范化并被命名为某种方法。现在有许多主要的过程被投入应用，或许您已经听说过其中的一些：瀑布、RUP（统一软件过程）、XP（极限编程），以及 Scrum。

随着我们对如何有效工作了解的增多，方法学也在不断的发展。瀑布法是一种古老的方法，该方法提倡软件开发过程中编写大量的文档，严格地事先（例如在项目刚启动的时候）定义需求和系统设计，并且将项目开发分成若干个串行的阶段。该方法的名字来源于这样的图：开发过程自顶向下，从一个阶段到下一个阶段。

尽管瀑布法适合某些场合，但是它的局限性在其他软件项目中会引发严重的问题。用瀑布

法意味着您很少能随着项目的进展而做出改变。因为这些原因，瀑布法有时候被视为重量级软件过程，因为瀑布法给开发者很重的负担，限制开发团队及时对变化做出响应。

与之相对，敏捷过程，是一种十分新颖、轻量级的过程。敏捷过程不强调编写文档和提前确定。敏捷过程致力于拥抱变化，XP（极限编程）或许就是最著名的敏捷过程的实例。

RUP 是另外一个著名的过程。从技术上讲 RUP 是一个过程框架。它提供了一个目录，您可以从中选取一部分来定制自己的过程。这样一个 RUP 过程实例可以打破重量级过程与敏捷过程之间的藩篱。

不论是重量级的还是敏捷的过程，您在构建软件的过程中都必须做下面的事情：

- 分析：通过收集和提炼需求，来决定让您的软件做什么
- 计划：推算用多长时间完成您的软件
- 设计：决定如何把所有您要做的东西装配在一起
- 编码：用一种或多种开发语言来构建您的软件
- 测试：保证工作的正确性
- 部署：将软件交付到实际环境
- 文档：向不同的用户描述软件，包括需要知道如何操作软件的最终用户和需要知道如何维护软件的程序员
- 评审：通过同级评审，保证软件的可维护性和高质量

如果没有以上的工作，您生成的是将会被抛弃的代码，而且您没有考虑到业务需求。某些过程，比如 XP（极限编程）看起来不提倡以上的每一项工作，但事实上它做了，用的是一种有点特立独行的方式。

本书中我很少提到方法学，您可以找到满书橱的专门论述方法学的书籍。绝大多数开发团队使用混合的方法——他们采用各种方法学中成功的技术，并调整它们以适应自己的需要。许多开发团队从来不承认什么方法，甚至这些团队也通常遵循某些非文档化的过程。

这本书是讲如何构建软件的。本书将注意力集中在一种叫测试驱动开发（TDD）的技术上。TDD 本身不是一种方法，而是一种可以应用在任何软件开发过程中的实践。TDD 起源于 XP（极限编程），是 XP 规定的很多实践当中的一个。您不必为了这本书或是为了 TDD，而去采用 XP。

在这本书里，我很少讨论 XP 或者其他方法。我将精力集中在帮助您用本书构建和改进您的代码。您将学会如何借助 TDD 让您的系统拥抱变化。即使您正在使用可以想象的最僵硬的过程，您依然可以使用本书中的技术来构建高质量的 Java 代码。

什么是Java

当您听到人们谈起 Java，他们通常指的是 Java 语言。Java 语言允许您编写指令或者代码，您的计算机通过解释代码来运行一个应用软件。应用软件的例子有 Microsoft Word、Netscape、在您显示器某个角落的小时钟。编码或者写程序，是编写应用软件的行为。

人们谈论 Java 也可以指的是 Java 平台。平台这个术语通常指底层的操作系统，比如 Windows 或者 Unix，在平台之上您可以运行应用程序¹。Java 扮演了平台的角色，Java 不仅定义了一门语言，同时也提供了开发和执行应用程序的完整的环境。Java 平台是介于应用和底层操作系统的中间层。Java 自身是一个小型的操作系统，它允许您用一种语言编写，在所有主流操作系统上运行您的代码。

您需要下载 Java 软件开发包 (SDK)，它提供了以下三个主要组件：

- 编译器 (javac)
- 虚拟机 (java)
- 一套类库或者 API (应用程序接口)

11

编译器是一个程序，用以读取 Java 源文件，保证它们包含正确的 Java 代码，然后输出 class 文件。源文件是一个包含代码的文本文件。编译器生成的 class 文件包含字节码，字节码描绘了您所输入的代码。字节码采用虚拟机可以快速读入和解释的格式。

虚拟机是一个程序，用以执行 class 文件中的代码。术语“虚拟机”来源于它的行为仿佛是一个完整平台或者操作系统的事实。从 Java 程序的立场看，您的代码不直接调用操作系统提供的编程接口(API)，或许您过去曾经用 C 或者 C++直接调用 Windows API 来开发 Windows 应用。

数字游戏

Java 历史上有几个重要的版本。第一个正式 Java 发布版 (在 1996) 是 Java 1.0，接着是 1.1、1.2、1.3，和 1.4，大致上按照每一年来区分。在版本 1.2，Sun 将这个平台的名称从 Java 改为 Java 2。对当前最近的版本，也许是为了表达这个版本的重要性，Sun 选择开始一种新的版本命名方案。本书中您将学习这个新版本，名字是 Java2 标准版 5.0。本书中我将使用其简称 J2SE 5.0。

您将只用 Java 语言编写代码，并且只用一部分 Java SDK 提供的类库。您也会看到虚拟机的主要功能是作为一个解释器，虚拟机解释您的代码并将其分派给底层操作系统。此外虚拟机负责分配和控制代码所需要的内存。

众所周知，Java 是一种面向对象的编程语言。面向对象编程语言的一个基本前提是您可以在代码里创建客观世界事物的抽象。举例来说，您可以用 Java 编写一个计算器，用其来模拟真

¹ [WhatIs2004].

实计算器的特征和行为。

如果您从来没有编写过程序，那么您非常幸运；如果您没有学过非面向对象的编程语言，比如 Cobol 或者 C，那么您将更容易从头开始学习面向对象编程。如果您被过程或声明语言玷污过，您或许应该把本章多读几遍。预计学习面向对象并不容易：好几个月内，就像一个灯泡吊在额前而又不让眨眼。

本章会介绍面向对象的基本概念。面向对象字面上比较容易理解，但您依然需要时间来理解它的内涵，特别是在没有具体代码实例的情况下。因此，本章很简短。当您开始学习用 Java 编码时，您将会接触到大量的面向对象的概念。

为什么面向对象

面向对象编程从上个世纪六十年代就已经出现了。但是直到上个世纪九十年代，面向对象才开始被真正接受。过去十年中我们发现了很多关于对象编程的优秀理念。其中最重要的是：合适的面向对象可以在应用软件成熟和扩展的过程中，提高您管理维护的能力。

什么是对象

对象是某些相关概念在代码级别的抽象。假如您正在构建一个工资系统，对象可以被编程来表现支票、一个员工的级别分类、或者一个支票打印机。同样对象也可以是活动的抽象，比如一个路由处理可以保证员工被导向到恰当的处理队列。抽象的最好的定义是：“放大本质，去掉无关的内容。”²

一个支票对象会包括一些细节，比如支票号、收款人和金额。但是工资系统对诸如支票尺寸、颜色之类的属性没有兴趣，所以您不用在工资对象中表现这些属性。你或许需要在支票打印系统中考虑它们。

让对象和客观世界相关是有价值的，但是您必须非常小心不要让客观世界过分干预您的代码。举例来说，工资系统中的员工对象，包含了诸如员工代号、薪水的属性。您也想保证您可以给员工增加薪水。来自客观世界的设计建议把增加薪水的代码放在别的什么地方。但是实际上，面向对象系统中最正确的方法是让员工对象执行代码去增加薪水。（“什么？员工可以给它自己增加薪水？”）在本书中，您将发现如何做出此类的设计决定。

² [Martin2003].



图 1 发送一个消息

一个面向对象系统首要关注的是行为。面向对象的核心概念是对象之间相互发送消息，从而影响行为。一个对象发送一条消息给另外一个对象，告诉它去做某些事情。举一个客观世界的例子，我发送一个消息给您，让您去锁上前门。在一个面向对象系统中，一个安全对象发送一条消息给一个门禁控制对象，告诉它把所控制的门置于安全状态（图 1）。

在这个例子中被发送消息的门禁控制器是接收者。接受者决定如何处理收到的消息。您决定通过转动插锁锁上前门，另一个人也许会选择打开门锁。在面向对象系统中，门禁控制对象会和一个硬件设备交互，激活电磁锁上门或者弹开电子插销。

也是在这个例子中，消息的发送者只对这样一个抽象概念有兴趣，那就是接收者负责门的安全。消息发送者不关心门是怎样被保证安全的，其中的细节只有接收者知道。封装是面向对象编程的一个关键概念：对系统中的其他对象，隐藏所有不必要的细节。

实际上可以有不同的实现，然而客户不知道或者不关心哪一种实现会被调用。另一个非常重要的面向对象的概念是多态。例子中的安全系统不知道或者不关心门禁控制器是一个电磁锁控制器，还是一个电子插销控制器。在后面的课程中您将学到有关多态的大量细节。

什么是类

在部署安全系统的建筑里有五个电磁门，都位于不同的区域。每个门在物理地址上都不相同。但是，所有的门都有共同的、需要被安全控制的行为，因此它们可以被归为一类。类提供了一种方式来定义一组相关对象的共同性。类是一个模板，或者蓝本，用以创建新的对象。

14

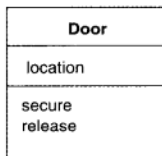


图 2 Door 类

面向对象的安全系统可以定义一个类：Door。这个类规定 Door 对象必须提供安全和释放（解锁）行为。更进一步，每一个 Door 对象应该保存它自己的位置信息。图 2 描述了 Door 类。

图 2 中最上面的格子里是类的名字，第二个格子里是每个 Door 对象都应该存储的属性信息，

第三个格子列出每个 Door 对象支持的行为，这些行为就是门对象收到消息后所能做出的行为。

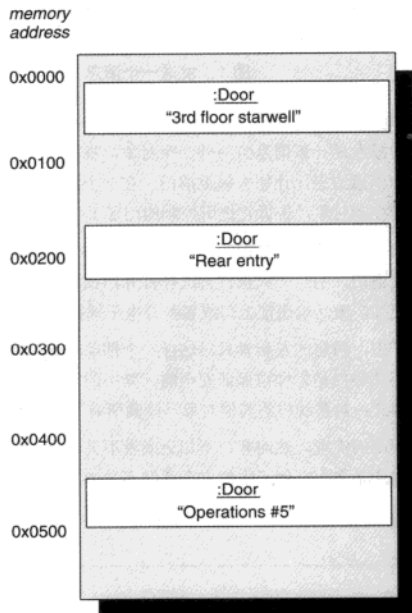


图3 内存中的 Door 对象

在一个面向对象的编程语言中，您使用 Door 类作为创建或者初始化新对象的基础。每一个初始化对象在内存中被分配唯一的地址空间。任何您对这个对象的改变都限定在这个地址空间里。

类图被用来展现面向对象系统的结构。它可以表现类和类之间的关系。类图可以让您对整个系统有一个快速直观的理解。同样，类图也是用来衡量系统设计质量高低的指示器。

在一个类图中，最基本的关系是关联。当一个类依赖于另一个类，或者两个类相互依赖的时候，两个类之间的关联就建立了。假如只有类 B 存在时，类 A 才能够工作，就可以说类 A 依赖于类 B。

类 A 依赖于类 B 的一个主要原因是：类 A 要发送一个或多个消息给类 B。就像安全对象发送一个消息给门禁控制对象，安全类依赖于门禁控制类。

您可以在图 4 中看到有向关联的两个类之间的单向依赖关系。

为什么采用 UML

本书使用 UML (统一建模语言) 标准来阐述某些代码。前面的类图就是采用的 UML。UML 是为面向对象系统建模的事实标准。UML 的主要好处是它可以被全世界所理解, UML 可以让其他程序员很快的理解您的设计思想。

UML 本身不是一种方法, 它是一种图形化语言。UML 是一种用来文档化任何面向对象系统的工具。UML 可以应用在使用任何方法的项目中。

本书会介绍 UML 初步的知识。一本可以获取对 UML 更好理解的书是 UML Distilled, 作者是 Martin Fowler³。此外, 您可以从网上得到 UML 规范的最新文档: <http://www.omg.org/technology/documents/formal/uml.htm>。



图 4 类间依赖

本书中的大多数 UML 图只给出最少的细节。我通常忽略属性, 而且只列出特别相关的行为。

大体而言, 我打算用 UML 给你一个有关系统设计的快速的展示。UML 展现的不是设计, 而是设计的模型⁴。代码就是设计。

就 UML 可以帮助您理解系统设计的广度而言, UML 是有价值并且值得去学习的。一旦 UML 被用来描述那些通过阅读源码就很容易理解的信息, UML 就变成了沉重的负担, 变成了昂贵的需要去维护的史前古董。把 UML 作为珍贵的沟通工具才是明智的。

什么是继承

继承是面向对象中的一个概念, 在这里对其进行简要介绍可以帮助您理解第一节课程中的某些概念。继承是系统里, 类与类之间的一种关系, 它允许一个类以其他类为基础, 增加自己特定的行为。

在客观世界里, 门是可以关闭和打开的一类事物, 门可以作为进入/退出点。除了这个共性, 您还可以定制这样的门: 自动门, 电梯门, 银行地下室门, 等等。所有的门都可以打开和关闭,

³ See [Fowler2000].

⁴ [Martin2003/Reeves1992].

但是每一种门也都有某些特定的行为。

在您的安全系统中，您为通用的 Door 类定义了安全和释放两种行为，并且允许每个 Door 对象存储自己的位置。但是，您也必须支持提供附加行为和属性的门。AlarmDoor 提供被激活时发出警铃的能力，同时也可以作为安全门。由于 AlarmDoor 类拥有和 Door 类相同的安全和释放两种行为，所以 AlarmDoor 类继承自 Door 类，只需要提供报警行为的定义。

继承为 AlarmDoor 类提供了重用安全/释放两种行为的能力，在 AlarmDoor 类中没有必要定义这两种行为。对一个 AlarmDoor 对象，这两种行为就仿佛它们已经在 AlarmDoor 类中做了定义。

17

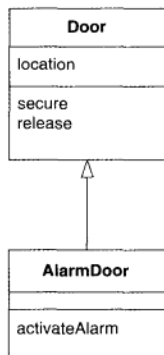


图 5 继承

请注意 AlarmDoor 和 Door 之间的关系也是有向关联。AlarmDoor 依赖于 Door，因为 AlarmDoor 从 Door 继承了行为和属性，所以不能脱离 Door 而独立存在。UML 用有向箭头来表示继承关系。

在第 1 课中，您将通过继承来使用一个测试框架。在更稍后的课程中，您会从更深的层次去理解继承。

为什么要 TDD

事实上这本书中所有的代码都用 TDD 开发。TDD 是一种有关指定什么地方为系统测试的技术。请您在编写实际代码之前编写测试代码，而不是在实际代码之后才提供测试代码⁵。

⁵ [Langr2001].

TDD 是一种简单、快速循环的机制。在编程的每一天，您都需要重复这种机制。一次循环持续若干秒到若干分钟。一次循环包含如下的步骤：

- 用代码写一个规格说明，要符合单元测试⁶的形式。
- 描述测试失败（您还没有实现这个规格说明呢）。
- 编写代码，实现这个规格说明。
- 测试通过。
- 重构，保证系统有一个优化的、干净的代码基线。

这就是 TDD。您一直针对整个系统运行所有的测试，确保新增的代码不会破坏系统中其他的任何代码。

测试在几个方面对系统进行积极正面的驱动：

- 质量：TDD 最大限度减少缺陷的数量。您测试了所有的方面。因为 TDD 促使您解耦，使类与类之间不再高度的互相依赖，这样您就改进了系统设计。
- 文档：一个测试用例定义了一个类的正确用法。
- 可扩展性：在适当的地方进行测试意味着您可以持续改进代码质量，而不用担心会破坏已有的代码。这带来了更低的维护成本。
- 匀速前进：由于 TDD 的每个循环非常短，所以很快得到反馈。于是您能够迅速发现是否遇到了麻烦。您学会去保持一致、可持续的开发进度。

TDD 的另一个好处是它帮助展示本书中的例子。您可以很快学到如何把规格说明变成测试代码，以及怎样把测试代码变成实际的 J2SE 5.0 代码。

从一个程序员的角度看，TDD 是有传染性的⁷。大多数程序员尝试了 TDD 后，会将其作为珍贵的工具保存在自己的开发工具箱里。我和许多程序员包括某些经验丰富的 Java 先驱，讨论过 TDD。他们异口同声说从来打算回到老路上去。1996 年 Kent Beck 在 Smalltalk 方案会议上介绍了为代码编写测试，这是我初次接触到 TDD。当时我的个人反应是：我是一个程序员，不是测试员。时至今日，我惊讶于 TDD 带给我的价值，以及它对我开发水平的极大提高。否则的话，我就不会使用 TDD。

⁶ 该测试验证代码中的一个功能点。Unit test 的另一个术语是程序员测试。

⁷ [Beck1998]。

本章介绍在您开始学习之前所要做的准备工作。

您需要的软件

IDE 或者程序编辑器

用 Java 编程意味着您要能够输入源代码到系统中，接着编译源代码，最后执行生成的 class 文件。完成您的工作有两种主要方式：

- 您可以使用集成开发环境(IDE)。IDE 提供 Java 开发所需要的全部东西。例如 IntelliJ IDEA, Eclipse, Borland JBuilder, 以及 NetBeans。

本书中我用 IntelliJ IDEA 编译所有的例子，您可以从 JetBrains 得到它 (<http://www.jetbrains.com>)。IDEA 非常出色，可以按照您的期望去工作。它完全支持 J2SE 5.0，同时也比其他 IDE 有更快的更新。

- 您可以选择使用程序编辑器。程序编辑器是一种应用软件，允许您输入代码并保存到文件系统。它允许您配置外部工具，比如：Java 编译器和 Java 虚拟机，这样您就可以从编辑器中执行您的 Java 程序。典型的程序编辑器提供额外的支持，使您的编程任务更加轻松。举例来说，大多数程序编辑器能够识别编程语言，从而提供有意义的色彩标记（这种功能叫做语法高亮）。

比较流行的程序编辑器有 emacs, vi, TextPad, UltraEdit, 以及 SlickEdit。我的选择是 TextPad，花费 27 美元您就可以从 <http://www.textpad.com> 下载它。尽管您可以用 Windows Notepad（或者 WordPad）作为编辑器，但是您会发现它缺乏很多编程特性，不是一个有效的编程工具。

那么程序编辑器和 IDE 之间有什么不同呢？

IDE 通常（但不总是）局限于某种语言。IntelliJ IDEA、JBuilder、以及 NetBeans 专门面向 Java，而 Microsoft's Visual Studio 和 Eclipse 提供 Java 语言以及其他语言的支持。现代 IDE 对语言和相应类库都有深入的理解，这些知识使得 IDE 拥有一些辅助编程的高级特性，比如自动补齐（您输入几个字符，IDE 完成您想接下来输入的字符）。现代 IDE 也提供复杂导航工具，这些工具理解代码的内部关系。最后，现代 IDE 通常都有一个调试器。调试器是一种工具，您可以通过它跟踪代码执行的步骤。使用调试器，您可以对代码如何工作有更好的理解。

较之 IDE，程序编辑器是通用工具，用来支持编辑任何类型的程序或者文件。最近几年，程序编辑器增加了理解越来越多语言特性的功能。但是，其强大程度依然比 IDE 差了很多。为了在程序编辑器中查找代码，您通常不得不执行文本搜索。相反，IDE 搜索简单到了可以用一次按键完成。就我所知，没有哪个程序编辑器包含调试器，但是您可以配置某个调试器作为外部工具。

有些人发现 IDE 限制您不得不按照 IDE 要求的方式去工作。多数 IDE 高度可配置，但是您可能会发现，总有什么东西不能满足您的期望。可反过来，一旦您熟悉了 IDE 提供的高级特性，让您退到程序编辑器您又会怀念它。

由于 IDE 总是要花上一段时间才能支持新的语言版本，所以本书中的大多数代码在 TextPad 中编写。您也许也会发现，您打算使用的 IDE 仍然不支持 J2SE5.0。对 IDE 提供商而言，要一年的时间才能使他们的产品跟上最新的 Java 版本。

本书不假定您使用任何特定的 IDE 或者编辑器。在本书中，有些讨论编译细节的例子从命令行执行，但是这样的例子很少。

假如您正在使用 IntelliJ IDEA，附录 C 告诉您如何开始本章的“Hello world”例子。附录 C 也向您展示了如何构建和运行第一节课程中的 TDD 例子。假如您正在使用其他 IDE，请阅读其帮助文档来确定如何配置它。

Java

您需要 Java 2 SDK 版本 5.0 来运行本书中的所有例子。如果您使用含有 SDK 的 IDE，我仍然强烈建议你安装这个 SDK，学习怎样在没有 IDE 的情况下编译和执行 Java 程序是十分明智的。

您可以从 <http://java.sun.com> 下载 SDK。您也可以从这个网址下载相应的文档。从这个网址您可以找到 SDK 的安装指南。

请注意：您可以选择下载 SDK 或者 JRE（Java 运行环境）。您最好下载 SDK（包含了 JRE）。

JRE 本质上就是 JVM（Java 虚拟机）。如果您打算只是运行而不去编译 Java 程序，您可以选择下载安装 JRE。单独提供 JVM，这样当您部署 Java 应用时，只需要在应用中绑定最小的组件集合。

安装 SDK 之后，您可以解开文档，这些文档是大量的网页，保存在一个 doc 的子目录中。

您最好把文档解压缩到 SDK 的安装目录（在 Windows，通常是 c:\Program Files\Java\jdk1.5.0）。一旦您解开了文档，您可以在浏览器中为文档中不同的 index.html 建立书签。您应该抽取出 API 文档，该文档位于 doc 的 api 子目录中。

最后，如果您打算从命令行编译和执行 Java 程序，您需要更新 path 环境变量以包含 JDK 安装目录中的 bin 子目录。您也要创建一个 JAVA_HOME 环境变量，用来指定 JDK 安装目录（假如安装 JDK 时没有创建这个环境变量的话）。请参考操作系统手册来得到如何设置 path 环境变量的信息。

检查您的 Java 安装

您可以对安装和配置进行快速简单的检查。从命令行，执行下面的命令：

```
java -version
```

23

您将会看到类似下面的输出：

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-12345)
Java HotSpot(TM) Client VM (build 1.5.0-12345, mixed mode, sharing)
```

相反，假如您看到这样的输出：

```
'java' is not recognized as an internal or external command,
operable program or batch file.
```

或者：

```
java: command not found
```

这就意味着您没有正确设置 path 环境变量。在 Windows 中查看 path 环境变量的值，可以输入下面的命令：

```
path
```

在 Unix 中，请输入：

```
echo $PATH
```

您会看到 Java 所在的目录。如果没有看到，试着重启一下 Shell 或者 command。

如果您看到了不同的 Java 版本，可能是 1.3 或者 1.4，而且其中一个 Java 版本在 path 环境变量中处于比 1.5 更靠前的位置。请把您的 1.5 安装目录移动到 path 环境变量的最前面¹。

¹ Windows 把最近的 Java 版本放在 SYSTEM32 目录。您可以谨慎地考虑删除这个版本。

您也许想测试是否正确设置了环境变量 `JAVA_HOME`。请执行（在 Windows 中）：

```
"%JAVA_HOME%\bin\java -version
```

您会看到和前面相同的输出。在 Unix 中：

```
"$JAVA_HOME/bin/java" -version
```

引号是为了处理路径名中可能存在空格的情况。如果您没有得到理想的结果，请看一下环境变量 `JAVA_HOME` 的值。在 Unix 中：

```
echo $JAVA_HOME
```

在 Windows 中：

```
echo %JAVA_HOME%
```

如果您不能通过以上三步，请不要继续。到这里寻求帮助：本章结束处的“仍有困难”小节。

JUnit

在开始学习之前您需要下载和安装 JUnit。JUnit 是 TDD 所需要的一个简单的单元测试框架。

多数主流的 IDE 对 JUnit 提供直接或者间接的支持。很多 IDE 产品中带有 JUnit。假如您从命令行或者程序编辑器中编译执行 Java 程序，您需要安装 JUnit。

您可以从 <http://www.junit.org> 免费下载 JUnit。安装很简单，就是将 zip 文件中的内容解压缩到硬盘上。

最重要的是，JUnit zip 文件中含有一个 `junit.jar` 文件。这个文件中含有 JUnit 类库，您需要在代码中引用这些类以编写测试用例。

本书中我用 JUnit 3.8.1 来编写测试用例。

Ant

您还需要下载和安装 Ant。Ant 是一个基于 XML 的编译工具，现在 Ant 已经成了编译和部署 Java 项目的标准。

多数主流的 IDE 对 Ant 提供了直接或间接的支持。很多 IDE 产品中带有 Ant。假如您从命令行或者程序编辑器中编译执行 Java 程序，您需要安装 Ant。

您可以从 <http://ant.apache.org> 下载 Ant。和 JUnit 一样，安装很简单，就是将 zip 文件中的

内容解压缩到硬盘上。

您可以选择推迟安装 Ant，直到您开始学习第 3 课，在第 3 课中第一次用到 Ant，并对 Ant 进行了简要的介绍。

假如您的 IDE 中没有内建 Ant 支持：您需要将 Ant 安装路径中的 bin 子目录加到 path 环境变量中。请参考操作系统手册来获取如何设置 path 的信息。你还需要设置环境变量 ANT_HOME 来指定 Ant 的安装路径。请参考操作系统手册来获取如何设置环境变量的信息。

本书中我用 Ant 1.6 来编译所有的例子。

25

可以正常工作吗

在开始之前，请编写“Hello World”程序。“Hello World”是经典的第一个 Java 程序。执行这个程序，会在终端上打印出文字：“hello world”。很多程序员通过创建“Hello World”程序来判断在新的语言环境中是否可以正确的编译和执行。

Javac 编译器

在 Sun 的 JDK 中，javac 的基本格式如下：

```
javac <options> <source files>
```

javac 提供了很多命令行选项。在命令行输入 javac 再按回车，您可以得到 javac 的选项列表。所有的选项都是可选的。编译时选项放在源文件的前面。下面是一个 javac 的例子：

```
javac -g:none Hello.java
```

在这个例子中，“-g:none”是一个选项，它告诉编译器不要生成调试信息（用来帮助您解决执行问题的有用信息）。

您可能经常用这个命令：

```
javac -version (原文为 java -version，怀疑为作者笔误)
```

没有源文件，用以查看当前 java 编译器的版本。

我把“hello world”放在这儿是为了很快得到反馈。这个例子所蕴涵的很多概念涉及到本书后面会讨论到的高级专题。这个例子用来确保您可以编译和执行 Java 程序。现在开始，把下面的代码输入到您的 IDE 或者编辑器中吧。

```
class Hello {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

用文件名 `Hello.java` 保存上面的代码。大小写对 Java 很重要。假如您保存成 `hello.java`，您将无法编译或者执行您的代码²。

编译 Hello World

您可以从 IDE（假如您的 IDE 要求显式编译³）或者命令行编译：

```
javac Hello.java
```

请从 `javac` 的其他选项获得 `javac` 命令行的更多信息。

您应该在 `Hello.java` 所在的目录。如果一切正常，编译器没有任何输出。如果您输入错了什么，将会收到编译错误。例如，您可能看到：

```
Hello.java:4: ';' expected
    }
    ^
1 error
```

上面的错误意味着您忘了在行尾输入分号。

您也许会看到多个错误。改正所有的错误，然后重新编译。如果还有问题，首先请确保您的代码和上面例子中的完全一样。最后，请参考本章最后的“仍有困难”小节。

成功编译 `Hello.java` 会生成文件 `Hello.class`。执行 `dir` 或者 `ls` 命令来列出当前目录中的所有文件，确保当前目录中存在 `Hello.class`。`Hello.class` 是源代码的二进制版本。Java 虚拟机读入并解释 `Hello.class`，从而执行您在 `Hello.java` 中输入的代码。

执行 Hello World

为了从命令行执行 Hello World 程序，请输入：

```
java Hello
```

请确定如上所示，`Hello` 的第一个字母是大写。

假如输入正确，您将会看到下面的输出：

```
hello world
```

执行 Hello World 可能遇到的问题

- 如果您看到

```
'java' is not recognized as an internal or external command, ...
```

² Windows 对大小写不怎么吹毛求疵。也许您会发现，Windows 并不总是要求大小写敏感。

³ 一些 IDE 要求您自己执行编译。而另一些 IDE，例如 Eclipse，在您每次保存修改后，会自动编译必要的源文件。

或者

```
java: Command not found.
```

那么，您没有安装 Java，或者 Java 的 bin 子目录没有在 path 环境变量中。请参考“检查您的 Java 安装”小节寻求帮助。

- 如果您看到:

```
Exception in thread "main" java.lang.NoClassDefFoundError: hello (wrong name: Hello) ...
```

这是您输入的大小写不正确，例如:

```
java hello
```

请输入正确的名字，重试一下。

- 如果您看到:

```
Exception in thread "main" java.lang.NoClassDefFoundError: Hello/class  
错误是因为您输入了后缀.class (java hello.class)。去掉.class 重试一下。
```

每次编译源代码，Java 编译器会覆盖已有的所有相关的 class 文件。

本书中只在相关的地方(比如，当我讨论 classpath)，我的确在展示如何使用 javac 和 java 命令。我强烈建议您熟悉命令行方式的编译和执行。Java 编译器和虚拟机在不同平台上的工作方式是相同的。然而，如果您用 IDE 编译和执行代码有什么困难，您将不得不去看 IDE 相关的文档。

仍有困难

编译和执行第一个 Java 程序，可能会让你感到沮丧，似乎好多地方都出了问题。不幸的是，我不能在本书中考虑到所有的可能性。如果您还有问题，网上有一些非常好的场所可以给您提供所需要的帮助。Sun 的 Java 官方网站 (<http://java.sun.com>) 提供了非常有价值的信息。

28

另一个可以找到答案的非常好的网址是 <http://www.javaranch.com>。为了使用该网站的服务您需要用真实姓名注册。这个网站的多数斑竹对 Java 新手特别友好，能给您很多帮助。找到“Java in General (beginner)”的论坛，开始发帖吧。

和任何公共论坛一样，发帖前请阅读 JavaRanch 指南。

链接 <http://www.faqs.org/faqs/usenet/primer/part1/> 包含了在新闻组发帖子的礼仪文档。其中的多数礼仪适用于任何公共论坛，包括 JavaRanch。用正确的方式寻求帮助可以给你带来更好的回复。

如果您依然有困难，我很抱歉。请发送邮件给我 agileJava@LangrSoft.com。

29

Contents

目 录

关于作者	xvii
前言	xix
致谢	xxi
引言	xxiii
“敏捷”综述	xxix
搭建环境	xxxix
第 1 章 起步	1
测试	1
设计	2
一个简单的测试	2
JUnit	4
增加一个测试	5
创建 Student 对象	7
创建 Student 类	7
构造函数	8
局部变量	9
从方法返回一个值	10
断言	12
实例变量	14
总结这个测试	16
重构	17
this	19
private	21
命名约定	22

空白区域	23
练习	24
第 2 章 Java 基础	25
课程安排	25
学生注册	27
int	27
初始化	29
默认构造函数	30
测试套件	30
SDK 和 java.util.ArrayList	31
增加对象	33
渐增重构	35
内存中的对象	36
包和 import 语句	37
java.lang 包	38
默认包和 package 语句	38
setUp 方法	40
更多的重构	41
类常量	42
Dates	43
重载构造函数	43
不赞成警告	47
重构	48
用 Calendar 创建日期	50
注释	51
Javadoc 注释	52
练习	54
第 3 章 字符串和包	57
字符和字符串	57
字符串	59
StringBuilder	60
系统属性	62
遍历所有的学生	63
单职责原则	64
重构	67

System.out	69
使用 System.out	71
重构	71
包结构	72
访问修饰符	73
使用 Ant	78
练习	81
第 4 章 类方法和类变量	83
类方法	83
类变量	86
使用类方法操作类变量	88
静态导入	90
增量	92
工厂方法	93
简单设计	95
静态的危险	95
使用静态所需要注意的	96
Jeff 静态规则	97
布尔型	97
测试就是文档	101
关于初始化的更多内容	103
异常	104
再看基本类型的初始化	105
练习	106
第 5 章 接口和多态	109
排序：准备工作	109
排序：Collections.sort	110
CourseReportTest	111
接口	112
为什么需要接口	113
实现 Comparable	114
根据学科和编号进行排序	116
If 语句	116
学生的成绩	118
浮点数	118

测试成绩	119
重构	121
枚举	123
多态	124
使用接口引用	129
ArrayList 和 List 接口	131
练习	131
第 6 章 继承	135
switch 语句	135
Case 标记只是标记	136
Map	138
继承	140
抽象类	143
方法扩展	144
重构	146
增强的枚举 Grade	147
夏季课程安排	148
调用基类的构造函数	149
重构	153
深入构造函数	155
继承和多态	156
子合约原则	157
练习	164
第 7 章 遗留元素	167
循环结构	168
分解学生全名	168
比较 Java 循环	175
重构	176
循环控制语句	177
三元操作符	179
遗留的集合类	180
迭代器	181
迭代器和 for-each 循环	182
类型转换	183
包装类	185

数组	188
重构	195
练习	197
第 8 章 异常和日志	201
异常	202
异常处理	204
检查异常	204
异常层次关系	206
创建自己的异常类型	207
检查异常和非检查异常	209
消息	209
捕获多个异常	211
重新抛出异常	212
堆栈跟踪	214
finally 块	215
重构	216
日志	218
Java 中的日志	219
测试日志	222
将日志定向到文件	225
日志的测试哲学	227
更多关于 FileHandler	228
日志等级	229
日志层次结构	230
日志补充说明	231
练习	232
第 9 章 Map 和相等性	237
逻辑操作符	237
短路	238
哈希表	239
课程	241
重构 Session	242
相等性	248
相等性的定义	250
苹果和橙子	251

集合与相等性	253
哈希表	254
冲突	255
一个理想的哈希算法	256
hashCode 最后一个要点	258
更多关于 HashMap	259
其它哈希表和 Set 实现	263
toString	264
字符串和相等性	266
练习	267
第 10 章 数学	269
BigDecimal	269
更多关于基本数字类型	273
整数运算	273
数字类型转换	274
运算优先级	275
NaN (Not a Number)	276
无穷大	277
数字溢出	278
位操作	279
java.lang.Math	285
数字包装类	287
随机数	288
练习	291
第 11 章 IO (输入/输出)	295
组织	295
字符流	296
写入文件	300
java.io.File	302
字节流与转换	304
学生用户界面	304
测试应用	307
数据流	309
CourseCatalog	309
高级流	312

对象流	312
随机存取文件	318
学生字典	320
sis.db.DataFileTest	321
静态内嵌 (static nested) 类和内联 (inner) 类	323
sis.db.DataFile	324
sis.db.KeyFileTest	327
sis.db.KeyFile	328
sis.util.IOUtilTest	329
sis.util.IOUtil	330
sis.util.TestUtil	331
方案的改进	331
练习	332
第 12 章 反射及其他高级主题	335
再顾 Mock 对象	335
Jim Bob ACH 接口	337
Mock 类	338
Account 类的实现	340
匿名内联类	342
适配器 (Adapter)	344
访问外围类中的变量	346
折衷	348
反射 (Reflection)	348
使用 JUnit 代码	349
Class 类	350
建立测试套件	352
类修饰符	354
动态代理	355
安全帐号类	356
建立安全帐号方案	358
SecureProxy 类	362
反射的问题	364
练习	365
第 13 章 多线程	367
多线程	367

搜索 (Search) 服务器	368
Search 类	369
更少依赖的测试	372
服务器	374
测试中的等待	376
创建并运行线程	377
合作式协作式 (cooperative) 与可抢占的 (preemptive) 多任务	380
同步	381
使用 Runnable 创建线程	383
synchronized	384
同步的集合类	385
BlockingQueue	385
停止线程	386
Wait/Notify	388
wait 和 notify 的补充注意事项	391
锁与条件	392
线程优先级	394
死锁	394
ThreadLocal	395
Timer 类	398
Thread 的杂项	400
总结: 同步的基本设计原则	404
练习	404
第 14 章 范型	405
参数化类型	405
集合框架 (Collection Framework)	406
多类型参数	406
创建参数化类型	407
擦拭法	409
上限 (Upper Bound)	411
通配符 (Wildcard)	413
使用通配符的隐含问题 (Implication)	415
范型方法	416
通配符捕获 (Wildcard Capture)	417
Super	418

附加限界	419
原始类型 (Raw Type)	420
Checked 集合	421
数组 (Array)	423
额外的局限	423
反射	424
最后的注意事项	425
练习	425
第 15 章 断言与注解	427
断言	427
assert 语句 vs. JUnit 的 Assert 方法	428
注解 (Annotation)	429
建立一个测试工具	430
TestRunnerTest	430
TestRunner	432
@TestMethod 注解	434
保留 (Retention)	436
注解的目标 (Annotation Targets)	437
跳过测试方法	438
修改 TestRunner	439
单值 (Single-Value) 注解	440
TestRunner 的用户界面类	442
数组参数	443
多个参数的注解	445
缺省值	446
附加返回类型与复式注解类型	447
包注解	449
兼容性考虑	450
关于注解的额外注意事项	451
总结	451
练习	452
附加课 I Swing, 第一部分	453
Swing	454
起步	454
Swing 应用的设计	458

面板 (Panel)	459
重构 (Refactory)	463
更多的控件	466
重构 (Refactory)	468
按钮点击与 ActionListener	471
列表 Model	473
应用	476
布局	478
继续前进	488
附加课 II Swing, 第二部分	489
界面美化的杂项	490
体验 (feel)	494
表格 (Table)	513
反馈 (Feedback)	518
响应性 (Responsiveness)	523
余下的任务	526
最后的注意事项	527
附加课 III Java 的杂项	529
JAR	529
正则表达式	532
克隆 (Cloning) 与协变 (Covariance)	536
JDBC	538
国际化 (Internationalization)	545
按引用调用 vs. 按值调用	553
Java 的边缘地带	554
还有哪些内容	563
附录 A 敏捷 Java 的术语表	569
附录 B Java 操作符的优先规则	581
附录 C IDEA 入门	583
IDEA	583
Hello 项目	584
运行测试	589
利用 IDEA 的优势	594
Agile Java References	597
索引	599

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

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

本课内容包括：

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

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

测试

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

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



图 1.1 测试类和生产类

设计

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

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

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



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

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

一个简单的测试

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

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

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

保存成文件 `StudentTest.java`。

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

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

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

33

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

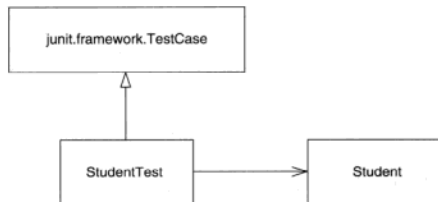


图 1.2 `StudentTest` 继承自 `junit.framework.TestCase`

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

在附加课 III 中, 我会进一步讨论 JAR 文件。在这里, 您只需理解怎样告诉 Java 到某个地方找到包含 JUnit 的 JAR 文件。您可以通过设置 `classpath` 实现这一点。

更多关于 Classpath

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

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

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

34

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

从命令行, 您可以在编译源文件的同时, 指定 classpath。

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

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

您需要保证和 StudentTest.java 在同一个目录。

假如您忽略了 classpath, Java 编译器会报告下面的出错信息:

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

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

JUnit

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

35

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

您再次指定了 classpath, 这次用的是关键字缩写 -cp。不止 Java 编译器需要知道 JUnit 类在什么地方, Java 虚拟机也需要找到这些类, 这样 Java 虚拟机就可以在运行时随需加载。此外, classpath 含有一个用以表示当前目录的 "."。这样 Java⁴就能够定位 StudentTest.class: 如果指定一个目录而不是 JAR 文件, Java 会扫描这个目录并查找必要的 class 文件。

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

执行 TestRunner, 您会看到如图 1.3 的窗口:

² 绝对路径是文件的完整路径, 以驱动器名称或者文件系统的根作为开始。相对路径是某一文件相对当前位置的路径。例如, 假如 StudentTest 位于 /usr/src/student, JUnit.jar 在 /usr/src/JUnit3.8.1, 您可以把相对路径指定为 ./JUnit3.8.1/JUnit.jar。

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

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

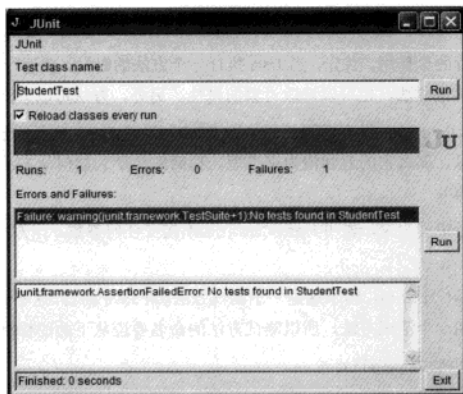


图 1.3 JUnit 执行出错（显示了一根红条）

36

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

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

增加一个测试

编辑 `StudentTest` 类，像下面这样：

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

新增的第二行和第三行在 `StudentTest` 类中定义了一个方法：

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

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

⁵ 如果您是色盲，红条下面的统计为您提供所需的信息。

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

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

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

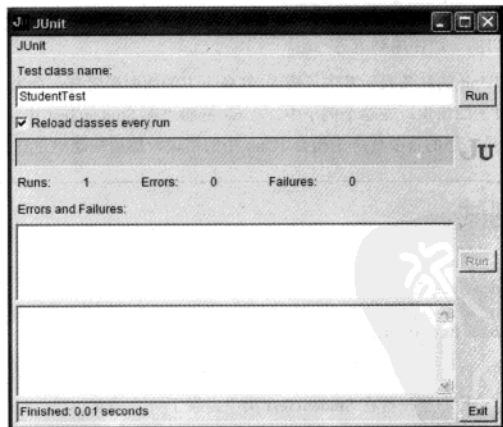
37

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

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

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

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



38

图 1.4 JUnit 执行成功（显示一根绿条）

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

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

创建 Student 对象

在 `testCreate` 方法中增加一行代码：

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

每行代码以分号结束。

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

`testCreate` 方法中新增的这行语句告诉 Java 去创建 `Student` 类的对象。
`new Student("Jane Doe");`

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

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

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

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

◀ 39

创建 Student 类

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

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

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

⁶ 您可能看到不同的错误，这取决于您的 Java 编译器或者 IDE。

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

简化编译和执行

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

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

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

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

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

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

```
class Student {
}
```

再次运行 javac，这次我们使用通配符来编译所有的源文件：

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

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

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

构造函数

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

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

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

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

41

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

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

再次编译，并且运行您的 JUnit 测试：

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

您会看到绿条。

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

局部变量

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

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

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

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

42

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

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

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

图 1.5⁷ 展示了 `student` 引用和 `Student` 对象。该图只是帮助您理解幕后发生了什么，您不必了解如何创建这样的示意图。

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

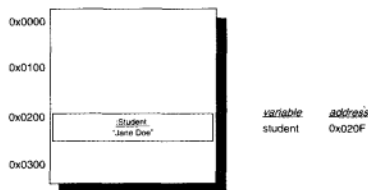


图 1.5 对象的引用

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

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

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

从方法返回一个值

下一步，您向测试中创建的 `Student` 对象请求学生的姓名。

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

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

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

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

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

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

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

您也必须指定消息的接收者——您打算向其发送消息的对象。为了达到目的，首先指定对象引用 student，后面跟上一个句点 (.)，再后面是消息 getName()。圆括号表明没有参数 ◀ 44

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

马上您就会看到如何编写 getName()。

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

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

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

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

    String getName() {
    }
}
```

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

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

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

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

    String getName() {
```



```

        return "";
    }
}

```

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

断言

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

```

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

```

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

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

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

```

public class StudentTest extends junit.framework.TestCase {

```

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

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

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

么测试通过。

重新编译并运行测试。JUnit 看起来会像图 1.6。

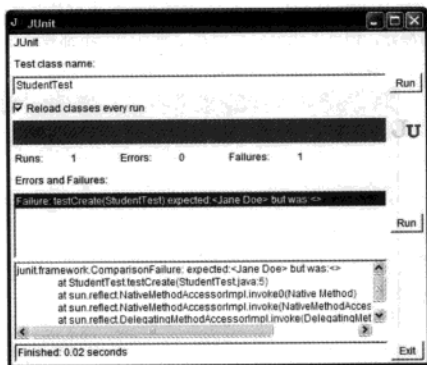


图 1.6 测试尚未全部完成

47

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

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

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

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

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

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

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

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

    String getName() {
```

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

```

        return "Jane Doe";
    }
}

```

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

48

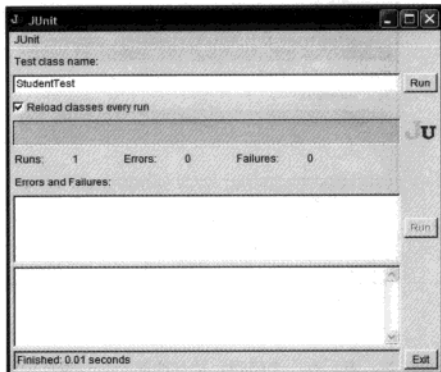


图 1.7 运行成功

实例变量

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

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

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

49

```

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

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

```

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

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

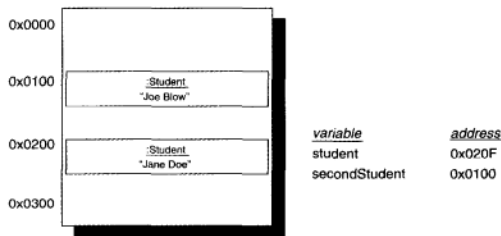


图 1.8 对象在内存中

本次测试失败:

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

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

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

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

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

```
Class student {
    String myName;

    Student(String name) {
    }

    String getName() {
```

⁹ 多数系统中，在命令的后面加上 `&`。

¹⁰ 在命令的前面加上 `start`，例如：`start java -cp .;c:\junit3.8.1\junit.jar junit.awtui.TestRunner StudentTest`。

¹¹ 如果能生效，请保证选中 JUnit 界面上的复选框 “Reload classes every run”。

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

在构造函数中，把参数 `name` 赋值给 `myName`。

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

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

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

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

51

再看一下这个测试方法：

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

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

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

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

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

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

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

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

总结这个测试

让我们逐行总结这个测试可预期的动作：

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

请求 student 的姓名，保存到局部变量

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

验证 student 的姓名是 "Jane Doe"

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

◀ 52

重构

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



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

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

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

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

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

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

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

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

◀ 53

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

代码看起来像这样：

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

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

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

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

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

消除此类冗余的方法（代码中增加一点表现力）是用字符串常量来替代一个字符串。

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

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

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

在下面您已经定义了常量来替代字符串：

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

54

编译并重新运行测试，以确保没有因不小心而破坏了什么。

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

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

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

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

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

您的开发循环是：

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

这样的循环，会很快成为一种根深蒂固、自然的开发流程。

this

审视 `Student` 类的代码，看看是否有可以改进的地方。

```
class Student {
    String myName;

    Student(String name) {
        myName = name;
    }

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

55

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

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

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

简单的用 `name` 命名如何？

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

```
class Student {
    String name;

    Student(String name) {
```



```

        name = name;
    }

    String getName() {
        return name;
    }
}

```

编译会通过（也许您会看到一条警告）。然而，运行测试，却会失败：

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

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

```
name = name;
```

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

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

56

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

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

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

```

class Student {
    String name;

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

    String getName() {
        return name;
    }
}

```

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

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

private

Java 允许访问对象的成员变量，就像您可以调用一个方法：

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

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

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

57

运行这个测试，会通过。但是，它展现了一种特别不好的面向对象编码风格。



不要把成员变量直接暴露给其他对象。

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

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

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

对 `StudentTest` 做出上面的修改。运行测试，测试显示失败。

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

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

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

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

您将得到一个编译报错：

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

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

```
1 error
```

删掉有问题的代码，重新编译和测试。

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

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

命名约定

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

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

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

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

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

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

¹² 想象一下：字符串展现了从侧面所观察到的骆驼的形象，大写字母表示驼峰。关于这个术语有一个有趣的讨论，请看 <http://c2.com/cgi/wiki?CamelCase>。您也许还听说过其他的名字，比如混合大小写。

¹³ 驼峰模式有时也被称为小写驼峰模式，以此来和大写驼峰模式有所区分。

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

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

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

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

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

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

60

空白区域

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

```
class Student
```

下面的空格是允许的（但是要避免）：

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

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

¹⁴ 单职责原则[Martin2003]。

¹⁵ 我打赌您会错误地以为是 `customer` 和 `number`。看看这多么容易让人误入歧途。

¹⁶ [Langr2000]。

¹⁷ [Vermeulen2000]。

为了有利于将来可以轻松的理解代码，您应该坚持这样。

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

61

练习

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

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

62

本课内容包括：

- 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 DeVughn");
        session.enroll(student2);
        assertEquals(2, session.getNumberOfStudents());
    }
}
```

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。

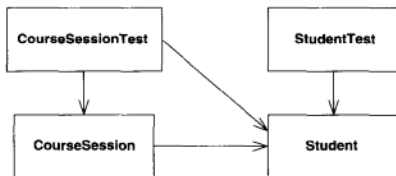


图 2.1 `CourseSession` 和 `Student` 类图

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() {
    }
    ...
}
```

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。

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

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。

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 `CourseSessionTest` 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。

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。

70

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，

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。

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`，`junit.framework.TestCase` 继承自 `java.lang.Object`。

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

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。

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。

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;
    }
}
```

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。

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)。

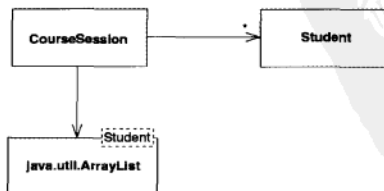


图 2.3 包含参数类型的类图

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。

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遍。参数化类型声明使得 `CourseSession` 与存储多个 `Student` 对象的 `ArrayList` 是一一对一的关系。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：

```
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。

```

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 DeV Vaughn");
    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。

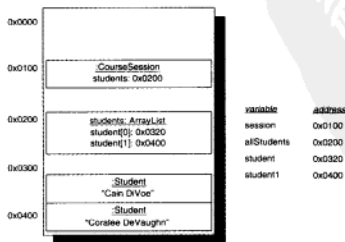


图 2.4 内存图示

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();
        ...
    }
}
```

（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。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`。如果对这个步骤不确信的话，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`、`CourseSessionTest.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

现在编辑五个 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`。

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做法。

我没有选择这样做，但是推荐您试一试，看看它怎样帮助您。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益于软件的成长。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得的。本课将学到如何为您的类创建多个构造函数。

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。

86

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），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 `getter/setters`。Sun 用这种最清洁的方法消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做的事情。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 =
        new CourseSession("ABCD", "200", startDate);
    year = 103;
    month = 3;
    date = 25;
    Date sixteenWeeksOut = new Date(year, month, date);
    assertEquals(sixteenWeeksOut, session.getEndDate());
}
```

87

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() {
        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 `Calendar` 紧密相关。第一个参数中的成员变量告诉 `Calendar` 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。

现在 `calendar` 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;

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

    public void setUp() {
        int year = 103;

```

⁶ 不要将其当作好的方法命名的例子。

```

        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());
    }
    ...

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

```

90

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：再次输入编译命令，并加上编译选项 `-Xlint:deprecation`。

⁷ 写这些文字，仿佛牙齿蛀烂了，坐在这儿。

```
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
```

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 `endDate`。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;
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速失去控制。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。

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。

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() {
```

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”。

```
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);
    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.*;
```

⁹ [Wiki2004].

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。

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關鍵字@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
```

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。

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，我感到困惑。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

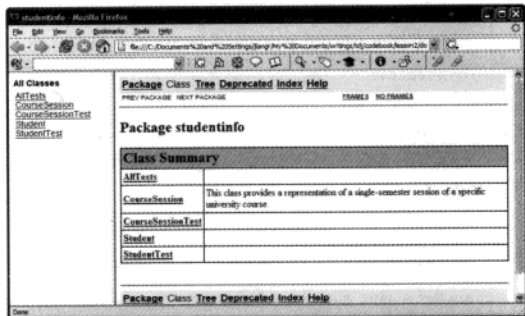


图 2.5 您的 API 页面

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。

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。



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

尽管本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：

```
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 X3.4¹ 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

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，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	'\"'

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].

字符串连接

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 连接操作作用下面的方式：

```
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。

107

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
```

108

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 13 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);
}
```

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) {
    // ... statements here ...
}
```

³ 相对过去的循环，功能有所增强。

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7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 来生成报名表。

```
package studentinfo;
```

⁴ [Martin2003]

```

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。

```
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，所以您不得不再将其再次添加到 `CourseSession`。

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

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 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。

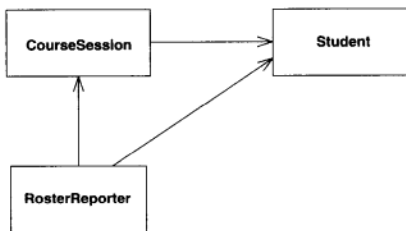


图 3.1 类图

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

◀ 117

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。

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);
    }
}
```

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。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。

119



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，会随着需求的变化而改变。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，来满足潜在的不断加重的重用，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.studentinfo`。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`、`Student` 在不同的包中。它们不再能够正确访问其它包中的类。

121

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：

```
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。

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;
    }

    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

```

124

PDF
PDG


```

*/
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();
        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。

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。

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的和正确的。

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。
- 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" />
```

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。

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 true，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(^)。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。
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)，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>。

131

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)：

⁷ 另一个可选任务是 junit，该任务将执行基于文本的 JUnit。

⁸ [Hatcher2002]。

```
.....  
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。
10. 创建 Ant 脚本，编译整个项目。然后，用命令运行所有的测试。



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();
    }
}
```

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

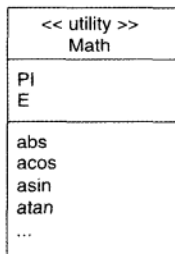


图 4.1 工具类 `Math`

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

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）。

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 声明某个类中的所有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 run²”。如果运行该测试两次 (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。

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

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。

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() {
    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

140

有趣的问题。



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加容易混淆。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 =
            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);
    }
}
```

³ 您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。

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);
}
```

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构造函数”⁴。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，就应该把代码移过去。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，随着需求的变化，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：

```

package sis.studentinfo;

public class Student {
    private static String name;

    public Student(String name) {

```

147

⁶ [Wiki2004b].

```

        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。

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。

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，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，也许你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 DeV Vaughn");
    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;
}
}
```

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

154

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。

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，您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，测试会变得更具有表现力。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 `true`，`"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 10 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(
            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?



本课的内容包括：

- 排序
- 接口
- Comparable 接口
- if 语句
- enum 枚举类型
- 多态

排序：准备工作



学校需要关于课程安排的报表。报表必须先根据学科排序，然后根据课程编号排序。学科相同的课程放在同一组。并且学科分组按照字母升序排列。同一个分组内，课程按照编号排序。

用一个简单的报表类来处理所有的课程安排。暂时不用担心排序问题。

```
package sis.report;

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

public class CourseReportTest extends TestCase {
    public void testReport() {
        final Date date = new Date();
        CourseReport report = new CourseReport();
        report.add(CourseSession.create("ENGL", "101", date));
        report.add(CourseSession.create("CZEC", "200", date));
        report.add(CourseSession.create("ITAL", "410", date));

        assertEquals(
```

```

        "ENGL 101" + NEWLINE +
        "CZEC 200" + NEWLINE +
        "ITAL 410" + NEWLINE,
        report.text());
    }
}

```

报表只是简单地在每一行列出学科和课程编号。当前报表按照添加到 `CourseReport` 对象的先后顺序对课程进行排序。

生产类 `CourseReport`，看起来和 `RosterReporter` 类似：

```

package sis.report;

import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

public class CourseReport {
    private ArrayList<CourseSession> sessions =
        new ArrayList<CourseSession>();

    public void add(CourseSession session) {
        sessions.add(session);
    }

    public String text() {
        StringBuilder builder = new StringBuilder();
        for (CourseSession session: sessions)
            builder.append(
                session.getDepartment() + " " +
                session.getNumber() + NEWLINE);
        return builder.toString();
    }
}

```

为了让 `CourseReport` 编译通过，需要将 `CourseSession` 的 `getDepartment` 与 `getNumber` 方法设置为 `public`。

排序：Collections.sort

164

可以对包含 `String` 对象的列表进行简单的排序，下面的语言测试使用了这个功能：

```

public void testSortStringsInPlace() {
    ArrayList<String> list = new ArrayList<String>();
    list.add("Heller");
    list.add("Kafka");
    list.add("Camus");
    list.add("Boyle");
    java.util.Collections.sort(list);
    assertEquals("Boyle", list.get(0));
    assertEquals("Camus", list.get(1));
}

```

```

    assertEquals("Heller", list.get(2));
    assertEquals("Kafka", list.get(3));
}

```

类 `java.util.Collections` 的静态方法 `sort` 以列表作为参数，并且对列表进行排序¹。如果您不想在当前列表进行排序——不打算修改当前列表——您可以创建一个新列表，并将这个列表作为 `sort` 的参数。

```

public void testSortStringsInNewList() {
    ArrayList<String> list = new ArrayList<String>();
    list.add("Heller");
    list.add("Kafka");
    list.add("Camus");
    list.add("Boyle");
    ArrayList<String> sortedList = new ArrayList<String>(list);
    java.util.Collections.sort(sortedList);
    assertEquals("Boyle", sortedList.get(0));
    assertEquals("Camus", sortedList.get(1));
    assertEquals("Heller", sortedList.get(2));
    assertEquals("Kafka", sortedList.get(3));

    assertEquals("Heller", list.get(0));
    assertEquals("Kafka", list.get(1));
    assertEquals("Camus", list.get(2));
    assertEquals("Boyle", list.get(3));
}

```

最后四个断言验证原始列表没有被修改。

CourseReportTest

在 `CourseReportTest` 的 `testReport` 中，修改断言以确保排序正确。当前的测试数据只考虑学科：

165

```

assertEquals(
    "CZEC 200" + NEWLINE +
    "ENGL 101" + NEWLINE +
    "ITAL 410" + NEWLINE,
    report.text());

```

测试将失败。您应该可以从 JUnit 中看到课程的排序不正确。

为了解决这个问题，使用 `Collections.sort` 对列表 `sessions` 进行排序。

```

public String text() {
    Collections.sort(sessions);
}

```

¹ 方法 `sort` 使用合并排序算法：将列表中的元素分成两组，分别对两组元素进行排序，然后将排序后的两个分组合并成一个完整的列表。

```

StringBuilder builder = new StringBuilder();
for (CourseSession session: sessions)
    builder.append(
        session.getDepartment() + " " +
        session.getNumber() + NEWLINE);
return builder.toString();
}

```

不幸的是，编译无法通过：

```

cannot find symbol
symbol : method sort(java.util.List<sis.studentinfo.CourseSession>)
location: class java.util.Collections
    Collections.sort(sessions);
                ^

```

即使对 Java 有相关的经验，也很难找到这个错误的根源。问题在于：sort 在声明时，要求排序对象必须是 java.lang.Comparable 类型。（第 14 课关于范型的讨论中，会介绍相关的高级语法。）

Comparable 是一种允许对象之间进行比较的类型。但是您打算对 CourseSession 对象进行比较。利用 Java 接口可以解决这个问题。接口允许某个对象拥有多种类型的行为。修改 CourseSession，使其既是 CourseSession 类型，同时也具有 java.lang.Comparable 类型的行为。

接口

Comparable 是一个接口，而不是类。接口包含若干个方法的定义。方法定义由方法名字和随后的分号构成。没有成对的大括号，也没有方法的实现代码。Java 的 Comparable 类型定义如下：

```

public interface Comparable<T> {
    public int compareTo(T o);
}

```

字符“T”是待比较类型的占位符。第 14 课有关范型的内容会进一步解释这个概念。目前，将其理解为一个可比较的引用。Java 编译器用实际类型来代替接口定义中所有的 T。

将接口定义和类定义进行对比。区别在于：使用 interface 关键字声明接口，使用 class 关键字声明类。接口不包含任何方法的实现。

使用 UML，可以用多种方式来表现接口。图 5.1 显示了一种方式。

方法 compareTo 的作用是：根据排序规则，返回一个值，通过值来判断某个对象应该排在另一个对象的前面还是后面。所有排序技术都包含这样的方法：每次只比较两个对象，必要的时候交换这两个对象。

例如，某个列表包含两个 `String` 对象：“E”和“D”。您希望对这个列表按照字母顺序进行排序。`Collections` 的方法发送消息 `compareTo` 到 `String` “E”，消息的参数是“D”。方法 `compareTo` 将返回一个值，表明“D”应该排在“E”的前面。然后方法 `sort` 根据这个返回值，在列表中交换两个字符串对象的位置。

可以定义实现某个接口的类。通过定义，类承诺实现接口中声明的所有方法。通过实现接口，从而允许类扮演一个以上的类型。

前面的例子，对包含字符串对象的集合进行了排序。查阅 J2SE API 文档，发现类 `String` 实现了三个接口，`Comparable` 接口是其中的一个。实现 `Comparable` 接口，使得 `String` 对象不仅被看作 `String` 类型的对象，而且被看作 `Comparable` 类型的对象。

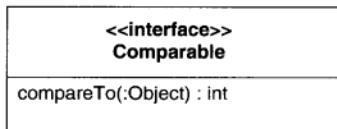


图 5.1 Comparable

从实际的 Java `String` 类的代码中，我们可以看到接口实现的语法：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    ...
}
```

如果进一步阅读类 `String` 的源代码，您可以找到 `Comparable` 所定义的 `compareTo` 方法的实现。

为了让 `CourseReport` 调用的排序可以正常工作，必须把消息 `compareTo` 发送给 `sessions` 中所包含的对象。不过，方法 `sort` 并不知道集合中所包含对象的类型。方法 `sort` 轮流取出集合中的每一个对象，并且将对象赋值给一个 `Comparable` 接口类型的变量。如果赋值成功，那么方法 `sort` 可以成功地将消息 `sort` 发送给这个对象。如果赋值不成功，Java 会报错。

为什么需要接口

接口是 Java 提供的非常强大和重要的功能。使用接口是实现可靠设计的关键之一。正确地使用接口可以帮助您划分软件，最小化不同代码之间的相互影响。

接口提供了更高的抽象。排序代码不需要知道排序对象的细节，不需要知道参与排序的是 `Student` 对象，还是 `Customer` 对象或者字符串对象。排序代码需要知道对象是否支持排序。排序代码只关注抽象的质量，不关心任何排序对象的细节。

可以将抽象排序概念视为消除重复的一种方式。接口允许某个算法操作不同类型的对象。排序算法没有用来判断对象类型的重复代码。



系统中，使用接口提供抽象层，从而有助于消除重复。

系统中的抽象层可以隔离代码，避免受到负面的影响。理想情况下，您一次性完成 `sort` 的编码，并且使其正常工作，之后就不再做任何改动。`sort` 必须知道的关于排序对象的唯一细节是：`compareTo` 方法返回一个整型值，`sort` 根据这个整型值进行处理。排序对象可以任意改变，但是任何改变都不会影响到排序代码。

168

通过编写符合接口规范的桩代码，您可以利用接口消除代码之间的依赖关系，这些代码可以不工作甚至不存在。在第12课，您将学习如何通过模拟技术来实现这个目标。接口是实现有效测试的基本工具。

通过实现 `Comparable` 接口，类 `String` 宣布可以支持 `sort` 排序。类 `String` 实现了 `compareTo` 方法，该方法决定了如何比较字符串对象。

为了对课程列表进行排序，您必须修改类 `CourseSession`，实现 `Comparable` 接口。

实现 Comparable

在 `Comparable` 接口中，方法 `compareTo` 的任务是指出两个对象中的哪一个应该排在前面——消息的接收者和消息的参数。您实现的 `compareTo` 方法将提供 `CourseSession` 的默认排序规则。换句话说，您应该如何对一个包含 `CourseSession` 对象的集合进行排序。

方法 `compareTo` 必须返回 `int` 类型的值。如果返回值是 0，那么两个对象相同。如果返回值为负数，那么接收者（消息 `compareTo` 的发送对象）应该排在参数的前面。如果返回值为正数，那么参数应该在接收者的前面。

为了按照字母顺序对字符串进行排序，类 `String` 实现了 `compareTo` 方法。下面的语言测试演示了：在三种可能的场景下，方法 `compareTo` 的返回值。

```
public void testStringCompareTo() {
    assertTrue("A".compareTo("B") < 0);
    assertEquals(0, "A".compareTo("A"));
    assertTrue("B".compareTo("A") > 0);
}
```

修改 `CourseSession`，实现 `Comparable` 接口，并且提供 `compareTo` 的定义。然后在 `CourseSessionTest` 中增加相应的测试：

```
public void testComparable() {
    final Date date = new Date();
    CourseSession sessionA = CourseSession.create("CMSC", "101", date);
    CourseSession sessionB = CourseSession.create("ENGL", "101", date);
}
```

169

```

assertTrue(sessionA.compareTo(sessionB) < 0);
assertTrue(sessionB.compareTo(sessionA) > 0);

CourseSession sessionC = CourseSession.create("CMSC", "101", date);
assertEquals(0, sessionA.compareTo(sessionC));
}

```

类 `CourseSession` 必须声明对 `Comparable` 接口的实现。同时，也必须指定 `CourseSession` 绑定类型——`CourseSession` 对象之间可以进行比较。

```

public class CourseSession implements Comparable<CourseSession> {
...

```

更多关于 this

在 `compareTo` 方法中，`return` 语句包含了表达式 `this.getDepartment()`。在第 1 课中学到过，`this` 是当前对象的引用。方法调用时，`this` 通常是不必要的，但是可以用 `this` 来区别当前对象和参数对象。

关键字 `this` 的另一个用法是构造函数链。您可以使用关键字 `this`，调用同一个类中的另一个构造函数。

```

class Name {
...
    public Name(String first, String mid, String last) {
        this.first = first;
        this.mid = mid;
        this.last = last;
    }

    public Name(String first, String last) {
        this(first, "", last);
    }
...

```

第二个构造函数中的单行代码调用第一个构造函数。这个例子中，我们将空字符串作为 `middle name` 的默认值。

调用其它构造函数的代码必须是某个构造函数的第一行代码。

构造函数链是一个帮助消除重复的有用的工具。如果没有构造函数链，那么您需要单独编写一个方法来实现公共初始化。

由于将 `Comparable` 的实现与 `CourseSession` 绑定在一起，所以您必须定义方法 `compareTo`，并接受 `CourseSession` 作为参数：

```

public int compareTo(CourseSession that) {
    return this.getDepartment().compareTo(that.getDepartment());
}

```


在方法 `compareTo` 的实现代码中，比较当前 `CourseSession` 的学科 (`this.getDepartment()`) 和参数 `CourseSession` 对象的学科 (`that.getDepartment()`)，并将比较的结果作为返回值。

此时，`CourseSessionTest` 的 `testComparable`，以及 `CourseReportTest` 的 `testReport`，都会通过。

根据学科和编号进行排序

您可以得到根据学科进行排序的课程报表。现在，您将增强报表的功能，从而可以根据学科以及编号进行课程排序。首先向测试中加入合适的测试数据。下面的测试（在 `CourseSessionTest`）中增加了两门课程。测试数据表明这两门课程具有相同的学科名称。

```
package sis.report;

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

public class CourseReportTest extends TestCase {
    public void testReport() {
        final Date date = new Date();
        CourseReport report = new CourseReport();
        report.add(CourseSession.create("ENGL", "101", date));
        report.add(CourseSession.create("CZEC", "200", date));
        report.add(CourseSession.create("ITAL", "410", date));
        report.add(CourseSession.create("CZEC", "220", date));
        report.add(CourseSession.create("ITAL", "330", date));

        assertEquals(
            "CZEC 200" + NEWLINE +
            "CZEC 220" + NEWLINE +
            "ENGL 101" + NEWLINE +
            "ITAL 330" + NEWLINE +
            "ITAL 410" + NEWLINE,
            report.text());
    }
}
```

为了对学生进行正确的排序，方法 `compareTo` 首先比较学科。如果学科不相同，那么对学科进行比较，然后提供返回值。如果学科相同，那么比较课程编号。

If 语句

使用 `if` 语句进行条件判断：如果条件为 `true`，执行一个分支，否则执行另外一个分支。分

支可以是单行代码或者是一个代码块。

增加进行更复杂比较的测试，来充实 `CourseSessionTest` 的方法 `testCompareTo`:

```
public void testComparable() {
    final Date date = new Date();
    CourseSession sessionA = CourseSession.create("CMSC", "101", date);
    CourseSession sessionB = CourseSession.create("ENGL", "101", date);
    assertTrue(sessionA.compareTo(sessionB) < 0);
    assertTrue(sessionB.compareTo(sessionA) > 0);

    CourseSession sessionC = CourseSession.create("CMSC", "101", date);
    assertEquals(0, sessionA.compareTo(sessionC));

    CourseSession sessionD = CourseSession.create("CMSC", "210", date);
    assertTrue(sessionC.compareTo(sessionD) < 0);
    assertTrue(sessionD.compareTo(sessionC) > 0);
}
```

然后，更新 `CourseSession` 中的 `compareTo` 方法:

```
public int compareTo(CourseSession that) {
    int compare =
        this.getDepartment().compareTo(that.getDepartment());
    if (compare == 0)
        compare = this.getNumber().compareTo(that.getNumber());
    return compare;
}
```

上面代码的流程是这样的：对当前对象的学科和参数对象的学科进行比较，将比较结果存储在局部变量 `compare` 中。如果 `compare` 的值为 0（学科名字相同），那么对当前对象的课程编号和参数对象的课程编号进行比较，并且将比较结果赋值给变量 `compare`。最后，返回局部变量 `compare` 中存储的值。

172

您也可能这样编写代码:

```
public int compareTo(CourseSession that) {
    int compare =
        this.getDepartment().compareTo(that.getDepartment());
    if (compare != 0)
        return compare;
    return this.getNumber().compareTo(that.getNumber());
}
```

如果第一次比较时，学科名称不相同（比较的结果不为 0），那么直接返回比较的结果，后面的代码不会执行。这种编码风格比较容易理解。但是要注意：在更长的方法中，多个返回语句使得方法更难被理解。所以，对于比较长的方法，我不推荐多个 `return` 语句。但是，真正的方案应尽可能不要让代码过长。

所有的测试都会通过。

学生的成绩



需要为所有学生提供成绩单。给定某个学生，能够计算出该学生的 GPA（平均成绩）。学生会收到若干个成绩。发送消息 `addGrade` 给 `Student` 对象，以成绩作为参数，从而存储学生的成绩。如果希望获得平均成绩，利用 `Student` 对象存储的成绩计算出平均成绩。

用十进制浮点数表示成绩。

浮点数

您已经熟悉了表示整数的基本类型 `int`。除了整数，Java 也支持 IEEE 754 的 32 位单精度浮点数。而且，Java 也支持 64 位的双精度浮点数。

表 5.1 列出了类型，类型所支持的数值范围，以及对应的例子。

表 5.1 浮点数类型

类型	范围	例子
<code>float</code>	<code>1.40239846e -45f to 3.40282347e + 38f</code>	<code>6.07f 6F 7.0f 7.0f 1.8e5f</code>
<code>double</code>	<code>4.94065645841246544e -324 to 1.79769313486231570e +308</code>	<code>1440.0 6 3.2545e7 32D 0d</code>

任何有小数点的数字，以及以 `f`、`F`、`d` 或者 `D` 作为后缀的数字，都是浮点数。

`float` 和 `double` 类型也可以使用科学计数法。任何中间有 `e` 或者 `E` 的数字也是浮点数。双精度数字 `1.2e6` 表示 1.2 乘以 10 的 6 次方，或者表示成科学计数法 `1.2×106`。

如果浮点数不加后缀，那么默认被认为是双精度浮点数。换句话说，您必须加上后缀 `F` 或者 `f` 来表示单精度浮点数。在本书中，我推荐最好使用双精度浮点数 `double`，因为它提供了更高的精度，而且不需要后缀。

浮点数是真实数字的基于位模式的近似值。因为真实数字是一个无限的集合，而浮点数只是一个提供 32 位或者 64 位精度的有限集合，所以浮点数不可能表示所有的真实数字。这意味着多数的真实数字只有一个近似的浮点表达形式。

例如，如果您执行下面的代码：

```
System.out.println("value = " + (3 * 0.3));
```

输出是：

```
value = 0.8999999999999999
```

输出不是预期的 0.9。

如果使用浮点数，您必须利用舍入技术来处理此类问题。另一种方案是 Java 类 `BigDecimal`，用小数部分来表示数字。第 10 课中讲述了 `BigDecimal`。

174

测试成绩

将下面的测试添加到 `StudentTest`，该测试枚举了所有的可能性。包括：学生没有收到成绩，这种情况下，学生的 GPA 为 0；另外，对从 A 到 F 所有可能的成绩进行测试。

```
private static final double GRADE_TOLERANCE = 0.05;
...
public void testCalculateGpa() {
    Student student = new Student("a");
    assertEquals(0.0, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("A");
    assertEquals(4.0, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("B");
    assertEquals(3.5, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("C");
    assertEquals(3.0, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("D");
    assertEquals(2.5, student.getGpa(), GRADE_TOLERANCE);
    student.addGrade("F");
    assertEquals(2.0, student.getGpa(), GRADE_TOLERANCE);
}
```

方法 `assertEquals` 有三个参数，而不是两个。由于浮点数并不精确等于真实数字，所以计算出来的值与期望值可能有一定的误差。例如，对于下面的代码：

```
assertEquals(0.9, 3 * 0.3);
```

测试会失败：

```
AssertionFailedError: expected:<0.9> but was:<0.8999999999999999>
```

如果需要比较浮点数，JUnit 提供了第三个参数。该参数表示可以容忍的误差范围：两个浮点数相差超过多少，JUnit 会认为测试失败。

规则是：误差范围为最小精度的二分之一。例如，如果最小精度是分(百分之一元)，那么您必须保证误差不超过半分。

您应该使平均成绩 GPA 精确到十分之一，所以误差范围是 5/100。类 `StudentTest` 中定义了一个静态常量 `GRADE_TOLERANCE`，使用这个常量作为 `assertEquals` 的第三个参数：

175

```
assertEquals(2.0, student.getGpa(), GRADE_TOLERANCE);
```

为了使测试得以通过，您必须修改类 `Student`，将所有成绩添加到一个 `ArrayList` 中²。然后编写方法 `getGpa`，来计算平均成绩。首先，逐个读取列表中的每一项成绩，得到总成绩。然后，用总成绩除以列表中元素个数，从而得到平均成绩 GPA。

```
import java.util.*;

...
class Student {
    private ArrayList<String> grades = new ArrayList<String>();
    ...
    void addGrade(String grade) {
        grades.add(grade);
    }
    ...
    double getGpa() {
        if (grades.isEmpty())
            return 0.0;
        double total = 0.0;
        for (String grade: grades) {
            if (grade.equals("A")) {
                total += 4;
            }
            else {
                if (grade.equals("B")) {
                    total += 3;
                }
                else {
                    if (grade.equals("C")) {
                        total += 2;
                    }
                    else {
                        if (grade.equals("D")) {
                            total += 1;
                        }
                    }
                }
            }
        }
        return total / grades.size();
    }
}
```

176

首先，方法 `getGpa` 用 `if` 语句来判断成绩列表是否为空。如果没有成绩，那么立刻返回平均成绩 0.0。出现在方法起始处，并且根据某个特定条件判断是否返回的 `if` 语句，被称之为“防卫语句”。方法 `getGpa` 的防卫语句过滤了没有成绩的情况，从而对方法中其余的代码进行防护。既然您通过除以成绩列表的元素个数来得到平均成绩 GPA，防卫语句消除了 0 作为除数的可能性。

该方法使用 `for-each` 循环从集合 `grades` 中取出每一门成绩。循环体对成绩和可能的字

² 当添加成绩后，您也需要计算平均成绩 GPA。

母成绩进行比较。循环体使用了 if 语句的扩展形式：if-else 语句。

接着是对 for-each 循环体的解释：如果成绩是“A”，那么总成绩增加 4³。否则，执行 else 后面的代码块。在这个例子中，else 后面的代码块包含另一个 if-else 语句：如果成绩为“B”，那么总成绩增加 3，否则执行另一个代码块，直到穷举所有的可能性（F 被忽略，如果是“F”，那么不增加总成绩）。

因为有很多大括号以及缩进，所以复杂的 if-else 语句不容易阅读。在重复内嵌 if-else 语句的情况下，您可以使用紧密格式。每一个 else-if 放在同一行。而且，由于 if 语句都是单行形式，所以这个例子中可以去掉大括号。

```
double getGpa() {
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (String grade: grades) {
        if (grade.equals("A"))
            total += 4;
        else if (grade.equals("B"))
            total += 3;
        else if (grade.equals("C"))
            total += 2;
        else if (grade.equals("D"))
            total += 1;
    }
    return total / grades.size();
}
```

这种风格使得方法 getGpa 更容易阅读，而且结果是一样的。

◀ 177

重构

方法 getGpa 还是有点长，而且难以阅读。从中抽取部分代码，创建一个新方法，该方法根据字母成绩返回一个双精度数值：

```
double getGpa() {
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (String grade: grades)
        total += gradePointsFor(grade);
    return total / grades.size();
}

int gradePointsFor(String grade) {
```

³ 将 int 与 double 类型的数值相加，没有什么不利的影响。但是，如果在表达式中混合 int 和 double 类型的数值，会引起意想不到的问题。第10课有关于 Java 数值的讨论。

```

    if (grade.equals("A"))
        return 4;
    else if (grade.equals("B"))
        return 3;
    else if (grade.equals("C"))
        return 2;
    else if (grade.equals("D"))
        return 1;
    return 0;
}

```

确保编译和测试都能通过。另一个改动是去掉 else 子句,用 return 语句来实现流程控制。一旦 Java 虚拟机在方法中遇到 return 语句,就不再执行方法中其余的代码。

```

int gradePointsFor(String grade) {
    if (grade.equals("A")) return 4;
    if (grade.equals("B")) return 3;
    if (grade.equals("C")) return 2;
    if (grade.equals("D")) return 1;
    return 0;
}

```

在方法 gradePointsFor 中,只要匹配到某一个成绩,虚拟机就不再继续比较,马上执行 return 语句。

您也注意到:我将 return 语句和对应的 if 子句放在同一行。通常,条件和 if 语句的主体应该放在不同的代码行。这个例子中,多个 if 子句是重复的、简洁的、而且视觉上是一致的。代码很容易阅读和理解。在其它情况下,将条件和主体放在同一行会导致阅读困难。

178

对测试进行重构:

```

public void testCalculateGpa() {
    Student student = new Student("a");
    assertGpa(student, 0.0);
    student.addGrade("A");
    assertGpa(student, 4.0);
    student.addGrade("B");
    assertGpa(student, 3.5);
    student.addGrade("C");
    assertGpa(student, 3.0);
    student.addGrade("D");
    assertGpa(student, 2.5);
    student.addGrade("F");
    assertGpa(student, 2.0);
}

private void assertGpa(Student student, double expectedGpa) {
    assertEquals(expectedGpa, student.getGpa(), GRADE_TOLERANCE);
}

```

枚举

J2SE 5.0 引入了枚举类型——在离散列表中包含所有可能的值。例如，需要实现一个表示纸牌的类，您知道可能的值是梅花、方片、黑桃、以及红心，没有其它的可能。对于成绩，您知道只有五种可能的字母成绩。

上面的 GPA 代码中，使用字符串表示可能的字母成绩。尽管这样可以工作，但是依然存在问题。首先，输入不同字符串的时候，很容易犯错误。您可以创建类常量来表示每一个字母成绩（我们在前面的代码中已经这样做了），只要所有的代码都使用这些常量，就不会有问题。

但是，即使提供了类常量，客户代码依然有可能传入无效的值。不能防止使用 Student 类的客户编写下面的代码：

```
student.addGrade("a");
```

结果可能不是您所期望的——您编写的代码只能处理大写的字母成绩。

相反，可以定义枚举类型的成绩。目前，最好的是类 Student：

```
public class Student {
    enum Grade {A, B, C, D, F};
    ...
}
```

179

使用 enum 可以声明一个新类型。在这个例子中，因为在类 Student 中定义了 Grade 枚举，所以您创建了一个新数据类型 Student.Grade。该枚举类型有五个可能的值，每个值表示一个命名的对象实例。

改变测试，使用枚举实例：

```
public void testCalculateGpa() {
    Student student = new Student("a");
    assertGpa(student, 0.0);
    student.addGrade(Student.Grade.A);
    assertGpa(student, 4.0);
    student.addGrade(Student.Grade.B);
    assertGpa(student, 3.5);
    student.addGrade(Student.Grade.C);
    assertGpa(student, 3.0);
    student.addGrade(Student.Grade.D);
    assertGpa(student, 2.5);
    student.addGrade(Student.Grade.F);
    assertGpa(student, 2.0);
}
```

在类 Student 中，将所有字符串类型的成绩声明改成 Grade 类型。由于枚举 Grade 定义在 Student 内部，所以可以直接引用它（用 Grade 替代 Student.Grade）。


```

public class Student implements Comparable<Student> {
    enum Grade {A, B, C, D, F};
    ...
    private ArrayList<Grade> grades = new ArrayList<Grade>();
    ...

    void addGrade(Grade grade) {
        grades.add(grade);
    }
    ...
    double getGpa() {
        if (grades.isEmpty())
            return 0.0;
        double total = 0.0;
        for (Grade grade: grades)
            total += gradePointsFor(grade);
        return total / grades.size();
    }
    ...
    int gradePointsFor(Grade grade) {
        if (grade == Grade.A) return 4;
        if (grade == Grade.B) return 3;
        if (grade == Grade.C) return 2;
        if (grade == Grade.D) return 1;

        return 0;
    }
    ...
}

```


180

方法 `gradePointsFor` 将参数 `grade` 和枚举数值逐个进行比较。每个枚举数值在内存中有唯一的实例，所以可以用 `==` 操作符进行比较，而不用 `equals` 方法。`==` 操作符用来比较两个对象的引用。这两个引用指向内存中的同一个对象吗？

这样，客户不再可能向方法 `addGrade` 中传入无效的数值。客户代码不能创建新的 `Grade` 实例。类 `Student` 的枚举类型 `Grade` 指定了全部五个实例。

多态

过度使用 `if` 语句，将很快使代码变得难以维护。多态的概念可以帮助您重组代码，最小化使用 `if` 语句的需要。

 学生成绩比上面的例子要用到更多的 `if` 语句。您需要支持荣誉学生。荣誉学生有特别的待遇。如果是“A”类，他们可以得到5分；如果是“B”类，他们可以得到4分；如果是“C”类，他们可以得到3分；如果是“D”类，他们可以得到2分。

进一步重构测试：

```

public void testCalculateHonorsStudentGpa() {
    assertGpa(createHonorsStudent(), 0.0);
    assertGpa(createHonorsStudent(Student.Grade.A), 5.0);
    assertGpa(createHonorsStudent(Student.Grade.B), 4.0);
    assertGpa(createHonorsStudent(Student.Grade.C), 3.0);
    assertGpa(createHonorsStudent(Student.Grade.D), 2.0);
    assertGpa(createHonorsStudent(Student.Grade.F), 0.0);
}

private Student createHonorsStudent(Student.Grade grade) {
    Student student = createHonorsStudent();
    student.addGrade(grade);
    return student;
}

private Student createHonorsStudent() {
    Student student = new Student("a");
    student.setHonors();
    return student;
}

```

181

相应的生产代码:

```

private boolean isHonors = false;
// ...
void setHonors() {
    isHonors = true;
}
// ...
int gradePointsFor(Grade grade) {
    int points = basicGradePointsFor(grade);
    if (isHonors)
        if (points > 0)
            points += 1;
    return points;
}

private int basicGradePointsFor(Grade grade) {
    if (grade == Grade.A) return 4;
    if (grade == Grade.B) return 3;
    if (grade == Grade.C) return 2;
    if (grade == Grade.D) return 1;
    return 0;
}

```

不过, 代码正在变得难以控制。下一步, 假设您必须支持另一种不同的成绩方案:

```

double gradePointsFor(Grade grade) {
    if (isSenatorsSon) {
        if (grade == Grade.A) return 4;
        if (grade == Grade.B) return 4;
        if (grade == Grade.C) return 4;
        if (grade == Grade.D) return 4;
        return 3;
    }
    else {
        double points = basicGradePointsFor(grade);
        if (isHonors)

```

```

        if (points > 0)
            points += 1;
        return points;
    }
}

```

现在，代码正变得凌乱。并且，院长指出马上会有更多的学生成绩方案。院长每增加一种方案，您就必须修改类 `Student` 的代码。修改 `Student`，很容易使 `Student` 以及依赖于它的类无法继续工作。

182

您应该冻结类 `Student`，不要对其进行任何改动。每次院长增加新的方案，您可以通过扩展系统⁴来满足需求，而不是每次都修改类 `Student`。

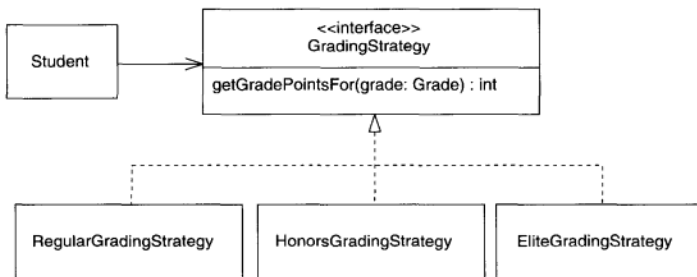


图 5.2 策略



设计系统，通过扩展来拥抱变化。不要通过修改来适应变化。

可以将成绩方案考虑成某种策略。根据不同类型的学生，策略会有所变化。您可以创建专门的学生类，例如 `HonorsStudent`，`RegularStudent`，以及 `PrivilegedStudent`，但是学生可以变更状态。将对象从一种类型转化成另一种类型是困难的。

相反，您可以为每个成绩方案创建一个类。然后，将合适的方案赋值给每一个学生。

您可以使用接口 `GradingStrategy` 来表示成绩策略这一抽象概念。类 `Student` 存储一个指向 `GradingStrategy` 接口类型的引用，如图 5.2 中的 UML 图。在图 5.2 中，有三个实现了 `GradingStrategy` 接口的类：`RegularGradingStrategy`，`HonorsGradingStrategy`，以及 `EliteGradingStrategy`。UML 用虚线箭头表示实现关系。按照 Java 术语，`RegularGradingStrategy` 实现了 `GradingStrategy` 接口。按照 UML 术语，`RegularGradingStrategy` 实现了 `GradingStrategy` 接口。

183

⁴ [Martin2003],p.99.

UML 和接口

通常，用 UML 类图来描述系统的结构——类之间的关系和依赖。同时，不要用 UML 图来表现细节。不要在 UML 中显示非 public 的方法，通常也不要显示某个特定类的方法。UML 图只需要表现有趣和有用的内容。

作为一种简单的图表语言，UML 最好用来表现系统中比较大的概念。如果将所有可能的细节都放到 UML 图中，那么将会使您希望表达的重要的东西变得模糊。

用 UML 图表示接口没有什么不同。通常用 UML 图来表现某个类实现了某个特定的接口。定义在接口中的方法通常可以被理解，而且相应的实现在代码里。UML 提供了可视的、简洁的方法，来表现某个类实现了某个接口。例如，用下面的 UML 图，可以表示类 String 实现了 Comparable 接口：



GradingStrategy 接口声明：任何实现该接口的类，必须提供根据给定 Grade 返回分数的能力。在源文件 GradingStrategy.java 中，定义 GradingStrategy 如下：

```
package sis.studentinfo;

public interface GradingStrategy {
    public int getGradePointsFor(Student.Grade grade);
}
```

您需要用单独的类来表示每一种策略。每一个策略类都实现了 GradingStrategy 接口，因而都提供了方法 getGradePointsFor 相应的实现代码。

接口定义的方法必须是实现类的公共接口的一部分。所以，默认的接口方法都是公共的。在接口中，您不需要为方法声明指定 public 关键字：

```
package sis.studentinfo;

public interface GradingStrategy {
    int getGradePointsFor(Student.Grade grade);
}
```

HonorsGradingStrategy:

```
package sis.studentinfo;

public class HonorsGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }
}
```

```

    }

    int basicGradePointsFor(Student.Grade grade) {
        if (grade == Student.Grade.A) return 4;
        if (grade == Student.Grade.B) return 3;
        if (grade == Student.Grade.C) return 2;
        if (grade == Student.Grade.D) return 1;
        return 0;
    }
}

```

RegularGradingStrategy:

```

package sis.studentinfo;

public class RegularGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        if (grade == Student.Grade.A) return 4;
        if (grade == Student.Grade.B) return 3;
        if (grade == Student.Grade.C) return 2;
        if (grade == Student.Grade.D) return 1;
        return 0;
    }
}

```

如果充分了解代码中存在重复的地方，请花时间去重构，从而消除重复。可以引入 **BasicGradingStrategy**，该方法提供了一个包含公用代码的静态方法。或者您可以选择等待（只是因为后面的原因——否则不要推迟消除重复的代码）：下一课将介绍另外一种消除重复的方法。

在 **StudentTest** 中，不再使用 **setHonors** 来创建一个荣誉学生，替代方法是发送消息 **setGradingStrategy** 给 **Student** 对象，并且传入 **HonorsGradingStrategy** 实例作为参数。

```

private Student createHonorsStudent() {
    Student student = new Student("a");
    student.setGradingStrategy(new HonorsGradingStrategy());
    return student;
}

```

请注意：您是如何在一个地方做出修改，从而消除了测试代码中的所有重复的。

然后修改类 **Student**，允许设置和存储策略。用表示默认策略的 **RegularGradingStrategy** 对象来初始化 **gradingStrategy** 实例变量。

```

public class Student {
    ...
    private GradingStrategy gradingStrategy =
        new RegularGradingStrategy();
    ...
    void setGradingStrategy(GradingStrategy gradingStrategy) {
        this.gradingStrategy = gradingStrategy;
    }
    ...
}

```

下一节介绍接口引用，解释为什么实例变量 `gradingStrategy` 和方法 `setGradingStrategy` 的参数都是 `GradingStrategy` 类型。

修改 `Student`，通过发送 `gradePointsFor` 消息来获得分数，消息的接收者是 `gradingStrategy` 所指向的对象。

```
int gradePointsFor(Grade grade) {
    return gradingStrategy.getGradePointsFor(grade);
}
```

去掉实例变量 `isHonors`，和该变量相关的 `setter` 方法，以及 `basicGradePointsFor` 方法。

既然方法 `gradePointsFor` 什么也不做（只是一个简单的代理），那么您可以将它的代码并入方法 `getGpa`。在某些情况下，诸如 `gradePointsFor` 的单行代理方法可以提供非常干净的代码。但是这个例子中，方法 `gradePointsFor` 没有带来任何好处，它的内容和名字表达了同样的意思，而且没有任何代码调用这个方法。

在 `getGpa` 中并入方法 `gradePointsFor` 的单行代码。然后您就可以彻底删除方法 `gradePointsFor`。

```
double getGpa() {
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (Grade grade: grades)
        total += gradingStrategy.getGradePointsFor(grade);
    return total / grades.size();
}
```

现在您拥有了一个可扩展的、闭合的方案。增加第三个 `GradingStrategy` 实现，例如 `EliteGradingStrategy` 是一个简单的练习——只需创建一个新类，实现一个方法，类 `Student` 中的代码可以保持不变——您已经冻结了它，针对成绩策略不需要做任何改动。而且，修改某个特定成绩策略的同时，不会影响其它所有的成绩策略。

使用接口引用

在类 `Student` 中，使用 `GradingStrategy` 类型的实例变量和参数：

```
public class Student implements Comparable<Student> {
    ...
    private GradingStrategy gradingStrategy =
        new RegularGradingStrategy();
    ...
    void setGradingStrategy(GradingStrategy gradingStrategy) {
        this.gradingStrategy = gradingStrategy;
    }
}
```

即使创建一个 `RegularGradingStrategy` 对象，并将其赋值给 `gradingStrategy` 实例变量（`GradingStrategy` 是 `GradingStrategy` 类型的变量，而不是 `RegularGradingStrategy` 类型的变量）。Java 规则是：某个类实现了某个接口，那么就是该接口类型。`RegularGradingStrategy` 实现了 `GradingStrategy` 接口，所以也是 `GradingStrategy` 类型。

将对象赋值给 `GradingStrategy` 接口类型的引用，该对象在内存中依然是 `RegularGradingStrategy` 对象。客户代码处理该 `GradingStrategy` 类型的引用。但是，客户代码只能通过引用发送 `GradingStrategy` 消息。

`Student` 中的代码不关心传入的是 `RegularGradingStrategy` 还是 `HonorsGradingStrategy`。方法 `getGpa` 发送消息 `getGradePointsFor` 给引用 `gradingStrategy`，而不需要知道引用所指向的实际类型。



最好使用接口类型的变量以及参数。

使用接口类型是实现良好面向对象系统的基石。接口是“抽象的围墙”。接口允许您通过抽象来定义对象交互的方法——抽象是不可能改变的事物。另一种方法是依赖于某个具体的类——类是可能改变的。

接口为什么重要？因为任何时候，如果对类 `X` 作了改动，那么您必须重新编译、重新测试所有依赖于 `X` 的类。您还必须重新编译、重新测试依赖于这些类的类。图 5.3 中，对类 `X` 的改动，将会反过来影响到依赖于它的类 `A` 和 `B`。

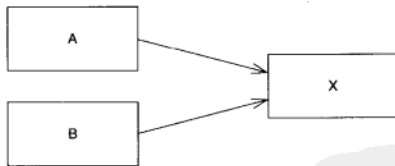


图 5.3 X 的改变会影响到 A 和 B

接口改变了这种依赖关系，使得客户类依赖于抽象⁵。实现接口的类同样依赖于抽象，但是客户类不再依赖于具体的、变化的类。接口 `XAbstraction` 表示源自 `XImplementation` 的抽象。类 `A` 和 `B` 通过接口与 `XImplementation` 类型的对象进行交互。类 `XImplementation` 同样依赖于这个抽象。图 5.4 中，所有类都不依赖于可能变化的实现细节。

设计面向对象系统的时候，您应该努力获得此类依赖关系。更多地依赖具体类型，那么修改系统就越困难。更多地依赖抽象类型（接口），那么修改系统就越容易。通过接口，您可以在客户和具体服务类之间构建起抽象屏障。

⁵ [Martin 2003], p.127.

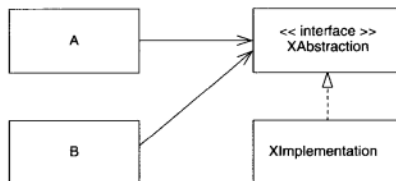


图 5.4 反转依赖

ArrayList 和 List 接口

前面的课程中，您创建了 `ArrayList` 实例，并将它们赋值给 `ArrayList` 类型的引用：

```
ArrayList<Student> students = new ArrayList<Student>();
```

浏览 Java API 文档，发现 `ArrayList` 实现了接口 `List`。浏览接口 `List` 的文档，会发现您发送给 `ArrayList` 实例的所有消息，都定义在 `List` 中。因此，最好将 `students`（在这个例子里）定义为 `List` 类型：

```
List<Student> students = new ArrayList<Student>();
```

类 `ArrayList` 在内存中定义了固定长度的数组。如果不在列表末尾频繁地插入或者删除元素，那么 `ArrayList` 的性能很好。如果需要在列表末尾频繁地插入或者删除元素，那么最好的数据结构是链表，因为链表使用了更加动态的内存管理。使用链表的缺点在于：访问链表中的任何元素，都需要从头开始顺序遍历链表。相反，因为只需要计算相对该数组内存起始地址的偏移量，所以在数组中访问某个元素非常快。

Java 用 `java.util.LinkedList` 实现了链表。如果在代码中始终使用 `List` 类型的引用，那么您在一个地方做出修改，就可以改变应用程序的性能特征：

```
List<Student> students = new LinkedList<Student>();
```

现在，请仔细检查您的源代码，尽可能将引用修改为接口类型。

练习

1. 创建和封装包括两种颜色的枚举类型，使类 `Piece` 知道该枚举类型的存在。根据您的实现，可能需要重大的重构。注意每个有改动的地方——针对某段代码，怎样才能避免修

改两次？

您或许也考虑为格子的值创建枚举类型。如果代码允许您简单地实现它，那么继续。或者您也可以等到第6课，在第6课您将学习如何关联数据和枚举值。

2. 为每个格子引入枚举值。您应该将格子的表达形式和格子的枚举值相脱离。
3. 为每种颜色/格子的组合（例如：`createWhitePawn`，`createBlackRook`），创建单独的工厂方法。

```
package pieces;

import junit.framework.TestCase;

public class PieceTest extends TestCase {
    public void testCreate() {
        verifyCreation(
            Piece.createWhitePawn(), Piece.createBlackPawn(),
            Piece.Type.PAWN, Piece.PAWN_REPRESENTATION);
        verifyCreation(
            Piece.createWhiteRook(), Piece.createBlackRook(),
            Piece.Type.ROOK, Piece.ROOK_REPRESENTATION);
        verifyCreation(
            Piece.createWhiteKnight(), Piece.createBlackKnight(),
            Piece.Type.KNIGHT, Piece.KNIGHT_REPRESENTATION);
        verifyCreation(
            Piece.createWhiteBishop(), Piece.createBlackBishop(),
            Piece.Type.BISHOP, Piece.BISHOP_REPRESENTATION);
        verifyCreation(Piece.createWhiteQueen(), Piece.createBlackQueen(),
            Piece.Type.QUEEN, Piece.QUEEN_REPRESENTATION);
        verifyCreation(Piece.createWhiteKing(), Piece.createBlackKing(),
            Piece.Type.KING, Piece.KING_REPRESENTATION);
        Piece blank = Piece.noPiece();
        assertEquals('.', blank.getRepresentation());
        assertEquals(Piece.Type.NO_PIECE, blank.getType());
    }

    private void verifyCreation(Piece whitePiece, Piece blackPiece,
        Piece.Type type, char representation) {
        assertTrue(whitePiece.isWhite());
        assertEquals(type, whitePiece.getType());
        assertEquals(representation, whitePiece.getRepresentation());

        assertTrue(blackPiece.isBlack());
        assertEquals(type, blackPiece.getType());
        assertEquals(Character.toUpperCase(representation),
            blackPiece.getRepresentation());
    }
}
```

使用 `if` 语句，将 `createWhite` 和 `createBlack` 方法合并到单个私有工厂方法。

4. 在 `Board` 中，给定颜色和格子的表示形式，返回格子的数目。例如，下面的棋盘，如果您查询黑卒的数目，那么返回值为 3。按照要求来计算格子的数目（换句话说，不要对所创建的格子进行计数）。

```

. K R . . . .
P . P B . . . .
. P . . Q . . .
. . . . . . .
. . . . . n q .
. . . . . p . .
. . . . . p . .
. . . . . r k .

```

5. 创建方法，根据给定的位置返回格子。对于初始的棋盘，如果请求“a8”，那么返回黑车。白方国王的位置是“e1”。

使用类 `Character` 的工具方法，将位置字符串中的每一个字符转换成数字索引。注意：棋盘可能是反的，换句话说，可能第一行在顶部，第八行在底部。您的代码必须对其进行反转。

```

R N B Q K B N R 8 (rank 8)
P P P P P P P 7
. . . . . . . 6
. . . . . . . 5
. . . . . . . 4
. . . . . . . 3
P P P P P P P 2
r n b q k b n r 1 (rank 1)
a b c d e f g h
files

```

下一个练习，您将创建方法：把棋子放在空的棋盘上。增加一个测试，验证新初始化的 `Board` 对象中没有任何棋子。该测试将引导您进行重构，您需要用 `set` 方法替换 `ArrayList` 的 `add` 方法。

6. 编写代码，将棋子放在棋盘合适的位置上。重构前面管理格子的代码。

```

. . . . . . . 8 (rank 8)
. . . . . . . 7
. K . . . . . 6
. R . . . . . 5
. . k . . . . 4
. . . . . . . 3
. . . . . . . 2
. . . . . . . 1 (rank 1)
a b c d e f g h
files

```

7. 在象棋程序中，棋盘位置的强弱关系对于决定棋子如何移动是很重要的。使用一种非常基本的棋盘评价函数，给棋盘上每一个格子赋值。按照下面的规则计算总和：王后加 5 分，车加 5 分，象加 3 分，马加 2.5 分，如果某个卒和另一个同颜色的卒在相同的列，那么加 0.5 分，否则加 1 分。记得一次只计算一方的分数总和。

```

. K R . . . . . 8
P . P B . . . . . 7
. P . . . Q . . . 6
. . . . . . . . . 5
. . . . . n q . . 4
. . . . . p . p . 3
. . . . . p p . . 2
. . . . . r k . . 1
a b c d e f g h

```

上面的例子中，黑方有 20 分，白方有 19.5 分。

以渐增的方式实现您的方案。开始，只把一个棋子添加到棋盘上。然后，逐个把其它棋子添加到棋盘上，每次添加棋子都要进行断言。最后，完成最复杂的、判断是否卒子在同一列的情况。

8. 一旦棋盘上的棋子布局完毕，将每一个格子的值赋给 Piece 对象。把双方（黑方和白方）的格子各自存放在一个集合中。保证列表中的格子根据值的大小，按照从高到低的顺序存储。
9. 检查您的代码，看是否有机会创建一个接口。系统看起来是干净还是混乱？提防没有必要的接口——牢记仅次于“没有重复”的简单规则：“最少的类，最少的方法”。
10. 检查所有的生产代码，寻找机会使用“尽早返回”和“防卫子句”，以此来简化代码。
11. 检查所有的生产代码，确保在任何可能的地方使用接口，而不是具体的实现。特别要检查对集合的使用，确保面向接口编程。

192

而且：尽管静态导入使代码难以理解，但是您可以在测试代码的合适地方使用它。测试代码通常和单个目标类进行交互。如果该目标类包含类常量，那么可以在测试类中针对这些类常量使用静态导入。上下文使得常量的定义位置变得清晰。

193

改变代码，合理地使用 import static。



本课内容包括：

- Switch 语句
- map
- 延迟初始化
- 继承
- 扩展方法
- 调用父类的构造函数
- 子合约原则

switch 语句

上一课，在 `HonorsGradingStrategy` 和 `RegularGradingStrategy` 中，您编写方法以字母的形式返回正确的 GPA。在该方法中，您使用了一连串的 `if` 语句。下面是相应的代码（取自 `HonorsGradingStrategy`）：

```
int basicGradePointsFor(Student.Grade grade) {  
    if (grade == Student.Grade.A) return 4;  
    if (grade == Student.Grade.B) return 3;  
    if (grade == Student.Grade.C) return 2;  
    if (grade == Student.Grade.D) return 1;  
    return 0;  
}
```

方法 `basicGradePointsFor` 中，每个条件语句都用某个值与同一个变量 `grade` 进行比较。Java 允许用 `switch` 语句来简化此类代码：

```
int basicGradePointsFor(Student.Grade grade) {  
    switch (grade) {  
        case A: return 4;  
        case B: return 3;  
    }
```

```

        case C: return 2;
        case D: return 1;
        default: return 0;
    }
}

```

switch 语句中，首先需要指定一个数值或者表达式，将其作为比较目标。该例子中，参数 grade 就是指定的目标：

```
switch (grade) {
```

然后，指定任意数量的 case 标记。每个 case 标记指定一个枚举值。Java 虚拟机执行到 switch 语句时，首先判断目标的值，然后将这个值与每个 case 标记轮流进行比较。

如果目标的值与某个 case 标记相匹配，Java 虚拟机会跳过目标与该 case 标记之间所有的代码，立刻将控制流转移到该 case 标记。如果目标的值和所有的 case 标记均不匹配，Java 虚拟机将控制流转移到 default 标记。default 标记是可选的。如果不存在 default 标记，虚拟机立刻将控制流转移到 switch 的下一行代码。

所以，如果变量 grade 的值是 Student.Grade.B，那么 Java 虚拟机会将控制流转移到：

```
case B: return 3;
```

Java 虚拟机执行 return 语句，立刻将控制流移出该方法。

Case 标记只是标记

Case 标记只是一个标记而已。Java 虚拟机仅在初次遇到 switch 语句的时候，才使用 case 标记。一旦 Java 找到匹配的 case 标记就将控制流转移到此处，而且会忽略其它所有的 case 标记。Java 在 switch 语句中，按照自上而下的顺序执行其它代码；Java 会忽略 case 标记和 default 标记，直到 switch 语句的末尾（或者直到遇见某个跳转语句，例如 return。）¹

下面的例子展示了 switch 语句的控制流：

```

public void testSwitchResults() {
    enum Score
    { fieldGoal, touchdown, extraPoint,
      twoPointConversion, safety };

    int totalPoints = 0;

    Score score = Score.touchdown;

```

¹ 在第 7 课中，您将了解更多的跳转语句。

```

switch (score) {
    case fieldGoal:
        totalPoints += 3;
    case touchdown:
        totalPoints += 6;
    case extraPoint:
        totalPoints += 1;
    case twoPointConversion:
        totalPoints += 2;
    case safety:
        totalPoints += 2;
}
assertEquals(6, totalPoints);
}

```

测试失败:

```

junit.framework.AssertionFailedError: expected<6> but was<11>

```

由于变量 `score` 被赋值为 `Score.touchdown`, 所以 Java 虚拟机将控制流转移到相应 case 标记中的代码:

```

case touchdown:
    totalPoints += 6;

```

Java 执行完该行代码, 接着执行下面的三行代码 (忽略所有的 case 标记):

```

case extraPoint: // ignored
    totalPoints += 1;
case twoPointConversion: // ignored
    totalPoints += 2;
case safety: // ignored
    totalPoints += 2;

```

`totalPoints` 的值通过 $6+1+2+2=11$ 得到。所以测试失败!

在每个 case 标记代码块的末尾插入一行 `break` 语句, 可以得到您所期望的行为。

```

public void testSwitchResults() {
    enum Score
    { fieldGoal, touchdown, extraPoint,
      TwoPointConversion, safety };

    int totalPoints = 0;

    Score score = Score.touchdown;

    switch (score) {
        case fieldGoal:
            totalPoints += 3;
            break;
        case touchdown:
            totalPoints += 6;
            break;
        case extraPoint:
            totalPoints += 1;

```

```

        break;
    case twoPointConversion:
        totalPoints += 2;
        break;
    case safety:
        totalPoints += 2;
        break;
    }
    assertEquals(6, totalPoints);
}

```

break 是另一种控制流跳转语句，它可以将控制流转移到 switch 的下一条语句。该例子中，执行任意 break 语句，将导致控制流转移到 assertEquals 语句。

在 switch 语句范围内，某行语句前面可以有多个 case 标记：

```

switch (score) {
    case fieldGoal:
        totalPoints += 3;
        break;
    case touchdown:
        totalPoints += 6;
        break;
    case extraPoint:
        totalPoints += 1;
        break;
    case twoPointConversion:
    case safety:
        totalPoints += 2;
        break;
}

```

如果变量 score 匹配 Score.twoPointConversion 或者 Score.safety，那么 totalPoints 将增加 2。

case 标记除了可以指定枚举类型的数值，也可以指定 char、byte、short、或者 int 类型的值。不幸的是，您不能使用 String 类型的值。

代码中不应该有大量的 switch 语句。而且，也不应该包含大量的多重 if 语句。通常，您可以使用多态来消除 switch 语句。如果能够消除冗余、使代码容易维护，就应该考虑用多态替换 switch 语句。

在进一步学习之前，重构 RegularGradingStrategy，用 switch 语句替换多重 if 语句。

Map

可以用 map 代替 switch 语句。Map 是一种集合，根据指定的关键字可以从集合中快速插

入和获取相应的值。举例来说，在线字典可以是一个集合，存储每个单词（关键字）以及该单词的定义（值）。

Java 提供了接口 `java.util.Map`，该接口定义了 `Map` 操作的公共行为。对于成绩单信息，可以使用实现类 `EnumMap`。`EnumMap` 约束所有的关键字必须是 `enum` 对象。

使用 `put` 方法，可以向 `Map` 中添加 `key-value` 对，`put` 方法以 `key` 和 `value` 作为参数。使用 `get` 方法，可以根据指定的参数 `key`，返回相应的 `value`。

假设您需要根据每个学生的成绩等级，在成绩单上打印出相应的信息。在包 `sis.reports` 中添加新类 `ReportCardTest`。

```
package sis.report;

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

public class ReportCardTest extends TestCase {
    public void testMessage() {
        ReportCard card = new ReportCard();
        assertEquals(ReportCard.A_MESSAGE,
            card.getMessage(Student.Grade.A));
        assertEquals(ReportCard.B_MESSAGE,
            card.getMessage(Student.Grade.B));
        assertEquals(ReportCard.C_MESSAGE,
            card.getMessage(Student.Grade.C));
        assertEquals(ReportCard.D_MESSAGE,
            card.getMessage(Student.Grade.D));
        assertEquals(ReportCard.F_MESSAGE,
            card.getMessage(Student.Grade.F));
    }
}
```

199

把 `Student` 中定义的枚举 `Grade` 修改成 `public` 类型，重新编译。

下面是类 `ReportCard`：

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;

public class ReportCard {
    static final String A_MESSAGE = "Excellent";
    static final String B_MESSAGE = "Very good";
    static final String C_MESSAGE = "Hmmm...";
    static final String D_MESSAGE = "You're not trying";
    static final String F_MESSAGE = "Loser";

    private Map<Student.Grade, String> messages = null;

    public String getMessage(Student.Grade grade) {
        return getMessages().get(grade);
    }

    private Map<Student.Grade, String> getMessages() {
        if (messages == null)
```



```

        loadMessages();
        return messages;
    }

    private void loadMessages() {
        messages =
            new EnumMap<Student.Grade, String>(Student.Grade.class);
        messages.put(Student.Grade.A, A_MESSAGE);
        messages.put(Student.Grade.B, B_MESSAGE);
        messages.put(Student.Grade.C, C_MESSAGE);
        messages.put(Student.Grade.D, D_MESSAGE);
        messages.put(Student.Grade.F, F_MESSAGE);
    }
}

```

200

类 `ReportCard` 中定义了一个实例变量 `messages`，该变量是参数化 `Map` 类型。其中，参数 `key` 的类型是 `Student.Grade`，参数 `value` 的类型是 `String`。当创建 `EnumMap` 实例时，您必须以 `Student.Grade.class` 作为参数。

方法 `getMessages` 使用延迟初始化，来将相应的字符串静态常量加载到 `EnumMap` 实例。

延迟初始化

可以在定义成员变量时进行初始化，或者在构造函数中进行初始化。另一种选择是在需要使用该成员变量时，才进行初始化。这种技术就是延迟初始化。

在 `getter` 方法中，例如 `getMessages`，首先测试某个成员变量是否已经初始化。对于某个实例变量的引用，可以测试该引用是否为 `null`。如果为 `null`，赋值给该成员变量进行初始化。最后，将该成员变量的值作为 `getter` 方法的结果返回。

延迟初始化的主要用途是推迟加载某个成员变量的可能的高成本的操作，只在必需的时候才进行此类操作。如果该成员变量从来没有被调用过，那么就不会加载它。每次访问该成员变量，都需要测试其是否为 `null`，虽然这造成了一点儿重复工作，但是这点儿重复工作相对来说是可以忽略的。

在这里，延迟初始化是一种保持复杂初始化与成员变量访问具有相近逻辑的方法。延迟初始化，有助于理解成员变量如何被初始化以及如何被使用。

方法 `getMessage` 中只有一行代码，这行代码中，以成绩等级作为关键字调用 `get` 方法获取相应的字符串。

`Map` 功能非常强大，存取数据的速度非常快，在应用程序中被广泛使用。参考第九课，可以看到关于其用法的详细讨论。

继承

类 `RegularGradingStrategy` 和 `HonorsGradingStrategy` 的代码中含有重复的逻辑。两个类中都

包含相同逻辑的 switch 语句:

```
switch (grade) {
    case A: return 4;
    case B: return 3;
    case C: return 2;
    case D: return 1;
    default: return 0;
}
```

201

可以使用继承来消除此类冗余。在本书开头的敏捷综述中，曾对继承进行了简要的讨论。您可以从类 `RegularGradingStrategy` 和 `HonorsGradingStrategy` 中找出共同性，从而抽象出基类 `BasicGradingStrategy`。然后将 `RegularGradingStrategy` 和 `HonorsGradingStrategy` 声明为 `BasicGradingStrategy` 的子类。作为子类，`RegularGradingStrategy` 和 `HonorsGradingStrategy` 拥有 `BasicGradingStrategy` 的所有行为。

下面以渐增的方式，将代码重构成基于继承的解决方案。第一步是创建类 `BasicGradingStrategy`:

```
package sis.studentinfo;

public class BasicGradingStrategy {
}
```

然后，使用关键字 `extends` 将 `RegularGradingStrategy` 和 `HonorsGradingStrategy` 声明为 `BasicGradingStrategy` 的子类。`extends` 子句必须在所有 `implements` 子句的前面。

```
// RegularGradingStrategy.java
package sis.studentinfo;

public class RegularGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}

// HonorsGradingStrategy.java
package sis.studentinfo;

public class HonorsGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
    }
}
```

```

        return points;
    }

    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}

```

现在，将通用代码移动到父类（也叫基类）`BasicGradingStrategy` 的某个方法中。首先，将 `basicGradePointsFor` 方法从 `HonorsGradingStrategy` 移动到 `BasicGradingStrategy`：

```

// HonorsGradingStrategy.java
package sis.studentinfo;

public class HonorsGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }
}

// BasicGradingStrategy.java
package sis.studentinfo;

public class BasicGradingStrategy {
    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}

```

重新编译和测试。

即使类 `HonorsGradingStrategy` 不再包含 `basicGradePointsFor` 方法，也可以调用此方法，就像已经定义了此方法。您可以在测试类中调用断言方法，因为测试类继承自 `junit.framework.TestCase`。从概念上讲，类 `HonorsGradingStrategy` 是一种 `BasicGradingStrategy`。

因此，`HonorsGradingStrategy` 可以使用 `BasicGradingStrategy` 中定义的任何非私有方法。

图 6.1 用 UML 表示了继承关系。

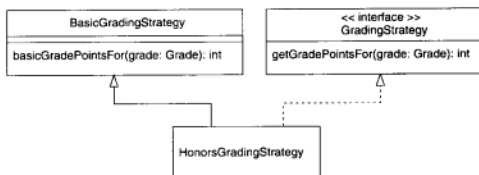


图 6.1 从 BasicGradingStrategy 继承

接着，修改 RegularGradingStrategy，以重用方法 basicGradePointsFor。

```

package sis.studentinfo;

public class RegularGradingStrategy
    extends BasicGradingStrategy
    implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return basicGradePointsFor(grade);
    }
}
  
```

抽象类

您可以停在这里，找一个更完善的解决方案。也可以通过在基类 BasicGradingStrategy 中直接实现接口 GradingStrategy，从而进一步消除冗余。唯一的问题是，您没有必要提供方法 getGradePointsFor 的实现。

Java 允许将某个方法定义为抽象方法，这意味着您不能或者不希望提供该方法的实现。一旦某个类中包含了哪怕一个抽象方法，那么也必须把类定义成抽象类。Java 不允许创建抽象类的实例，就像不能直接实例化某个接口。

◀ 204

继承自抽象类的任何子类，必须实现父类中的所有抽象方法，否则子类也必须定义成抽象类。

改变 BasicGradingStrategy 的定义，实现接口 GradingStrategy。然后提供方法 getGradePointsFor 的抽象定义。

```

package sis.studentinfo;

abstract public class BasicGradingStrategy implements GradingStrategy {
    abstract public int getGradePointsFor(Student.Grade grade);

    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
        }
    }
}
  
```

```

        case B: return 3;
        case C: return 2;
        case D: return 1;
        default: return 0;
    }
}

```

在定义子类时，不再需要声明它继承接口 `GradingStrategy`：

```

// HonorsGradingStrategy.java
package sis.studentinfo;

public class HonorsGradingStrategy extends BasicGradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = basicGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }
}

// RegularGradingStrategy.java
package sis.studentinfo;

public class RegularGradingStrategy extends BasicGradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return basicGradePointsFor(grade);
    }
}

```

205

方法扩展

第三种方案²将 `RegularGradingStrategy` 和 `BasicGradingStrategy` 视为相同的类。定义在 `HonorsGradingStrategy` 中的方法 `getGradePointsFor`，针对 `RegularGradingStrategy` 中同名的方法进行了扩展。换句话说，您可以在 `BasicGradingStrategy` 中提供 `getGradePointsFor` 的默认定义，然后在 `HonorsGradingStrategy` 中对该方法进行扩展。

将方法 `getGradePointsFor` 的定义从 `RegularGradingStrategy` 移动到 `BasicGradingStrategy`。因为所有的方法都提供了定义，所以不再将 `BasicGradingStrategy` 声明为抽象类。

```

// BasicGradingStrategy.java
package sis.studentinfo;

public class BasicGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return basicGradePointsFor(grade);
    }
}

```

² Java 提供多种方式，允许您编写出尽可能具有表现力的代码。当然，没有一种方式是尽善尽美的。

```

    }

    int basicGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}

// RegularGradingStrategy.java
package sis.studentinfo;

public class RegularGradingStrategy extends BasicGradingStrategy {
}

```

RegularGradingStrategy 中不再定义任何方法！

(编译，测试。)

接着，修改 HonorsGradingStrategy 以扩展基类方法 getGradePointsFor。在子类中定义同名的方法，可以扩展某个方法。在子类的方法中，调用基类方法以影响其行为，接着提供附加的、特定的行为。

◀ 206

相对扩展方法，Java 允许重载方法，在子类中提供和父类方法没有任何关系、崭新的方法实现。对 Java 编译器而言，扩展和重载没有任何区别。如果在子类方法中调用了父类的同名方法，那么就是方法的扩展。



相对重载，最好使用扩展。

在子类中以某种方式，定制基类方法中的行为，而不是彻底改变基类方法。

在子类中，用关键字 super 显式地调用基类方法，如下面黑体所示：

```

package sis.studentinfo;

public class HonorsGradingStrategy extends BasicGradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        int points = super.getGradePointsFor(grade);
        if (points > 0)
            points += 1;
        return points;
    }
}

```

使用关键字 super，Java 虚拟机会在父类中查找相应的方法。

重构

现在，类 `BasicGradingStrategy` 的方法 `basicGradePointsFor` 是多余的。可以清除该方法。

```
package sis.studentinfo;

public class BasicGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        switch (grade) {
            case A: return 4;
            case B: return 3;
            case C: return 2;
            case D: return 1;
            default: return 0;
        }
    }
}
```

207

子类 `RegularGradingStrategy` 有必要存在吗？该类中没有什么内容了：

```
package studentinfo;

public class RegularGradingStrategy
    extends BasicGradingStrategy {
}
```

类 `Student` 中，将 `BasicGradingStrategy` 作为引用 `gradeStrategy` 的默认值，从而可以清除子类 `RegularGradingStrategy`。

```
public class Student {
    ...
    private GradingStrategy gradeStrategy =
        new BasicGradingStrategy();
    ...
}
```

到目前为止，只是修改了生产代码，还没有修改任何测试类（希望您针对每一次变更，都运行了测试）。通常而言，每一个生产类都至少对应一个测试类。您应该分别对类 `HonorsGradingStrategy` 和 `BasicGradingStrategy` 进行测试。

然而，`Student` 中的测试包含了“管理学生”这个上下文环境中的行为。你会想要对此额外提供单元级别的测试。一般规则是，对于每个被测类至少配备一个测试类。你应该单独地分别测试 `HonorsGradingStrategy` 和 `BasicGradingStrategy`。

下面是对 `BasicGradingStrategy` 的测试：

```

package sis.studentinfo;

import junit.framework.*;

public class BasicGradingStrategyTest extends TestCase {
    public void testGetGradePoints() {
        BasicGradingStrategy strategy = new BasicGradingStrategy();
        assertEquals(4, strategy.getGradePointsFor(Student.Grade.A));
        assertEquals(3, strategy.getGradePointsFor(Student.Grade.B));
        assertEquals(2, strategy.getGradePointsFor(Student.Grade.C));
        assertEquals(1, strategy.getGradePointsFor(Student.Grade.D));
        assertEquals(0, strategy.getGradePointsFor(Student.Grade.F));
    }
}

```

208

对 HonorsGradingStrategy 对应的测试如下:

```

package sis.studentinfo;

import junit.framework.*;

public class HonorsGradingStrategyTest extends TestCase {
    public void testGetGradePoints() {
        GradingStrategy strategy = new HonorsGradingStrategy();
        assertEquals(5, strategy.getGradePointsFor(Student.Grade.A));
        assertEquals(4, strategy.getGradePointsFor(Student.Grade.B));
        assertEquals(3, strategy.getGradePointsFor(Student.Grade.C));
        assertEquals(2, strategy.getGradePointsFor(Student.Grade.D));
        assertEquals(0, strategy.getGradePointsFor(Student.Grade.F));
    }
}

```

不要忘了将上面两个测试类添加到 AllTests 中的测试套件。

增强的枚举 Grade

学生成绩可以直接对应到某个枚举值。可以增强枚举的定义，在某个枚举类型定义中包含实例变量、构造函数、以及方法，就像类定义一样。主要限制是：枚举不能继承，一个枚举类型不能继承自另一个枚举类型。

修改类 Student 中枚举类型 Grade 的定义：

```

public class Student {
    public enum Grade {
        A(4),
        B(3),
        C(2),
        D(1),
        F(0);

        private int points;
    }
}

```



```

    Grade(int points) {
        this.points = points;
    }

    int getPoints() {
        return points;
    }
}
...

```

209

在枚举类型定义中，将若干枚举值作为参数，然后用分号作为参数的结尾。分号后面，定义了该枚举类型的成员变量、构造函数以及方法。其中，枚举值作为参数传入 `Grade` 的构造函数，并保存在私有成员变量 `points` 中。最后，可以通过方法 `getPoints` 获取 `points`。

现在，您可以简化 `BasicGradingStrategy` 中的代码：

```

package sis.studentinfo;

public class BasicGradingStrategy implements GradingStrategy {
    public int getGradePointsFor(Student.Grade grade) {
        return grade.getPoints();
    }
}

```

不再有 `switch` 语句了！

夏季课程安排



目前，类 `CourseSession` 支持春季和秋季的课程安排（15 周课程，一周休息）。大学也需要有夏季课程安排。夏季学期从六月初开始，一共有八周，期间没有中断。

一种方法是将学期长度作为参数传入类 `CourseSession`，使用这个参数来计算学期结束时间。

但是，夏季学期和春/秋学期还有很多其它的区别，包括最大课程学时、学分，等等。所以，创建一个新类 `SummerCourseSession`。

实现 `SummerCourseSession` 最简单的方法是：`SummerCourseSession` 继承自 `CourseSession`。`SummerCourseSession` 也是 `CourseSession`，但在细节上有所不同（见图 6.2）。

为了便于组织，在新包 `summer` 中创建类 `SummerCourseSession`（包括生产类和测试类）。

在测试类中验证夏季学期结束时间是否正确：

```

package sis.summer;

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

```

210

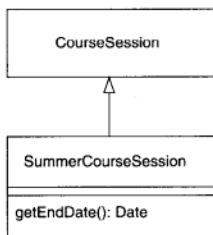


图 6.2 SummerCourseSession 定义

```

public class SummerCourseSessionTest extends TestCase {
    public void testEndDate() {
        Date startDate = DateUtil.createDate(2003, 6, 9);
        CourseSession session =
            SummerCourseSession.create("ENGL", "200", startDate);
        Date eightWeeksOut = DateUtil.createDate(2003, 8, 1);
        assertEquals(eightWeeksOut, session.getEndDate());
    }
}

```

调用基类的构造函数

下面是未经编译的 SummerCourseSession:

```

// this code does not compile!
package sis.summer;

import java.util.*;
import sis.studentinfo.*;

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

    private SummerCourseSession(
        String department,
        String number,

        Date startDate) {
        super(department, number, startDate);
    }

    Date getEndDate() {
        GregorianCalendar calendar = new GregorianCalendar();
    }
}

```

```

        calendar.setTime(startDate);
        int sessionLength = 8;
        int daysInWeek = 7;
        int daysFromFridayToMonday = 3;
        int numberOfDays =
            sessionLength * daysInWeek - daysFromFridayToMonday;
        calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
        return calendar.getTime();
    }
}

```

请注意用**黑体标注**的关键代码。

第一，在类声明时使用关键字 `extends`，将 `SummerCourseSession` 声明为 `CourseSession` 的子类。

第二，`SummerCourseSession` 的构造函数中，使用关键字 `super`，以学科、课程编号、开始日期作为参数，调用父类的构造函数。

第三，把 `CourseSession` 中的方法 `getEndDate` 拷贝到此处，并且将 `sessionLength` 的值从 16 改成 8。

编译无法通过。会看到三个编译错误，您应该知道如何修改第一个错误：

```

getEndDate() is not public in sis.studentinfo.CourseSession; cannot be accessed
from outside
package
    assertEquals(eightWeeksOut, session.getEndDate());
                                ^

```

将 `CourseSession` 方法 `getEndDate` 的定义改为 `public`，然后重新编译：

```

public Date getEndDate() {

```

您会看到两个和 `getEndDate` 相关的编译错误：

```

getEndDate() in sis.summer.SummerCourseSession cannot override getEndDate() in
sis.studentinfo.CourseSession; attempting to assign weaker access privileges; was
public
    Date getEndDate() {
        ^

```

如果要在子类中重载父类的某个方法，那么子类方法相对父类中的同名方法，必须具有相同或者更宽松的权限控制。换句话说，`CourseSession` 对方法 `getEndDate` 的控制必须比其子类更加严格。如果某个方法在父类的权限控制是 `public`，那么子类重载该方法时，也必须将其控制权限设置为 `public`。

在 `SummerCourseSession` 中将方法 `getEndDate` 的权限控制设置为 `public`：

```

public Date getEndDate() {

```

重新编译，依然有两个错误：

```
CourseSession(java.util.Date) has private access in
sis.studentinfo.CourseSession
    super(department, number, startDate);
    ^
startDate has private access in sis.studentinfo.CourseSession
    calendar.setTime(startDate);
    ^
```

毫无疑问，原因是 `CourseSession` 中以日期为参数的构造函数的权限控制是 `private`：

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

把构造函数设置为 `private`，是为了强制客户使用类方法来构建 `CourseSession` 实例。问题在于，`private` 修饰符限制了对构造函数的调用，只有 `CourseSession` 内部的代码才可以调用该构造函数。

现在，您的目的是允许 `CourseSession` 自身以及子类都可以访问 `CourseSession` 的构造函数。即使 `CourseSession` 的子类定义在不同的包中，也可以访问 `CourseSession` 的构造函数。与此同时，不允许其它类访问 `CourseSession` 的构造函数。

`public` 修饰符的控制过于宽松，其它包中的非子类也可以直接创建 `CourseSession` 对象。`Package` 修饰符不允许其它包的子类访问 `CourseSession` 的构造函数。

`Java` 提供了第四种（也是最后一种）访问修饰符——`protected`。用关键字 `protected` 修饰的成员变量或者方法，可以被所在的类访问，也可以被同一个包中的其它任何类访问，也可以被它的任何子类访问。即使子类 and 父类不在同一个包中，子类也可以访问父类中 `protected` 类型的成员变量和方法。

把 `CourseSession` 的构造函数类型从 `private` 改为 `protected`：

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

关于构造函数的编译错误消失了。但是还有一个错误：`getEndDate` 直接引用了父类中的私有成员变量 `startDate`。

```
startDate has private access in sis.studentinfo.CourseSession
    calendar.setTime(startDate);
    ^
```

把成员变量 `startDate` 从 `private` 改为 `protected`，就可以消除这个编译错误。一个更好的解决方案是：在子类中使用方法来存取私有成员变量。

把 `getStartDate` 的类型控制从 `package` 改为 `protected`。

```
public class CourseSession implements Comparable<CourseSession> {
    ...
    protected Date getStartDate() {
        return startDate;
    }
    ...
}
```

修改 CourseSession 和 SummerCourseSession，使用这个 getter 方法：

```
// CourseSession.java
public Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getStartDate());
    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();
}

// SummerCourseSession.java:
public Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getStartDate());
    int sessionLength = 8;
    int daysInWeek = 7;
    int daysFromFridayToMonday = 3;
    int numberOfDays =
        sessionLength * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
    return calendar.getTime();
}
```

由于打算偷点儿懒，直接拷贝粘贴 getEndDate 的代码，所以不得不付出额外的努力，要在两个地方进行修改。

214

在 SummerCourseSession 中，方法 getEndDate 调用 getStartDate，但是 getStartDate 并没有定义在 SummerCourseSession 中。由于没有使用范围限定修饰符，所以 Java 首先在 SummerCourseSession 中查找 getStartDate，因为该方法不存在，然后 Java 沿着继承链向上搜索，直到在父类中找到方法 getStartDate。

Protected 具有比 package 稍微宽松的访问控制。假设某个包的某个类中定义了若干 protected 元素（方法或者成员变量），那么同一个包中的其它类可以存取 protected 元素，不同包中的子类也可以存取 protected 元素，但是不同包的非子类不能访问 protected 元素。

在 CourseSession 这个例子中，如果您希望只有子类可以访问 CourseSession 的构造函数，同一个包以及不同包的子类均不能访问 CourseSession 的构造函数。依照 Java 现有的能力，无法实现这样的约束。

重构

现在是对前面仓促编写的代码进行重构的时候了。

`CourseSession` 和 `SummerCourseSession` 中都定义了方法 `getEndDate`，该方法唯一的区别是学期时间长短不同。在 `CourseSession` 中定义一个 `protected` 类型的方法：

```
public class CourseSession implements Comparable<CourseSession> {
    ...
    protected int getSessionLength() {
        return 16;
    }
    ...
}
```

修改 `getEndDate`，在 `getEndDate` 中调用 `getSessionLength`：

```
public Date getEndDate() {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(getStartDate());
    final int daysInWeek = 7;
    final int daysFromFridayToMonday = 3;
    int numberOfDays =
        getSessionLength() * daysInWeek - daysFromFridayToMonday;
    calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
    return calendar.getTime();
}
```

◀ 215

在 `SummerCourseSession` 中，重载方法 `getSessionLength`。夏季学期有 8 周，而不是 16 周。在这个例子中，因为没有改变行为，只是改变了和行为有关的数据，所以重载是一种比扩展更好的方案。

```
public class SummerCourseSession extends CourseSession {
    ...
    @Override
    protected int getSessionLength() {
        return 8;
    }
    ...
}
```

重载某个方法时，您应该在前面加上 `@Override` 注释。使用注释来标记特定的代码段。编译器、IDE 以及测试工具，能够分析这些注释。Java 编译器能够理解 `@Override` 注释。

对于前面加上 `@Override` 的方法，编译器要求其父类也必须有同名、同参数的方法。如果您犯了某些错误，比如将 `getSessionLength` 敲错成 `getSessionLentgh`，编译器会报错：

```
method does not override a method from its superclass
```

Java 并不强制您必须在代码中加上 `@Override` 注释。但是，这些注释是非常有价值的文档，

而且可以避免一些简单的错误。您可以在第 15 课中获得有关注释的更多内容。

现在可以从 `SummerCourseSession` 中删除方法 `getEndData` 了。图 6.3 中, `getSessionLength` 出现了两次, 子类 `SummerCourseSession` 重载了方法 `getSessionLength`。请注意我在每个方法前面都加上了访问控制类型信息。由于我通常只在 UML 草图中显示 `public` 信息, 所以我更倾向于省略表示 `public` 的 + (加号)。

但是, 在这个例子中, `getSessionLength` 是 `protected` 类型, 而不是 `public`, 所以在图 6.3 中添加了访问控制类型信息。UML 规定用英镑符号 (#)¹ 表示 `protected`。

216

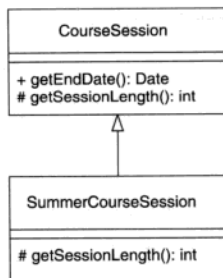


图 6.3 重载一个被保护类型的方法

特性

由于 Java 并不强制在方法前面加上 `@Override` 注释, 所以编码时很容易忘掉 (结对编程可以有效避免这种情况)。 `@Override` 注释是 J2SE 5.0 的新特性, 所以请原谅我可能在本书的某些地方忘了使用这个特性。

您会注意到, 我没有在方法 `setUp` 和 `tearDown` 前面加上 `@Override`, 即使该注释可以用在这些重载的测试方法上。使用 `@Override` 是个好主意 (特别是可以避免将 `setUp` 拼写成 `setup`)。但是旧习惯是很难改变的。我劝说自己, 这仅仅是测试代码而已, 况且无论如何测试用例会帮助我改正任何错误。不过, 我建议您不要有我这样的坏习惯。

计算学期结束时间的逻辑基本上是固定的。对于 `CourseSession` 和 `SummerCourseSession` 而言, 唯一的差异是学期长度, 用独立方法 `getSessionLength` 来处理两者之间的差异。类 `CourseSession` 提供一个 `getSessionLength` 的实现, `SummerCourseSession` 提供另外一个实现。

`CourseSession` 中的方法 `getEndDate` 在行为上像一个模板。该方法提供了计算学期结束日期的绝大部分算法。其余的小段算法是可插拔的, 也就是说这一部分算法可以在其它地方出现,

¹ UML 规定用减号 (-) 表示 `private` 访问控制权限, 用波浪号 (~) 表示包访问控制权限。

例如可以在子类中提供这一小段算法。`getEndDate` 中的模板算法是模板方法⁴（一种设计模式）的一个例子。

217

深入构造函数

前面，我们在子类中使用关键字 `super` 调用父类的构造函数。调用父类构造函数的代码必须出现在子类构造函数的第一行。

子类可以对父类进行扩展。您可以把子类对象想象成一个包裹层、或者外壳，里面包裹着父类对象。在子类对象构造完成之前，Java 必须正确地初始化父类对象。接下来的语言测试中，用 `SuperClass` 和其子类 `SubClass` 来展示对象的初始化过程。

```
// SuperClassTest.java
import junit.framework.TestCase;

public class SuperClassTest extends TestCase {
    public void testConstructorCalls() {
        SuperClass superClass = new SubClass();
        assertTrue(SuperClass.constructorWasCalled);
    }
}

// SuperClass.java
class SuperClass {
    static boolean constructorWasCalled = false;

    SuperClass() {
        constructorWasCalled = true;
    }
}

// SubClass.java
class SubClass extends SuperClass {
    SubClass() {
    }
}
```

Java 虚拟机要求每个类至少有一个构造函数。如果没有显式地定义构造函数，那么 Java 会自动替您生成一个默认的、没有任何参数的构造函数。

如果子类的构造函数中没有 `super` 调用，那么 Java 会自动插入一个没有参数的父类的构造函数。即使子类的构造函数有参数，Java 也会调用父类的没有参数的构造函数。

```
// SuperClassTest.java
import junit.framework.TestCase;
public class SuperClassTest extends TestCase {
```

218

⁴ [Gamma1995].


```

    public void testConstructorCalls() {
        SuperClass superClass = new SubClass("parm");
        assertTrue(SuperClass.constructorWasCalled);
    }
}

// SubClass.java
class SubClass extends SuperClass {
    SubClass(String parm) {
    }
}

```

测试代码中传入 `String` 作为参数，创建了一个 `SubClass` 实例。参数用黑体显示。测试结果表明在创建 `SubClass` 实例的过程中，执行了 `SuperClass` 的无参数的构造函数。

请注意，如有类中显式地定义了构造函数，那么 `Java` 不会再自动提供无参数的默认构造函数。下面的例子中，所有子类必须显式调用父类的构造函数，否则会导致编译错误。

```

// This code will not compile
// SuperClass.java
class SuperClass {
    static boolean constructorWasCalled = false;

    SuperClass(String parm) {
        constructorWasCalled = true;
    }
}

// SubClass.java
class SubClass extends SuperClass {
    SubClass(String parm) {
    }
}

```

上面的代码会导致编译错误：

```

SubClass.java: cannot find symbol
symbol : constructor SuperClass()
location: class studentinfo.SuperClass
    SubClass(String parm) {

```

正确的子类的构造函数应该是：

```

class SubClass extends SuperClass {
    SubClass(String parm) {
        super(parm);
    }
}

```

继承和多态

类 `SummerCourseSession` 继承自 `CourseSession`。就像类实现某个接口，子类继承父类意味着子类也是父类类型。所以，`SummerCourseSession` 也是 `CourseSession`。

就像前面所讨论的，最好使用接口作为引用类型，对于基类和子类，也最好使用基类作为引用类型。把 `SummerCourseSession` 赋值给 `CourseSession` 类型的引用：

```
CourseSession session = new SummerCourseSession();
```

即使变量 `session` 的类型是 `CourseSession`，但是该变量在内存中指向 `SummerCourseSession`。

我们在 `CourseSession` 和 `SummerCourseSession` 中都定义了方法 `getEndDate`。使用变量 `session` 发送消息 `getEndDate`，`SummerCourseSession` 会接收到这个消息。Java 执行 `SummerCourseSession` 中定义的 `getEndDate`。

客户认为消息发送给了 `CourseSession`，但是 `CourseSession` 某个子类的对象接收并执行这个消息。这是多态的另一种形式。

子合约原则

客户使用 `CourseSession` 类型的变量发送消息，期望按照某种方式被解释和执行。在 TDD 中，测试用例定义了解释和执行的方式。测试用例首先描述了通过某个类的接口与该类进行交互的一系列行为，然后确保若干后置条件为真。

`CourseSession` 的子类，不必改变期望的行为。`CourseSession` 的任何要求为真的后置条件，同样适用于 `CourseSession` 的子类。总而言之，子类应该保留或者增加后置条件。子类扩展了父类的行为，也意味着针对子类的测试，要相应地增加后置条件。

TDD 要求在父类级别上，用测试用例的形式定义合约。所有子类必须遵守这些合约例。您可以用抽象测试⁵（一种设计模式）来构建这些合约。

◀ 220

从技术上讲，并不是必需进行这样的重构。对于 `CourseSession` 和 `SummerCourseSession`，即便存在较小的差异，`CourseSession` 的测试也能保持相同的合约。`CourseSession` 显式地跟踪创建实例的数目，而 `SummerCourseSession` 不必进行跟踪。

创建基类 `Session`，会使继承关系更加清晰和易于理解。概念上讲，`CourseSession` 和 `SummerCourseSession` 在继承关系上处于相同的层次。

是否值得进行这样的重构，取决于您。如果重构可以带来更简单的解决方案，那么继续。如果只是增加了没有任何好处的人造类，那就纯粹是浪费时间。

在 `CourseSession` 这个例子中，您应该重构 `CourseSession` 和 `SummerCourseSession`，让这两个类继承自同一个抽象基类，而不是 `SummerCourseSession` 继承自 `CourseSession`。见图 6.4。

⁵ [George2002].

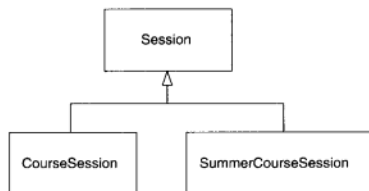


图 6.4 Session 继承层次

创建抽象测试类 `SessionTest`，该类定义了 `Session` 及其子类的基本单元测试。把 `testCreate`、`testComparable` 和 `testEnrollStudents` 从 `CourseSessionTest` 移动到 `SessionTest`。这些测试是所有 `Session` 子类都必须共同遵守的合约。

`SessionTest` 包含了一个抽象工厂方法 `createSession`。`SessionTest` 的子类中提供了 `createSession` 的实现，以返回适当类型的对象（`CourseSession` 或者 `SummerCourseSession`）。

```

package sis.studentinfo;

import junit.framework.TestCase;
import java.util.*;
import static sis.studentinfo.DateUtil.createDate;

abstract public class SessionTest extends TestCase {
    private Session session;
    private Date startDate;
    public static final int CREDITS = 3;

    public void setUp() {
        startDate = createDate(2003, 1, 6);
        session = createSession("ENGL", "101", startDate);
        session.setNumberOfCredits(CREDITS);
    }

    abstract protected Session createSession(
        String department, String number, Date startDate);

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

    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));
}

public void testComparable() {
    final Date date = new Date();
    Session sessionA = createSession("CMSC", "101", date);
    Session sessionB = createSession("ENGL", "101", date);
    assertTrue(sessionA.compareTo(sessionB) < 0);
    assertTrue(sessionB.compareTo(sessionA) > 0);

    Session sessionC = createSession("CMSC", "101", date);
    assertEquals(0, sessionA.compareTo(sessionC));

    Session sessionD = createSession("CMSC", "210", date);
    assertTrue(sessionC.compareTo(sessionD) < 0);
    assertTrue(sessionD.compareTo(sessionC) > 0);
}
}

```

222

您必须修改继承自 SessionTest 的测试子类: CourseSessionTest 和 SummerCourseSessionTest。
图 6.5 用 UML 草图描述了重构后的测试。

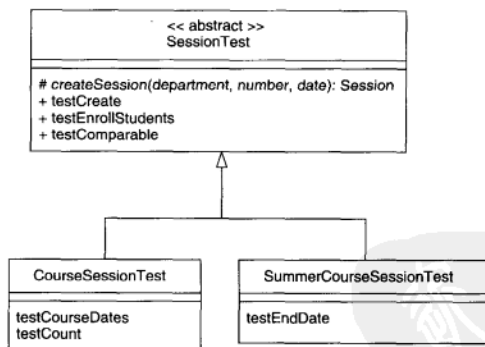


图 6.5 抽象测试

每个测试子类都需要提供 createSession 的实现。

```

// SummerCourseSessionTest.java
package sis.summer;

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

public class SummerCourseSessionTest extends SessionTest {

```

```

public void testEndDate() {
    Date startDate = DateUtil.createDate(2003, 6, 9);
    Session session = createSession("ENGL", "200", startDate);
    Date eightWeeksOut = DateUtil.createDate(2003, 8, 1);
    assertEquals(eightWeeksOut, session.getEndDate());
}

protected Session createSession(
    String department,
    String number,
    Date date) {
    return SummerCourseSession.create(department, number, date);
}
}

```

在 CourseSessionTest 中，将方法 createCourseSession 替换成 createSession。

```

// CourseSessionTest.java
package sis.studentinfo;

import junit.framework.TestCase;
import java.util.*;
import static sis.studentinfo.DateUtil.createDate;

public class CourseSessionTest extends SessionTest {
    public void testCourseDates() {
        Date startDate = DateUtil.createDate(2003, 1, 6);
        Session session = createSession("ENGL", "200", startDate);
        Date sixteenWeeksOut = createDate(2003, 4, 25);
        assertEquals(sixteenWeeksOut, session.getEndDate());
    }

    public void testCount() {
        CourseSession.resetCount();
        createSession("", "", new Date());
        assertEquals(1, CourseSession.getCount());
        createSession("", "", new Date());
        assertEquals(2, CourseSession.getCount());
    }

    protected Session createSession(
        String department,
        String number,
        Date date) {
        return CourseSession.create(department, number, date);
    }
}

```

方法 testCourseDates 中创建了 CourseSession 类型的局部实例。这样，SessionTest 的 setUp 方法中创建的 CourseSession 对象就不再有效。直接在子类的测试方法中创建 CourseSession 对象会使测试更具可读性。

现在，CourseSessionTest 和 SummerCourseSessionTest 都提供了抽象方法 createSession 的实现。

请注意，最重要的是 CourseSessionTest 和 SummerCourseSessionTest 都继承自 SessionTest，

所以它们共享 `SessionTest` 中的所有测试。JUnit 会将 `SessionTest` 中的每一个测试都执行两次：一次在 `CourseSessionTest` 实例中执行，另一次在 `SummerCourseSessionTest` 实例中执行。这样保证了 `Session` 的所有子类都拥有正确的行为。

◀ 224

JUnit 执行类 `SessionTest` 的方法时，会导致对 `createSession` 的调用。由于 `SessionTest` 中的 `createSession` 是抽象方法，并没有具体的实现，所以 Java 虚拟机调用相应测试子类的 `createSession`。 `createSession` 创建某个 `Session` 子类的实例：`CourseSession` 或者 `SummerCourseSession`。

重构生产类：

```
package sis.studentinfo;

import java.util.*;

abstract public class Session implements Comparable<Session> {
    private static int count;
    private String department;
    private String number;
    private List<Student> students = new ArrayList<Student>();
    private Date startDate;
    private int numberOfCredits;

    protected Session(
        String department, String number, Date startDate) {
        this.department = department;
        this.number = number;
        this.startDate = startDate;
    }

    public int compareTo(Session that) {
        int compare =
            this.getDepartment().compareTo(that.getDepartment());
        if (compare != 0)
            return compare;
        return this.getNumber().compareTo(that.getNumber());
    }

    void setNumberOfCredits(int numberOfCredits) {
        this.numberOfCredits = numberOfCredits;
    }

    public String getDepartment() {
        return department;
    }

    public String getNumber() {
        return number;
    }

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

    public void enroll(Student student) {
```

◀ 225

```

        student.addCredits(numberOfCredits);
        students.add(student);
    }

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

    protected Date getStartDate() {
        return startDate;
    }

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

    abstract protected int getSessionLength();

    public Date getEndDate() {
        GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(getStartDate());
        final int daysInWeek = 7;
        final int daysFromFridayToMonday = 3;
        int numberOfDays =
            getSessionLength() * daysInWeek - daysFromFridayToMonday;
        calendar.add(Calendar.DAY_OF_YEAR, numberOfDays);
        return calendar.getTime();
    }
}

```

类 `Session` 的大部分代码来自 `CourseSession`。`getSessionLength` 成为抽象方法，强制每个子类都实现该方法，并提供以周为单位的学期长度。下面是 `CourseSession` 和 `SummerCourseSession`，两者都继承自 `Session`：

```

// CourseSession.java
package sis.studentinfo;

import java.util.*;

public class CourseSession extends Session {
    private static int count;
    public static CourseSession create(
        String department,
        String number,
        Date startDate) {
        return new CourseSession(department, number, startDate);
    }

    protected CourseSession(
        String department, String number, Date startDate) {
        super(department, number, startDate);
        CourseSession.incrementCount();
    }

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

```

```

static void resetCount() {
    count = 0;
}

static int getCount() {
    return count;
}

protected int getSessionLength() {
    return 16;
}
}

// SummerCourseSession.java
package sis.summer;

import java.util.*;
import sis.studentinfo.*;

public class SummerCourseSession extends Session {
    public static SummerCourseSession create(
        String department,
        String number,
        Date startDate) {
        return new SummerCourseSession(department, number, startDate);
    }

    private SummerCourseSession(
        String department,
        String number,
        Date startDate) {
        super(department, number, startDate);
    }

    protected int getSessionLength() {
        return 8;
    }
}

```

227

子类中包含了构造函数、静态方法、以及重载的模板方法。构造函数不能继承。通用方法，例如 `enroll` 和 `getAllStudents`，只出现在父类 `Session` 中。子类的通用成员变量也只出现在父类 `Session` 中。

测试方法 `testCourseDates` 同时出现在 `SessionTest` 和它的子类中。您可以在 `SessionTest` 中定义一个通用的断言，但是这样的断言应该如何定义呢？一个简单的断言是：结束时间必须晚于开始时间。子类中通过验证某个指定的日期，从而增强这个断言。

另一种方法是断言结束日期等于开始日期加上正确的周数，再减去相应的周末天数。这样导致测试代码和 `getEndDate` 重复了相同的逻辑。

一个更有价值的做法是：在抽象测试中，对方法 `getSessionLength` 进行断言：

```

public void testSessionLength() {
    Session session = createSession(new Date());
    assertTrue(session.getSessionLength() > 0);
}

```


通过在抽象类中设置后置条件，从而建立了所有子类都必须遵守的子合约。
这次重构进行了大量的工作。希望您以渐增的方式，花些时间逐步改进您的代码。

练习

1. 第五课的练习中，您编写了根据棋子类型和棋盘位置来计算棋子大小的方法。用 `switch` 替代这个方法中的 `if` 语句。
2. 修改计算棋子大小的方法，使用 `Map` 在棋子类型和棋子权值之间建立映射。为了在 `Map` 中存储 `double` 类型的数值，需要将 `Map` 声明成 `Map<Piece.Type, Double>`。⁶
3. 使用延迟初始化，在需要的时候加载棋子类型和棋子大小之间的映射表。注意这样对代码可读性的影响。什么因素驱使您使用延迟初始化？
4. 把棋子对应的分值移动到某个枚举类型中，调用该枚举类型的某个方法来获取相应的分值。
5. 把表示棋子的字母移动到某个枚举类型中。棋盘仍然用大写字母表示黑方。
6. 实现国王在不同方向上的移动操作。基本规则是国王一次可以往任何方向走一格（暂时忽略相邻棋子可能会限制国王的移动，也忽略相邻棋子可能被国王吃掉）。
7. 类 `Board` 中的代码越来越复杂。`Board` 负责管理一个网格，`Board` 负责存储所有的棋子，`Board` 决定棋子的移动是否有效（迄今为止，只处理国王），`Board` 管理棋子并决定它们的大小关系。如果把其它棋子（王后、卒、车、相、马）的移动逻辑也添加到 `Board`，那么类 `Board` 会在逻辑上变得十分混乱。

有多种分解问题的方法。按照我的观点，目前类 `Board` 实现了两个主要功能。第一，用列表来存储棋子，实现了 `8x8` 的棋盘。类 `Board` 中的很多代码用来操作该列表类型的数据结构，而和象棋规则无关。第二，类 `Board` 定义了象棋的游戏规则。

将类 `Board` 分解成两个类：表示象棋游戏规则的游戏，存储棋子并且理解棋盘布局的 `Board`。

这是规模比较大的一次重构。请尽可能以渐增的方式来移动代码，保证测试在每次移动之后都是绿色的。如果必要，在重构的时候保留两份相同的代码，当一切都工作正常之后再删除多余的那份代码。而且，确保增加必要的测试。

借此这个机会，封装类 `Board` 的实现细节。在类 `Game` 中调用类 `Board` 的 `put` 方法，而不用请求获取棋子等级、添加棋子。

8. 王后一次可以沿着一条直线移动任意个格子。实现王后的移动操作。注意：在原始实现中，要求使用 `if` 语句来判断棋子是国王还是王后。

⁶ 第7课，在“包裹类型”一节解释了为什么需要这样声明。

实现中可能含有递归——方法内部调用该方法自身。您必须小心提供可以终止递归的条件，否则将陷入无限循环。

229

9. 实现国王和王后移动操作的代码中包含了一个 `if` 语句。可以想象，如果进一步支持其它棋子的移动操作，将会导致一簇复杂的条件语句。为了消除复杂的条件语句，请创建棋子的继承层次关系，以 `Piece` 作为基类，`Piece` 有若干子类，例如 `King`、`Queen`、`Bishop`，等等。

在创建 `Piece` 的子类之前，把方法 `getPossibleMoved` 移动到基类 `Piece`。

10. 现在，开始创建 `Piece` 的子类。创建 `Piece` 的子类 `Queen` 和 `King`，您会发现枚举 `Piece.Type` 变得多余了。下个练习中，将清除这个枚举。把相关的测试从 `PieceTest` 移动到 `QueenTest` 和 `KingTest`。然后，根据棋子类型改变类厂方法。

重构代码，消除冗余。如果因为测试方法被移动到新类，从而导致缺失了某些测试，就请增加这些测试。

11. 最后一步是清除枚举。既然针对每一种棋子都创建了子类，那么您可以把与棋子类型相关的代码移动到相应的子类中去。

您可能需要测试子类的类型。但是无需获取每种棋子的 `Piece.Type`，可以简单地调用下面的方法：

```
piece.getClass()
```

然后在断言中进行比较：

```
Class expectedClass = Queen.class;
assertEquals(expectedClass, piece.getClass());
```

而且，您应该尽可能清除请求对象类型的代码——这是创建子类的主要原因之一。看看您是否可以把诸如 `if (piece.getClass().equals(Pawn.class))` 的代码移动到适当的类中。

230

遗留元素

这一课，您将了解 Java 遗留元素。Java 是一种不断进化和发展中的语言。早期 Java 不论语言特性，还是类库设计，都存在不成熟的地方。随着时间的推移，Sun 在 Java 中加入了很多旨在提高健壮性的语言特性和类库。同时，Sun 仅仅删除了很少的不成熟的语言特性和类库，结果导致 Java 中包含了很多 Sun 建议您不要继续使用的功能。

J2SE5.0 引入了很多新的语言元素，其中一些是为了替代现有的 Java 元素。举一个最有代表性的例子，Java 设计者在 J2SE5.0 中改变了对集合进行迭代的方式。从前，Java 通过结合类库和过程式循环结构来支持迭代。现在，Java 语言直接通过 `for-each` 循环支持迭代。

本课中，您将了解源自 Java 语法祖先——C 语言的很多元素。这些元素本质上是过程式的，但是对于功能完善的编程语言而言，这些元素依然是必须的。即便如此，大多数时候您还是应该用面向对象来替代本课中介绍的这些遗留元素。

虽然对于这些 Java 遗留元素的使用我并不重视，但是您必须理解它们，然后才能完全掌握 Java。这些 Java 遗留元素依然是 Java 语言的基础。在某些情况下，您必须使用这些遗留元素，这些遗留元素提供了针对某些问题的最佳解决方案。

本课内容包括：

- `for`、`while` 和 `do` 循环
- 循环控制语句
- 遗留的集合元素
- 迭代器
- 强制类型转换
- 包装类
- 数组
- 可变参数

所有这些元素，除了可变参数以外，在 Java 出现以前都已经存在。集合、迭代器、以及包

装类都是面向对象的结构。其它元素都源自 C 语言。第 6 课中学到的 switch 语句同样也是源自 C 语言的遗留元素。

循环结构

通过 for 循环，可以迭代访问集合中的每一个元素。

此外，还有更通用的循环。或许，您需要执行一个无限循环的代码段，直到满足某个条件，或者将某条消息发送十次为止。Java 使用三种循环结构（while、do 和 for）来满足这些需求。

分解学生全名



现在，类 Student 使用单一字符串作为参数来构建 Student 对象，该字符串表示学生的全名。您需要修改类 Student，把表示学生全名的字符串分解成若干部分。例如，类 Student 把全名字符串 “Robert Cecil Martin” 分解为名 “Robert”、中名 “Cecil”，以及姓 “Martin”。

类 Student 假定：如果字符串可以分解为两个部分，那么分别表示名和姓。如果字符串只能分解成一个部分，那么表示姓。

首先，修改 StudentTest 中的方法 testCreate：

```
public void testCreate() {
    final String firstStudentName = "Jane Doe";
    Student firstStudent = new Student(firstStudentName);
    assertEquals(firstStudentName, firstStudent.getName());
    assertEquals("Jane", firstStudent.getFirstName());
    assertEquals("Doe", firstStudent.getLastName());
    assertEquals("", firstStudent.getMiddleName());
    final String secondStudentName = "Blow";
    Student secondStudent = new Student(secondStudentName);
    assertEquals(secondStudentName, secondStudent.getName());
    assertEquals("", secondStudent.getFirstName());
    assertEquals("Blow", secondStudent.getLastName());
    assertEquals("", secondStudent.getMiddleName());

    final String thirdStudentName = "Raymond Douglas Davies";
    Student thirdStudent = new Student(thirdStudentName);
    assertEquals(thirdStudentName, thirdStudent.getName());
    assertEquals("Raymond", thirdStudent.getFirstName());
    assertEquals("Davies", thirdStudent.getLastName());
    assertEquals("Douglas", thirdStudent.getMiddleName());
}
```

在类 Student 中加入成员变量和相应的 getter 方法：

```

private String firstName;
private String middleName;
private String lastName;
...
public String getFirstName() {
    return firstName;
}

public String getMiddleName() {
    return middleName;
}

public String getLastName() {
    return lastName;
}

```

修改 Student 构造函数的参数命名，从而表明您的意图：

```

public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    setName(nameParts);
}

```

面向意图编程是一种有用的技术，它可以帮助您在编码之前规划出所需要做的内容¹。面向意图编程意味着把某个问题分解成若干更小的抽象。在这里，您知道需要把全名分解成若干不同的部分，但是您还不知道如何去分解。

233



使用面向意图编程，把某个问题分解成若干更小的目标。

一旦弄清楚如何将全名字符串分解成三个姓名部分，相应的实现方法 setName 就显而易见了。方法 setName 中包含几个 if-else 语句。每个 if 语句测试全名字符串被分解成了几个部分。

```

private void setName(List<String> nameParts) {
    if (nameParts.size() == 1)
        this.lastName = nameParts.get(0);
    else if (nameParts.size() == 2) {
        this.firstName = nameParts.get(0);
        this.lastName = nameParts.get(1);
    }
    else if (nameParts.size() == 3) {
        this.firstName = nameParts.get(0);
        this.middleName = nameParts.get(1);
        this.lastName = nameParts.get(2);
    }
}

```

¹ [Astels2003], p.45.

方法 `setName` 无论在什么情况下，都给变量 `lastName` 赋值。但是，并不总是赋值给变量 `firstName` 和 `middleName`。您需要确保类 `Student` 对这三个变量都进行了正确的初始化。

```
private String firstName = "";
private String middleName = "";
private String lastName;
```

While 循环

对于分解全名字符串这个问题，Java 提供了几种方法。首先，我们使用更经典、也更乏味的方法：遍历全名字符串中的每一个字符，找到空格字符，使用空格作为姓名分解的分隔符。稍后，您会看到两种更简单、也更面向对象的方法。

方法 `split` 以全名字符串作为参数，返回一个包含全名字符串各部分的列表。使用 `while` 循环进行遍历，以空格作为上个部分的结束和下个部分的开始，并把分解后的每个部分添加到结果列表。

```
private List<String> tokenize(String string) {
    List<String> results = new ArrayList<String>();

    StringBuffer word = new StringBuffer();
    int index = 0;
    while (index < string.length()) {
        char ch = string.charAt(index);
        if (ch != ' ') // prefer Character.isSpace. Defined yet?
            word.append(ch);
        else
            if (word.length() > 0) {
                results.add(word.toString());
                word = new StringBuffer();
            }
        index++;
    }
    if (word.length() > 0)
        results.add(word.toString());
    return results;
}
```

使用 `while` 循环，只要条件为真，就反复执行循环体中的代码段。

本例中，Java 将计数器 `index` 初始化为 0，然后开始执行 `while` 循环。只要 `index` 的值小于字符串长度 (`index < string.length()`)，Java 就会反复执行 `while` 循环体（紧随 `while` 循环条件之后，在花括号对之间的所有代码）。

循环体根据当前索引，从字符串中取出相应的字符。然后测试字符是否为空格（' '）。如果不是空格，字符被追加到当前 `StringBuffer` 实例。如果是空格并且当前 `StringBuffer` 实例中有内容，那么当前 `StringBuffer` 实例中的内容是一个完整的单词，把这个单词添加到结果列表中。然

后创建一个代表新单词的 `StringBuffer` 实例，并将其赋值给变量 `word`。

每当循环体执行完毕，控制转移到 `while` 循环条件。如果条件表达式返回 `false`，那么中止 `while` 循环，控制转移到 `while` 循环体的后一行代码。本例中，如果循环条件为 `false`，那么控制转移到紧随 `while` 循环体之后的 `if` 语句，该语句确保分解出来的最后一个单词被添加到结果列表中。

修改后的测试可以通过。

重构

方法 `setName` 存在冗余。因为全名字符串分解后的每个部分都存储在列表中，而且您可以操作该列表，根据这个条件对方法 `setName` 进行重构。

```
private void setName(List<String> nameParts) {
    this.lastName = removeLast(nameParts);
    String name = removeLast(nameParts);
    if (nameParts.isEmpty())
        this.firstName = name;
    else {
        this.middleName = name;
        this.firstName = removeLast(nameParts);
    }
}

private String removeLast(List<String> list) {
    if (list.isEmpty())
        return "";
    return list.remove(list.size() - 1);
}
```

235

重构后的代码比以前更多了。但是，您消除了重复，清除了方法 `setName` 中对索引的硬编码。而且，创建了一个有用的通用方法 `removeLast`。

for 循环

从前面的例子可以看出，在循环使用计数器和限制条件，是一种通用的操作。`for` 循环提供了更加简洁的方式。`for` 循环允许您将三个通用循环部件组织在一起，每个 `for` 循环声明包括：

- 初始化代码
- 条件表达式，用来决定是否继续执行循环体
- 每次循环结束时，Java 虚拟机执行一个更新表达式

在 `for` 循环声明中，使用分号把声明分隔成三个部分。

for 循环的经典用法是循环体执行若干确定的次数。通常，但非必须，初始化代码将计数器设置为 0，在条件表达式中测试计数器是否小于某个上限，最后更新表达式将计数器增加 1。

```
private List<String> split(String name) {
    List<String> results = new ArrayList<String>();

    StringBuffer word = new StringBuffer();
    for (int index = 0; index < name.length(); index++) {
        char ch = name.charAt(index);
        if (!Character.isWhitespace(ch))
            word.append(ch);
        else
            if (word.length() > 0) {
                results.add(word.toString());
                word = new StringBuffer();
            }
    }
    if (word.length() > 0)
        results.add(word.toString());
    return results;
}
```

上面的例子中，初始化代码将 `index` 定义为 `int` 类型，并且将其初始化为 0。条件表达式测试 `index` 是否小于某个字符串的长度。只要条件表达式的结果为 `true`，Java 执行循环体。每次执行完循环体之后，Java 将 `index` 的值增加 1 (`index++`)；然后把控制转移到条件表达式，一旦条件表达式返回 `false`，Java 将控制转移到紧跟 for 循环体的后面一行代码。

该例中，针对输入字符串中的每一个字符，Java 执行一次循环体。计数器 `index` 的最小值是 0，最大值是输入字符串的长度减 1。

for 循环中的多重初始化与更新表达式

初始化和更新表达式，都允许使用逗号隔开的多重表达式。方法 `countChars`² 返回字符串中某个特定字符出现的次数，该方法展示了如何将变量 `i` 和 `count` 都初始化为 0。

```
public static int countChars(String input, char ch) {
    int count;
    int i;
    for (i = 0, count = 0; i < input.length(); i++)
        if (input.charAt(i) == ch)
            count++;
    return count;
}
```

将字母 `i` 作为循环计数器是十分常见的习惯用法。这是允许对变量名进行缩写、可接受的、

² 该方法，以及随后的几个方法，例如 `isPalindrome`，都和学生信息系统没有任何关系。这些方法只是用来解释 Java 语法的某些细微之处。

少有的几种标准之一。多数程序员倾向于用 `i` 来代替全名 `index`。

下面的方法 `isPalindrome`，同时从字符串的头部和尾部依次取出字符，如果取出的字符相同，那么返回 `true`。该方法向您展示如何在初始化部分同时初始化一个以上的变量，在更新表达式的时候，如何增加变量 `forward` 的同时，减小变量 `backward`。

```
public static boolean isPalindrome(String string) {
    for
        (int forward = 0, backward = string.length() - 1;
         forward < string.length();
         forward++, backward--)
        if (string.charAt(forward) != string.charAt(backward))
            return false;
    return true;
}

// tests
public void testPalindrome() {
    assertFalse(isPalindrome("abcdef"));
    assertFalse(isPalindrome("abccda"));
    assertTrue(isPalindrome("abccba"));
    assertFalse(isPalindrome("abcxba"));
    assertTrue(isPalindrome("a"));
    assertTrue(isPalindrome("aa"));
    assertFalse(isPalindrome("ab"));
    assertTrue(isPalindrome(""));
    assertTrue(isPalindrome("aaa"));
    assertTrue(isPalindrome("aba"));
    assertTrue(isPalindrome("abbba"));
    assertTrue(isPalindrome("abba"));
    assertFalse(isPalindrome("abbbaa"));
    assertFalse(isPalindrome("abcda"));
}
```

在 `for` 循环的初始化代码中声明的局部变量，只在 `for` 循环的范围中有效。所以，在方法 `countChars` 中，由于在循环之后返回 `count`，因此您必须在循环的前面声明变量 `count`。

Java 规定“如果存在一个声明局部变量的表达式，并且表达式中存在逗号，那么逗号分割的每个部分都被视为局部变量声明”³。所以必须修改 `countChars`，声明 `i` 为局部变量并且只对 `count` 进行初始化。Java 认为下面的初始化代码会将 `i` 和 `count` 同时声明为局部变量。

```
// this code will not compile!
public static int countChars(String input, char ch) {
    int count;
    for (int i = 0, count = 0; i < input.length(); i++)
        if (input.charAt(i) == ch)
            count++;
    return count;
}
```

³ [Arnold2000].

由于已经在 for 循环的前面声明了变量 count，所以 Java 编译器会报错：

```
count is already defined in countChars(java.lang.String,char)
```

声明 for 循环的所有三个部分——初始化、条件表达式、更新表达式，都是可选的。如果忽略条件表达式，那么 Java 会无限次地执行循环体。下面的 for 循环是无限循环的惯用法：

```
for (;;) {  
}
```

更有表现力的、创建无限循环的方法是：使用 while 循环：

```
while (true) {  
}
```

前面提到过，for 循环的通常用法是将计数器的范围限定为 0 到 n-1 之间，n 是某个集中元素的个数。但是，没有什么可以限制您使用不同的方式来创建 for 循环。方法 isPalindrome 展示了如何在增加某个计数器的同时，减小另一个计数器。

下面的语言测试表明在更新表达式中，不止可以对计数器使用加 1 或者减 1 的操作：

```
public void testForSkip() {  
    StringBuilder builder = new StringBuilder();  
    String string = "123456";  
    for (int i = 0; i < string.length(); i += 2)  
        builder.append(string.charAt(i));  
    assertEquals("135", builder.toString());  
}
```

do 循环

do 循环工作起来和 while 循环一样，只有一点不同：do 循环在循环体结束的地方测试条件表达式。使用 do 循环可以保证循环体至少被执行一次。

十三世纪，菲波那契提出用来表示兔子繁殖⁴的菲波那契数列。该数列以 0 和 1 开始，后面的每个数字都是该数列中前两个数字之和。下面的代码和测试展示了使用 do 循环的菲波那契实现。

```
public void testFibonacci() {  
    assertEquals(0, fib(0));  
    assertEquals(1, fib(1));  
    assertEquals(1, fib(2));  
    assertEquals(2, fib(3));  
    assertEquals(3, fib(4));  
    assertEquals(5, fib(5));  
}
```

⁴ [Wikipedia2004].

```

    assertEquals(8, fib(6));
    assertEquals(13, fib(7));
    assertEquals(21, fib(8));
    assertEquals(34, fib(9));
    assertEquals(55, fib(10));
}

private int fib(int x) {
    if (x == 0) return 0;
    if (x == 1) return 1;
    int fib = 0;
    int nextFib = 1;
    int index = 0;
    int temp;
    do {
        temp = fib + nextFib;
        fib = nextFib;
        nextFib = temp;
    } while (++index < x);
    return fib;
}

```

比较 Java 循环

以上三种循环方式都可以满足编程中任何循环处理的要求。接下来的测试和三个方法，使用三种不同的技术来显示一个用逗号分隔的数字列表。

```

public void testCommas() {
    String sequence = "1,2,3,4,5";
    assertEquals(sequence, sequenceUsingDo(1, 5));
    assertEquals(sequence, sequenceUsingFor(1, 5));
    assertEquals(sequence, sequenceUsingWhile(1, 5));

    sequence = "8";
    assertEquals(sequence, sequenceUsingDo(8, 8));
    assertEquals(sequence, sequenceUsingFor(8, 8));
    assertEquals(sequence, sequenceUsingWhile(8, 8));
}

String sequenceUsingDo(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    int i = start;
    do {
        if (i > start)
            builder.append(',');
        builder.append(i);
    } while (++i <= stop);
    return builder.toString();
}

String sequenceUsingFor(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    for (int i = start; i <= stop; i++) {
        if (i > start)

```

```

        builder.append(',');
        builder.append(i);
    }
    return builder.toString();
}

String sequenceUsingWhile(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    int i = start;
    while (i <= stop) {
        if (i > start)
            builder.append(',');
        builder.append(i);
        i++;
    }
    return builder.toString();
}

```

通过上面的例子，您会发现通常存在一种循环方式比另外两种更加合适。该例中，因为循环基于计数器，所以 for 循环是最佳的方式。

如果不需要维护计数器或者其它增量变量，那么 while 循环通常比较合适。多数情况下，每次进入循环体之前，需要测试某个条件，while 循环比较适用。很少用到 do 循环，因为在测试条件之前要求必须至少执行一次循环体的情况比较少见。

重构

isPalindrome

方法 isPalindrom 花费了两倍时间的代价，来判断某个字符串是否回文（字符串从前向后看，和从后向前看都是一样的）。尽管通常情况下，您不必考虑性能，除非出现了明显的性能问题。但是，一个算法必须是干净的。如果在算法中做了不必要的事情，那么说明您对问题的理解还不够充分，也会给将来的程序员带来混淆。

只需从字符串的两头遍历到字符串的中间。当前的实现是从两头开始，遍历了整个字符串，从而导致每个字符串都比较了两次。更好的实现是：

```

public static boolean isPalindrome(String string) {
    if (string.length() == 0)
        return true;
    int limit = string.length() / 2;
    for
        (int forward = 0, backward = string.length() - 1;
         forward < limit;
         forward++, backward--)
        if (string.charAt(forward) != string.charAt(backward))
            return false;
    return true;
}

```

菲波那契和递归

Java 支持递归方法调用。递归方法在函数体中调用方法本身。用递归来解决菲波那契数列问题要更合适一些：

```
private int fib(int x) {
    if (x == 0)
        return 0;
    if (x == 1)
        return 1;
    return fib(x - 1) + fib(x - 2);
}
```

使用递归一定要注意递归的终止条件。如果递归中没有终止条件，那么会导致无限递归调用。所以递归中需要防卫子句，就像方法 fib（有两个防卫子句）。

循环控制语句

Java 提供了多种在循环中改变控制流的方法：continue 语句、break 语句、标签 break 语句、标签 continue 语句。

242

break 语句

也许您希望提前终止某个循环。一旦 Java 虚拟机在循环体中遇见 break 语句，就会立即把控制流转移给紧跟循环体的后面一条语句。

类 String 中定义了方法 trim，该方法可以删除某个字符串两端的所有空白字符⁵。下面创建一个字符串处理方法，该方法只从字符串的末尾删除所有的空白字符。

```
public void testEndTrim() {
    assertEquals("", endTrim(""));
    assertEquals(" x", endTrim(" x "));
    assertEquals("y", endTrim("y"));
    assertEquals("xaxa", endTrim("xaxa"));
    assertEquals("", endTrim(" "));
    assertEquals("xxx", endTrim("xxx "));
}

public String endTrim(String source) {
    int i = source.length();
    while (i >= 0)
        if (source.charAt(i) != ' ')
            break;
    return source.substring(0, i + 1);
}
```

⁵ 包括空格字符、tab 键、换行、回车、换页。

continue 语句

当遇见 continue 语句，Java 立刻把控制流转移到循环开始处的条件表达式。



下面的例子，计算某门课程的兼职学生的平均学分。在 SessionTest 中创建测试：

```
public void testAverageGpaForPartTimeStudents() {
    session.enroll(createFullTimeStudent());

    Student partTimer1 = new Student("1");
    partTimer1.addGrade(Student.Grade.A);
    session.enroll(partTimer1);

    session.enroll(createFullTimeStudent());

    Student partTimer2 = new Student("2");
    partTimer2.addGrade(Student.Grade.B);
    session.enroll(partTimer2);
    assertEquals(3.5, session.averageGpaForPartTimeStudents(), 0.05);
}

private Student createFullTimeStudent() {
    Student student = new Student("a");
    student.addCredits(Student.CREDITS_REQUIRED_FOR_FULL_TIME);
    return student;
}
```

相应的实现（在类 Session 中）展示了如何利用 continue 语句来忽略所有的非兼职学生。

```
double averageGpaForPartTimeStudents() {
    double total = 0.0;
    int count = 0;
    for (Student student: students) {
        if (student.isFullTime())
            continue;
        count++;
        total += student.getGpa();
    }
    if (count == 0) return 0.0;
    return total / count;
}
```

您可以很容易地用 if-else 语句替代 continue 语句。但是，某些时候 continue 语句提供了更雅致、更可读的表达式。

标签 break 和 continue 语句

针对复杂嵌套，标签 break 和 continue 语句可以帮助您实现控制流转移。

下面的例子测试标签 `break`。不要让有趣的 `table` 声明吓着了您。`List<List<Integer>>` 声明一个列表，并且允许向该列表中加入成员为 `Integer` 的列表。

```
public void testLabeledBreak() {
    List<List<String>> table = new ArrayList<List<String>>();

    List<String> row1 = new ArrayList<String>();
    row1.add("5");
    row1.add("2");
    List<String> row2 = new ArrayList<String>();
    row2.add("3");
    row2.add("4");

    table.add(row1);
    table.add(row2);
    assertTrue(found(table, "3"));
    assertFalse(found(table, "8"));
}

private boolean found(List<List<String>> table, String target) {
    boolean found = false;
    search:
    for (List<String> row: table) {
        for (String value: row) {
            if (value.equals(target)) {
                found = true;
                break search;
            }
        }
    }
    return found;
}
```

244

该例中，如果没有在 `break` 语句中指定标签 `search`，那么 Java 只会终止内部 `for` 循环（即：`Integer num : row`）。相反，`break` 语句会从标签 `search` 所在位置跳出循环，也就是跳出本例中的外部循环。

标签 `continue` 和标签 `break` 的工作方式几乎一样。唯一的区别在于：Java 虚拟机将控制流转移到标签所在位置，而不是跳出循环到标签所在位置。

很少会用到标签 `break` 和标签 `continue`。多数情况下，可以通过代码重构来消除标签 `break` 和标签 `continue`，或者使用 `if-else` 语句，或者更好的情况是把方法分解成更小、更紧凑的子方法。只有在有助于更好地理解代码的情况下，才使用标签语句。

三元操作符

第 5 课中介绍了 `if` 语句。您经常需要把值赋给某个变量，或者将某个 `if` 条件的值作为方法的结果返回。例如：


```
String message =
    "the course has " + getText(sessions) + " sessions";
...
private String getText(int sessions) {
    if (sessions == 1)
        return "one";
    return "many";
}
```

对于必须返回新值的简单条件表达式，Java 提供了叫作“三元操作符”的简洁形式。三元操作符将 if-else 语句压缩成单行表达式。三元操作符的一般形式如下：

```
conditional ? true-value : false-value
```

如果 conditional 返回 true，那么整个表达式的结果为 true-value。否则整个表达式的结果为 false-value。使用三元操作符，可以重写前面生成消息的代码：

```
String message =
    "the course has " + (sessions == 1 ? "one" : "many") + " sessions";
```

如果 sessions 的值为 1，那么三元操作符返回字符串“one”，否则返回“many”。然后结果字符串和其它字符串串接起来，组成更大的字符串。

三元操作符源自 C 语言。通常不要用三元操作符代替 if 语句。三元操作符最好用在简单的、单行表达式中，就像例中所示。如果有更复杂的需求，或者代码超过一行，那么最好使用 if 语句。滥用三元操作符会导致晦涩的、高维护成本的代码。

遗留的集合类

在 JDK1.2 之前，Java 中没有集合框架。Java 包含了两个主要的集合类：java.util.Vector 和 java.util.Hashtable⁶，这两个类一直保留到现在。在最新的 Java 中，相对应的类分别是 java.util.ArrayList 和 java.util.HashMap。类 java.util.HashMap 提供了和类 java.util.EnumMap 相似的功能。参考第 9 课可以得到更多有关 HashMap 的资料。

Vector 和 Hashtable 都是具体的类，它们没有共同的接口。默认情况下，它们一起工作可以支持多线程编程⁷，但是经常会导致重复工作。不幸的是，今天您依然会看见大量使用 Vector 的代码。

由于 Java 的初始版本中没有这些类的相应接口，所以您不得不把 Vector 或者 Hashtable 的实例赋值给某个引用：

```
Vector names = new Vector();
```

⁶ 过去也有这两种集合类：BitSet 和 Stack（Vector 的子类），这两个类几乎都没有用处。

⁷ 参考 13 课有关多线程编程的讨论。

随着集合框架的引入, Sun 重构了类 `Vector` 和 `Hashtable`。类 `Vector` 实现接口 `List`, 类 `Hashtable` 实现接口 `Map`。或许因为某些原因, 您不得不继续使用 `Vector` 或者 `Hashtable`。如果这样, 您可以使用新的方法, 把实例赋值给某个接口引用:

```
List names = new Vector();
Map dictionary = new Hashtable();
```

而且, Sun 重构 `Vector` 和 `Hashtable`, 使之可以接受参数化类型。您可以这样编码:

```
List<String> names = new Vector<String>();
Map<Student.Grade,String> dictionary =
    new Hashtable<Student.Grade,String>();
```

迭代器

前面讲过如何使用 `for-each` 循环, 遍历集中的每一个元素。这种形式的 `for` 循环是 J2SE5.0 中最新引入的。

在引入 `for-each` 循环之前, 遍历集合的最佳方法是使用 `java.util.Iterator` 对象。迭代器对象维护一个指向集合元素的内部指针。可以请求迭代器返回集合的下一个可用元素, 在集合返回下一个元素之后, 迭代器将其内部指针加 1 从而指向集合的下一个元素。除此以外, 迭代器还可以判断集合中是否还有剩余的元素。

例如, 修改方法 `averageGpaForPartTimeStudents`, 在该方法中用迭代器替换 `for-each` 循环:

```
double averageGpaForPartTimeStudents() {
    double total = 0.0;
    int count = 0;

    for (Iterator<Student> it = students.iterator();
         it.hasNext(); ) {
        Student student = it.next();
        if (student.isFullTime())
            continue;
        count++;
        total += student.getGpa();
    }
    if (count == 0) return 0.0;
    return total / count;
}
```

通常, 您可以通过向集合发送消息 `iterator`, 从而得到一个迭代器, 然后把这个迭代器对象赋值给予集合中元素类型绑定在一起的某个迭代器引用。该例中, 把迭代器引用和集合存

储的 `Student` 元素类型绑定在一起。

接口 `Iterator` 定义了三个方法：`hasNext`、`next`，以及 `remove`（可选的，而且很少使用）。如果集合中存在没有被存取的元素，那么方法 `hasNext` 返回 `true`。方法 `next` 返回集合中的下一个元素，并且将迭代器的内部指针增加 1。

遗留集合 `Vector` 和 `Hashtable` 使用一种叫做 `enumeration` 的类似技术。这种方法和迭代器实质上是一样的，只是名字不同而已。在 `Session` 中将实例 `student` 重新定义成 `Vector` 类型。

```
package sis.studentinfo;

import java.util.*;

abstract public class Session
    implements Comparable<Session> {
    ...
    private Vector<Student> students = new Vector<Student>();

    double averageGpaForPartTimeStudents() {
        double total = 0.0;
        int count = 0;

        for (Enumeration<Student> it = students.elements();
             it.hasMoreElements(); ) {
            Student student = it.nextElement();
            if (student.isFullTime())
                continue;
            count++;
            total += student.getGpa();
        }
        if (count == 0) return 0.0;
        return total / count;
    }
}
```

248

相对于使用方法 `iterator` 来获取迭代器，您可以使用方法 `elements` 来获取一个枚举。相对于 `hasNext`，您可以使用 `hasMoreElements` 来测试是否还有可用的元素。相对于 `next`，您可以使用方法 `nextElement` 来获取下一个元素。大家认为类名 `Enumeration` 及其方法名比较长，所以 `Sun` 引入迭代器来消除大家的抱怨。

由于 `Vector` 实现了接口 `List`，所以您通过向 `Vector` 发送消息 `iterator`，从而获得一个迭代器。

把您的 `Session` 代码改回成使用新集合和 `for-each` 循环的形式。

迭代器和 for-each 循环

`for-each` 循环构建在迭代器之上。为了遍历集合，集合必须能够通过发送消息 `iterator`，以返回一个迭代器。`for-each` 循环通过检验集合是否实现了接口 `java.lang.Iterable`，从而判断

某个集合是否可以迭代。

在下面的例子中，修改类 `Session` 以支持对集合元素的迭代。

在 `SessionTest` 中编写代码：

```
public void testIterate() {
    enrollStudents(session);

    List<Student> results = new ArrayList<Student>();
    for (Student student: session)
        results.add(student);

    assertEquals(session.getAllStudents(), results);
}

private void enrollStudents(Session session) {
    session.enroll(new Student("1"));
    session.enroll(new Student("2"));
    session.enroll(new Student("3"));
}
```

重新编译，因为 `Session` 不支持迭代，所以会有下面的错误：

```
foreach not applicable to expression type
    for (Student student: session)
        ^
```

◀ 249

修改类 `Session`，让类 `Session` 实现接口 `Iterable`。因为您希望 `for-each` 循环中迭代器指向的集合只能包含 `Student` 类型的元素，所以必须把接口 `Iterable` 绑定到 `Student` 类型。

```
abstract public class Session
    implements Comparable<Session>, Iterable<Student> {
    // ...
}
```

现在，您只需在 `Session` 中实现方法 `iterator`。因为 `Session` 仅封装了保存 `Student` 对象的 `ArrayList`，所以方法 `iterator` 中只需要简单地返回 `ArrayList` 的迭代器对象。

```
public Iterator<Student> iterator() {
    return students.iterator();
}
```

类型转换

如果程序员知道对象的确切类型，但是编译器却不接受对象的类型，这种情况下就需要类型转换。通过类型转换，告诉编译器可以安全地把某种类型的对象赋值给其它类型的引用。

Sun 引入参数化类型，从而消除了 Java 中大多数的类型转换。在 J2SE5.0 之前，集合类不支持参数化类型，只有 Object 类型的引用可以插入集合，就像接口 List 的 add 方法所表现的：

```
public boolean add(Object o)
```

因为存入的元素都是 Object 类型，所以取出的元素类型也必须是 Object 类型：

```
public Object get(int index)
```

您，作为程序员，知道自己要把 Student 对象存入集合，但是集合只允许插入 Object 类型的引用。从集合中获取元素的时候，您知道取出的对象应该是 Student 类型。这意味着从集合中获取元素的时候，您必须把 Object 引用转换成 Student 引用。

为了进行类型转换，必须在被转换对象的前面插入一对括号，括号中是目标转换类型。

250

```
Student student = (Student)students.get(0);
```

该例中，被转换对象是表达式 students.get(0) 的返回值。在其两端加上括号，代码显得更加清楚一些：

```
(Student)(students.get(0));
```

请注意：被转换对象和目标转换类型之间没有空格。尽管可以在它们中间插入空格，但是最好不要这么做。

类型转换不会对被转换对象进行任何修改，理解这一点非常重要。对 Student 进行类型转换的结果，只是针对同一块内存区域创建了一个新的引用。因为这个新引用是 Student 类型，所以 Java 编译器允许您通过该引用发送 Student 相关的消息。没有类型转换，您只能发送定义在 Object 中的消息。

有了参数化类型，就没有必要再进行类型转换。如果发现自己的代码中存在类型转换，请检查是否可以使用参数化类型，以避免类型转换。

作为练习，下面的测试展示如何使用旧的方法：不使用 for-each 循环，不使用参数化类型。尽管代码在 J2SE5.0 中可以运行，但是由于没有使用参数化类型，所以您会看到警告信息。

```
public void testCasting() {
    List students = new ArrayList();
    students.add(new Student("a"));
    students.add(new Student("b"));

    List names = new ArrayList();

    Iterator it = students.iterator();
    while (it.hasNext()) {
        Student student = (Student)it.next();
```

```

        names.add(student.getLastName());
    }

    assertEquals("a", names.get(0));
    assertEquals("b", names.get(1));
}

```

对基本类型进行类型转换

不能对基本类型进行类型转换，将其赋值给其它类型的引用，反之亦然。

针对数字类型的转换，存在着另一种类型转换的方式。第 10 课中，会有相应的讨论。

◀ 251

包装类



学生信息系统需要把学费添加到某个集合，并计算总和。用整型变量表示每一笔费用。

基本类型不是对象——没有从类 `java.lang.Object` 继承。接口 `java.util.List` 只提供了以引用类型为参数的方法 `add`。基本类型不能重载 `add` 方法。

为了将整型数值存入集合，您必须将其转换成对象。通过将整型数值存入某个包装对象，可以完成这种转换。

对于每一种基本类型，`java.lang` 都定义了相应的包装类⁸。

基本类型	包装类
<code>char</code>	<code>Character</code>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>Boolean</code>	<code>Boolean</code>

每个包装类都提供了以相应基本类型作为参数的构造函数。可以向包装类中存储基本类型，并且包装类提供访问基本类型的 `getter` 方法。对于包装类 `Integer`，您可以通过向包装类发送消息 `intValue` 来访问原始的整型数值。

⁸ 表中列出了所有基本类型以及相对应的包装类。您将在第 10 课熟悉这些数据类型。

下面是 StudentTest 中的测试:

```
public void testCharges() {
    Student student = new Student("a");
    student.addCharge(500);
    student.addCharge(200);
    student.addCharge(399);
    assertEquals(1099, student.totalCharges());
}
```

在 J2SE5.0 以前, 相应的实现代码是这个样子的:

```
public class Student implements Comparable {
    ...
    private List charges = new ArrayList();
    ...
    public void addCharge(int charge) {
        charges.add(new Integer(charge));
    }

    public int totalCharges() {
        int total = 0;
        Iterator it = charges.iterator();
        while (it.hasNext()) {
            Integer charge = (Integer)it.next();
            total += charge.intValue();
        }
        return total;
    }
    ...
}
```

在方法 addCharge 中, 使用包装类 Integer 的实例来封装 int, 然后将该实例作为方法 add 的参数, 添加到集合 charges 中。在方法 totalCharges 中遍历集合的时候, 您必须把每一个取出的对象强制类型转换成 Integer 类型。只有通过方法 intValue, 才能取出原始的整型数值。

J2SE5.0 提供了参数化类型和 for-each 循环, 从而可以简化上面的代码。

```
public class Student implements Comparable<Student> {
    ...
    private List<Integer> charges = new ArrayList<Integer>();
    ...
    public void addCharge(int charge) {
        charges.add(new Integer(charge));
    }

    public int totalCharges() {
        int total = 0;
        for (Integer charge: charges)
            total += charge.intValue();
        return total;
    }
    ...
}
```

除了封装基本类型, 包装类也提供了很多可以用来操作基本类型的类方法。您在前面

已经用过 `Character` 的类方法 `isWhitespace`。

自动装箱是 J2SE5.0 引入的另一个新特性，利用该特性可以进一步简化代码。

自动装箱和自动拆箱

自动装箱的意思是：编译器自动调用相应的包装类，对基本类型进行封装。当 Java 发现类型不匹配的时候就会进行自动装箱。类型匹配意味着方法调用及其参数和类定义中的完全一致，其中如果方法调用的参数是类定义中参数的子类型，也被认为是类型匹配。

例如，类 `Box` 中定义了一个方法：

```
void add(List list) {...}
```

调用方法 `add`：

```
Box b = new Box();
b.add(new List());
b.add(new ArrayList());
```

因为 `ArrayList` 是 `List` 的子类，所以第二个 `add` 也可以工作。

如果 Java 发现没有直接的类型匹配，那么就试图寻找基于包装类的类型匹配。

在方法 `addCharge` 中，没有必要对 `int` 进行显式的封装：

```
public void addCharge(int charge) {
    charges.add(charge);
}
```

Java 自动在幕后为整型数值创建相应的 `Integer` 实例，理解这一点非常重要。

目前只可以对参数进行自动装箱。如果您试图给某个基本类型发送消息的时候，也可以自动装箱，那将会非常棒。但是 Java 目前还没有这个能力：

```
6.toString() // this does not work
```

254

当您试图访问被包装的基本类型的时候，会发生自动拆箱。下面的两个测试都是自动拆箱的例子：

```
public void testUnboxing() {
    int x = new Integer(5);
    assertEquals(5, x);
}

public void testUnboxingMath() {
    assertEquals(10, new Integer(2) * new Integer(5));
}
```


遍历某个集合，依次访问集合中存储的每个基本类型的包装对象，自动拆箱在这种场合下显得更加有用。通过自动装箱和自动拆箱，可以把 Student 中的代码改成下面这样：

```
public void addCharge(int charge) {
    charges.add(charge);
}

public int totalCharges() {
    int total = 0;
    for (int charge: charges)
        total += charge;
    return total;
}
```

在第 10 课中，您将了解到更多关于包装类的知识。

数组

本书中，已经多次提到数组这个概念。数组的长度固定，而且数组的每个元素所对应的内存地址也是连续的。和 ArrayList 以及其它集合对象不一样，数组可以被填满，填满以后不允许再增加任何新的元素。向填满的数组中增加元素的唯一途径是：创建另一个更大的数组，然后把所有的元素拷贝到这个更大的数组中去。

Java 对数组提供了特别的语法支持。实际上，您一直通过 ArrayList 在间接地使用数组。ArrayList 可以存储添加到 Java 数组中的所有元素。类 ArrayList 封装了这样的行为：当添加的元素超出范围的时候，ArrayList 会自动增加数组的长度。

下面的例子中，创建新类 Performance，CourseSession 使用该类来记录每个学生每次测试的成绩。类 Performance 提供计算平均成绩的方法。

```
package sis.studentinfo;

import junit.framework.*;

public class PerformanceTest extends TestCase {
    private static final double tolerance = 0.005;
    public void testAverage() {
        Performance performance = new Performance();
        performance.setNumberOfTests(4);
        performance.set(0, 98);
        performance.set(1, 92);
        performance.set(2, 81);
        performance.set(3, 72);

        assertEquals(92, performance.get(1));
    }
}
```

```

assertEquals(85.75, performance.average(), tolerance);
}
}

```

类 Performance:

```

package sis.studentinfo;

public class Performance {
    private int[] tests;
    public void setNumberOfTests(int numberOfTests) {
        tests = new int[numberOfTests];
    }

    public void set(int testNumber, int score) {
        tests[testNumber] = score;
    }

    public int get(int testNumber) {
        return tests[testNumber];
    }

    public double average() {
        double total = 0.0;
        for (int score: tests)
            total += score;
        return total / tests.length;
    }
}

```

您把测试成绩存放在整型数组 `tests` 中:

```
private int[] tests;
```

256

上面的定义表明变量 `tests` 是整型数组。成对的方括号指示 `tests` 是一个数组。在方括号中对中使用下标，即数组的索引，它可以告诉 Java 您打算访问数组中的哪个元素。

Java 也允许用下面的方式来定义数组:

```
private int tests[]; // 不要用这种方式定义数组
```

避免用这种方式来定义数组。如果使用这种方式，就表明您是一个 C/C++ 程序员。

上面两种定义数组的方式都不会导致内存分配。如果您访问数组的任何元素，都会导致报错⁹。

当调用类 `Performance` 的方法 `setNumberOfTests` 时，赋值语句对数组 `tests` 进行初始化。

```

public void setNumberOfTests(int numberOfTests) {
    tests = new int[numberOfTests];
}

```

⁹ `NullPointerException`。请参考第 8 课有关异常的内容。

使用关键字 `new` 来为数组分配内存。在关键字 `new` 后面, 指定数组的类型 (本例中为 `int`) 以及数组中元素的个数 (`numberOfTests`)。

一旦完成数组的初始化, 您就可以向数组的任何单元添加元素。元素的类型必须和数组类型相匹配。例如, 您只能把 `int` 类型的值赋给 `int[]` 数组。

```
public void set(int testNumber, int score) {
    tests[testNumber] = score;
}
```

方法 `set` 中, 把变量 `score` 赋值给数组 `tests` 的第 `testNumber` 个单元。数组的索引从 0 开始。数组的第一个单元的索引值为 0, 第二个单元的索引值为 1, 依此类推。

方法 `get` 展示了如何根据指定的索引, 来访问数组的元素:

```
public int get(int testNumber) {
    return tests[testNumber];
}
```

就像处理集合一样, 您可以用 `for-each` 循环来遍历某个数组。

```
public double average() {
    double total = 0.0;
    for (int score: tests)
        total += score;
    return total / tests.length;
}
```

257

方法 `average` 中, 我们用数组的成员变量 `length`, 来获取数组中元素的个数。

用经典的 `for` 循环来遍历数组:

```
public double average() {
    double total = 0.0;
    for (int i = 0; i < tests.length; i++)
        total += tests[i];
    return total / tests.length;
}
```

可以将数组声明成任何类型或者引用。您也许需要一个表示学生数组的实例变量:

```
private Student[] students;
```

数组初始化

为数组分配内存的时候, Java 针对数组的每一个元素设置默认值。例如, Java 将数值类型数组的每一个元素初始化成 0, 将布尔数组的每一个元素初始化成 `false`, 将对象数组的每一个元素初始化成 `null`。

Java 提供了特殊的数组初始化语法, 您可以在声明数组的时候使用该初始化语法。

```
public void testInitialization() {
    Performance performance = new Performance();
    performance.setScores(75, 72, 90, 60);
    assertEquals(74.25, performance.average(), tolerance);
}
```

方法 `setScores` 将四个表示成绩的参数传给了一个数组初始化表达式。

```
public void setScores(int score1, int score2, int score3, int score4) {
    tests = new int[] { score1, score2, score3, score4 };
}
```

上面例子中的数组拥有四个单元，每个单元都填充了 `int` 类型的元素。

更简单的方法是在数组声明赋值语句中，进行数组初始化：

```
int[] values = {1, 2, 3};
```

258

在数组初始化表达式中，可以用逗号来表示数组列表的结束。Java 编译器忽略最后一个逗号（不会在数组中创建一个多余的单元）。例如，前面的声明和初始化语句等同于：

```
int[] values = {
    1,
    2,
    3,
};
assertEquals(3, values.length);
```

将初始化语句写成多行，有助于您理解该特性的用处。该特性允许您在数组初始化中，自由地添加、删除、和移动元素，而不必担心在最后一个元素的后面留有一个逗号。您可以在每个元素的后面都加上一个逗号。

何时使用数组

集合相对数组，代码更简单、更接近纯面向对象、更灵活。在代码中，用 `LinkedList` 替换 `ArrayList` 比较容易，但是用 `LinkedList` 替换数组就比较费事。

某些情况下，您必须使用数组。例如，很多 API 返回数组，或者以数组作为参数。

某些情况下，数组能更好地满足要求。很多数学算法和结构（例如矩阵），用数组实现要好一些。

访问数组某个元素之前，要知道该元素存储在数组的哪个单元，然后乘以单元的长度，最后加上内存偏移量。图 7.1 描绘了包含 3 个元素的整型数组。数组在内存中的起始地址是 `0x100`。如果要访问第 3 个元素 (`x[2]`)，Java 用下面的表达式计算该元素在内存中的位置：

```
0x100 starting address + 2nd slot * 4 bytes per slot
```

因为数组中元素的类型为 `int`¹⁰，所以每个单元的长度是 4 个字节¹⁰。在地址 `0x108` 可以找到元素 55。

259

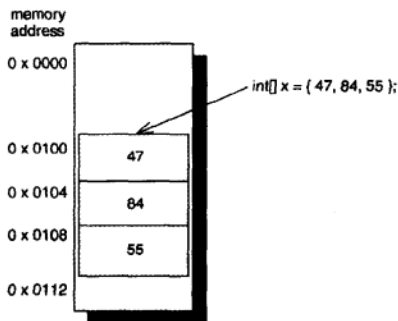


图 7.1 数组在内存中

数组提供了比集合类更好的性能。但是，最好使用集合类而不是数组。在把集合类改为数组之前，总是要从性能的角度来衡量是否确实值得。



面向对象的集合类优于数组。

可变参数

如果方法的参数个数不确定，但是参数类型都一样，那么您可以将这些参数放到一个集合中。前面的方法 `setScores` 要求 4 个测试成绩，也许您希望将任意个数的成绩传入某个 `Performance` 对象。在 J2SE5.0 之前，通常定义一个数组作为参数，来传入任意个数的参数：

```
public void testArrayParam() {
    Performance performance = new Performance();
    performance.setScores(new int[] {75, 72, 90, 60 });
    assertEquals(74.25, performance.average(), tolerance);
}
```

方法 `setScores` 可以直接存储数组：

```
public void setScores(int[] tests) {
    this.tests = tests;
}
```

260

您也许会尝试用更短的数组声明/初始化语法：如

¹⁰ Java 将基本类型直接存储在数组中。至于对象数组，Java 在数组的每个单元中存储指向其它内存位置的引用。

```
performance.setScores({ 75, 72, 90, 60 }); // 编译无法通过
```

但是 Java 不允许这样的语法。

J2SE5.0 允许在方法声明中使用可变参数。参数位于方法参数列表的末尾，而且所有参数的类型必须相同。

在圆括号中使用省略号，来定义可变参数。程序员应该熟悉 C 语言中有类似语法的可变参数声明。

```
public void setScores(int... tests) {
    this.tests = tests;
}
```

调用方法 `setScores` 时，可以传入可变数目的测试成绩：

```
public void testVariableMethodParms() {
    Performance performance = new Performance();
    performance.setScores(75, 72, 90, 60);
    assertEquals(74.25, performance.average(), tolerance);

    performance.setScores(100, 90);
    assertEquals(95.0, performance.average(), tolerance);
}
```

执行 Java 程序时，虚拟机通过包含 `String` 对象的数组，将命令行参数传入 `main` 方法：

```
public static void main(String[] args)
```

也可以这样定义 `main` 方法：

```
public static void main(String... args)
```

多维数组

Java 支持多维数组。用多维数组表示矩阵是经典的用法。可以创建最多 255 维的数组，但是通常很少使用三维以上的数组。下面的语言测试展示了如何分配、赋值、以及访问一个二维数组。

```
// 0 1 2 3
// 4 5 6 7
// 8 9 10 11
public void testTwoDimensionalArrays() {
    final int rows = 3;
    final int cols = 4;
    int count = 0;
    int[][] matrix = new int[rows][cols];
    for (int x = 0; x < rows; x++)
        for (int y = 0; y < cols; y++)
```

```

        matrix[x][y] = count++;
        assertEquals(11, matrix[2][3]);
        assertEquals(6, matrix[1][2]);
    }

```

不需要针对所有的维度一次性分配全部的内存。可以先为最左边的维度分配内存，最右边的维度暂时不分配内存。稍后，需要的时候再为最右边的维度分配内存。方法 `testPartialDimensions` 中，将 `matrix` 初始化成具有三行元素的二维整型数组。然后，为二维数组的每一行分配长度不同的内存单元。

```

// 0
// 1 2
// 3 4 5
public void testPartialDimensions() {
    final int rows = 3;
    int[][] matrix = new int[rows][];
    matrix[0] = new int[]{0 };
    matrix[1] = new int[]{1, 2 };
    matrix[2] = new int[]{3, 4, 5 };
    assertEquals(1, matrix[1][0]);
    assertEquals(5, matrix[2][2]);
}

```

方法 `testPartialDimensions` 表明，Java 并不限制必须将多维数组定义成矩形，您可以将多维数组定义成锯齿型。

可以使用数组初始化语法来初始化多维数组。下面的初始化代码将产生和方法 `testPartialDimensions` 一样的数组：

```
int[][] matrix2 = {{0 }, {1, 2 }, {3, 4, 5 }};
```

数组类

由于数组不是对象，所以不能向数组发送消息。尽管可以通过变量 `length` 来获取数组的长度，但是 Java 将其视为特殊语法。

类 `java.util.Arrays` 提供了若干通用的操作数组的类方法。使用数组类，您可以对一维数组进行下面的处理：

262

- 执行折半搜索（假定数组是排好序的）（`binarySearch`）
- 对数组排序（`sort`）
- 将数组转换成实现接口 `List` 的对象（`asList`）
- 和另一个相同类型的一维数组进行比较（`equals`）
- 获取哈希值（参考第9课）（`hashCode`）

- 用指定值填充数组的所有元素，或者子集 (fill)
- 获取数组的可打印形式 (toString)

下面会对方法 `equals` 进行深入的讨论。关于其它方法的更多信息，请参考 Java API 文档关于 `java.util.Arrays` 的部分。

Arrays.equals

无论数组中包含的是基本类型还是引用，数组本身都是引用类型。如果创建了两个相同的数组，Java 虚拟机会将它们存储在两个不同的内存位置，这意味着两个数组间“==”比较的结果是 `false`。

```
public void testArrayEquality() {
    int[] a = {1, 2, 3};
    int[] b = {1, 2, 3};
    assertFalse(a == b);
}
```

由于数组是引用，所以您可以使用方法 `equals` 来比较两个数组。但是，即使两个数组具有相同的维度和相同的内容，比较的结果也是 `false`。

```
public void testArrayEquals() {
    int[] a = {1, 2, 3};
    int[] b = {1, 2, 3};
    assertFalse(a.equals(b));
}
```

方法 `Arrays.equals` 比较两个数组的内容，而不是内存位置。

```
public void testArraysEquals() {
    int[] a = {1, 2, 3};
    int[] b = {1, 2, 3};
    assertTrue(Arrays.equals(a, b));
}
```

263

重构

分隔字符串

尽管 `Student` 的 `split` 方法可以工作，但是代码相当复杂。Java 提供了两种以上对 `String` 对象进行分隔的方法。第一，可以使用类 `StringTokenizer` 将某个输入字符串分隔成多个独立的单元。`StringTokenizer` 是 Java 的遗留类，Sun 建议用更好的方案来替代遗留类。对于

StringTokenizer, 更好的方案是使用类 String 的 split 方法来分隔字符串。一旦您在代码中发现 StringTokenizer, 最好修改代码使用 String.split。

在 Java1.4 以前, Sun 一直推荐使用 StringTokenizer 来分隔字符串。所以, 您会遇见大量使用了 StringTokenizer 的代码。在 Java1.4 中, Sun 引入了处理正则表达式的 API。正则表达式是对文本进行模式匹配的规约。方法 String.split 接受正则表达式, 所以比 StringTokenizer 更有效率。有关正则表达式 API 的更多信息, 请参考附加课程三。

类 java.util.StringTokenizer 的构造函数有两个参数: 输入字符串, 表示字符分隔符列表的字符串。StringTokenizer 将输入字符串分隔成若干独立的单元, 然后您可以迭代访问每一个独立的单元。while 循环提供了迭代访问每个独立单元的最佳途径:

```
private List<String> split(String name) {
    List<String> results = new ArrayList<String>();
    StringTokenizer tokenizer = new StringTokenizer(name, " ");
    while (tokenizer.hasMoreTokens())
        results.add(tokenizer.nextToken());
    return results;
}
```

发送消息 hasMoreTokens 给 StringTokenizer, 一旦不存在可用的独立单元, 那么返回 false。消息 nextToken 返回下一个可用的独立单元。StringTokenizer 使用分隔符列表来决定如何分隔输入字符串, 本例中仅以空格来对输入字符串进行分隔。

264

分割字符串: String.split

类 String 提供了方法 split, 使用该方法对姓名全称进行分隔, 甚至比 StringTokenizer 还要简单。采用 String.split, 我们仅用一行代码就可以替换整个方法 Student.split。

下面是重构后的代码:

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    setName(nameParts);
}

private List<String> split(String name) {
    return Arrays.asList(name.split(" ")); //不能工作
}

private void setName(List<String> nameParts) {
    this.lastName = removeLast(nameParts);
    String name = removeLast(nameParts);
    if (nameParts.isEmpty())
        this.firstName = name;
    else {
```

```

        this.middleName = name;
        this.firstName = removeLast(nameParts);
    }
}

private String removeLast(List<String> list) {
    if (list.isEmpty())
        return "";
    return list.remove(list.size() - 1);
}

```

为了对其它代码带来尽可能小的影响，我们保留了方法 `Student.split`，使用 `String` 的方法 `split` 来作为其实现代码：

```
return Arrays.asList(name.split(" ")); // 不能工作
```

方法 `split` 用空格作为分隔符，返回一个包含若干单词的数组。方法 `asList` 将数组转换成列表形式。

您可能认为这行代码可以作为 `StringTokenizer` 的有效替代方案，但是它不能工作。当方法 `removeLast` 中调用 `remove` 时，JUnit 会报告 `UnsupportedOperationException` 错误。 ◀ 265

原因在于处理数组的 `asList` 方法。参考 Java API 文档中关于 `asList` 的说明，发现该方法返回某个数组的列表形式。返回的列表只是数组的另一种视图，而数组本身并没有消失，对列表的任何操作最终都反映在数组上。

由于列表的背后是数组，又不能删除数组的任何元素，所以 Java 禁止 `remove` 调用。

一种可行的办法是在 `split` 中使用 `for-each` 循环。就像 `StringTokenizer` 方案，直接在方法 `split` 中将每一个单词添加到结果数组。

```

private List<String> split(String fullName) {
    List<String> results = new ArrayList<String>();
    for (String name: fullName.split(" "))
        results.add(name);
    return results;
}

```

方法 `split` 中可以应用 Java 的正则表达式功能。正则表达式是用来进行文本匹配的语法单元或者符号的集合。例如，查看目录下的文件，可以使用简单的正则表达式。用星号 (*) 作为通配符来匹配文件名中任意长度的字符序列。参考附加课Ⅲ，可以获得关于强大的正则表达式功能的概要信息。

练习

1. 用不同循环类型来实现求阶算法（不用递归），阶的定义如下：

如果 $n=0$ ，那么 n 的阶数为 1。
 如果 $n=1$ ，那么 n 的阶数为 1。
 如果 n 大于 1，那么 n 的阶数为 $1*2*3*...*n$ 。

编写测试，然后用 while 循环实现求阶方法。接着，修改求阶方法，用 for 循环替代 while 循环。最后，再用 do-while 循环替代 for 循环。注意不同循环结构的区别。同时，注意用一种循环结构就可以解决任何循环问题。

- a) 为什么您倾向于某一种循环结构，而不是其它循环结构？
- b) 哪种循环结构的代码是最短的？
- c) 哪种循环结构对于该问题是最合适的？为什么？
- d) 使用

```
while(true) {
```

代替您使用的循环结构，在循环的末尾使用关键字 break，检验是否可以通过测试。

2. 展示您对关键字 continue 的理解。创建一个返回字符串的方法，该字符串包含从 1 到 n 的数字，每个数字之间用空格分隔。每 5 个数字之间用星号隔开。例如，从 1 到 12 的字符串如下：

```
"1 2 3 4 5* 6 7 8 9 10* 11 12".
```

3. a) 将练习 2 的字符串分割成子字符串，并存储在 Vector 中。用空格字符分隔输入字符串。例如，字符串 "1 2 3 4 5* 6 7" 被分割成子字符串 "1", "2", "3", "4", "5*", "6 ", 和 "7"，并存储在 Vector 中。
 b) 迭代访问 Vector 的每一个元素，重新创建字符串。
 c) 删除所有的参数化类型信息，注意编译错误，在必要的地方修改代码。
4. 在以前的练习中，您编写过验证棋子移动是否有效的测试方法。简化断言：

```
public void testKingMoveNotOnEdge() {
    Piece piece = Piece.createBlackKing();
    board.put("d3", piece);
    assertContains(piece.getPossibleMoves("d3", board),
        "c4", "d4", "e4", "c3", "e3", "c2", "d2", "e2");
}
```

按照该方法，重构其它测试。

5. 您可能注意到让您花了很多时间的棋盘实际上是一个二维数组。备份类 Board。然后修改类 Board，使用二维数组作为底层数据结构。

您将会遇到的一个困难是：客户代码（例如 Game 和 Pawn）依赖类 Board 的实现。该练习中，修改客户代码以遍历二维数组。这是一个不太理想的方案：通常您应该为类设计一个稳定的、不会改变的公共接口。如果修改了类 Board 的实现细节，类 Game 不会受

到任何影响。

一种替代方法是：每次客户需要迭代访问时，将二维数组转换成以 *Piece* 对象列表为元素的列表。但是最佳方案不应该要求客户自己来进行迭代。避免客户自行迭代的方案需要应用 *visitor* 设计模式，该方案比较复杂。请参考设计模式¹¹得到进一步信息。目前，把类 *Board* 的二维数组暴露给客户——这样暂时可以满足要求。

◀ 267

编译并比较两个版本的 *Board* 类。哪一个可读性更好、更容易理解？哪一个的代码更少？

6. 使类 *Board* 具备可迭代性，这样您可以通过 *for-each* 循环访问每一个格子。单独创建一个跟踪格子索引的迭代类是一种可选的方案，但是该方案比较复杂。更简单的方案是：遍历棋盘上的每一个格子，将其添加到 *List* 对象，然后返回 *List* 对象的迭代器。

一旦类 *Board* 可迭代，就可以将现有的循环结构改为 *for-each* 循环。寻找任何重构的机会，尽可能使用 *for-each* 循环结构。例如，您可以修改代码，让棋子对象存储它自己当前的行列位置。

7. 阅读类 *ArrayList* 的 *javadoc* API，找到一种方法来简化本课的最后一个例子——分隔学生全名，返回包含所有名字单元的列表。重构方法 *split*，做到不使用循环——注意看 *ArrayList* 的构造函数。

◀ 268

¹¹ [Gamma1995].

8

Exceptions and Logging

异常和日志

这一课，您将了解异常和日志。在 Java 中，异常是一种代码流程控制的机制。代码中，通过产生异常来告知出现了问题。然后，编写代码接受或者处理异常。

Java 提供了扩展性很好的日志功能，您可以利用日志来记录应用程序执行过程中产生的信息。您可以选择把异常、有趣的事情或者希望跟踪的事件记入日志。

本课内容包括：

- try-catch 块
- 检查异常和非检查（运行时）异常
- 错误和异常
- throws 字句
- 异常层次结构
- 创建自己的异常类型
- 异常消息
- 处理多个异常
- 重新抛出异常
- 处理堆栈跟踪
- finally 块
- 格式化类
- Java 日志 API
- 将日志记录到文件
- 日志处理
- 日志属性
- 日志层次结构



异常

第4课中，当访问没有正确初始化的引用时，会生成 `NullPointerException` 异常。

异常是一种对象，用来表示异常的情况。您可以创建异常对象，然后抛出异常对象。如果您意识到可能会有异常发生，请编写代码显式的处理、或者捕捉异常。您也可以在代码中声明忽略异常，让别人或者在别的地方来处理异常。

异常抛出意味着控制的转移。您可以在代码的任何地方抛出异常；同样，其它 API 可能在任何时间抛出异常。甚至 Java 虚拟机也可以抛出异常。异常被抛出的地方，是处理或者捕捉异常的第一位置。如果没有代码捕捉异常，那么将导致程序的异常中止¹。



最终，您将创建一个用户界面，借助该界面输入某课程的学生成绩。用户界面可能出现任何类型的错误：用户输入的数字过大，用户什么也没有输入，或者用户输入了乱码。您的任务是尽可能快的处理各种无效输入。

输入用户界面的成绩被视为 `String` 对象。您需要首先将字符串转换成整型。使用 `Integer` 包装类的工具方法 `parseInt` 可以完成此类转换。如果转换成功，`parseInt` 返回相应的整型数值。如果字符串含有无效输入，那么 `parseInt` 会抛出异常 `NumberFormatException`。

编写一个简单的测试，来预期成功的情况：

```
package sis.studentinfo;

import junit.framework.TestCase;

public class ScorerTest extends TestCase {
    public void testCaptureScore() {
        Scorer scorer = new Scorer();
        assertEquals(75, scorer.score("75"));
    }
}
```

使测试得以通过：

```
package sis.studentinfo;

public class Scorer {
    public int score(String input) {
        return Integer.parseInt(input);
    }
}
```

编写第二个测试来展示将无效输入传入方法 `score` 后，会发生什么：

¹ 对于多线程程序，并非必然如此，请参考有关多线程的章节。



```
public void testBadScoreEntered() {
    Scorer scorer = new Scorer();
    scorer.score("abd");
}
```

虽然这不是完整的测试，但是可以展示当异常抛出时会发生什么。编译代码并运行测试。JUnit 会报告错误而不是测试失败。错误意味着测试代码（当然，也包括调用的代码）中产生了未经处理的异常。JUnit 在细节窗口中打印该异常的堆栈跟踪，结果表明类 `Integer` 生成了一个 `NumberFormatException` 异常。

测试表明：当客户传入无效输入时，方法 `score` 生成了一个异常。您期望在该测试用例中捕获一个异常，如果异常发生那么测试通过，如果异常没有发生那么测试失败。

Java 提供了 `try-catch` 块结构来捕获异常。标准的 `try-catch` 包含两块代码。`try` 包含可能抛出异常的代码块。`catch` 包含处理异常的代码。

```
public void testBadScoreEntered() {
    Scorer scorer = new Scorer();
    try {
        scorer.score("abd");
        fail("expected NumberFormatException on bad input");
    }
    catch (NumberFormatException success) {
    }
}
```

方法 `testBadScoreEntered` 中的代码是测试异常的习惯用法。在 `try` 块中捕获 `score` 可能生成的 `NumberFormatException` 异常。如果 `score` 生成了一个异常，那么 Java 虚拟机立刻抛出该异常，并将控制转移到 `catch` 块。

271

如果 `score` 没有产生异常，那么 Java 执行 `try` 块中的下一行代码。在 `testBadScoreEntered` 中，下一行代码是 JUnit 方法 `fail`。该测试从 `junit.framework.TestCase` 继承（间接地）了 `fail` 方法。执行 `fail` 方法，会立刻中止测试，Java 虚拟机不会再执行任何别的代码。然后 JUnit 报告测试失败。调用 `fail` 等同于调用 `assertTrue(false)`（或者 `assertFalse(true)`）。

您期望 `score` 方法产生一个异常。如果没有，表明出了问题，这时您应该中止测试。如果 `score` 方法产生了一个异常，那么 Java 虚拟机忽略 `fail` 调用，直接将控制转移到 `catch` 块。

`catch` 块是空的。可能您的 `catch` 块永远是空的。通常情况下，您应该在 `catch` 块中处理捕获的异常。本课中，我将向您展示处理被捕获异常的可能方法。

在 `testBadScoreEntered` 中接受异常是一件好事情。在 `catch` 子句中，用 `success` 来命名 `NumberFormatException` 对象。您遇到的大多数代码一般会使用通用的变量名（例如 `e`）来命名异常对象。通过异常命名可以描述您为什么期望该异常。

该异常测试说明了客户代码不得不进行处理的事务。在用户界面的其它地方，也会有和

testBadScoreEntered 中类似的代码。该测试也证明了 score 方法产生 NumberFormatException 异常的潜在可能性，以及在什么情况下会产生异常。

异常处理

异常管理的策略是：在尽可能靠近源头（异常产生的地方）的地方捕获异常。一旦捕获某个异常，就编写代码对其进行处理，从而将问题转变成一种可接受的合理的行为。

除此之外，您也可以避免异常。在类 Scorer 中，提供对用户输入字符串进行合法性验证的方法 isValid，把负担转移给了客户代码，由客户代码避免使用无效输入调用 score 方法。

```
// ScorerTest.java
public void testIsValid() {
    Scorer scorer = new Scorer();
    assertTrue(scorer.isValid("75"));
    assertFalse(scorer.isValid("bd"));
}
// Scorer.java
public boolean isValid(String input) {
    try {
        Integer.parseInt(input);
        return true;
    }
    catch (NumberFormatException e) {
        return false;
    }
}
```

客户代码首先调用方法 isValid，如果该方法返回 true，那么客户代码可以继续安全地调用方法 score。如果 isValid 返回 false，那么客户代码通知用户输入了无效的字符串。

这种方法的好处是可以避免在客户代码中编写 try-catch 块。您应该尽可能少的通过异常来控制代码流。在尽可能靠近源头的地方捕获异常，可以消除客户代码通过 try-catch 块来管理控制流的必要。

针对 score 方法，存在一种需要 try-catch 块的可能性：当捕获 NumberFormatException 异常，score 方法返回-1 或者别的客户代码能够识别的特殊整型数值。

检查异常



学校的每门课程安排都有相应的 Web 页面。创建一个 Session 对象之后，您需要给该对象

发送一个表示 Web 页面 URL² 的字符串。然后, Session 对象使用该字符串创建一个 java.net.URL 对象。类 Java.net.URL 有一个接受正确格式 URL 字符串的构造函数。URL 由若干部分(例如: 协议名称, 主机名称)组成, 每个部分都有若干规则。如果使用错误格式的 URL 来构造 java.net.URL, 那么 java.net.URL 的构造函数会抛出异常 java.net.MalformedURLException。

在 SessionTest 中创建一个小测试: 首先, 将字符串形式的 URL 存入 session 对象, 接着以 java.net.URL 对象的形式取出该 URL。最后, 测试保证 URL 对象的字符串形式 (toString) 和 URL 字符串参数相匹配。

273

```
public void testSessionUrl() {
    final String url = "http://course.langrsoft.com/cmsc300";
    session.setUrl(url);
    assertEquals(url, session.getUrl().toString());
}
```

Session 中的实现代码只是一对 setter 和 getter:

```
package sis.studentinfo;

import java.util.*;
import java.net.*;

abstract public class Session
    implements Comparable<Session>, Iterable<Student> {
    ...
    private URL url;
    ...
    public void setUrl(String urlString) {
        this.url = new URL(urlString);
    }

    public URL getUrl() {
        return url;
    }
    ...
}
```

但是, 编译之后会发现错误:

```
unreported exception java.net.MalformedURLException; must be caught or declared
to be thrown
    this.url = new URL(urlString);
        ^
```

Java 将 MalformedURLException 类型的异常定义为检查异常 (该名词相对于非检查异常)。检查异常是您必须在代码中显式地处理的异常。您可以忽略非检查异常 (例如: NumberFormatException), 但是请注意忽略异常可能是不安全的。

对于能够产生检查异常的代码, 我们必须加以处理。可以使用 try-catch 块来处理检查

² Uniform Resource Locator, 统一资源定位符, 一种指向互联网资源的指针。

异常，或者通过简单的声明将异常传递给调用者。使用 `throws` 子句来声明某个方法需要传递异常：

```
public void setUrl(String urlString) throws MalformedURLException {
    this.url = new URL(urlString);
}
```

274

子句 `throws MalformedURLException` 表明方法 `setUrl` 可能产生此类异常。任何调用 `setUrl` 方法的代码要么使用 `try-catch` 块来处理 `setUrl` 调用，要么也在调用 `setUrl` 的方法的声明中使用 `throws` 子句。

所以，对于测试本身，要么捕获 `MalformedURLException`，要么在声明中使用子句 `throws MalformedURLException`。编译，然后看看结果如何。

```
public void testSessionUrl() throws MalformedURLException {
    final String url = "http://course.langrsoft.com/cm300";
    session.setUrl(url);
    assertEquals(url, session.getUrl().toString());
}
```

(不要忘了在 `SessionTest` 中加上 `import java.net.*`)。

除非您将异常接收作为测试的一部分，否则一般都简单地声明测试 `throws` 异常。不要用 `try-catch` 包裹 `setUrl` 调用。对这个测试而言，您知道 `URL` 是有效的。在正面测试的场合中，可以假设所有情况都在您的控制之下，测试执行过程中不可能抛出异常，所以可以忽略任何异常。即使发生了不太可能的情况，导致抛出了异常，JUnit 也会捕获该异常，然后用某个错误来报告测试失败。

当然，您也需要否定测试，通过否定测试来展示什么情况下会出现问题：

```
public void testInvalidSessionUrl() {
    final String url = "http://course.langrsoft.com/cm300";
    try {
        session.setUrl(url);
        fail("expected exception due to invalid protocol in URL");
    }
    catch (MalformedURLException success) {
    }
}
```

您需要导入类 `MalformedURLException`。

异常层次关系

为了能够将对象作为异常抛出，该对象必须是 `Throwable` 类型。类 `Throwable` 定义在包

java.lang 中，位于异常层次关系的最顶部。Throwable 有两个子类：Error 和 Exception。检查异常和非检查异常都继承自 Exception。 ◀ 275

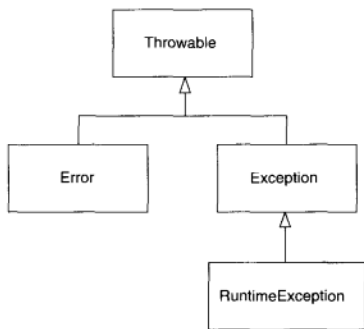


图 8.1 异常层次关系

Sun 用类 Error 记录 Java 运行环境自身可能出现的严重问题。Error 属于非检查类型，通常不必去捕获 Error 类的实例。Error 包括诸如 OutOfMemoryError 和 InternalError 的错误。您不要期望可以编写代码从任何 Error 中恢复。

类 Exception 是其它所有异常类的父类。Sun 将虚拟机或者 Java 类库产生的异常定义为 Exception 的子类。您自己定义的任何异常也必须是 Exception 的子类。有时，我们也把 Exception 的子类通称为应用程序异常。

非检查应用程序异常必须继承自 RuntimeException。RuntimeException 直接继承自 Exception。

创建自己的异常类型

您可以创建并抛出现有的 Java 异常对象。但是，定制异常可以增加程序的清晰度。定制的异常也可以使测试变得简单。

就像前面所提到的，由于某些原因无法对异常进行控制，所以把异常传递到上一级调用者。但是，最好有一种方法能够阻止异常扩散。 ◀ 276

在 URL 的例子中，构造函数必须接受有效的参数。您可以通过静态方法调用（例如：`public static boolean URL.isValid(String url)`）来进行有效性验证。但是，在构造函数中抛出异常比强制客户调用静态方法，要更合理一些。



例如，当姓名单元超过3的时候，您应该拒绝创建 Student 对象。

```
public void testBadlyFormattedName() {
    try {
        new Student("a b c d");
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException success) {
    }
}
```

实现该测试的第一步是创建一个新的异常子类。在扩展异常之前，您应该仔细阅读 Sun 提供的 Java 类库中所有的异常类型，从中寻找一个最有可能作为基类的异常类型。在本例中，`java.lang.IllegalArgumentException` 最合适做基类。创建类 `StudentNameFormatException`，该类继承自 `IllegalArgumentException`：

```
package sis.studentinfo;

public class StudentNameFormatException
    extends IllegalArgumentException {
}
```

以上就是定义一个异常类型所需要做的全部工作。编译测试，然后运行测试，因为还没有产生异常，所以测试失败。修改代码使测试得以通过：

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    final int maximumNumberOfNameParts = 3;
    if (nameParts.size() > maximumNumberOfNameParts)
        throw new StudentNameFormatException();
    setName(nameParts);
}
```

下面这行代码表明：创建异常对象和创建其它任何对象都是一样的：

277

```
throw new StudentNameFormatException();
```

使用关键字 `throw` 抛出异常对象：

```
throw new StudentNameFormatException();
```

因为 `StudentNameFormatException` 是 `IllegalArgumentException` 的子类，又因为 `IllegalArgumentException` 是 `RuntimeException` 的子类，所以您不必在 `Student` 的构造函数声明中加上 `throws` 子句。

注意：没有针对 `StudentNameFormatException` 的测试类。原因在于：事实上不存在导致该测试失败的代码。的确，您可能会忘记从 `Exception` 继承，但是如果这样代码就无法编译通过。

检查异常和非检查异常

作为代码设计者，您可能指定 `StudentNameFormatException` 为检查异常或者非检查异常。究竟哪一种更好一些，目前还存在非常大的争论。一种意见认为：检查异常造成的问题比它的价值还要多³。因为在方法声明中添加 `throws` 子句，可能会导致客户代码无法工作——此类潜在的客户代码非常多。

现实情况是：多数代码对大多数的异常不做任何处理。管理异常的唯一适当的地方是用户界面层：“应用程序出现严重错误，请联系支持人员”。系统底层代码产生的异常会逐层上升，最终反映到用户界面层。强制所有中间层次的类必须处理此类问题，会导致代码的混乱。

通常，我们建议尽可能的避免要求客户代码去处理异常。可以考虑返回某个值，客户代码将处理返回值作为正常流程的一部分。或者考虑，根据约定在客户代码中执行有效性验证，就像前面的例子中 `Scorer` 首先要调用方法 `isValid`。

另一些程序员支持检查异常，强调强制编写客户代码的程序员尽早并且始终接受和处理异常是有益的。不幸的是，面对检查异常，很多程序员选择了十分糟糕的处理方法：

278

```
try {
    doSomething();
}
catch (Exception e) {
    // log or do nothing
}
```

在 `catch` 块中什么也不做是一种反模式：空 `catch` 语句⁴。将异常记入日志等同于什么也没做。空 `catch` 语句几乎在任何情况下都是不合适的。而且，空 `catch` 语句还可能隐藏潜在的、严重的问题，这些问题也许将来会导致更严重的问题。



避免传递异常，但也不要创建空 `catch` 块。

消息

大多数异常都会存储一条相关的、供程序员使用的消息。只在个别情况下，才将异常消息呈现给最终用户（尽管异常消息是正确的最终用户提示信息的基础）。异常消息通常用来调试程

³ [Venners2003].

⁴ [Wiki2004a].

序。也可以将异常消息存储到应用程序的日志文件，从而记录是什么原因导致了错误的发生⁵。

定制的异常对象可以包含消息，也可以不包含消息。使用方法 `getMessage` 可以从异常对象中获取相应的消息。如果异常对象不包含消息，那么 `getMessage` 返回 `null`。

您也许希望在 `StudentNameFormatException` 对象中存储一条错误消息，以便提供更多的信息。

```
public void testBadlyFormattedName() {
    try {
        new Student("a b c d");
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        assertEquals(
            "Student name 'a b c d' contains more than 3 parts",
            expectedException.getMessage());
    }
}
```

279

这里将捕获异常对象的临时变量声明为 `expectedException`，而不是 `success`。这样命名是为了表示仅仅捕获异常并不能说明测试成功，而且存储在 `expectedException` 中的消息必须和预期的一致。

因为 `getMessage()` 缺省返回 `null`，所以测试将失败。下面修改 `StudentNameFormatException` 的构造函数，构造函数以消息字符串为参数，并且构造函数直接调用相应父类的构造函数。

```
package sis.studentinfo;

public class StudentNameFormatException
    extends IllegalArgumentException {
    public StudentNameFormatException(String message) {
        super(message);
    }
}
```

然后，在 `Student` 的构造函数中将消息整合在一起，传递给异常的构造函数。

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    final int maximumNumberOfNameParts = 3;
    if (nameParts.size() > maximumNumberOfNameParts) {
        String message =
            "Student name '" + fullName +
            "' contains more than " + maximumNumberOfNameParts +
            " parts";
        throw new StudentNameFormatException(message);
    }
}
```

⁵ 细节请参考本课第二部分关于日志的讨论。

```

    }
    setName(nameParts);
}

```

的确，异常消息在测试和实现代码中存在重复。您可以选择现在就对之进行重构。稍后，您将看到如何使用 `String` 的 `format` 方法来动态构造消息，而且我会在那个时候对这部分代码进行重构。

捕获多个异常

一行代码有可能产生多个异常，而且每个异常都属于不同的异常类型。抛出多个异常的方法需要逐个列出所有的异常类型：

280

```
public void send() throws ObjectStreamException, UnknownHostException
```

如果两个异常继承自同一个类，那么您可以在声明中用它们的父类异常来代替：

```
public void send() throws Exception
```

`try-catch` 块中有多个 `catch` 子句，每个子句处理一种异常：

```

try {
    new Student("a b c d");
}
catch (StudentNameFormatException e) {
    ...
}
catch (NoSuchElementException e) {
    ...
}

```

如果 `try` 块中的代码抛出某个异常，那么 Java 虚拟机将控制转移到第一个 `catch` 块。如果第一个 `catch` 块中声明的异常类型和抛出的异常类型相匹配，那么虚拟机执行第一个 `catch` 块中的代码。否则，虚拟机将控制转移到下一个 `catch` 块，依此类推。一旦某个 `catch` 块被执行，那么虚拟机忽略其它所有的 `catch` 块。

下面的代码中，第二个 `catch` 块捕获任何类型的异常。由于所有的（普通）异常都继承自 `Exception`，所以第二个 `catch` 块可以捕获除了 `StudentNameFormatException`（因为该异常定义在第一个 `catch` 块中）以外的任何异常。

```

try {
    new Student("a b c d");
}
catch (StudentNameFormatException e) {
    ...
}

```



```
catch (Exception e) {
    ...
}
```

在需要捕获无法预期的异常的情况下，能够捕获任何异常的 `catch` 块非常有用。通常，您只在最上层的、和用户界面最接近的类中使用这种 `catch` 块。在这种 `catch` 块中，您需要做的事情只是记录错误，尽可能展现给最终用户某些有用的信息。进行这样的处理还可以避免应用程序崩溃或者明显的错误。

使用能够捕获任何异常的 `catch` 块，需要您极强的判断能力。因为该 `catch` 块中的代码有可能产生新的异常，而且您又可能没有意识到新的异常，即便它是一个检查异常。错误被隐藏是可怕的。

281

重新抛出异常

没有人可以阻止您捕获异常，然后在 `catch` 块中重新抛出这个异常（或者其它异常）。这样的行为被称之为“重新抛出异常”。

通常，重新抛出异常的目的在于：在尽可能接近源头的地方捕获异常，并进行日志，然后再传递该异常。这种技术使得追溯问题的源头变得更加容易。

```
public void setUrl(String urlString) throws MalformedURLException {
    try {
        this.url = new URL(urlString);
    }
    catch (MalformedURLException e) {
        log(e);
        throw e;
    }
}

private void log(Exception e) {
    // logging code. The second half of this lesson contains more
    // info on logging. For now, this method is empty.
}
```

在方法 `setUrl` 的 `catch` 块中，把捕获的异常对象传递给方法 `log`，然后重新抛出该异常。这种技术的好处在于：您可以创建并抛出应用程序特定的异常，可以封装产生异常的特定实现细节。

接下来，抛出一个应用程序特定的异常 `SessionException`，而不是标准异常 `MalformedURLException`。一旦创建了下面的异常类：

```
package sis.studentinfo;

public class SessionException extends Exception {
```

}

您就可以修改测试代码，以反映新的异常类型：

```
public void testSessionUrl() throws SessionException {
    final String url = "http://course.langrsoft.com/cmsc300";
    session.setUrl(url);
    assertEquals(url, session.getUrl().toString());
}

public void testInvalidSessionUrl() {
    final String url = "https://course.langrsoft.com/cmsc300";
    try {
        session.setUrl(url);
        fail("expected exception due to invalid protocol in URL");
    }
    catch (SessionException success) {
    }
}
```

262

最后，修改实现代码，抛出新异常。

```
public void setUrl(String urlString) throws SessionException {
    try {
        this.url = new URL(urlString);
    }
    catch (MalformedURLException e) {
        log(e);
        throw new SessionException();
    }
}
```

这种方案的缺点是容易丢失信息。如果异常被抛出，那么真正的原因是什么呢？在不同的环境下，try 块中的代码可能抛出不同的异常。重新抛出应用程序特定的异常可能会隐藏异常的根源。

为了解决这个问题，您可以从根异常中取出消息，然后把消息存储到 SessionException 实例。但是，尽管这样，您依然会失去了最初的堆栈跟踪信息。

在 J2SE1.4 中，Sun 在类 Throwable 中增加了存储问题根源的能力。类 Throwable 增加了两种构造函数：一种以 Throwable 作为参数，另一种以消息字符串和 Throwable 作为参数。Sun 也在 Exception 和 RuntimeException 中增加了这两个构造函数。您可以调用 initCause 方法来设置问题根源。然后，通过发送消息 getCause，从异常中获取 Throwable 对象。

首先，修改方法 testInvalidSessionUrl，从 SessionException 实例中获取问题根源。作为测试验证的一部分，确保问题根源和期望的相符合。

```
public void testInvalidSessionUrl() {
    final String url = "https://course.langrsoft.com/cmsc300";
    try {
        session.setUrl(url);
        fail("expected exception due to invalid protocol in URL");
    }
}
```

263

```

catch (SessionException expectedException) {
    Throwable cause = expectedException.getCause();
    assertEquals(MalformedURLException.class, cause.getClass());
}
}

```

`assertEquals` 确保问题的根源是 `MalformedURLException`，该行代码利用了 Java 的反射机制——运行时动态获取各种类型信息的能力。在第 12 课中，我会深入介绍 Java 的反射机制。

简而言之，您可以向任何对象发送消息 `getClass`，该方法将返回一个合适的类常量。类常量是类名加上 `.class`。`MalformedURLException.class` 是类 `MalformedURLException` 的类常量。

测试将失败：如果没有显式地设置问题根源，那么 `getCause` 返回 `null`。首先，修改 `SessionException`，在构造函数中捕获问题根源：

```

package studentinfo;

public class SessionException extends Exception {
    public SessionException(Throwable cause) {
        super(cause);
    }
}

```

然后，修改实现代码，将问题根源嵌入到 `SessionException` 实例：

```

public void setUrl(String urlString) throws SessionException {
    try {
        this.url = new URL(urlString);
    }
    catch (MalformedURLException e) {
        log(e);
        throw new SessionException(e); // < here's the change
    }
}

```

堆栈跟踪

第 4 课中，我向您演示了如何阅读异常的堆栈，来帮助破译异常发生的根源。

使用 `Throwable` 中定义的方法 `printStackTrace`，可以将存储在异常中的堆栈跟踪打印输出到流或者 `writer`。`PrintStackTrace` 有一种不接受任何参数的实现，缺省情况下把堆栈跟踪打印到系统控制台。`printStackTrace` 提供了日志方法的天然实现：

```

private void log(Exception e) {
    e.printStackTrace();
}

```

(尝试一下) 但是对于生产系统,您可能需要一个更健壮的日志解决方案。Sun 在 J2SE1.4 中引入了日志 API。本课第二部分将讨论日志。

`printStackTrace` 打印出来的堆栈跟踪包含了对于高级编程非常有用的信息。您可以自己分析堆栈跟踪来提取此类有用的信息,很多程序员都这样做。或者,对于 J2SE1.4,您可以发送消息 `getStackTrace` 到异常对象,从而直接获取已经经过分析的信息。

finally 块

当异常被抛出之后,Java 虚拟机立刻把控制转移到与异常类型相匹配的第一个 `catch` 块。Java 不再继续执行 `try` 块中的其余代码。如果不存在 `catch` 块,那么虚拟机立刻将控制转移到产生异常的方法。但是,您可能需要在控制转移之前,确保有一小段代码会被执行。

`finally` 块是可选的,不管是否有异常抛出,Java 虚拟机总是会执行 `finally` 块中的代码片断。您可以将一个 `finally` 块跟在 `try-catch` 块的后面。

使用 `finally` 块的原因是为了回收局部变量。如果打开了一个文件,`finally` 块可以确保该文件能被正确的关闭。如果打开了一个数据库连接,`finally` 块可以确保连接被关闭。

285

```
public static Student findByLastName(String lastName)
    throws RuntimeException {
    java.sql.Connection dbConnection = null;
    try {
        dbConnection = getConnection();
        return lookup(dbConnection, lastName);
    }
    catch (java.sql.SQLException e) {
        throw new RuntimeException(e.getMessage());
    }
    finally {
        close(dbConnection);
    }
}
```

(注意:上面的例子只是为了演示。参考附加课程三,您可以获取通过 JDBC 与数据库交互的摘要。)

方法 `lookup` 和 `getConnection` 都可能产生 `SQLException` 对象。在获得数据库连接之后调用方法 `lookup`。如果随后的 `lookup` 方法抛出异常,您应该确保正确关闭数据库连接。

如果没有抛出异常,那么执行完 `try` 块中的代码之后,Java 直接将控制转移到 `finally` 块。本例中,`finally` 块调用了关闭数据库连接的方法。如果抛出了异常,那么 Java 执行 `catch` 块中的代码。一旦 `catch` 块执行完毕,或者虚拟机直接将控制转移到 `catch` 块之外(本例中,使用 `throw` 导致控制转移),Java 都会执行关闭数据库连接的 `finally` 块。

如果提供了 `finally` 块,那么 `catch` 块就是可选的。例如,您可能不打算在

`findByLastName` 中处理 SQL 异常，如果是这样的话，将方法 `findByLastName` 声明为可以抛出 `SQLException`。但是您依然需要在虚拟机将控制转移出 `findByLastName` 之前，关闭数据库连接。

```
public static Student findByLastName(String lastName)
    throws java.sql.SQLException {
    java.sql.Connection dbConnection = null;
    try {
        dbConnection = getConnection();
        return lookup(dbConnection, lastName);
    }
    finally {
        close(dbConnection);
    }
}
```

286

决不要在 `finally` 块中调用 `return` 语句——这样可能会吃掉 `catch` 块中抛出的异常。一些程序员甚至建议更严格的用法：在 `try` 块和 `catch` 块也不要调用 `return` 语句，仅在整个 `try-catch` 块之后调用 `return` 语句。

重构

将若干子字符串和基本类型以及对象的可打印形式串接起来，组成一个大的字符串。`Student` 的构造函数提供了一个例子：

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    final int maximumNumberOfNameParts = 3;
    if (nameParts.size() > maximumNumberOfNameParts) {
        String message =
            "Student name '" + fullName +
            "' contains more than " + maximumNumberOfNameParts +
            " parts";
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

在 `Student` 的构造函数中，您把五个分离的元素组合成一个完整的字符串：三个字符串，全名，表示名字单元的最大数目的常量。同样，您也必须在测试方法中构建这样的字符串。加入更多信息，连接更大的字符串，将使代码会变得难以阅读和维护。

类 `String` 提供了类方法 `format`。您可以向 `format` 方法中传入格式化字符串以及若干可变数目的参数（参考第7课的可变参数）。`format` 方法返回如您所需的、格式化好的字符串。格式化字符串中包含了格式化因子。这些格式化因子是参数的占位符。格式化因子告诉 `format`

方法如何解释和格式化每一个参数。通常，但并不总是，您需要为每一个参数指定一个格式化因子。

对于熟悉 C 语言的程序员，Java 的字符串格式化能力和 C 语言中的 `printf` 非常接近，但是 Java 提供了更多的功能，也更安全。不要假设 Java 格式化因子的执行效果和 C 语言的完全一样。

287

从 `testBadlyFormattedName` 开始，用 `format` 方法替换原有的字符串连接操作。同时，引入表示名字单元的最大数目的类常量。

```
public void testBadlyFormattedName() {
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        assertEquals(
            String.format("Student name '%s' contains more than %d parts",
                studentName, Student.MAX_NAME_PARTS),
            expectedException.getMessage());
    }
}
```

`format` 有三个参数：格式化字符串，两个格式化字符串参数。在格式化字符串中有两个格式化因子：`%s` 和 `%d`。第一个格式化因子对应第一个参数 `studentName`。第二个格式化因子对应第二个参数 `Student.MAX_NAME_PARTS`。格式化因子以百分号开头（`%`），以约定的转换字符结束。约定转换字符告诉 `format` 方法如何解释对应的参数。

`%s` 中的字符 `s` 表示字符串转换。当方法 `format` 遇到字符串转换因子，就用相应的参数替换该格式化因子。该测试中，方法 `format` 用 `studentName` 的内容替换 `%s`：

```
Student name 'a b c d' contains more than %d parts
```

`%d` 中的字符 `d`，表示将整型数值转换成十进制数值。该测试中，方法 `format` 用 `Student.MAX_NAME_PARTS`（整型常量，值为 3）的值来替换 `%d`：

```
Student name 'a b c d' contains more than 3 parts
```

`String` 的 `format` 方法是一个工具方法，该方法代理类 `java.util.Formatter` 所有的工作。类 `Formatter` 提供了超过一打的不同类型的转换，包括日期转换和更复杂的数值转换。

288

一种非常有用的转化因子是 `n`，`n` 代表换行。如果您提供了格式化因子 `%n`，那么就会用平台相关的换行符（通常是“`\n`”或者“`\r\n`”）来替换该因子。这样节省了您的时间——不用编写代码来获取系统的 `line separator` 属性。

`Formatter` 还提供下面的功能：

- 国际化
- 参数的顺序可以和格式化因子的顺序不匹配
- 以渐增的方式，将输出送到 `StringBuilder` 或者别的接收器

`Java.util.Formatter` 的文档非常长、并且特别详细。参考该文档，了解其它转换以及如何使用 `Formatter` 的其它功能。

最后，重构代码，用类常量表示错误消息。下面的代码是重构后的 `Student` 代码：

```
// StudentTest.java
public void testBadlyFormattedName() {
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        assertEquals(
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                studentName, Student.MAX_NAME_PARTS),
            expectedException.getMessage());
    }
}

// Student.java
static final String TOO_MANY_NAME_PARTS_MSG =
    "Student name '%s' contains more than %d parts";
...
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                fullName, MAX_NAME_PARTS);
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

日志

多数项目中，记录应用程序执行过程中有趣的历史信息是非常有价值的。日志代码的职责是记录重要的事件以及支持这些事件的数据。作为程序员，您的职责是判断如何记录日志才是充分的。一旦做出决定，您将日志代码插入到重要的地方，从而将信息写到某个日志文件或者一组日志文件。然后，您可以根据这些日志文件回答有关性能、出错等问题。

找出恰当的、必要的日志是最难的工作。现在，您没有做任何日志，所以针对发生的错误

丢失了所有的信息。对发生的错误缺乏相关的数据将剥夺您解决这些问题的能力。更糟糕的是，您可能直到太晚的时候（这是避免空 catch 块的原因之一）才发觉出了问题。

最低限度，您应该对所有不期望的错误条件（异常）进行日志。您也应该对关键算法、比较麻烦的事物处理、以及感兴趣的数据进行日志。事实上，您也可以对所有发生的事情进行日志，例如记录每一次方法调用。

记录所有的事情，或者记录的过多，都会造成问题。日志文件会以很快的速率增长。日志对性能会有小的影响，但是过多的日志会导致系统变慢。而且，一个更糟糕的问题是：如果记录了过多的日志，您将无法对海量的数据进行分析。问题会消失在森林里，日志也变得没有用处。从代码的角度看，日志也会使代码变的混乱和臃肿。

记录多少日志完全取决于您。如果可以比较容易地部署更新的代码（例如，web 应用程序），那么日志可以比较少。如果部署更新的代码比较困难，那么日志应该多一些。如果您知道某部分代码中可能存在问题、或者存在高风险（例如，控制关键组件的代码），那么就on应该记录比较多的信息。

Java 中的日志

Java 在包 `java.util.logging`⁶ 中提供了完整的日志功能。下面有一个练习，您将学习如何使用日志包来捕获异常事件。`Student` 的构造函数抛出一个异常，我们对其进行记录。这里对 `Student` 的构造函数做了微小的修改——增加了一个日志占位符。

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                          fullName, MAX_NAME_PARTS);
        // log the message here
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

第一个问题是：如何测试，当异常抛出后，正确记录了相应的信息。或者您是否需要为这样一个测试担忧？

请记住测试的首要原则：测试可能造成问题的任何代码。日志也的确可能会带来问题。所以进行下面的测试：

⁶ 尽管 Java 日志功能可以满足绝大多数需求，但是很多程序员更倾向于使用自由的 Log4j 包 (<http://logging.apache.org/log4j/docs>)。


```

public void testBadlyFormattedName() {
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                studentName, Student.MAX_NAME_PARTS);
        assertEquals(message, expectedException.getMessage());
        assertTrue(wasLogged(message));
    }
}

private boolean wasLogged(String message) {
    // ???
}

```

291

如何判断消息已经被记录呢？方法 `wasLogged` 的命名表现了您的意图。`wasLogged` 的内容将说明您将如何判断。

学习如何测试不熟悉的 API，例如日志 API。有时，我们带着游戏的态度来摆弄这些 API，直到理解它们如何工作为止。如果对这些 API 没有完整的理解，您不可能知道如何去测试它们。该练习中，为了更好的理解日志 API，您需要编写一些临时代码，然后再抛弃这些临时代码。编写测试，以及符合测试要求的生产代码。

在 `Student` 的构造函数中创建一个“有目的的消息”：

```

public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                fullName, MAX_NAME_PARTS);
        log(message);
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}

log 方法:

private void log(String message) {
    Logger logger = Logger.getLogger(getClass().getName());
    logger.info(message);
}

```

为了记录消息，您需要一个 `Logger` 对象。为了获得一个 `Logger` 对象，您需要调用 `Logger` 的工厂方法 `getLogger`。传入需要记录日志的子系统的名字作为 `getLogger` 的参数。在本课

稍后的部分（日志层次关系），您将了解更多关于子系统名字的信息。通常，传入类名就可以了。

如果针对某个子系统名字的 `Logger` 对象已经存在，那么 `getLogger` 返回已存在的日志对象。否则，`getLogger` 创建并返回一个新的日志对象。

方法 `log` 的第二行调用了日志方法 `info`，并且传入 `message` 作为参数。通过调用方法 `info`，您请求记录信息级别的日志。日志 API 支持多种等级的日志信息。类 `Logger` 针对每个等级提供了相应的方法。从最高到最低，一共有七个等级：严重（severe）、警告（warning）、信息（info）、配置（config）、略细（fine）、较细（finer）、最细（finest）。 292

每个日志对象都维护了它自己的日志等级。如果您试图记录一条比日志对象自身等级要低的日志信息，那么该日志对象会拒绝这条日志信息。例如，如果日志对象的等级被设置为“警告”，那么该日志对象只能记录“严重”和“警告”级别的信息。

调用方法 `getLevel` 和 `setLevel` 可以获取/设置某个日志对象的日志级别。利用 `Level` 对象可以表示日志级别。类 `Level` 定义了一组常量，每一个常量表示一种日志级别。类 `Level` 还提供了两种额外的类常量：`Level.ALL` 和 `Level.OFF`，分别用来表示记录所有的信息或者任何信息也不记录。

修改 `StudentTest` 的方法 `wasLogged`，返回 `false`。然后运行所有的测试。由于返回 `false`，所以导致 `testBadlyFormattedName` 测试失败。这条测试失败信息提醒您充实相应的测试。

在终端控制台，您可以看到类似下面的信息：

```
Apr 14, 2005 2:45:04 AM sis.studentinfo.Student log
INFO: Student name 'a b c d' contains more than 3 parts
```

您成功地记录了一条信息！

图 8.2 的 UML 类图展现了 `Logger` 和 `Level` 之间的关系，同时也包括了 `Logger` 和其它相关类的关系。您将在本课后面学习这些相关类。

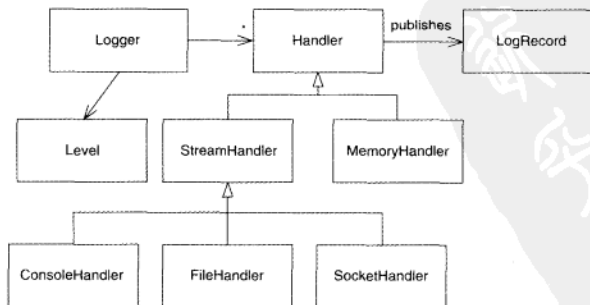


图 8.2 日志层次结构

测试日志

现在的问题是找到测试日志的办法。首先，就像我已经承诺的，您必须抛弃 `Student` 中的临时代码。删除 `log` 方法，并且删除构造函数中的 `log` 调用。

```
public Student(String fullName) {
    this.name = fullName;
    credits = 0;
    List<String> nameParts = split(fullName);
    if (nameParts.size() > MAX_NAME_PARTS) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                          fullName, MAX_NAME_PARTS);
        throw new StudentNameFormatException(message);
    }
    setName(nameParts);
}
```

不要忘了：编写没有测试的代码，容易陷入困境。首先，要对即将进行的编码指定相应的测试。

您已经成功地记录了一条信息，并将信息输出到终端控制台。但是，为了编写测试，如何截获这条消息呢？

Java 日志功能允许您直接将信息输出到除了控制台以外的其它目的地。您也可以一次将日志信息同时输出到多个目的地⁷。Logger 中存储用以表示日志目的地的 handler 对象。handler 对象是 `java.util.logging.Handler` 的子类的实例。类 `ConsoleHandler` 表示将输出发送到终端控制台的行为。多数开发团队倾向于使用 `FileHandler`，把日志输出重定向到文件，这样程序员可以仔细阅读这些日志文件。通过发送 handler 对象，您可以告诉 logger 将消息路由到不同目的地。

不幸的是，因为还没有学习 Java IO（您将在第 11 课学习 Java IO），所以暂时您还不能编写读取日志文件的测试代码。作为替代，您可以创建一个新的 Handler 类：TestHandler。在该类中捕获所有日志信息。另外您还需要提供返回最后一条日志信息的 getter 方法。一旦完成这个 Handler 类，可以将该 Handler 类的实例作为参数传入 logger。

对于每一条日志信息，Logger 对象都调用 Handler 的方法 `publish`，并将一个 `java.util.logging.LogRecord` 对象作为参数传入 `publish` 方法。

在 Handler 的子类 TestHandler 中，重载方法 `publish` 以捕获日志信息。同时，您也需要提供另外两个抽象方法的实现：`flush` 和 `close`，这两个方法可以为空。

```
package sis.studentinfo;
```

⁷ 图 8.2 的 UML 图通过在两个类之间使用星号 (*) 说明：一个 Logger 可以有多个 Handler 对象。星号 (*) 是多样性指示器。如果没有多样性指示器，那么表示 1，或者对多样性不感兴趣，或者跟多样性不相关，Logger 和 Handler 之间是一对多的关系。

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

class TestHandler extends Handler {
    private LogRecord record;

    public void flush() {}
    public void close() {}
    public void publish(LogRecord record) {
        this.record = record;
    }

    String getMessage() {
        return record.getMessage();
    }
}
```

在 `testBadlyFormattedName` 中, 首先获取 `Student` 所需要的、同样的 `logger`。因为 `Student` 的测试代码和实现代码都把 `Student` 类名作为参数传入 `getLogger`, 所以它们将引用相同的 `Logger` 对象。接着, 创建一个 `TestHandler` 实例, 并将其作为句柄添加到 `logger`。

```
public void testBadlyFormattedName() {
    Logger logger = Logger.getLogger(Student.class.getName());
    TestHandler handler = new TestHandler();
    logger.addHandler(handler);
    ...
}
```

您所记录的任何消息都会被路由到 `TestHandler` 对象。为了证实 `Student` 确实记录了日志, 您可以请求 `handler` 返回接受到的最后一条信息。

```
public void testBadlyFormattedName() {
    ...
    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                           studentName, Student.MAX_NAME_PARTS);
        assertEquals(message, expectedException.getMessage());
        assertTrue(wasLogged(message, handler));
    }
}

private boolean wasLogged(String message, TestHandler handler) {
    return message.equals(handler.getMessage());
}
```

从风格上讲, 我更倾向于下面的测试代码:

```
public void testBadlyFormattedName() {
    Logger logger = Logger.getLogger(Student.class.getName());
```

```

Handler handler = new TestHandler();
logger.addHandler(handler);

final String studentName = "a b c d";
try {
    new Student(studentName);
    fail("expected exception from 4-part name");
}
catch (StudentNameFormatException expectedException) {
    String message =
        String.format(Student.TOO_MANY_NAME_PARTS_MSG,
            studentName, Student.MAX_NAME_PARTS);
    assertEquals(message, expectedException.getMessage());
    assertTrue(wasLogged(message, (TestHandler)handler));
}
}

```

不再把 `TestHandler` 实例赋值给 `TestHandler` 引用，而是赋值给 `Handler` 实例。这样可以使测试更加清晰：方法 `addHandler` 期望以 `Handler` 作为参数（而不是 `TestHandler`）。方法 `wasLogged` 需要调用 `TestHandler` 中创建的 `getMessage` 方法，所以必须将 `handler` 强制类型转换成 `TestHandler`。

测试会失败，因为 `Student` 中没有符合测试要求的生产代码（您删除了它，不是吗？）。在继续之前，演示一下失败的测试。您知道，新的测试代码可能什么也没做，如果得到了绿条，上面的警告会消失。（有时候，您忘记了编译代码，这可能会发生。）坚持这种方法，可以使您保持冷静，并减少错误。

296

删除临时代码的另一个原因是：我们需要更好的、经过重构的版本。现在存在冗余的代码：`StudentTest` 和 `Student` 中都包含了复杂的获取 `Logger` 对象的代码。

除了在 `Student` 中使用类变量 `Student.logger` 以外，还在测试中内联 `wasLogged` 方法：

```

public void testBadlyFormattedName() {
    Handler handler = new TestHandler();
    Student.logger.addHandler(handler);

    final String studentName = "a b c d";
    try {
        new Student(studentName);
        fail("expected exception from 4-part name");
    }
    catch (StudentNameFormatException expectedException) {
        String message =
            String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                studentName, Student.MAX_NAME_PARTS);
        assertEquals(message, expectedException.getMessage());
        assertEquals(message, ((TestHandler)handler).getMessage());
    }
}

```

下面是经过重构的 `Student` 代码：

```

package sis.studentinfo;

import java.util.*;
import java.util.logging.*;

public class Student {
    ...
    final static Logger logger =
        Logger.getLogger(Student.class.getName());
    ...
    public Student(String fullName) {
        this.name = fullName;
        credits = 0;
        List<String> nameParts = split(fullName);
        if (nameParts.size() > MAX_NAME_PARTS) {
            String message =
                String.format(Student.TOO_MANY_NAME_PARTS_MSG,
                              fullName, MAX_NAME_PARTS);
            Student.logger.info(message);
            throw new StudentNameFormatException(message);
        }
        setName(nameParts);
    }
    ...
}

```

297

将日志定向到文件

Sun 将日志功能设计得非常灵活，允许您在运行时改变日志的特性。您可以很快地将日志信息的目的地从终端控制台改变到文件，与此同时，不需要修改代码、重新编译、以及重新部署。

缺省情况下，日志信息被输出到终端控制台。但是该行为并没有在类 `Logger` 中硬编码。您可以通过修改外部配置文件，来改变日志信息的输出行为。

浏览 Java 的安装目录。进入到子目录 `jre/lib`（如果使用 Windows，进入 `jre\lib`）。在该子目录下，您会看到文件 `logging.properties`⁸。使用任何编辑器都可以编辑该文件。

在很多环境中，您都可以使用属性配置文件，程序运行时可以改变配置文件中的属性，从而改变程序的行为。Sun 允许您使用属性配置文件来配置日志的行为。属性配置文件被设计成自解释型。在一行的开始使用 `#` 来表示注释行。空行被自动忽略。其余所有行都是成对的“key—value”，就像哈希表中的每一项。每一对“key—value”，或者叫属性，都是下面的形式：

```
key = value
```

打开 `logging.properties`，您将看到：

```

# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.

```

⁸ 如果没有该文件，请创建一个。

```
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.
handlers= java.util.logging.ConsoleHandler

# To also add the FileHandler, use the following line instead.
#handlers=java.util.logging.FileHandler,java.util.logging.ConsoleHandler
```

交换注释行。注释掉将 handlers 设置为 ConsoleHandler 的行，打开将 handlers 设置为 FileHandler 和 ConsoleHandler 的行。这样，您告诉 logger 同时把输出定向到文件和终端控制台。

在接近 logging.properties 的底部，您将看到下面的内容：

```
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

以上几行决定了 FileHandler 如何工作。

java.util.logging.FileHandler.pattern 对应的值定义了日志文件名的命名格式。例子中的 %u 和 %h 叫做域。域 %h 告诉 FileHandler 将日志文件存储在用户主目录。

FileHandler 会用一个唯一的数字来代替域 %u。这样做的目的是为了避免两个日志文件具有同一个文件名。

重新运行测试，然后进入到您的用户主目录。在多数 Windows 版本中，使用下面的命令⁹：

```
cd %USERPROFILE%
```

在多数 Unix shells 中，通过执行下面的命令可以进入用户主目录：

```
cd $HOME
```

在用户主目录中，您将看到文件 java0.log。数字 0 被用来替换域 %u。

打开文件 java0.log，您将看到：

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
<date>2004-04-15T03:27:05</date>
<millis>1082021225078</millis>
<sequence>0</sequence>
<logger>sis.studentinfo.Student</logger>
<level>INFO</level>
<class>sis.studentinfo.Student</class>
<method>log</method>
<thread>10</thread>
<message>Student name 'a b c d' contains more than 3 parts</message>
</record>
</log>
```

⁹ 您必须将当前驱动器切换到用户配置文件所在的驱动器。通过执行命令 set USERPROFILE，可以看到主目录的全路径。

这不是您所期望的只有两行日志信息的日志文件。这里的日志信息是 XML 格式¹⁰。Java 日志允许您为每一个 handler 指定不同的格式。每种格式都是 `java.util.logging.Formatter` 的子类。Sun 提供了两种格式实现：`SimpleFormatter` 可以生成如您所期望的两行日志信息；`XMLFormatter` 可以生成如您在 `java0.log` 中所看到的日志信息。您可以编写自己的格式类，来产生任何您所希望的日志输出格式。

299

在 `logging.properties` 中找到属性 `java.util.logging.FileHandler.formatter`，该属性的当前值是 `java.util.logging.XMLFormatter`。将该属性的值修改成 `java.util.logging.SimpleFormatter`，然后重新运行测试。

文件 `java0.log` 的内容和通过 `ConsoleHandler` 输出的结果一模一样。

日志的测试哲学

前一节中，您通过修改文件 `logging.properties`，改变了日志的行为。通过配置文件动态改变程序的行为是一种非常强大的功能，可以使系统变得更富有弹性。

注意：您编写的测试不关心日志输出到什么地方。测试代码只证明指定的消息被发送到了 logger 对象。您通过检查从 logger 发送到 handler 的消息，可以进行证明。

但是，日志输出目的地也是一种需要测试的需求细节。针对属性配置文件的修改很有可能无效的。当应用程序发布以后，这些无效的修改可能造成严重的问题。所以，您不仅要测试 Java 代码，也要测试系统的配置文件会怎样影响 Java 代码。

使用 JUnit 测试确保正确地创建了日志文件¹¹。测试策略应该是：

1. 用正确的数据编写配置文件。
2. 强制日志系统加载配置文件。
3. 执行记录日志的代码。
4. 读取日志文件，确保文件中包含期望的日志信息。

您也可以在集成测试（有些开发团队称之为客户测试，或者系统测试，或者验收测试）中考虑关于配置文件的测试。您已经测试了单元测试代码——`Student`——和日志系统进行了正确的交互。日志系统在动态配置的情况下是否能够正常工作——这已经超出了单元测试的范围。

300

不管怎样定义——不管您是否考虑这样的单元测试——对于要交付的系统，测试其配置子系统都是必须的。您可以选择手动测试，就像前一节那样观察日志文件的内容。您也可以将上面的四步骤策略变成一个可执行的集成测试。您可以将该测试作为任何回归测试套件的一部分。

¹⁰ 可扩展标记语言。参考 <http://www.w3.org/XML/>。

¹¹ 一旦看到关于如何处理文件的代码，请参考第 11 课。

回归测试通过执行一套针对整个系统的完整的测试集合，从而确保新的改动不会破坏已有的代码。

日志是一种支持性工具，就像其它功能需求一样，也是一种系统需求。某些级别的测试日志是绝对必要的。但是，您是否应该为每一条日志信息都编写一个测试？

简单的回答是“不”。一旦证明构建日志系统的基本机制是正确的，您就可以认为编写其它的日志测试是在浪费时间。通常，日志系统的唯一作用是：程序员希望通过日志来分析代码究竟在做什么。增加新的关于 logger 的调用不大可能破坏现有的系统。

更好的答案是“是”。针对每一条日志信息维护一个测试，这是令人厌烦的。但是，它会避免因为某些新的原因而使您陷入困境。

很多程序员使用 try-catch 块，是因为编译器要求必须这样做，而不是因为规格要求或者为了测试。他们没有处理异常发生的解决方案。对于这些程序员，往往不假思索的在 catch 块中记录一条日志信息。这样做导致了容易隐藏严重错误的“空 catch 子句”。

通过为所有的日志信息编写测试，您无需再面对“空 catch 子句”的问题。但是还有两种情况会发生：第一，为了编写测试，您必须知道如何模拟被怀疑的异常。有时候，仅是这个模拟过程，就可以教会您如何消除“空 catch 子句”。第二，测试将愚蠢的“空 catch 子句”变得更加可视，测试可以帮助您发现潜在的问题。

为每一条日志信息编写测试的另一个“好处”是：它会带来痛苦。有时候痛苦是一件好事。痛苦迫使您仔细考虑每一条可能进入系统的日志信息。“我确实需要在这里加上一条日志信息吗？这样有什么好处？这样是否符合开发团队的日志策略？”如果针对每一个问题都仔细寻求正确的答案，那么您将避免过度日志带来的严重问题。

301

最后，日志代码可能导致现有系统无法正常工作。虽然不绝对，但是有可能。

更多关于 FileHandler

FileHandler 定义了很多域，包括 `%t` 和 `%g`。FileHandler 用系统临时目录代替域 `%t`。在 Windows 中，通常是 `c:\temp`；在 Unix 中，通常是 `/tmp`。

FileHandler 用循环计数代替域 `%g`。您可以联合使用 `java.util.logging.FileHandler.count`（后面我将用简称 `count`）和 `java.util.logging.FileHandler.limit(limit)` 属性。属性 `limit` 代表以字节为单位的日志文件长度的上限。如果 `limit` 的值为 0，那么表示对日志文件的大小不做限制。当日志文件的大小达到了上限，FileHandler 对象会关闭该文件。域 `count` 指定 FileHandler 可以循环使用多少个日志文件。如果 `count` 的值为 1，那么每当达到上限，FileHandler 会继续使用原来的日志文件。

否则，FileHandler 会保持一个循环计数。首先，日志文件使用循环计数 0 来代替域 `%g`。如

果存在一个日志文件名模板 `java%g.log`，那么替换后的文件名是 `java0.log`。每当日志文件的大小达到上限，`FileHandler` 增加循环计数，用新的循环计数来代替 `%g`。`FileHandler` 将新的日志记录到新的日志文件中。这里，第二个日志文件是 `java1.log`。一旦 `FileHandler` 关闭了最大循环计数(count)的日志文件，那么将循环计数重新设置为 0。

日志等级

您也许希望针对某些特殊目的日志信息，例如记录某个问题方法的执行时间的信息、引入方便程序员跟踪系统执行流程的消息。程序正常执行的时候，此类信息不会出现在日志中。但是，需要具有随时开关此类信息的能力，与此同时，不用重新编码、重新编译、重新部署。

就像前面所提到的，Java 日志 API 支持七种日志等级：严重 (severe)、警告 (warning)、信息 (info)、配置 (config)、略细 (fine)、较细 (finer)、最细 (finest)。您应该限制正常生产系统只记录严重、警告和信息等级的日志。其它等级的日志只在特殊的情况下短暂的使用。

假设您需要记录 `Student` 方法 `getGpa` 的执行时间，因为您关心是否该方法在重负载的情况下也具有良好的表现。在 `getGpa` 的开头和结束的地方，使用 `Logger` 的 `fine` 方法记录日志。

302

```
double getGpa() {
    Student.logger.fine("begin getGpa " + System.currentTimeMillis());
    if (grades.isEmpty())
        return 0.0;
    double total = 0.0;
    for (Grade grade: grades)
        total += gradingStrategy.getGradePointsFor(grade);
    double result = total / grades.size();
    Student.logger.fine("end getGpa " + System.currentTimeMillis());
    return result;
}
```

重新编译和运行测试¹²。在终端控制台和日志文件中，都看不到新的日志信息。

为了能够看到新的日志信息，您必须配置 handler 接受更低等级的日志。通常，您可以修改 `logging.properties` 达到目的。下面黑体的部分就是修改的部分：

```
...
.level= FINE

#####
# Handler specific properties.
# Describes specific configuration info for Handlers.
#####
```

¹² 这里，我选择采用延迟日志——不测试——因为这是临时日志信息。

```
# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.level = FINE
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
...
```

将全局等级设置 (level) 和 FileHandler 设置 (java.util.logging.FileHandler.level) 改为可以接受略细以及略细等级以上的任何日志信息。Java 日志系统首先检查请求的日志等级是否和全局日志等级一样或者更高; 如果不是, 就不将此条消息发送到 handler。如果 logger 将消息发送到 handler, handler 同样也需要判断是否应该记录日志。

换句话说, 全局日志等级应该等于或者低于任何 handler 等级, 如果不是这样, 那么不会有任何日志。例如, 如果将全局等级设置为 INFO, 将 FileHandler 设置为 FINE, 那么 FileHandler 将不会记录任何日志。

重新运行测试, 保证日志消息出现在日志文件中, 而不是终端控制台。

日志层次结构

通过 Logger 的类方法 getLogger, 可以获取一个新的 Logger 对象, 或者返回已经存在的、使用同一个名字创建的 Logger 对象。调用 Logger.getLogger(Student.class.getName()), 将会以“sis.studentinfo.Student”为名字创建一个 logger。通常, 我们使用类名的全称来创建 Logger。

接下来的语言测试展示: 两次使用同一个名字调用 getLogger, 将返回相同的 Logger 对象:

```
public void testLoggingHierarchy() {
    Logger logger = Logger.getLogger("sis.studentinfo.Student");
    assertTrue(logger == Logger.getLogger("sis.studentinfo.Student"));
}
```

类名全称表示了类的层次关系。层次关系 (或者叫树) 的最上层是 sis, 下一层是 studentinfo。studentinfo 中的每一个类都是树上的一片叶子。浏览编译生成的 class 文件的目录结构, 您可以更清楚地看到类的层次关系。

日志可以利用类命名层次关系的优点。Java 日志系统将 Logger 对象组织成类似的层次关系。如果您创建了名字为 sis.studentinfo.Student 的 logger, 那么它的父 Logger 是 sis.studentinfo。sis.studentinfo 的父 Logger 是 sis。请看接下来的语言测试:

```

public void testLoggingHierarchy() {
    Logger logger = Logger.getLogger("sis.studentinfo.Student");
    assertTrue(logger == Logger.getLogger("sis.studentinfo.Student"));

    Logger parent = Logger.getLogger("sis.studentinfo");
    assertEquals(parent, logger.getParent());
    assertEquals(Logger.getLogger("sis"), parent.getParent());
}

```

这种层次关系的好处在于：您可以在较高的层次设置日志等级。例如，可以将 `sis` logger 的日志等级设置为 `Level.ALL`。子 logger 如果没有设置自己的日志等级，那么就使用父亲的日志等级。

304

日志补充说明

- Java 日志系统允许向 logger 或者 handler 添加过滤器。过滤器接口定义了 `isLoggable` 方法，该方法以 `LogRecord` 为参数。如果记录信息，那么 `isLoggable` 返回 `true`；如果不记录，那么 `isLoggable` 返回 `false`。例如，您可以编写一个忽略任何超过某一长度的日志信息的过滤器实现。
- Java 日志系统支持国际化。参考 Java API 文档可以获得更多信息。附加课程三对国际化进行了简要介绍。
- 可以在程序执行过程中修改日志属性配置文件。一旦修改了日志属性配置文件，可以调用类 `LogManager` 的方法 `readConfiguration` 来重新加载属性配置文件。关于如何用输入流表示属性文件，请参考第 11 课“如何通过输入流读取文件”。
- 您知道在运行时，可以修改 `logging.properties` 来重新定义日志特性。您也可以自己定制属性配置文件，而不使用 `logging.properties`。为了实现定制，您需要设置系统属性 `java.util.logging.config.file`。在命令行启动 Java 程序时，也可以实现定制，例如：

```
java -Djava.util.logging.config.file=sis.properties sis.MainApp
```

- 类 `Logger` 针对特殊情况，提供了一些便利的方法。通过使用 `entering` 和 `exiting`，可以简单记录每个方法的调用和退出。使用方法 `throwing` 可以简单记录每一次异常。请参考 Java API 文档，获取有关日志的更多信息。

练习

305

1. 编写调用方法 `blowsUp` 的测试。编写方法 `blowsUp`，该方法抛出携带消息“Somebody should catch this!”的 `RuntimeException` 异常。运行测试。验证测试失败，并且查看相应的异常堆栈跟踪。
2. 在不删掉 `blowsUp` 调用的情况下，使测试得以通过。如果 `blowsUp` 不抛出异常，确保测试失败。
3. 确保捕获的异常中包含正确的消息。改变消息，观察测试失败。再将消息改回到最初的形式，让测试得以通过。
4. 创建调用新方法 `rethrows` 的测试。方法 `rethrows` 调用 `blowsUp`，并且捕获异常，然后在一个新的 `RuntimeException` 中包裹捕获到的异常，并重新抛出。使用 `getCause`，确保捕获到的异常中包含最初的异常。
5. 修改测试，当调用方法 `blowsUp` 时，期望一种新的异常类型 `SimpleException`。`SimpleException` 继承自 `RuntimeException`。
6. 下面的哪一个测试方法无法编译？在编译能够通过的测试方法当中，哪些失败，哪些成功？

```
public void testExceptionOrder1() {
    try {
        blowsUp();
        rethrows();
        fail("no exception");
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
    catch (RuntimeException success) {
    }
}

public void testExceptionOrder2() {
    try {
        rethrows();
        blowsUp();
        fail("no exception");
    }
    catch (SimpleException success) {
    }
    catch (RuntimeException failure) {
        fail("caught wrong exception");
    }
}

public void testExceptionOrder3() {
```

```
try {
    blowsUp();
    rethrows();
    fail("no exception");
}
catch (RuntimeException success) {
}
catch (SimpleException yours) {
    fail("caught wrong exception");
}
}

public void testExceptionOrder4() {
    try {
        blowsUp();
        rethrows();
        fail("no exception");
    }
    catch (RuntimeException fail) {
        fail("exception unacceptable");
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
    finally {
        return;
    }
}

public void testExceptionOrder5() {
    try {
        blowsUp();
        rethrows();
        fail("no exception");
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
    catch (RuntimeException success) {
    }
}

public void testExceptionOrder6() {
    try {
        rethrows();
        blowsUp();
        fail("no exception");
    }
    catch (SimpleException yours) {
        fail("caught wrong exception");
    }
    catch (RuntimeException success) {
    }
}

public void testExceptionOrder7() {
    try {
        rethrows();
        blowsUp();
        fail("no exception");
    }
}
```

306

307

```
        catch (SimpleException success) {
        }
        catch (RuntimeException fail) {
            fail("caught wrong exception");
        }
    }

    public void testErrorException1() {
        try {
            throw new RuntimeException("fail");
        }
        catch (Exception success) {
        }
    }

    public void testErrorException2() {
        try {
            new Dyer();
        }
        catch (Exception success) {
        }
    }

    public void testErrorException3() {
        try {
            new Dyer();
        }
        catch (Error success) {
        }
    }

    public void testErrorException4() {
        try {
            new Dyer();
        }
        catch (Throwable success) {
        }
    }

    public void testErrorException5() {
        try {
            new Dyer();
        }
        catch (Throwable fail) {
            fail("caught exception in wrong place");
        }
        catch (Error success) {
        }
    }

    public void testErrorException6() {
        try {
            new Dyer();
        }
        catch (Error fail) {
            fail("caught exception in wrong place");
        }
        catch (Throwable success) {
        }
    }
}
```

```

    }

    public void testErrorException7() {
        try {
            new Dyer();
        }
        catch (Error fail) {
            fail("caught exception in wrong place");
        }
        catch (Throwable success) {
        }
        finally {
            return;
        }
    }

    Dyer:
    class Dyer {
        Dyer() {
            throw new RuntimeException("oops.");
        }
    }
}

```

7. 下面的代码会产生什么样的编译错误？把下面的代码拷贝到您自己的测试类中，并使其通过测试。

```

public void testWithProblems() {
    try {
        doSomething();
        fail("no exception");
    }
    catch (Exception success) {}
}

void doSomething() {
    throw new Exception("blah");
}

```

8. 下面的代码存在什么错误？

```

public void doSomething() {
    try {
        complexOperationWithSideEffects();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

9. 编写一个记录异常的方法，并且以相反的顺序记录堆栈跟踪（错误信息的源头放在日志信息的尾部）。
10. 定制一个 Handler，该 Handler 放弃收到的任何信息，但是根据不同的日志等级分别对收到的信息的个数进行统计。使用 Map 来存储不同日志等级的信息计数。
11. 定制一个日志信息格式化类，可以传入 CountingLogHandler 作为参数。如果没有传入

CountingLogHandler，那么该格式化类会产生如下形式的输出：

```
LEVEL: message
```

例如：

```
WARNING: watch out
```

如果传入 CountingLogHandler，那么对于当前等级的每一条日志信息，都会显示其当前计数。例如：

```
WARNING: watch out (WARNING total = 1)
```

确保在两种情况下，测试都可以通过。

然后，让 CountingLogHandler 以定制的格式化类作为缺省值。修改 CountingLogHandler，将格式化输出存储到 StringBuilder 对象，这样测试可以要求得到完整的日志摘要。

最后，编辑日志属性配置文件，将定制的格式化类赋值给 ConsoleHandler。观察并确保发送到 ConsoleHandler 的日志信息和预期的一致。

Map 和相等性

这一课，您将学习 Java 中的哈希表数据结构类，以及相等性的基本概念。理解这两个知识点以及相关主题，对于掌握 Java 非常关键。但是，很多有着多年 Java 开发经验的程序员对这些概念也不是完全理解。缺乏对这些概念的完全理解，容易在编程中产生难以发觉的错误、以及严重的性能问题。不过，在正式开始之前，您需要了解一个同样重要的主题：逻辑操作符。

本课内容包括：

- 逻辑操作符
- 哈希表
- 相等性
- toString
- Map 和 Set 实现
- 字符串和相等性

逻辑操作符

假设需要在两个条件都为真的情况下，执行一行代码。您可以使用多层 if 语句进行条件判断：

```
if (isFullTime)
    if (isInState)
        rate *= 0.9;
```

使用逻辑运算符，可以把多个条件语句组合成为一条复杂的布尔表达式：

```
if (isFullTime && isInState)
    rate *= 0.9;
```

&& 是逻辑与操作符。逻辑与是二元操作符——有两个独立的布尔表达式作为运算符。如果

两个运算符的结果都为 true，那么整个条件表达式的结果为 true。如果任何一个运算符的结果为 false，那么整个条件表达式的结果为 false。下面是一个表示所有可能逻辑操作的、测试驱动真值表 (TDD¹)：

```
assertTrue(true && true);
assertFalse(true && false);
assertFalse(false && true);
assertFalse(false && false);
```

逻辑或 (||) 也是二元操作符。任何一个运算符为 true 或者两个运算符都为 true，那么整个条件表达式为 true。只有两个运算符都为 false，整个条件表达式才为 false。相应的测试驱动真值表如下：

```
assertTrue(true || true);
assertTrue(true || false);
assertTrue(false || true);
assertFalse(false || false);
```

逻辑非 (!) 是一元操作符，返回和某个表达式相反的布尔值。如果源表达式为 true，那么整个表达式的结果为 false，反之亦然。相应的测试驱动真值表如下：

```
assertFalse(!true);
assertTrue(!false);
```

逻辑异或 (^) 是二元操作符，但是很少使用。只有两个运算符都返回相同的值，整个表达式的值才为 false。相应的测试驱动真值表如下：

```
assertFalse(true ^ true);
assertTrue(true ^ false);
assertTrue(false ^ true);
assertFalse(false ^ false);
```

可以把多个逻辑操作组合在一起。如果没有提供括号，那么从高到低的优先级顺序为：！大于&&，&&大于^^，最后是||。

312

短路

操作符&&和||都是短路逻辑操作符。当 Java 虚拟机遇到短路逻辑操作符，首先计算最左边的运算符，最左边运算符的结果可能立刻决定整个表达式的值。

对于逻辑或 (||)，如果最左边运算符的值为 true，那么就没有必要计算右边的其它运算

¹ 一个自造、但是很准确的缩写。

子，可以直接判断整个表达式的值为 `true`。Java 虚拟机通过只计算表达式的一部分，从而节约时间。对于逻辑与 (`&&`)，如果最左边运算符的值为 `false`，那么整个表达式的值为 `false`，Java 虚拟机不会计算右边的其它运算符。

您很少需要执行表达式中的所有运算符，然而有时候你会希望右边的运算符做一些事情：这是一种不好的设计。第 4 课中指出，方法要么返回信息，要么影响行为，而不是两者兼有。同样，运算符应该被视为原子操作：要么是命令、要么是查询。您应该重构代码，让行为发生在复杂条件语句之前。

如果您仍然感到必须要执行表达式中所有的运算符，您可以使用非短路逻辑操作符。非短路逻辑与操作符是 `&`。非短路逻辑或操作符是 `|`。

注意：异或 (`^`) 永远是非短路操作符。Java 不能通过单个运算符，知道整个表达式的值。参考前面的测试驱动真值表，您就可以发现原因。

哈希表



现在，您需要创建一个包含所有学生的学生目录。系统给每个学生指定唯一的 ID。并且可以通过 ID 从学生目录中获取相应的学生信息。

为了实现这个 story，您需要创建类 `StudentDirectory`。尽管可以使用 `ArrayList` 存储所有的学生对象，但是这里我们使用 `Map`。第 6 课中您已经对 `Map` 有过一些了解。`Map` 是一种基于关键字存取元素的数据结构。`Map` 是包含若干 “key-value” 对的集合。首先，把关键字以及对应的值存入 `Map`，然后以关键字为参数从 `Map` 中获取相应的值。

313

第 6 课中，您学习了实现接口 `Map` 的 `EnumMap`。如果 `Map` 中的关键字为枚举类型，那么您应该使用 `EnumMap`。否则，通常使用 `Map` 的另一个实现 `java.util.HashMap`。`HashMap` 是一种基于哈希表的通用 `Map`。多数有关 `Map` 的代码都利用 `HashMap` 实现（或者使用遗留 `HashTable` 实现，参考第 7 课）。

哈希表用一种特殊的方法实现了接口 `map`。哈希表的设计目标是：快速插入和获取。

为了构建学生目录，首先从测试开始：

```
package sis.studentinfo;

import junit.framework.*;
import java.io.*;

public class StudentDirectoryTest extends TestCase {
    private StudentDirectory dir;

    protected void setUp() {
        dir = new StudentDirectory();
    }
}
```

```

public void testStoreAndRetrieve() throws IOException {
    final int numberOfStudents = 10;

    for (int i = 0; i < numberOfStudents; i++)
        addStudent(dir, i);

    for (int i = 0; i < numberOfStudents; i++)
        verifyStudentLookup(dir, i);
}

void addStudent(StudentDirectory directory, int i)
    throws IOException {
    String id = "" + i;
    Student student = new Student(id);
    student.setId(id);
    student.addCredits(i);
    directory.add(student);
}

void verifyStudentLookup(StudentDirectory directory, int i)
    throws IOException {
    String id = "" + i;
    Student student = dir.findById(id);
    assertEquals(id, student.getLastName());
    assertEquals(id, student.getId());
    assertEquals(i, student.getCredits());
}
}

```

314

测试 `testStoreAndRetrieve` 使用 `for` 循环, 在该循环中创建若干 `Student` 对象, 并且把这些对象插入到 `StudentDirectory`。在第二个 `for` 循环中, 验证是否可以通过方法 `findById` 获取学生目录中的每一个 `Student` 对象。

`StudentDirectory` 的代码非常简单。

```

package sis.studentinfo;

import java.util.*;

public class StudentDirectory {
    private Map<String, Student> students =
        new HashMap<String, Student>();

    public void add(Student student) {
        students.put(student.getId(), student);
    }

    public Student findById(String id) {
        return students.get(id);
    }
}


```

`StudentDirectory` 封装了一个 `HashMap` 的实例 `students`。客户调用方法 `add`, 并以 `Student` 对象为参数, 从而把 `Student` 对象插入到 `HashMap`。插入的 `Student` 对象以学生 ID 为关键字。

ID 必须是唯一的。调用 `put` 方法插入新值，如果新值对应的 ID 在哈希表中已经存在，那么新值会覆盖哈希表中原来的值。

本课程的后面，您将再次学习哈希表。但是，首先您需要学习相等性。在学习相等性之前，需要先实现一个 `story`。

课程

 `Session` 表示给定开始时间的某门课程的课程安排。多数课程一个学期只教授一次。可能多个 `Session` 实例²中都含有同一门课程的课程信息。

目前，类 `Session` 中包含科目、课程编号、以及学分。如果一门课程对应多个 `Session` 实例，那么现在的实现由于每个 `Session` 对象中都包含课程信息，所以效率低下、而且容易出问题。

◀ 315

改进的方法是：用多个 `Session` 对象对应一个 `Course` 对象。UML 图 9.1 展示了从类 `Session` 到类 `Course` 的多对一关系。

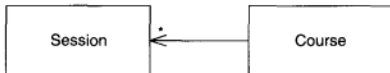


图 9.1 一门课程对应多个课程安排

重构代码，实现上面的多对一关系。首先创建包含科目、课程编号、学分的类 `Course`。编写一个简单的测试：以科目和课程编号为参数，创建一个 `Course` 对象。科目和课程编号联合起来，作为 `Course` 的关键字。两门不同的 `Course` 不能有相同的科目和课程编号。

```

package sis.studentinfo;

import junit.framework.*;

public class CourseTest extends TestCase {
    public void testCreate() {
        Course course = new Course("CMSC", "120");
        assertEquals("CMSC", course.getDepartment());
        assertEquals("120", course.getNumber());
    }
}
  
```

目前，`Course` 只是一个简单的数据对象，包含一个构造函数和两个 `getter` 方法。

```

package sis.studentinfo;

public class Course {
  
```

² 这是一个小型学校，所以忽略这样的可能性：为了容纳大量的学生，需要将 `Session` 分成两个部分。

```

private String department;
private String number;

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

public String getDepartment() {
    return department;
}

public String getNumber() {
    return number;
}
}

```

316

重构 Session

下面是重构的步骤：

1. 使用 `Course` 对象来构造 `Session` 对象。受影响的类有：`SessionTest`，`Session`，`CourseSessionTest`，`SummerCourseSessionTest`，`RosterReportTest`，`CourseReportTest`。
2. 修改 `CourseSession` 和 `SummerCourseSession` 的创建方法，接受 `Course` 为参数。
3. 修改 `CourseSession` 和 `SummerCourseSession` 的构造函数，接受 `Course` 为参数。
4. 修改 `Session` 的构造函数，接受 `Course` 为参数。
5. 在 `Session` 中存储一个 `Course` 引用，而不再存储科目、课程编号。

以渐增的方式进行以上重构。每次修改之后，利用编译器帮助您发现问题。然后运行测试，以确保没有造成任何破坏。

首先重构 `SessionTest`：

目前，抽象方法 `createSession` 以科目、课程编号为参数，创建 `Session` 对象：

```

abstract protected Session createSession(
    String department, String number, Date startDate);

```

修改上面的方法，以 `Course` 对象为参数，创建 `Session` 对象：

```

abstract public class SessionTest extends TestCase {
    ...
    abstract protected Session createSession(
        Course course, Date startDate);
}

```

上面的改动使得其它部分的修改成为必要，不过所有修改所花的时间应该不会超过五分钟。在 `SessionTest` 中，需要修改方法 `setUp` 和 `testComparable`。下面是修改前的 `setUp` 方法（黑

体是需要修改的代码):

317

```
public void setUp() {
    startDate = createDate(2003, 1, 6);
    session = createSession("ENGL", "101", startDate);
    session.setNumberOfCredits(CREDITS);
}
```

下面是修改后的 setUp 方法; 用类似的方法修改 testComparable:

```
protected void setUp() {
    startDate = new Date();
    session = createSession(new Course("ENGL", "101", startDate);
    session.setNumberOfCredits(CREDITS);
}
```

在重构过程中, 使用编译器作为您的向导。每当解决一个问题, 编译器马上告诉您下一个需要解决的问题。

您需要修改 SummerCourseSessionTest 和 CourseSessionTest。渐增地进行修改, 尽量每次只完成可以保证测试通过的、最小规模的改动。您可以在不改动 CourseSession 和 SummerCourseSession 的情况下, 修改 CourseSessionTest 和 SummerCourseSessionTest。

下面是对这些测试的修改。这一次, 我保留旧版的代码, 但是用删除线把它们标注出来, 这样您可以看到新旧两版代码的区别。对于其它重构, 我也采用这种方式。

```
// CourseSessionTest
...
public class CourseSessionTest extends SessionTest {
    public void testCourseDates() {
        Date startDate = DateUtil.createDate(2003, 1, 6);
Session session = createSession("ENGL", "200", startDate);
        Session session = createSession(createCourse(), startDate);
        Date sixteenWeeksOut = createDate(2003, 4, 25);
        assertEquals(sixteenWeeksOut, session.getEndDate());
    }

    public void testCount() {
        CourseSession.resetCount();
createSession("", "", new Date());
        createSession(createCourse(), new Date());
        assertEquals(1, CourseSession.getCount());
createSession("", "", new Date());
        createSession(createCourse(), new Date());
        assertEquals(2, CourseSession.getCount());
    }

    private Course createCourse() {
        return new Course("ENGL", "101");
    }

protected Session createSession(
String department, String number, Date date){
return CourseSession.create(department, number, date);
}
```

318


```

→
protected Session createSession(Course course, Date date) {
    return CourseSession.create(
        course.getDepartment(), course.getNumber(), date);
}

// SummerCourseSessionTest.java
package sis.summer;

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

public class SummerCourseSessionTest extends SessionTest {
    public void testEndDate() {
        Date startDate = DateUtil.createDate(2003, 6, 9);
Session session = createSession("ENGL", "200", startDate);
        Session session =
            createSession(new Course("ENGL", "200"), startDate);
        Date eightWeeksOut = DateUtil.createDate(2003, 8, 1);
        assertEquals(eightWeeksOut, session.getEndDate());
    }

protected Session createSession(
String department, String number, Date date) {
return SummerCourseSession.create(department, number, date);
→
protected Session createSession(Course course, Date date) {
    return SummerCourseSession.create(
        course.getDepartment(), course.getNumber(), date);
}
}

```

此时，测试会通过。现在，修改 Session 的创建方法，直接接受 Course 参数。

```

// CourseSessionTest.java

protected Session createSession(Course course, Date date) {
return CourseSession.create(
course.getDepartment(), course.getNumber(), date);
    return CourseSession.create(course, date);
}

// SummerCourseSessionTest.java
protected Session createSession(Course course, Date date) {
return SummerCourseSession.create(
course.getDepartment(), course.getNumber(), date);
    return SummerCourseSession.create(course, date);
}

```

319

上面的修改会影响到 RosterReportTest:

```

package sis.report;

import junit.framework.TestCase;

```

```

import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;

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

```

同时, 也会影响到 CourseReportTest³:

```

package sis.report;

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

public class CourseReportTest extends TestCase {
    public void testReport() {
        final Date date = new Date();
        CourseReport report = new CourseReport();
        report.add(CourseSession.create("ENGL", "101", date));
        report.add(CourseSession.create("CZEC", "200", date));
        report.add(CourseSession.create("ITAL", "410", date));
        report.add(CourseSession.create("CZEC", "220", date));
        report.add(CourseSession.create("ITAL", "330", date));
        report.add(create("ENGL", "101", date));
        report.add(create("CZEC", "200", date));
        report.add(create("ITAL", "410", date));
        report.add(create("CZEC", "220", date));
        report.add(create("ITAL", "330", date));

        assertEquals(
            String.format(
                "CZEC 200%n" +
                "CZEC 220%n" +
                "ENGL 101%n" +
                "ITAL 330%n" +
                "ITAL 410%n"),
            report.text());
    }

    private Session create(String name, String number, Date date) {
        return CourseSession.create(new Course(name, number), date);
    }
}

```

320

³ 使用您学过的 Java 特性, 修改 RosterReporterTest 和 CourseReportTest。例如, 将 CourseSession 对象赋值给 Session 引用。这样做, 也会强制您把 RosterReporter 改为 CourseReport。

修改 CourseSession 和 SummerCourseSession 的创建方法，接受 Course 参数。但是——现在——继续将科目和课程编号作为参数，传入构造函数。

```
// CourseSession.java
public static Session create(
String department, String number, Date startDate) {
incrementCount();
return new CourseSession(department, number, startDate);
}
public static Session create(Course course, Date startDate) {
    incrementCount();
    return new CourseSession(
        course.getDepartment(), course.getNumber(), startDate);
}

// SummerCourseSession.java
public static SummerCourseSession create(
String department, String number, Date startDate) {
return new SummerCourseSession(department, number, startDate);
}

public static Session create(Course course, Date startDate) {
    return new SummerCourseSession(
        course.getDepartment(), course.getNumber(), startDate);
}
```

测试通过。现在，把 Course 传入构造函数，重新运行测试。

```
// CourseSession.java
public static Session create(Course course, Date startDate) {
    incrementCount();
    return new CourseSession(course, startDate);
}

protected CourseSession(
String department, String number, Date startDate) {
super(department, number, startDate);
}

protected CourseSession(Course course, Date startDate) {
    super(course.getDepartment(), course.getNumber(), startDate);
}

// SummerCourseSession.java
public static Session create(Course course, Date startDate) {
    return new SummerCourseSession(course, startDate);
}

private SummerCourseSession(
String department, String number, Date startDate) {
super(department, number, startDate);
}

private SummerCourseSession(Course course, Date startDate) {
    super(course.getDepartment(), course.getNumber(), startDate);
}
```

乏味吗？或许。简单吗？重构的每一步都很基础，花上几秒钟就可以完成。安全吗？当然特别安全。总共花费的时间只有几分钟，但是一切都正常。在渐增重构的过程中，您运行了若干次 JUnit 测试，并且测试通过。重构方法的权威，Martin Fowler，建议永远不要过于频繁的执行测试⁴。

另一种方法是：一次修改完所有的代码，期望修改后的代码可以正常工作。这样做，您也许可以节约几分钟时间，但是存在犯错误的可能性。修改错误所花的时间会远远超过前面节约的时间。

您已经接近成功了。将 Course 对象放入基类 Session 的构造函数。

```
// CourseSession.java
protected CourseSession(Course course, Date startDate) {
    super(course, startDate);
}

// SummerCourseSession.java
private SummerCourseSession(Course course, Date startDate) {
    super(course, startDate);
}

// Session.java
protected Session{
    String department, String number, Date startDate){
    this.department = department;
    this.number = number;
    this.startDate = startDate;
}
protected Session(Course course, Date startDate) {
    this.department = course.getDepartment();
    this.number = course.getNumber();
    this.startDate = startDate;
}
```

322

最后，修改类 Session，存储 Course，而不是字符串 department 和 number。

```
// Session.java
abstract public class Session implements Iterable<Student> {
    private String department;
    private String number;
    private Course course;
    // ...
    protected Session(Course course, Date startDate) {
        this.course = course;
        this.startDate = startDate;
    }

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

⁴ [Fowler2000], p.94.

```
String getNumber() {
    return number;
    return course.getNumber();
}
// ...
```

相等性



在后面的课程中，您将创建一个课程目录。课程目录的要求之一是：目录中不能有重复的课程。如果两门课程的科目和课程编号都相同，那么两门课程在语义上是相同的课程。

现在已经有可用的类 `Course`。接下来需要重载 `equals` 方法，以提供相等性的语义学定义。两个 `Course` 对象，如果它们的科目和课程编号都相同，那么对这两个 `Course` 对象进行比较的结果应该为 `true`。

323

```
// in CourseTest.java:
public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);
}
```

测试失败。回顾第一课中的内容，我们知道对象 `courseAPrime` 和对象 `courseA` 的内存地址并不相同。方法 `assertEquals` 可以被粗略地⁵翻译成下面的代码：

```
if (!courseA.equals(courseAPrime))
    fail();
```

换句话说：`assertEquals` 中的两个对象，通过调用接收者的 `equals` 方法来进行比较。接收者是第一个参数，是被 JUnit 所引用的期望值。该例中，接收者是 `courseA`。

您还没有在 `Course` 中定义方法 `equals`。因此，Java 沿着继承链，向上寻找 `equals` 方法。`Course` 的父类只有一个：`Object`。您可以在类 `Object` 中发现方法 `equals` 的缺省实现：

```
package java.lang;
public class Object {
    // ...
    public boolean equals(Object obj) {
        return (this == obj);
    }
    // ...
}
```

⁵ 实际代码要更复杂一些。

Object 中的 equals 比较引用——比较接收者和参数对象的内存位置。



如果子类没有提供 equals 实现，那么使用缺省的比较内存位置的 equals 实现。

如果两个引用指向内存中的同一个对象，那么这两个引用就内存而言是相等的。

您应该在 Course 对象中重载 equals 方法。如果接收者对象在语义上和参数对象相等，那么重载的 equals 方法返回 true。对于 Course，如果接收者的科目以及课程编号与参数 Course 对象的相同，那么 equals 返回 true。

以较小的步伐前进，一次只前进一小步。现在，在 Course 中编写方法 equals，使测试得以通过：

324

```
@Override
public boolean equals(Object object) {
    return true;
}
```

方法 equals 必须满足 assertEquals 所预期的参数形式。方法 assertEquals 要求所调用的 equals 方法必须接受一个 Object 对象作为参数。如果你让 equals 接收一个 Course 对象作为参数，assertEquals 方法就无法找到你的 equals 方法。

```
@Override
public boolean equals(Course course) { // 这行不通
    return true;
}
```

方法 assertEquals 用该方法替换 Object 的 equals 实现。

在测试中增加一个断言，该断言将失败（因为 equals 总是返回 true）：

```
public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);

    Course courseB = new Course("ARTH", "330");
    assertFalse(courseA.equals(courseB));
}
```

一旦看见测试失败，请修改 equals 方法：

```
@Override
public boolean equals(Object object) {
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}
```

首先，必须将参数 `object` 强制类型转换成 `Course` 引用，这样您就可以引用它的实例变量。局部变量的命名可以帮助区分接收者（`this`）和参数（`that`）。在 `return` 语句中比较 `this` 和 `that` 的科目以及课程编号。如果都（`&&`）相等，那么方法 `equals` 返回 `true`。

325

相等性的定义

Java API 文档关于 `Object` 的 `equals` 方法的部分，提供了一个列表，详细的定义了两个对象之间的相等关系：

自反性：	<code>x.equals(x)</code>
对称性：	<code>x.equals(y)</code> 当且仅当 <code>y.equals(x)</code>
传递性：	如果 <code>x.equals(y)</code> 并且 <code>y.equals(z)</code> ，那么 <code>x.equals(z)</code>
一致性：	给定一致的状态， <code>x.equals(y)</code> 返回一致的结果
可以与 <code>null</code> 比较：	<code>!x.equals(null)</code>

对于非 `null` 的 `x` 和 `y`，上面几个准则都必须成立。在单元测试中，您可以通过这些准则，来验证相等性是否成立⁶。

```
public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);

    Course courseB = new Course("ARTH", "330");
    assertFalse(courseA.equals(courseB));

    // reflexivity
    assertEquals(courseA, courseA);

    // transitivity
    Course courseAPrime2 = new Course("NURS", "201");
    assertEquals(courseAPrime, courseAPrime2);
    assertEquals(courseA, courseAPrime2);

    // symmetry
    assertEquals(courseAPrime, courseA);

    // consistency
    assertEquals(courseA, courseAPrime);

    // comparison to null
    assertFalse(courseA.equals(null));
}
```

326

⁶ 可以从 <http://sourceforge.net/projects/junit-addons> 找到提供等式自动化测试的 JUnit 扩展。

运行测试，除了最后一个断言，其它测试都可以通过。您会看到 `NullPointerException` 异常：参数 `that` 为 `null`，方法 `equals` 试图访问 `that` 的成员变量（`department` 和 `object`）。您可以增加一个在参数为 `null` 的情况下返回 `false` 的防卫子句：

```
@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}
```

可以在一个测试中只验证相等性的一个方面（自反性，传递性，对称性，一致性，和 `null` 比较）。关于 TDD，存在不同的看法：一些程序员认为每个测试应该只包含一个断言⁷。我认为每个测试的目标是验证单一的功能点。如果一个测试验证一个功能点，可能需要多条断言/后置条件。该例中，多条断言验证了一个完整的功能点——相等性。

苹果和橙子

由于方法 `equals` 的参数类型为 `Object`，所以您可以将任何类型的参数传递给方法 `equals`（代码依然可以编译通过）。方法 `equals` 应该能够处理这种情况：

```
// apples & oranges
assertFalse(courseA.equals("CMSC-120"));
```

尽管参数中的字符串和 `courseA` 的科目、课程编号相匹配，但是 `Course` 毕竟不是 `String`。您期望比较的结果是 `false`，但是当 Java 执行方法 `equals` 时，却抛出异常 `ClassCastException`。下面黑体的代码行是抛出异常的地方：

```
@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}
```

代码试图将 `String` 类型的参数强制类型转换成 `Course`。这种无效的类型转换导致了异常 `ClassCastException`。

⁷ [Astels2004].

增加一个防卫子句：如果有人向方法 `equals` 中扔进一个“橙子”，立刻返回 `false`。该防卫子句确保参数的类型与接收者的类型相匹配。

```
@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    if (this.getClass() != object.getClass())
        return false;
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}
```

您在第8课学习了方法 `getClass`，该方法返回类常量。抛出异常 `ClassCastException` 的例子中，接收者的类常量为 `Course.class`，参数的类常量为 `String.class`。类常量是唯一的——所有类对象共享一个实例 `Course.class`。因为类常量的唯一性，所以可以使用操作符 `!=`（不等于）来比较类常量，而不必一定使用方法 `equals`。

有些程序员倾向于使用操作符 `instanceof`。如果某个对象是目标类的实例，或者目标类的子类的实例，那么操作符 `instanceof` 返回 `true`。下面用 `instanceof` 改写方法 `equals`。

```
@Override
public boolean equals(Object object) {
    if (object == null)
        return false;
    if (!(object instanceof Course))
        return false;
    Course that = (Course)object;
    return
        this.department.equals(that.department) &&
        this.number.equals(that.number);
}
```

推荐使用方法 `getClass` 而不是 `instanceof`，来比较两个类的类型。只有两个对象属于同一个类，`getClass` 才返回 `true`。如果您需要在继承层次上不考虑层次位置的比较对象，那就要用到 `instanceof`。请参考第12课，获取更多关于 `instanceof` 的信息。

JUnit 和相等性

有时候，我希望相等性测试能够更直接一些。一般的方式是：

```
assertEquals(courseA, courseB);
```

但是，我更喜欢这样的相等性测试：

```
assertTrue(courseA.equals(courseB));
```

第二种方式直观地表明：方法 `equals` 才是您实际要测试的方法。

您可能需要比较两个引用是否相同。换句话说，这两个引用是否指向内存中同一个对象？相应的比较代码如下：

```
assertTrue(courseA == courseB);
```

或者这样：

```
assertSame(courseA, courseB);
```

因为第二种方式提供了更好的错误消息，所以我推荐第二种形式的断言。不过，和以前一样，基于构建相等性测试的目的，您可能会选择明确的“=”比较。

集合与相等性

学生信息系统要求创建包含所有 `Course` 对象的课程目录。所以，您必须能够把课程对象存入集合，然后验证集合中包含该对象。

```
package sis.studentinfo;

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

public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);
    ...

    // containment
    List<Course> list = new ArrayList<Course>();
    list.add(courseA);
    assertTrue(list.contains(courseAPrime));
}
```

方法 `contains` 遍历 `ArrayList` 包含的所有对象，并且调用方法 `equals` 和参数进行比较。这个测试无需修改就能通过。事实上，该测试是一个验证 `ArrayList` 功能的语言测试，它示范了 `ArrayList` 使用 `equals` 方法来测试其内容，内容测试不属于相等性契约的一部分。请在验证结束之后，删除该测试。

将 `Course` 作为 `HashMap` 对象的关键字，如何？

```
public void testEquality() {
```

```

Course courseA = new Course("NURS", "201");
Course courseAPrime = new Course("NURS", "201");

// ...

Map<Course, String> map = new HashMap<Course, String>();
map.put(courseA, "");
assertTrue(map.containsKey(courseAPrime));
}

```

上面的测试演示了 Map 的 `containsKey` 方法的用法。如果和 Map 中的某个关键字相匹配，那么返回 `true`。尽管 Map 和 List 一样，都是集合，但是该测试将失败。在理解测试失败的原因之前，您必须理解类 `HashMap` 是如何实现的。

哈希表

`ArrayList` 封装了一个数组，数组在内存中是一块连续的空间。为了查询某个元素，必须遍历整个数组。类 `java.util.HashMap` 基于本节开始时提到的哈希表结构，它在内存中也是一块连续的空间。

图 9.2 展示了包含十个单位空间的哈希表。

向哈希表中插入元素，首先需要计算哈希值。简单的哈希值就是一个整数，而且理想情况下是唯一的。哈希值的定义是基于类的相等性定义：如果两个对象相同，那么它们的哈希值必须相同。如果两个对象不相同，那么它们的哈希值在理想情况下不相同（但是，并不必须）。

330

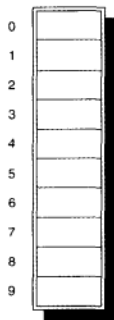


图 9.2 十个单位空间的哈希表

一旦确定了哈希值，您可以通过下面的算法来决定插入元素的位置：

```
hash code % table size = slot number
```

(%是求余运算符, 返回除法的余数) 例如, 如果将某个 Course 对象插入到大小为 10 的哈希表, 而且该 Course 对象的哈希值为 65, 那么该对象被插入到第五个内存单元:

$$65 \% 10 = 5$$

和数组一样, Java 使用下面的公式来计算某个内存单元的起始地址:

$$\text{offset} + (\text{slot size} * \text{slot number})$$

图 9.3 展示了插入到哈希表的 Course 对象。

哈希值是整型的, 并且通过发送消息 hashCode 到对象, 可以获取哈希值。类 java.lang.Object 提供了方法 hashCode 的缺省实现, 该方法返回一个基于对象内存地址的唯一的数字。

调用对象的 hashCode 方法, 计算对象的内存单元, 然后通过偏移量计算公式, 从哈希表中获取相应的对象。

331

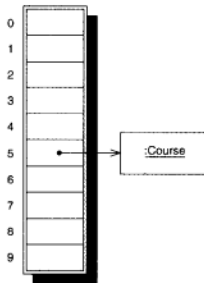


图 9.3 在第五个单元存放 Course 对象的哈希表

方法 hashCode 是如何实现的呢? 该方法必须返回一个整型值, 两个相同的对象必须返回同一个哈希值。最简单而又最差的方案是: 返回一个常量, 例如数字 1。

冲突

对象的哈希值应该尽可能唯一。如果两个不同的对象返回相同的哈希值, 那么这两个不同的对象在哈希表中对应同一个内存单元, 这种情况就是冲突。冲突意味着需要额外的逻辑和时间去维护一个冲突对象列表。有多个解决冲突的方案, 其中最简单的方案是为每个内存单元维护一个冲突对象列表, 图 9.4 形象的刻画了该方案。

因为存在冲突的可能性, 所以当访问对象的时候, 必须验证从某个内存单元取出的对象, 以确保取出的对象是所期望的对象。如果不是, 必须遍历冲突对象列表, 找到匹配的对象。

332

如果所有对象都返回同一个哈希值，例如 1，那么所有对象之间互相冲突，所有对象都对应哈希表的同一个内存单元。结果导致：所有插入和删除操作，都必须遍历冲突对象列表。该列表包含了所有插入的对象。在这种极端的情况下，最好使用 `ArrayList`。

如果所有对象都产生不同的哈希值，那么就可以得到最理想的哈希表。性能是最好的：所有插入和获取的操作，都可以在常量时间内完成。不需要遍历冲突对象列表。

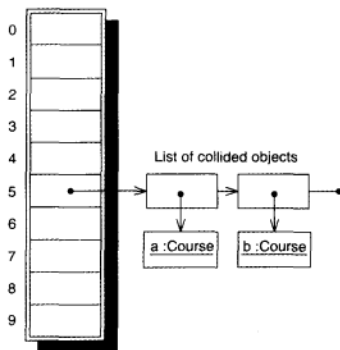


图 9.4 哈希表冲突

一个理想的哈希算法

Java 在类 `String` 中提供了优秀的 `hashCode` 实现，该实现比 `HashMap` 的关键字类型更常用。数字包装类 (`Integer`, `Long`, `Byte`, `Char`, `Short`) 通常也使用 `HashMap` 关键字：它们的哈希值就是它们所包装的数字。

通常，用来唯一标示对象的成员变量是 `String` 类型或者数字包装类类型。通过返回唯一标示对象的关键字，可以获得一个理想的哈希值。如果使用一个以上的成员变量作为关键字，那么需要提供哈希值的生成算法。

把下面的代码添加到类 `Course`，通过对 `department` 和 `number` 的哈希值进行算术组合，最终生成 `Course` 的哈希值。哈希值运算通常使用质数，这样，在对哈希表长度进行求余运算时可以得到较好的分布。

333

```
@Override
public int hashCode() {
    final int hashMultiplier = 41;
    int result = 7;
    result = result * hashMultiplier + department.hashCode();
    result = result * hashMultiplier + number.hashCode();
}
```

```
    return result;
}
```

一旦您实现了 hashCode 方法⁸，测试就将顺利通过。针对哈希值创建一个新的测试方法，强制类实现完整的 hashCode 方法。

```
public void testHashCode() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");

    Map<Course, String> map = new HashMap<Course, String>();
    map.put(courseA, "");
    assertTrue(map.containsKey(courseAPrime));

    assertEquals(courseA.hashCode(), courseAPrime.hashCode());
    // consistency
    assertEquals(courseA.hashCode(), courseA.hashCode());
}
```

测试证明：两个语义上相同的 Course 对象返回同一个哈希值。该测试也证明了一致性：每次调用 hashCode，只要不改变 Course 对象的关键字，返回值都是相同的。

还需要测试 Course 对象能够被正确的插入和删除。最终的、针对相等性和哈希值的测试如下：

```
public void testEquality() {
    Course courseA = new Course("NURS", "201");
    Course courseAPrime = new Course("NURS", "201");
    assertEquals(courseA, courseAPrime);

    Course courseB = new Course("ARTH", "330");
    assertFalse(courseA.equals(courseB));

    // reflexivity
    assertEquals(courseA, courseA);

    // transitivity
    Course courseAPrime2 = new Course("NURS", "201");
    assertEquals(courseAPrime, courseAPrime2);
    assertEquals(courseA, courseAPrime2);

    // symmetry
    assertEquals(courseAPrime, courseA);

    // consistency
    assertEquals(courseA, courseAPrime);

    // comparison to null
    assertFalse(courseA.equals(null));

    // apples & oranges
    assertFalse(courseA.equals("CMSC-120"));
}
```

⁸ 从 <http://mindprod.com/jgloss/hashcode.html> 的例子中得到该算法。

```

    }

    public void testHashCode() {
        Course courseA = new Course("NURS", "201");
        Course courseAPrime = new Course("NURS", "201");

        assertEquals(courseA.hashCode(), courseAPrime.hashCode());
        // consistency
        assertEquals(courseA.hashCode(), courseA.hashCode());
    }

```

这些测试包含了不少代码！再一次强调，请考虑使用 JUnit 扩展（参考脚注 6）来自动化测试相等性和哈希值。如果您决定不使用 JUnit 扩展，请重构代码，消除测试代码中的冗余。

hashCode 最后一个要点



如果您为某个类编写了 equals 方法，那么应该同时编写 hashCode 方法。

如果没有提供 hashCode 方法，编译器不会报错，而且也可能不会遇到任何问题。但是，将对象插入到基于冲突的哈希表（类 java.util.Set 也使用同样的哈希表实现）可能导致不可预料的后果。解决产生的问题，可能花费大量的时间。所以一定要养成这样的习惯：如果编写了方法 equals，同时提供相应的 hashCode 方法。

如果存在性能问题，请编写针对哈希值的单元测试。一种技术是：事先向 HashSet 中插入大量的元素，然后测试能否在合理的时间内完成一次插入操作。如果不能，说明哈希表可能管理着大量的冲突。

335

```

public void testHashCodePerformance() {
    final int count = 10000;
    long start = System.currentTimeMillis();
    Map<Course, String> map = new HashMap<Course, String>();
    for (int i = 0; i < count; i++) {
        Course course = new Course("C" + i, "" + i);
        map.put(course, "");
    }
    long stop = System.currentTimeMillis();
    long elapsed = stop - start;
    final long arbitraryThreshold = 200;
    assertTrue("elapsed time = " + elapsed,
        elapsed < arbitraryThreshold);
}

```

有很多性能测试的方法，但是上面的测试提供了一种简单的、适当的机制。类 java.lang.System 提供了静态方法 currentTimeMillis，该方法返回当前时间毫秒级的时间戳。开始，用局部变量（start）存储当前时间戳。然后，在 for 循环中执行需要进行性能测试的代码。循环结束后，再次捕获时间戳（stop）。用开始时间减去结束时间，可以得到执行代码

所花费的时间。断言确保循环能够在合理的时间范围内 (`arbitraryThreshold`) 结束。

方法 `assertTrue` 的第一个参数是字符串, 如果断言失败, JUnit 将该字符串作为错误信息的一部分打印输出。该例中, 错误信息将显示循环的实际执行时间。

您可以把性能测试放到一个单独的测试套件中。定期运行性能测试 (每天, 或者每次重要的修改之后)。但是, 性能测试需要花费时间, 您可能不希望性能测试给功能测试带来持续的、负面的影响。

如果您需要更加全面的单元测试, 请自己构建旨在消除冗余的测试框架, 或者使用工具 JUnitPerf⁹。JUnit 自身也提供了一些帮助您进行重复测试的简单工具。

为了证明性能测试的重要性, 请修改 `Course` 的 `hashCode` 实现, 返回常量数字:

```
@Override
public int hashCode() {
    return 1;
}
```

336

即使在只插入 10 000 个元素到 `HashMap` 的情况下, 测试也会失败。将方法 `hashCode` 改回到开始的状况, 测试通过。

测试 `hashCode` 的另一种技术是: 确保生成的哈希值具备良好的正态分布。另一种测试 `hashCode` 的技术是, 保证哈希值具有较大的方差。方差是用来衡量数值分布情况的, 对于所有数值偏离其平均数的差, 所有这些差的平方的平均值就是方差 (译注: 简言之, 方差越大, 这组数据就越离散, 数据的波动也就越大; 方差越小, 这组数据就越聚合, 数据的波动也就越小)。

两种测试技术的共同问题是: 什么是可接受的标准。作为程序员, 您可以选择一个值作为上限, 如果没有造成性能问题, 就以该值作为测量的标准。

测试的最小集合是: 你至少编写了足够充分的测试来证明 `hashCode` 的功能是正确的。

更多关于 HashMap

像列表一样, 您可以迭代访问哈希表的每一个元素。可以只迭代访问所有的关键字, 或者只迭代访问所有的值, 或者同时迭代访问关键字和值。

您在第 6 课创建了类 `ReportCard`, `ReportCard` 存储成绩到相应成绩卡片信息的映射。下面是 `ReportCard` 的源代码:

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;
```

⁹ 参考 <http://www.clarkware.com/software/JUnitPerf.html>。


```

public class ReportCard {
    static final String A_MESSAGE = "Excellent";
    static final String B_MESSAGE = "Very good";
    static final String C_MESSAGE = "Hmmm...";
    static final String D_MESSAGE = "You're not trying";
    static final String F_MESSAGE = "Loser";

    private Map<Student.Grade, String> messages = null;

    public String getMessage(Student.Grade grade) {
        return getMessages().get(grade);
    }

    private Map<Student.Grade, String> getMessages() {
        if (messages == null)
            loadMessages();
        return messages;
    }

    private void loadMessages() {
        messages =
            new EnumMap<Student.Grade, String>(Student.Grade.class);
        messages.put(Student.Grade.A, A_MESSAGE);
        messages.put(Student.Grade.B, B_MESSAGE);
        messages.put(Student.Grade.C, C_MESSAGE);
        messages.put(Student.Grade.D, D_MESSAGE);
        messages.put(Student.Grade.F, F_MESSAGE);
    }
}

```

337

接下来的三个针对 `ReportCard` 的测试展示了三种迭代访问哈希表的方式(不是必须的测试,更像是语言测试)。三个测试展示了不相同、但是有效的测试方法。

```

package sis.report;

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

public class ReportCardTest extends TestCase {
    private ReportCard card;

    protected void setUp() {
        card = new ReportCard();
    }

    public void testMessage() {
        // remove declaration of card since it is declared in setUp
        ...
    }

    public void testKeys() {
        Set<Student.Grade> expectedGrades = new HashSet<Student.Grade>();
        expectedGrades.add(Student.Grade.A);
        expectedGrades.add(Student.Grade.B);
        expectedGrades.add(Student.Grade.C);
        expectedGrades.add(Student.Grade.D);
    }
}

```

```

        expectedGrades.add(Student.Grade.F);

        Set<Student.Grade> grades = new HashSet<Student.Grade>();
        for (Student.Grade grade: card.getMessages().keySet())
            grades.add(grade);
        assertEquals(expectedGrades, grades);
    }
}

```

testKeys 是第一个测试，该测试提供了一种保证您可以迭代访问集合中所有期望值的技术。首先创建一个 set，把期望的对象（本例中是成绩对象）添加到该 set 中。然后创建第二个 Set，并把迭代访问的每一个元素添加到第二个 Set 中。最后，比较两个 Set，确保两个 Set 是相同的。

338

Set 是一种无序的集合，集合中的每个元素都是唯一的。如果试图把一个重复的元素插入到 Set 中，Set 会拒绝此类操作。在 Java 中，接口 java.util.Set 定义了 Set 的行为。类 java.util.HashSet 实现了 Set 接口。类 HashSet 使用方法 equals 比较两个对象是否相同。

基于对哈希表工作方式的理解，很显然不可能保证哈希表中的关键字是有序的，但是每个关键字都是唯一的。哈希表中的任意两个元素不可能拥有相同的关键字。发送消息 keySet 到 HashMap 对象，会返回 Set 类型的关键字集合。

在方法 testKeys 中，和迭代访问列表一样，可以使用 for-each 循环迭代访问 Set。

和关键字相反，哈希表中的值允许重复。例如，针对成绩从 D 到 F 的学生，都可以打印相同的信息（“get help”）。所以，HashMap 返回的值集合不能是 Set 类型，而是 java.util.Collection 类型。

Collection 是一种接口，HashSet 和 ArrayList 都实现了该接口。接口 Collection 定义了集合的基本功能，包括向集合中插入对象、获取集合的迭代器。接口 Collection 没有定义集合的有序性，没有提供类似接口 List、基于索引访问元素的方法。Java.util.Set 和 java.util.List 接口都从接口 Collection 扩展而来。

```

public void testValues() {
    List<String> expectedMessages = new ArrayList<String>();
    expectedMessages.add(ReportCard.A_MESSAGE);
    expectedMessages.add(ReportCard.B_MESSAGE);
    expectedMessages.add(ReportCard.C_MESSAGE);
    expectedMessages.add(ReportCard.D_MESSAGE);
    expectedMessages.add(ReportCard.F_MESSAGE);

    Collection<String> messages = card.getMessages().values();
    for (String message: messages)
        assertTrue(expectedMessages.contains(message));
    assertEquals(expectedMessages.size(), messages.size());
}

```

测试 testValues 使用了略有不同的技术。创建一个存储预期值的 Set，然后把所有可能的值添加到该 Set。发送消息 values 到 HashMap 对象，获取所有的值。

339

迭代访问所有的值，验证存储预期值的 `Set` 中存在与其一一对应的值。方法 `contains` 调用 `equals` 来判断集合中是否含有某个对象。最后，您必须测试第二个 `Set` 的尺寸，从而保证第二个 `Set` 没有包含多余的元素。

第三个测试 `testEntries`，同时迭代访问关键字和对应的值。发送消息 `entrySet` 到 `HashMap`，返回包含成对的“关键字—值”的 `Set`。使用 `for-each` 循环迭代访问该 `Set`。每次执行循环体，得到一个存储“关键字—值”的 `Map.Entry` 引用。调用方法 `getKey` 和 `getValue`，可以分别从 `Map.Entry` 中获取关键字和值。`Map.Entry` 和 `HashMap` 对象具有相同的关键字类型（本例中是 `Student.Grade`），以及相同的值类型（`String`）。

```
public void testEntries() {
    Set<Entry> entries = new HashSet<Entry>();

    for (Map.Entry<Student.Grade,String> entry:
        card.getMessages().entrySet())
        entries.add(
            new Entry(entry.getKey(), entry.getValue()));

    Set<Entry> expectedEntries = new HashSet<Entry>();
    expectedEntries.add(
        new Entry(Student.Grade.A, ReportCard.A_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.B, ReportCard.B_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.C, ReportCard.C_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.D, ReportCard.D_MESSAGE));
    expectedEntries.add(
        new Entry(Student.Grade.F, ReportCard.F_MESSAGE));

    assertEquals(expectedEntries, entries);
}
```

有必要测试通过 `for-each` 循环获取的每一个 `Entry`。就像前面的测试，创建第二个存储期望值的 `Set`，比较存储期望值的 `Set` 和通过迭代获取的 `Set`。您需要将“关键字—值”对儿存入预期 `Set`。但是，`Map.Entry` 是接口，没有存储“关键字—值”对儿的具体类。

所以，您需要创建用以存储“关键字—值”对儿的 `Entry` 类。在迭代的过程中，实例化 `Entry` 对象，并将 `Entry` 对象添加到集合中。同时，用 `Entry` 对象填充预期 `Set`。然后，比较两个 `Set`，确保它们包含相同的 `Entry` 对象。

```
class Entry {
    private Student.Grade grade;
    private String message;
    Entry(Student.Grade grade, String message) {
        this.grade = grade;
        this.message = message;
    }

    @Override
    public boolean equals(Object object) {
        if (object.getClass() != this.getClass())
```

```

        return false;
    }
    Entry that = (Entry) object;
    return
        this.grade == that.grade &&
        this.message.equals(that.message);
    }

    @Override
    public int hashCode() {
        final int hashMultiplier = 41;
        int result = 7;
        result = result * hashMultiplier + grade.hashCode();
        result = result * hashMultiplier + message.hashCode();
        return result;
    }
}

```

类 `Entry` 只用来测试，所以不需要为该类编写测试。类 `Entry` 提供了 `equals` 方法，以及随之而来的 `hashCode` 方法。由于类 `Entry` 是一个快速完成并且不需要较高质量的测试类，所以我可以选择忽略 `hashCode` 方法，但是成功的测试需要该方法。在没有 `hashCode` 的情况下进行测试，确信您理解为什么没有 `hashCode` 的情况下，测试会失败。

其它哈希表和 Set 实现

包 `java.util` 包含了接口 `Map` 和 `Set` 的其它几个实现。下面对这些实现进行简要介绍。请参考 Java API 文档关于 `java.util` 的部分，获取如何使用它们的细节。

341

EnumSet

您在第 6 课学习了 `EnumMap`，类 `EnumSet` 的工作方式与之有所不同。`EnumSet` 被定义成抽象类。创建一个 `EnumSet` 的子类，然后可以调用几乎一打的 `EnumSet` 类方法。我在表 9.1 中总结了这些工厂方法。

利用 `EnumSet`，可以简化 `ReportCardTest` 的方法 `testKeys`：

```

public void testKeys() {
    Set<Student.Grade> expectedGrades =
        EnumSet.allOf(Student.Grade.class);
    Set<Student.Grade> grades =
        EnumSet.noneOf(Student.Grade.class);
    for (Student.Grade grade: card.getMessages().keySet())
        grades.add(grade);
    assertEquals(expectedGrades, grades);
}

```

TreeSet 和 TreeMap

TreeSet 的元素按照自然顺序（您使用 Comparable 定义的顺序）或者通过您使用 Comparator 对象定义的顺序进行排序。TreeMap 按照类似的方式对所有的关键字进行排序。例如，学生成绩卡片的信息可以按照成绩进行排序。使用 TreeSet 或者 TreeMap，每次插入和存取都可以立即完成。但是，如果使用 HashMap 或者 HashSet，执行时间相对集合尺寸会有对数的增长。

表 9.1 EnumSet 的工厂方法

方法	描述
allOf	用指定枚举类型的所有元素创建 EnumSet。
complementOf	创建一个 EnumSet，并作为其它 EnumSet 的一部分。例如，新 EnumSet 包含了源 EnumSet 所没有的枚举实例。
copyOf(+1 variant)	利用已有集合的元素创建 EnumSet。
noneOf	根据指定类型，创建一个空 EnumSet。
of(+4 variants)	使用五个不同初值创建 EnumSet
range	

342

LinkedHashSet 和 LinkedHashMap

LinkedHashSet 维护了一个链表，元素的顺序就是元素被插入的顺序。性能和 HashSet 不相上下：add、contains，以及 remove 操作都在常量时间内完成，但是相对 HashSet 略有增加。迭代访问 LinkedHashSet 比普通 HashSet 要快。LinkedHashMap 和 LinkedHashSet 类似。

IdentityHashMap

IdentityHashMap 使用等号（==）来比较关键字，而不用相等性（equals）。如果需要用关键字来表示唯一的实例，可以使用这种 Map 实现。因为 IdentityHashMap 没有使用 equals，所以不符合 Map 的定义，不是一种通用的 Map 实现。

toString

前面，您了解通过 toString 方法，可以从 StringBuilder 对象中获取字符串消息。

toString 是 Java 设计者在 java.lang.Object 中定义的为数不多的方法之一，它的主要目的

是返回对象的可打印形式 (String)。您可以将这种可打印形式作为应用程序呈现给最终用户的一部分，但是最好不要这样做。方法 toString 对程序员非常有用，有助于调试程序或者解释 JUnit 消息。

您希望类 Course 的对象在 JUnit 错误消息中打印一条有用的字符串。一种方法是从 Course 对象中提取相应的字符串，然后和其它字符串连接成一个更大的字符串。另一种方法是使用 toString。

```
public void testToString() {
    Course course = new Course("ENGL", "301");
    assertEquals("ENGL 301", course.toString());
}
```

343

因为 toString 继承自 Object，所以此处调用 toString 的缺省实现，缺省实现导致测试失败：

```
expected: <ENGL 301> but was: <studentinfo.Course@53742e9>
```

下面的实现，可以使测试得以通过：

```
@Override
public String toString() {
    return department + " " + number;
}
```

很多 IDE 通过发送 toString 消息，在观察窗口中显示对象。JUnit 在 assertEquals 失败的情况下，使用 toString 显示相应的消息。如果编写代码：

```
assertEquals(course, new Course("SPAN", "420"));
```

那么，JUnit 会显示 ComparisonFailure 信息：

```
expected: <ENGL 301> but was: <SPAN 420>
```

通过加号(+)操作符可以将对象连接到某个字符串的后面。Java 自动调用对象的 toString 方法，将对象转换成 String 对象。例如：

```
assertEquals("Course: ENGL 301", "Course: " + course);
```

toString 的缺省实现几乎没有用处。大多数情况下，您需要提供自己的 toString 实现。如果只是出于调试或者理解代码的目的，就没有必要为 toString 编写测试。

事实上，toString 定义是不稳定的，它依赖于程序员在某一时期期望什么样的字符串信息。创建不必要的 toString 测试，会给生产类的修改带来困难。

编写测试的主要目的之一是：方便您有把握地修改代码。如果缺乏测试，那么修改代码时，您的信心就会减小。编写测试方便了代码的修改。拥有的测试越多，修改代码所花的时间就越

少。

不要把维护测试的困难，当作逃避编写测试的借口。但是，也不要编写不必要的测试。同时，编写的测试应该尽可能稳定。找到平衡需要经验和良好的判断。

当然，最后还是由您自己决定。我的建议是：开始时，编写尽可能多的，甚至是非必要的测试。当非必要测试造成了问题，而且无法解决时，再删除它们。

字符串和相等性

类 `String` 针对重载做了优化。在多数应用程序中，创建的 `String` 对象的数量超过其它任何类型对象的数量。

当前的 Java 版本提供了一组用来切换不同特征配置的命令行开关。在命令行输入命令：

```
java -agentlib:hprof=help
```

可以得到详细的信息¹⁰。

运行堆评测观察 JUnit 运行的当前测试套件中的大概 60 个测试。Java 虚拟机在堆中创建了大约 19 250 个对象。除了这些对象，还有超过 7 100 个 `String` 对象。所有对象中，出现频率为第二位的类型实例化了大概 2 000 个对象，该类型的对象相对于 `String` 也少很多。

前面的数字证明类 `String` 在 Java 中扮演了一个关键的角色。在一个典型应用程序中，`String` 对象占被创建对象总数的三分之一到二分之一。所以，类 `String` 的性能特别关键。而且，在创建特别多的 `String` 对象的同时，保证不占用过多的内存也特别关键。

为了最小化内存的消耗，类 `String` 使用了一个字符池。主要思想是：如果两个 `String` 对象包含了相同的字符，那么这些字符共享同样的内存空间（字符池）。字符池实现了一种叫做享元（Flyweight）¹¹的设计模式。享元模式基于共享，设计目标是有效地处理大量的细粒度的对象。

为了演示：

```
String a = "we have the technology";
String b = "we have the technology";
assertTrue(a.equals(b));
assertTrue(a == b);
```

字符串 `a` 和 `b` 在语义上是相同的（`equals`）：两个字符串包含相同的字符序列。同时，字符串引用在内存上也是相同的：它们指向内存中的同一个 `String` 对象。

作为 Java 编程的初学者，出于优化的目的，很可能用“==”来比较字符串对象。

¹⁰ 不能保证将来的 Java 版本也支持该功能。

¹¹ [Gamma1995]。

```
if (name == "David Thomas")
```

这是错误的。尽管两个字符串包含相同的字符序列，但是可能存储在不同的内存位置，这种情况之所以发生的原因是：Java 不对 `StringBuilder` 和连接操作¹²进行字符串优化。



使用 `equals` 比较字符串，而不是 `==`。

```
String c = "we have";
c += " the technology";
assertTrue(a.equals(c));
assertFalse(a == c);
```

测试通过。Java 虚拟机将 `c` 和 `a` 存储在不同的内存位置。

在某些受约束的情况下，您可以使用 `=="` 来比较字符串对象。这样对性能会有所提升。但是，使用 `=="` 比较字符串的原因应该只是为了优化，而且必须对此加以注释。使用 `=="` 比较字符串对象可能导致难以预料、而且难以解决的错误。

练习

1. 创建一个字符串对象，内容是练习 1 的前两句。创建类 `WordCount`，该类遍历字符串对象，统计每个单词出现的次数。单词列表中不包含标点符号，标点符号作为单词的一部分或者把标点符号视为独立的单词。使用 `Map` 存储单词出现的频率（词频）。类 `WordCount` 返回一个字符串集合，每个元素包括单词和对应的词频。统计词频时不考虑大小写。也就是说，拼写相同但是大小写不同的两个单词被认为是同一个单词。提示：结合正则表达式 `"\w+"` 和 `String` 的 `split` 方法，可以从字符串中提取单词。

346

2. 创建只包含一个 `String` 类型成员变量的类 `Name`。创建用于字符串比较的 `equals` 方法，不要创建 `hashCode` 方法。构建测试，验证类 `Name` 符合相等性定义。
3. 创建一个 `Set<Name>` 对象，该对象包含了多个 `Name` 对象，其中必须包含 `new Name("Foo")`。调用 `contains` 方法，查询集合中是否包含 `Name("Foo")` 实例，验证返回结果的为 `false`。首先创建：

```
Name foo = new Name("Foo");
```

集合中的确包含 `Foo`。

4. 修改 `Name` 的测试（参考练习 3），在集合只包含 `new Name("Foo")` 的情况下，使测试得以通过。

347

¹² 编译时，使用 `+` 进行字符串连接的代码被解释成对应的利用 `StringBuffer` 类的代码。

这一课，您将学习 Java 对数学的支持。Java 被设计成多用途语言。除了标准、通用的算术支持，Java 也支持一些高级数学概念，例如无穷大。Java 通过语言特性和类库，提供对这些高级数学概念的支持。

Java 支持固定精度和任意精度的浮点运算，以及整数运算。Java 的数字表达形式以及相关功能，符合 IEEE 标准。您可以选择为速度优化的运算法则，也选择严格符合公共标准的运算法则。

本课的内容包括：

- BigDecimal
- 其它整型和操作
- 数字类型转换
- 表达式运算顺序
- NaN (Not a Number)
- 无穷大
- 数字溢出
- 位操作
- java.lang.Math
- 静态导入

BigDecimal

类 `java.math.BigDecimal` 允许任意精度（基于十进制）的浮点运算。`BigDecimal` 的工作方式就像您在小学学习的初等数学一样。类 `BigDecimal` 提供了丰富的运算方法集合，以及多种舍入

方法。BigDecimal 对象可以表示一个任意精度的十进制数字，这意味着您可以自己决定有效数字的位数。BigDecimal 是不可更改的——您不能改变 BigDecimal 对象中存储的数字。

BigDecimal 应用最频繁的场所是金融类应用程序。金融类应用程序经常要求把金额精确到分，甚至更高的精度。

Java 也实现了浮点数，比如：基本类型 float 和 double。但是，二进制¹不可能精确地表示某个十进制数字（例如 0.1），所以 float 和 double 不能按照指定的大小、准确地表示数字。BigDecimal 可以精确地表示任何数字。

使用 BigDecimal 有两个主要的缺点。第一，硬件可以优化浮点运算，但是 BigDecimal 的十进制运算通过软件实现。如果用 BigDecimal 进行非常大量的数学运算，您可能面临性能问题。

第二，没有支持 BigDecimal 的数学运算符。必须通过方法调用来完成诸如加法、乘法的运算。这样会导致代码冗长，难以编写和阅读。

使用 BigDecimal



学生信息系统必须维护每个学生的收费和存款的帐目信息。第一个测试展示 Account 可以跟踪收支余额。收支余额基于实际收费和存款。

```
package sis.studentinfo;

import java.math.BigDecimal;
import junit.framework.*;

public class AccountTest extends TestCase {
    public void testTransactions() {
        Account account = new Account();
        account.credit(new BigDecimal("0.10"));
        account.credit(new BigDecimal("11.00"));
        assertEquals(new BigDecimal("11.10"), account.getBalance());
    }
}
```

构建 BigDecimal 的最好方式是：传入字符串到 BigDecimal 的构造函数，该字符串包含了需要用 BigDecimal 表示的数值。您也可以传入一个 double，但是这种构造方式不能保证 BigDecimal 所表示的数值完全和您所期望的一致，原因在于 Java 浮点数表示方式的本质²。

您也可以用它其它 BigDecimal 对象来构造新的 BigDecimal 对象。您可以直接将一个返回 BigDecimal 的表达式作为构造函数的参数。

Account 的实现：

¹ 请参考 <http://www.alphaworks.ibm.com/awnsf/FAQs/bigdecimal>。

² 参考 Java 术语表 <http://mindprod.com/jgloss/floatpoint.html>，此处有关于浮点数表示方式的讨论。

```

package sis.studentinfo;

import java.math.BigDecimal;

public class Account {
    private BigDecimal balance = new BigDecimal("0.00");

    public void credit(BigDecimal amount) {
        balance = balance.add(amount);
    }

    public BigDecimal getBalance() {
        return balance;
    }
}

```

BigDecimal 对象是不可改变的。发送消息 `add` 到某个 **BigDecimal** 对象，不会改变对象的值。方法 `add` 取出参数的值，与 **BigDecimal** 对象的值相加，然后创建并返回一个新的、表示总和的 **BigDecimal** 对象。前面的 `Account` 实现代码中，发送 `add` 消息到 `balance`，返回的结果是一个新的 **BigDecimal** 对象。为了记录总和，您必须将这个新的 **BigDecimal** 赋值给 `balance`。

BigDecimal 提供了表示数学运算的、丰富的方法集合，包括 `abs`、`add`、`divide`、`max`、`min`、`multiply`、`negate` 以及 `subtract`。此外，还提供了用来管理精度以及将 **BigDecimal** 转换成其他类型的方法。

351

精度

在 `testTransactions` 中，断言期望结果是精确到小数点后面两位的 **BigDecimal**，“11.10”。而如果您期望结果是“11.1”形式的 `float` 或者 `double` 型量，那么增加一个断言：

```
assertEquals(new BigDecimal("11.1"), account.getBalance());
```

使测试失败。


小数点后面的位数代表了数字的精度。对 **BigDecimal** 进行数学运算，结果依然是 **BigDecimal**，而且结果的精度是接收方 **BigDecimal** 和参数 **BigDecimal** 中的精度较大者。例如：

```
assertEquals(new BigDecimal("5.300"),
    new BigDecimal("5.000").add(new BigDecimal("0.3")));
```

精度为 1 的数字与精度为 3 的数字相加，结果的精度为 3。

除法和舍入

对数字进行除法，商的小数点后面的位数通常比除数和被除数的都要多。缺省情况下，`BigDecimal` 限定商的小数点后面的位数是除数和被除数两者之间的较大者。但是，您可以显式的定义结果的精度。

 限定精度，意味着 Java 需要对除法的结果进行舍入。Java 提供了八种不同的舍入模式。也许您最熟悉的舍入模式是 `BigDecimal.ROUND_HALF_UP`。这种模式的规则是：如果大于等于 0.5，那么进一位；否则，向下舍入。例如，如果 5.935 保留两位精度，那么舍入后的结果是 5.94。如果 5.934 保留两位精度，那么舍入后的结果是 5.93。

对于某个帐户，必须能够计算平均交易金额。对应的测试如下：

```
public void testTransactionAverage() {
    Account account = new Account();
    account.credit(new BigDecimal("0.10"));
    account.credit(new BigDecimal("11.00"));
    account.credit(new BigDecimal("2.99"));
    assertEquals(new BigDecimal("4.70"), account.transactionAverage());
}
```

修改后的 `Account` 类：

```
package sis.studentinfo;

import java.math.BigDecimal;

public class Account {
    private BigDecimal balance = new BigDecimal("0.00");
    private int transactionCount = 0;

    public void credit(BigDecimal amount) {
        balance = balance.add(amount);
        transactionCount++;
    }

    public BigDecimal getBalance() {
        return balance;
    }

    public BigDecimal transactionAverage() {
        return balance.divide(
            new BigDecimal(transactionCount), BigDecimal.ROUND_HALF_UP);
    }
}
```

请参考 Java API 文档，获取其它七种舍入模式的细节。

更多关于基本数字类型

到目前为止，您已经学习了基本数字类型 `int`、`double` 和 `float`，以及基本运算符、`int` 的自增/自减运算符、复合运算符。

其它整数类型

前面，您使用 `int` 来表示整数。除了 `int` 以外，还有其它可以用来表示整数的数字类型，每种类型的字节数不同。表 10.1 列出了所有的整数类型，以及它们的尺寸，支持的数字范围。

`char` 也是数字类型。请参考第 3 课，获取有关 `char` 的更多信息。

通常，用十进制表示数字。Java 也支持用十六进制或者八进制。

在十六进制数字前面加上前缀 `0x`。

```
assertEquals(12, 0xC);
```

353

表 10.1 整数类型

类型	尺寸	范围	例子
byte	8 bits	-128 到 127	0 24
char	16 bits	0 到 65535	17 'A' '%' \u0042 \062
short	16 bits	-32768 到 32767	1440 0x1AFF
int	32 bits	-2,147,483,648 到 2,147,483,647	-1440525 0x41F0 083
long	64 bits	-9223372036854775808 到 9223372036854775807	-90633969363693 77L 42

在八进制数字前面加上前缀 `0`：

```
assertEquals(10, 012);
```

整数在缺省情况下是 `int` 类型。在数字后面加上后缀 `L`，可以将其强制转换成 `long` 类型。可以用小写的 `l`，但是最好使用大写的 `L`，因为小写的 `l` 很容易和数字 `1` 混淆。

整数运算

让我们重新审视类 `Performance`（在第 7 课）中计算平均值的代码。

```
public double average() {
    double total = 0.0;
    for (int score: tests)
        total += score;
    return total / tests.length;
}
```

假设局部变量 `total` 不是 `double` 类型，而是 `int` 类型。因为每门测试的成绩是 `int` 类型，所以这种假设是合理的。将 `int` 与 `int` 相加，结果是另一个 `int` 类型的数字。

354

```
public double average() {
    int total = 0;
    for (int score: tests)
        total += score;
    return total / tests.length;
}
```

表面上看起来，修改没有任何问题，但是实际上会导致 `PerformanceTest` 中的几个测试失败。问题在于：用 `int` 除以另一个 `int`，商依然是 `int` 类型的数字。整数除法的结果是整数，任何余数都会被抛弃。接下来的两个断言演示了正确的整数除法：

```
assertEquals(2, 13 / 5);
assertEquals(0, 2 / 4);
```

如果您期望获得整数除法的余数，请使用求模运算符 (`%`)。

```
assertEquals(0, 40 % 8);
assertEquals(3, 13 % 5);
```

数字类型转换

Java 支持不同取值范围的多种数字类型。`float` 类型用 32 位比特表示数字，`double` 类型使用 64 位比特表示数字。整数基本类型 `char`、`byte`、`short`、`int`、`long`——每种类型的取值范围都不相同。

您总是可以把一个取值范围较小的基本类型赋值给另外一个取值范围较大的基本类型。例如，您可以把一个 `float` 引用赋值给另外一个 `double` 引用：

```
float x = 3.1415f;
double y = x;
```

在赋值的背后，Java 隐舍地进行了一次从 `float` 到 `double` 的类型转换。

尝试另一种途径——把取值范围较大的基本类型赋值给取值范围较小的基本类型——这样

会导致信息的丢失。如果存在一个 `double` 类型的数字，而且该数字超出 `float` 类型的最大表示范围，把该数字赋值给某个 `float` 类型，将会导致信息的截断。对于这种情况，Java 承认信息可能会丢失，并且强制必须进行类型转换，这样您可以明确地认识到可能存在着问题。

◀ 355

如果您试图编译下面的代码：

```
double x = 3.1415d;
float y = x;
```

您将看到编译错误信息：

```
possible loss of precision
found   : double
required: float
float y = x;
      ^
```

必须把 `double` 强制类型转换成 `float`：

```
double x = 3.1415d;
float y = (float)x;
```

为了避免整数除法，您可以将整数基本类型强制类型转换成浮点型。对于 Performance 的求平均值问题，一个可行的方案是：在做除法之前，把被除数强制类型转换成 `double`：

```
public double average() {
    int total = 0;
    for (int score: tests)
        total += score;
    return (double)total / tests.length;
}
```

另外要注意：如果表达式中既有整型也有 `float` 类型的数字，那么 Java 把表达式的结果转换成相应的 `float` 类型：

```
assertEquals(600.0f, 20.0f * 30, 0.05);
assertEquals(0.5, 15.0 / 30, 0.05);
assertEquals(0.5, 15 / 30.0, 0.05);
```

运算优先级

了解复杂表达式中的运算优先级是非常重要的。基本规则如下：

- 括号的优先级最高，内括号的优先级高于外括号的优先级
- 一些运算符的优先级高于其它运算符的优先级；例如，乘法的优先级高于加法的优先级
- 否则，表达式按照从左到右的顺序计算

◀ 356

特别地，因为优先级规则不容易记忆，所以您应该在复杂表达式中应该使用括号。事实上，某些优先级规则是不直观的。括号有利于消除对表达式优先级的错误理解。但是，不要过分使用括号——多数程序员比较熟悉基本的优先级规则。



使用括号，使理解复杂表达式变得简单。

下面有几个例子，通过这些例子可以看出优先级规则是如何影响一个表达式的结果：

```
assertEquals(7, 3 * 4 - 5); // left to right
assertEquals(-11, 4 - 5 * 3); // multiplication before subtraction
assertEquals(-3, 3 * (4 - 5)); // parentheses evaluate first
```

如果有疑问，请编写测试！

NaN (Not a Number)



如果没有成绩，那么 `Performance` 的平均成绩为零。您还没有编写针对这种情况的测试。

在 `PerformanceTest` 中增加一个简单的测试。

```
public void testAverageForNoScores() {
    Performance performance = new Performance();
    assertEquals(0.0, performance.average());
}
```

运行测试，得到异常 `NullPointerException`。通过将 `Performance` 中的整型数组初始化为空数组，可以解决这个问题：

```
private int[] tests = {};
```

运行测试，得到另外一个异常：

```
junit.framework.AssertionFailedError: expected:<0.0> but was:<NaN>
```

`NaN` 是一个常量，`NaN` 定义在 `java.lang.Float` 和 `java.lang.Double` 中，`NaN` 意味着“Not a Number”（不是一个数字）。

如果没有成绩，`tests.length()` 返回 0，意味着下面这行代码中除数为 0：

```
return total / tests.length;
```

在整型的情况下，Java 抛出异常 `ArithmeticException`，指出除数为 0 的情况。对于浮点数，您将得到 `NaN`，`NaN` 是一个合法的浮点数。

在 `average` 一开始的地方对成绩进行检查，可以解决上面的问题：

```
public double average() {
    if (tests.length == 0)
        return 0.0;
    int total = 0;
    for (int score: tests)
        total += score;
    return (double)total / tests.length;
}
```

```
}
```

NaN 存在一些有趣的特性。任何关于 NaN 的布尔表达式总是返回 false，就像下面的语言测试所展现的：

```
assertFalse(Double.NaN > 0.0);
assertFalse(Double.NaN < 1.0);
assertFalse(Double.NaN == 1.0);
```

您也许希望方法 `average` 返回 NaN。但是，因为不能将 NaN 与任何浮点数进行比较，如何编写测试呢？为了解决这个问题，Java 在 `Float` 和 `Double` 中，都提供了静态方法 `isNaN`：

```
public void testAverageForNoScores() {
    Performance performance = new Performance();
    assertTrue(Double.isNaN(performance.average()));
}
```

无穷大

类 `java.lang.Float` 提供了两个表示无穷大的常量：`Float.NEGATIVE_INFINITY` 和 `Float.POSITIVE_INFINITY`。类 `java.lang.Double` 也提供了类似的常量。

尽管整数除以 0 会导致错误，但是 `double` 和 `float` 除以 0 在数学上产生合理的无穷大。下面的断言展示了无穷大常量的用法：

```
final float tolerance = 0.5f;
final float x = 1f;

assertEquals(
    Float.POSITIVE_INFINITY, Float.POSITIVE_INFINITY * 100, tolerance);
assertEquals(Float.NEGATIVE_INFINITY,
    Float.POSITIVE_INFINITY * -1, tolerance);

assertEquals(Float.POSITIVE_INFINITY, x / 0f, tolerance);
assertEquals(Float.NEGATIVE_INFINITY, x / -0f, tolerance);
assertTrue(Float.isNaN(x % 0f));

assertEquals(0f, x / Float.POSITIVE_INFINITY, tolerance);
assertEquals(-0f, x / Float.NEGATIVE_INFINITY, tolerance);
assertEquals(x, x % Float.POSITIVE_INFINITY, tolerance);

assertTrue(Float.isNaN(0f / 0f));
assertTrue(Float.isNaN(0f % 0f));

assertEquals(
    Float.POSITIVE_INFINITY, Float.POSITIVE_INFINITY / x, tolerance);
assertEquals(
    Float.NEGATIVE_INFINITY, Float.NEGATIVE_INFINITY / x, tolerance);
```

358

```

assertTrue(Float.isNaN(Float.POSITIVE_INFINITY % x));

assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY % Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY / Float.NEGATIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.POSITIVE_INFINITY % Float.NEGATIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY / Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY % Float.POSITIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY / Float.NEGATIVE_INFINITY));
assertTrue(
    Float.isNaN(Float.NEGATIVE_INFINITY % Float.NEGATIVE_INFINITY));

```

数字溢出

处理整型变量的时候，必须非常小心，以避免数字溢出。数字溢出会导致错误的结果。

对于每一种数字类型，Java 都提供了表示其取值范围的常量。例如，常量 `Integer.MAX_VALUE` 和 `Integer.MIN_VALUE` 分别表示 `int` 的最大值 ($2^{31}-1$) 和最小值 (-2^{31})。

Java 内部自动将表达式的结果赋值给某个适当大小的基本类型。下面的例子测试中：将 1 与 `byte` 的最大值 (127) 相加，Java 将相加的结果存储到一个更大的基本类型。

359

```

byte b = Byte.MAX_VALUE;
assertEquals(Byte.MAX_VALUE + 1, b + 1);

```

但是，当表达式的结果超过 `byte` 的最大值，把该表达式的结果赋值给 `byte`，最后将产生预料之外的结果。下面的测试会通过，表明 1 与 `byte` 的最大值相加的结果，等于 `byte` 的最小值：

```

byte b = Byte.MAX_VALUE;
assertEquals(Byte.MAX_VALUE + 1, b + 1);
b += 1;
assertEquals(Byte.MIN_VALUE, b);

```

原因在于：Java 内部使用二进制存储数字，抛弃任何溢出的位。

浮点数溢出会导致无穷大：

```

assertTrue(Double.isInfinite(Double.MAX_VALUE * Double.MAX_VALUE));

```

浮点数可能向下溢出——存在无限趋近于 0 的值。Java 让这样的值等于 0。

位操作

Java 支持整数的位操作。或许您使用位操作进行数学运算，或许使用位操作提升性能（对于某些数学运算，使用位操作会加快速度），或许使用位操作进行编码，或许使用位操作处理压缩的标志集合。

二进制数字

在计算机内部，数字最终以位流的形式保存。使用二进制数 0 或者 1，表示每一位。下表列出了从 0 到 15 的二进制数字。

该表同时也列出了每个数字的十六进制表示形式。Java 不允许直接使用二进制，所以理解位操作的最简单的方法是：用十六进制表示法，每四个二进制位表示一个十六进制数。

Java 二进制表示法

Java 用 32 位表示 `int`。最高位为 0 表示正数，最高位为 1 表示负数。Java 用自然的二进制位表示正数，用补码表示负数。计算数字补码的算法是：正数的补码是其本身；忽略负数的负号，对应的正数求反码，再对反码加 1。

360

十进制	二进制	十六进制	十进制	二进制	十六进制
0	0	0x0	8	1000	0x8
1	1	0x1	9	1001	0x9
2	10	0x2	10	1010	0xA
3	11	0x3	11	1011	0xB
4	100	0x4	12	1100	0xC
5	101	0x5	13	1101	0xD
6	110	0x6	14	1110	0xE
7	111	0x7	15	1111	0xF

例如，数字 17 的补码为：

```
0000_0000_0000_0000_0000_0000_0001_00013
```

数字 -17 的补码为：

```
1111_1111_1111_1111_1111_1111_1110_1111
```

³ 加下划线，是为了提高可读性。

逻辑位操作

Java 提供了四种支持位运算的逻辑操作符：位与、位或、位求反、位异或。在真值表中指出所有可能的位组合，可以看出每种操作符的工作方式。

位与、位或、位异或都是二元操作符。位求反是一元操作符。

对于二元操作符，Java 按位比较两个操作数。如果对两个 32 位的整数进行按位与操作，那么需要 32 次位与操作。如果对某个整数进行求反操作，那么 Java 按位求整数每一位的相反数。

可以对两个 int 型或者取值范围更小的类型，进行逻辑位操作。但是，在实际进行位操作之前，Java 自动把取值范围比 int 小的类型——short、char、byte——转换成 int 类型。

位与操作符(&)也被称之为按位乘法。如果都是 1，那么按位乘法的结果为 1，否则结果总为 0。下面的测试驱动真值表描述了位与操作。

```
assertEquals(0, 0 & 0);
assertEquals(0, 0 & 1);
assertEquals(0, 1 & 0);
assertEquals(1, 1 & 1);
```

位或操作符(|)也被称之为按位加法。如果都是 0，那么结果为 0，否则结果总为 1。下面的测试驱动真值表描述了位或操作。

```
assertEquals(0, 0 | 0);
assertEquals(1, 0 | 1);
assertEquals(1, 1 | 0);
assertEquals(1, 1 | 1);
```

位异或操作符(^)也被称之为按位求异。如果都相同，那么返回 0；否则总为 1。下面的测试驱动真值表描述了位异或操作。

```
assertEquals(0, 0 ^ 0);
assertEquals(1, 0 ^ 1);
assertEquals(1, 1 ^ 0);
assertEquals(0, 1 ^ 1);
```

位求反操作符(~)处理整数的每一位，把 0 变成 1，1 变成 0。

```
int x = 0x7FFFFFFF; //0111_1111_1111_1111_1111_1111_1111_0001
assertEquals(0x8000000E, ~x); //1000_0000_0000_0000_0000_0000_0000_1110
```

Java 支持逻辑位操作的复合赋值运算。所以，x &= 1 等同于 x = x & 1。

使用位与、位或、位求反

可以用一个整型数来压缩表示若干个标志位(布尔值)。如果每个标志位都用一个 boolean 变量表示，那么就不是很有效。整型数的每一位可以表示一个标志位。因此，用一个 byte 可以表示八个标志位。例如，二进制数 00000001 可以表示第一个标志为真，其它标志都为假。二

进制 00000101 可以表示第一个和第三个标志为真，其它标志为假。

为了设置每一个标志位，首先需要定义每一个二进制位的掩码。第一位的掩码为 00000001（十进制 1），第二位的掩码为 00000010（十进制 2），依此类推。利用位或设置标志位，利用位与获取标志位。



您需要为每个学生设置四个“是/否”标志：学生是否住校，是否免税，是否是少数民族，是否是麻烦制造者？您需要处理 200 000 个学生，然而内存是很珍贵的。

```
public void testFlags() {
    Student student = new Student("a");
    student.set(
        Student.Flag.ON_CAMPUS,
        Student.Flag.TAX_EXEMPT,
        Student.Flag.MINOR);
    assertTrue(student.isOn(Student.Flag.ON_CAMPUS));
    assertTrue(student.isOn(Student.Flag.TAX_EXEMPT));
    assertTrue(student.isOn(Student.Flag.MINOR));

    assertFalse(student.isOff(Student.Flag.ON_CAMPUS));
    assertTrue(student.isOff(Student.Flag.TROUBLEMAKER));

    student.unset(Student.Flag.ON_CAMPUS);
    assertTrue(student.isOff(Student.Flag.ON_CAMPUS));
    assertTrue(student.isOn(Student.Flag.TAX_EXEMPT));
    assertTrue(student.isOn(Student.Flag.MINOR));
}
```

最好提供清晰的查询方法，以获取每个标志位（isOnCampus、isTroublemaker，等等）。但是，如果标志位非常多，而且需求是动态变化的，那么上面的方法更可取。

Student 中相应的代码如下：

```
public enum Flag {
    ON_CAMPUS(1),
    TAX_EXEMPT(2),
    MINOR(4),
    TROUBLEMAKER(8);

    private int mask;

    Flag(int mask) {
        this.mask = mask;
    }
}

private int settings = 0x0;
...
public void set(Flag... flags) {
    for (Flag flag: flags)
        settings |= flag.mask;
}

public void unset(Flag... flags) {
    for (Flag flag: flags)
        settings &= ~flag.mask;
}
```

```

    }

    public boolean isOn(Flag flag) {
        return (settings & flag.mask) == flag.mask;
    }

    public boolean isOff(Flag flag) {
        return !isOn(flag);
    }
}

```

每个标志都是定义在 `Student` 中的枚举常量 `Flag` 中。实例化枚举类型 `Flag` 的时候，把一个表示掩码的整型数传入 `Flag` 的构造函数。把 `int` 类型的变量 `settings` 显示地初始化为 0，将标志存储在 `settings` 中。

方法 `set` 以若干 `Flag` 枚举类型的对象为参数，循环访问每一个 `Flag` 对象的掩码，并且利用位或操作给变量 `settings` 赋值。

方法 `unset` 遍历 `Flag` 枚举类型的对象，对变量 `settings` 的反码进行位与运算。

下面的例子，演示了位或操作如何正确地设置每一位的值：

```

Flag.MINOR                                0100

settings                                  0000

settings = settings | Flag.MINOR          0100

Flag.ON_CAMPUS                            0001

settings | Flag.ON_CAMPUS                  0101

```

方法 `isOn` 用 `flag` 的掩码，和编码 `settings` 进行位与操作：

```

settings                                  0000

Flag.MINOR                                0100

settings & Flag.MINOR                     0000

settings & Flag.MINOR == Flag.MINOR       false

settings = settings | Flag.MINOR          0100

settings & Flag.MINOR                     0100

settings & Flag.MINOR == Flag.MINOR       true

```

最后，方法 `unset` 中使用位取反操作：

```

settings                                  0101

~Flag.MINOR                               1011

settings & ~Flag.MINOR                    0001

```

使用位操作来存储多个标志位，是一种经典的技术。该技术在给定内存的条件下，最大程度地进行了信息的压缩。因为 Java 通过数组或其它包含 boolean 值的集合，提供了更清晰简单的表示方法，所以不推荐、也没有必要在 Java 开发中使用该技术。

使用异或

异或操作符具有可逆性：

```
int x = 5;           // 101
int y = 7;           // 111
int xPrime = x ^ y;   // 010
assertEquals(2, xPrime);
assertEquals(x, xPrime ^ y);
```

异或操作是奇偶校验的基础。数据传输中，可能会有若干数据位失真。奇偶校验对发送的数据进行异或运算，从而计算出校验和，并将校验和作为附加信息发送出去。接收方针对数据和校验和执行同样的算法，如果校验和不匹配，那么发送方需要重新发送数据。

奇偶校验是二进制的：对数据流可以进行奇校验或者偶校验。如果数据流中 1 的位数是偶数，那么校验位是偶数。如果数据流中 1 的个数是奇数，那么校验位是奇数。通过异或计算校验位。下面是一个简单的测试：

```
public void testParity() {
    assertEquals(0, xorAll(0, 1, 0, 1));
    assertEquals(1, xorAll(0, 1, 1, 1));
}

private int xorAll(int first, int... rest) {
    int parity = first;
    for (int num: rest)
        parity ^= num;
    return parity;
}
```

对偶数个 1 进行异或运算，结果总是偶校验（0）。对奇数个 1 进行异或运算，结果总是奇校验（1）。 ▶ 365

原因在于：异或相当于两个数字相加，然后对 2 进行求余运算。下面的真值表扩展清晰地做了说明：

$0 \wedge 0 = 0$	$(0 + 0) \% 2 = 0$
$0 \wedge 1 = 1$	$(0 + 1) \% 2 = 1$
$1 \wedge 0 = 1$	$(1 + 0) \% 2 = 1$
$1 \wedge 1 = 1$	$(1 + 1) \% 2 = 0$

对2求余,说明结果要么是1,要么是0。若干二进制数字相加,值为1的二进制位决定总和。所以,用总和对2求余,可以知道1的个数为奇数还是偶数。

进一步,计算任何整数的校验和。类 `ParityChecker` 计算 `byte` 数组的校验和。下面的测试说明:破坏数组中的某个 `byte`,将导致不同的校验和。

```
package sis.util;

import junit.framework.*;

public class ParityCheckerTest extends TestCase {
    public void testSingleByte() {
        ParityChecker checker = new ParityChecker();
        byte source1 = 10; // 1010
        byte source2 = 13; // 1101
        byte source3 = 2; // 0010

        byte[] data = new byte[] {source1, source2, source3};

        byte checksum = 5; // 0101

        assertEquals(checksum, checker.checksum(data));

        // corrupt the source2 element
        data[1] = 14; // 1110

        assertFalse(checksum == checker.checksum(data));
    }
}
```

`ParityChecker` 循环访问所有的字节,对每个字节进行异或运算,计算累计校验和。测试中,我用注释指明每个十进制数字(`source1`、`source2` 和 `source3`)的二进制形式。这样,方便您明白对每一列进行异或运算的结果是5。

```
package sis.util;

public class ParityChecker {
    public byte checksum(byte[] bytes) {
        byte checksum = bytes[0];
        for (int i = 1; i < bytes.length; i++)
            checksum ^= bytes[i];
        return checksum;
    }
}
```

简单的异或奇偶校验检查不能发现所有的错误。更复杂的方法是:除了在整个数据流的末尾附加一个校验字节之外,还为传输的每一个字节附加一个校验位。这样就提供了一种可以最小化传输错误的矩阵模式。

位移操作

Java 提供了三种可以左移或者右移的位移操作符。

使用左移操作符 (<<) 或者右移操作符 (>>), 向左或者向右移位, 并且管理符号位。

左移操作使所有二进制位都向左移动一位, 最左的数据位丢失, 最右的数据位用 0 填充。

例如, 左移 1011 的结果是 0110。下面是有注释的 Java 位移操作的例子:

```
// 101 = 5
assertEquals(10, 5 << 1); // 1010 = 10
assertEquals(20, 5 << 2); // 10100 = 20
assertEquals(40, 5 << 3); // 101000 = 40
assertEquals(20, 40 >> 1);
assertEquals(10, 40 >> 2);

// ... 1111 0110 = -10
assertEquals(-20, -10 << 1); // ... 1110 1100 = -20
assertEquals(-5, -10 >> 1); // ... 1111 1011 = -5
```

请注意: 左移一位相当于乘以 2。右移一位相当于除以 2。

无符号右移操作不考虑符号位。最右边的数据位丢失, 最左边数据位用 0 填充。

```
assertEquals(5, 10 >>> 1);
assertEquals(2147483643, -10 >>> 1);
// 1111_1111_1111_1111_1111_1111_0110 = -10
// 0111_1111_1111_1111_1111_1111_1011 = 2147483643
```

请注意: 无符号右移的结果总是正数。

◀ 367

在密码运算和图形操作中, 位移操作会经常用到。您也可以使用位移操作来代替某些数学运算, 例如除以 2 或者乘以 2。当然, 只有在您需要特别快的性能、并且性能测试证明数学运算导致了性能问题的情况下, 才使用位移操作。

BitSet

Java API 库提供了类 `java.util.BitSet`。该类封装了一个以二进制位作为元素的向量, 并且可以根据需要自动增加长度。`BitSet` 对象是易变的——每个二进制位都可以被设置为 1 或者 0。可以在 `BitSet` 对象之间进行位与、位或、以及位异或操作, 而且也可以对某个 `BitSet` 对象进行位求反操作。`BitSet` 的优点不多, 其中一个优点是: `BitSet` 支持对超出 `int` 取值范围的数字进行位操作。

java.lang.Math

类 `java.lang.Math` 提供了针对数学方法的工具库, 同时提供了一些常量定义, 例如表示自然

对数的 `Math.E`、表示圆周率的 `Math.PI`。这些常量精确到小数点后面 15 位。

此外，Java API 类库提供了类 `java.lang.StrictMath`，该类更加严格地提供了函数公认的标准的结果。多数场合下，`Math` 的结果是可接受的，并且有更好的性能。

我在接下来的表中，总结了类 `Math` 提供的函数。请参考 Java API 文档获取更详细的信息。

函数名称	功能
<code>abs</code>	绝对值；所有基本数字类型都重载该函数
<code>max</code> , <code>min</code>	最大值，最小值
<code>acos</code> , <code>asin</code> , <code>atan</code>	反余弦，反正弦，反正切
<code>cos</code> , <code>sin</code> , <code>tan</code>	余弦，正弦，正切
<code>atan2</code>	将坐标 (x, y) 转换成极坐标
<code>ceil</code> , <code>floor</code> , <code>round</code>	无条件进位，无条件舍去，四舍五入后返回整数
<code>exp</code>	自然指数
<code>log</code>	自然对数
<code>pow</code>	次方
<code>sqrt</code>	平方根
<code>random</code>	0.0 到 1.0 的随机数
<code>rint</code>	以 <code>double</code> 为参数，返回最接近的 <code>int</code> 整数
<code>toDegrees</code> , <code>toRadians</code>	度与弧度之间的转换
<code>IEEERemainder</code>	返回余数

下面的例子程序：求直角三角形的斜边（直角所对应的边）。

```
// util.MathTest.java:
package util;

import junit.framework.*;

public class MathTest extends TestCase {
    static final double TOLERANCE = 0.05;

    public void testHypotenuse() {
        assertEquals(5.0, Math.hypotenuse(3.0, 4.0), TOLERANCE);
    }
}
```

```
// in util.Math:
package util;

import static java.lang.Math.*;

public class Math {
    public static double hypotenuse(double a, double b) {
        return sqrt(pow(a, 2.0) + pow(b, 2.0));
    }
}
```

用函数 `pow` 求边 `a` 和 `b` 的平方。然后计算平方的和，最后函数 `hypotenuse` 调用 `sqrt` 返回和的平方根。

深入了解类 `Math` 之后，您会发现使用第 4 课介绍的静态导入功能是有道理的。否则，如果代码中存在大量的数学方法调用，每个调用前面都加上包名，这样会使代码难以阅读。

◀ 369

就像 `java.lang.Math`，我也命名了自己的类 `Math`。尽管这样做不是好的实践（可能引起混淆），但是例子表明 `Java` 可以消除两者之间的歧义。

数字包装类

在第 7 课中介绍过，`Java` 为每一个基本类型提供了相应的包装类：`Integer`、`Double`、`Float`、`Byte`、`Boolean`，等等。包装类的主要作用是把基本类型转换成对象，这样可以将其存储在集合中。

数字包装类还有别的用途。前面的课程中讲过，每个数字包装类都提供了常量 `MIN_VALUE` 和 `MAX_VALUE`。而且，类 `Float` 和 `Double` 还提供了表示 `NaN`（不是一个数字）和无穷大的常量。

可打印形式

类 `Integer` 提供了分别将十六进制、八进制、二进制的 `int` 转换成可打印形式的静态工具方法。

```
assertEquals("101", Integer.toBinaryString(5));
assertEquals("32", Integer.toHexString(50));
assertEquals("21", Integer.toOctalString(17));
```

`Java` 提供了通用的，用以根据指定基数返回正确的字符串表示的方法。下面的断言展示：指定基数 3，返回 `int` 类型数字的字符串形式。

```
assertEquals("1022", Integer.toString(35, 3));
```

字符串转换为数字

在第8课，您学习了方法 `Integer.parseInt`，该方法以字符串为参数，返回相应的 `int` 值。通常，使用方法 `parseInt` 将用户界面输入的字符串转换成数字形式。输入字符串中的所有字符都必须是十进制数字，第一个字符除外（可能为负号）。如果输入字符串没有包含可解析的整型值，那么 `parseInt` 抛出异常 `NumberFormatException`。

`parseInt` 的另一种形式：把输入字符串的基数，作为第二个参数。方法 `valueOf` 的作用和 `parseInt` 一样，区别在于前者返回一个新的 `Integer` 对象，而不是 `int` 值。

每个数字包装类都提供了对应的解析方法。例如，使用 `Double.parseDouble` 可以把字符串转换成 `double` 值，或者使用 `Float.parseFloat` 把字符串转换成 `float` 值。

方法 `parseInt` 只能接受十进制输入。方法 `decode` 可以接受十六进制或者八进制输入，请看下面的断言：

```
assertEquals(253, Integer.decode("0xFD"));
assertEquals(253, Integer.decode("0XFD"));
assertEquals(253, Integer.decode("#FD"));
assertEquals(15, Integer.decode("017"));
assertEquals(10, Integer.decode("10"));
assertEquals(-253, Integer.decode("-0xFD"));
assertEquals(-253, Integer.decode("-0XFD"));
assertEquals(-253, Integer.decode("-#FD"));
assertEquals(-15, Integer.decode("-017"));
assertEquals(-10, Integer.decode("-10"));
```

随机数

类 `Math` 包含方法 `random`，该方法返回从 0.0 到 1.0 之间的 `double` 类型的伪随机数。通过方法 `random` 生成的数字不是一个真正的随机数——一组伪随机数序列，每个元素之间近似没有关联⁴。对于大多数应用程序，伪随机数已经够用了。

类 `java.util.Random` 提供了更加全面的、生成伪随机数的方案。类 `java.util.Random` 可以生成各种类型的伪随机序列，包括 `boolean`、`byte`、`int`、`long`、`float`、高斯，以及 `double`。伪随机序列生成器不是凭空生成随机序列，而是基于特定数学算法。

创建 `Random` 实例的时候，可以指定种子，也可以不指定。本质上，种子是某个随机序列的唯一标识符。如果用同一个种子创建两个 `Random` 对象，那么这两个对象有着相同的序列。

⁴ http://en.wikipedia.org/wiki/Pseudo-random_number_generator.

创建 `Random` 对象的时候如果没有指定种子，那么类 `Random` 采用系统时钟作为种子的基础⁵。

重复调用 `Random` 的 `nextBoolean` 方法，对翻转硬币进行模拟。`True` 表示正面朝上，`False` 表示反面朝上。下面的测试表明：相同的两个种子，可以产生两个相同的关于翻转硬币的伪随机序列。

◀ 371

```
public void testCoinFlips() {
    final long seed = 100L;
    final int total = 10;
    Random random1 = new Random(seed);
    List<Boolean> flips1 = new ArrayList<Boolean>();
    for (int i = 0; i < total; i++)
        flips1.add(random1.nextBoolean());

    Random random2 = new Random(seed);
    List<Boolean> flips2 = new ArrayList<Boolean>();
    for (int i = 0; i < total; i++)
        flips2.add(random2.nextBoolean());

    assertEquals(flips1, flips2);
}
```

方法 `nextInt`、`nextDouble`、`nextLong`、`nextFloat` 有着类似的工作方式。方法 `nextInt` 还有另一种形式：传入某个表示最大值的参数，总是返回 0 和最大值之间的某个数字。

测试随机代码

针对使用了随机序列的代码，可以用多种方法来编写对应的单元测试。一种方法是：提供一个 `Random` 的子类，代替代码中的 `Random`。子类提供某种已知的序列。这种技术被称之为模拟 `Random` 类。稍后我会深入介绍模拟技术。



第 8 课中，您编写了仅供测试的日志处理类。这也是模拟的一种形式。您需要给每个学生分配一个密码，这样他们可以在线登陆。密码必须是长度为 8 的字符串，每个字符必须在某个字符范围之内。

```
package sis.util;

import junit.framework.*;

public class PasswordGeneratorTest extends TestCase {
    public void testGeneratePassword() {
        PasswordGenerator generator = new PasswordGenerator();
        generator.setRandom(new MockRandom('A'));
        assertEquals("ABCDEFGH", generator.generatePassword());
        generator.setRandom(new MockRandom('C'));
        assertEquals("CDEFGHIJ", generator.generatePassword());
    }
}
```

◀ 372

⁵ 如果在同一个纳秒，创建两个 `Random` 对象，那么您将得到两个相同的随机序列。在早期 Java 中，`Random` 的构造函数使用毫秒级的当前系统时间，这样的粒度显著增加了两个 `Random` 对象拥有相同序列的可能性。

```

    }
}

```

测试方法 `testGeneratePassword` 将 `MockRandom` 的实例 `random` 作为参数，传入类 `PasswordGenerator` 的实例。类 `MockRandom` 继承自类 `Random`。参考 Java API 文档可知，创建类 `Random` 的子类的方法是：重载方法 `next(int bits)`，该方法返回基于某个序列的数字；其它所有的 `Random` 方法都以方法 `next(int bits)` 为基础。

`MockRandom` 的构造函数以 `int` 型变量作为参数，表示起始字符。然后，根据起始字符计算随机序列的初始值，并把初始值存入变量 `i`。初始值是一个和密码的最小有效字符相关的数字。方法 `next(int bits)` 简单地返回变量 `i`，并将 `i` 加 1。

```

package sis.util;

import junit.framework.*;

public class PasswordGeneratorTest extends TestCase {
    ...
    class MockRandom extends java.util.Random {
        private int i;
        MockRandom(char startCharValue) {
            i = startCharValue - PasswordGenerator.LOW_END_PASSWORD_CHAR;
        }
        protected int next(int bits) {
            return i++;
        }
    }
}

```

类 `MockRandom` 定义在 `PasswordGeneratorTest` 的内部，是 `PasswordGeneratorTest` 的嵌套类。您可以在 `PasswordGeneratorTest` 内部，直接实例化 `MockRandom` 对象，但是不能在其它任何类中实例化 `MockRandom` 对象。此外，还有另外几种嵌套类，每种嵌套类之间存在细微的差别。第 11 课中，您将深入了解嵌套类。

下面是类 `PasswordGenerator` 的实现。

```

package sis.util;

import java.util.*;

public class PasswordGenerator {
    private String password;
    private static final int PASSWORD_LENGTH = 8;
    private Random random = new Random();

    static final char LOW_END_PASSWORD_CHAR = 48;
    static final char HIGH_END_PASSWORD_CHAR = 122;

    void setRandom(Random random) {
        this.random = random;
    }

    public String generatePassword() {
        StringBuffer buffer = new StringBuffer(PASSWORD_LENGTH);
        for (int i = 0; i < PASSWORD_LENGTH; i++)
            buffer.append(getRandomChar());
    }
}

```

```

        return buffer.toString();
    }

    private char getRandomChar() {
        final char max = HIGH_END_PASSWORD_CHAR - LOW_END_PASSWORD_CHAR;
        return (char)(random.nextInt(max) + LOW_END_PASSWORD_CHAR);
    }

    public String getPassword() {
        return password;
    }
}

```

请注意：如果方法 `setRandom` 没有被调用，那么变量 `random` 将一直保持它的初始值——一个合法的 `java.util.Random` 实例。因为 `MockRandom` 符合 `java.util.Random` 的定义，所以可以在测试中用它来代替 `Random`。

方法 `testGeneratePassword` 的目标是：证明类 `PasswordGenerator` 能够生成并且返回一个随机的密码。该测试不需要重新证明类 `Random` 固有的功能，但是，该测试需要证明 `PasswordGenerator` 对象能够通过接口方法 `nextInt(int max)`，和 `Random` 正确的交互。该测试还需要证明类 `PasswordGenerator` 正确的使用了 `nextInt` 的返回值。该例中，`PasswordGenerator` 利用 `nextInt` 的返回值，构建 8 个字符长度的随机字符串序列。

Java 提供了另一个随机类 `java.util.SecureRandom`，可以生成基于标准的、强加密的伪随机数。

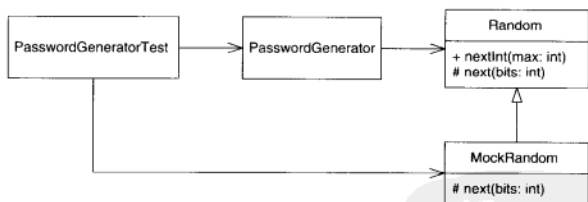


图 10.1 对类 `Random` 进行模拟

374

练习

- 编写一个展示 `BigDecimal` 不变性的测试。将第二个 `BigDecimal` 与第一个相加，验证第一个 `BigDecimal` 仍然保持原来的值。
- 用“10.00”创建一个 `BigDecimal`，用“1”创建第二个 `BigDecimal`，然后验证它们不相等。使用乘法，从第二个 `BigDecimal` 返回一个新的 `BigDecimal`，使得新 `BigDecimal` 和第一个相等。然后反过来，使用乘法，从第一个 `BigDecimal` 返回一个新的 `BigDecimal`，使

得新 `BigDecimal` 和第二个相等。

- 验证 `0.9` 和 `0.005*2.0` 不相等(都是 `float` 型)。它们各自的精度是多少? 如果都是 `double` 型呢?
- 为什么下面的代码无法编译通过? 用两种方法解决问题。

```
public class CompilerError {
    float x = 0.01;
}
```
- 编写一个简单的程序, 求 `0XDEAD` 的十进制的值。如何用八进制表示 `0XDEAD`?
- 尽您所能, 找出 `NaN` 和无穷大尽可能多的变化。
- 挑战: 找出包装类, 例如 `Float`, 与相应的基本类型在行为上有什么不同?
- 创建一个方法(当然, 要测试优先), 该方法以可变数量的 `int` 值作为参数, 然后只返回其中能被 3 整除的数。两次编写代码: 第一次, 只使用求模操作符(`%`)。第二次, 使用除法操作符(`/`), 必要时使用乘法操作符, 但是不要使用求模操作符。测试从 1 到 10 之间的整数序列。
- 修改“国际象棋程序”的代码, 在适当的地方使用类型转换。消除类 `Board` 中的 `toChar` 方法。
- 下面哪一行代码能够编译通过? 为什么?

```
float x = 1;
float x = 1.0;
float x = (int)1.0;
```
- `(int)1.9` 的结果是多少?
- `Math rint(1.9)` 的结果是多少?
- `Math rint` 是如何进行舍入的? `1.5` 等于 1 还是 0? `2.5` 呢?
- 下面每个表达式的最终结果是多少, 假设 `x` 等于 5, `y` 等于 10, 而且 `x` 和 `y` 都是 `ints` 类型。同时, 标记无法编译通过的代码行。最后 `x` 和 `y` 的值各是多少? (认为每个表达式都是不连续的——每个表达式的 `x` 和 `y` 的初始值都是 5、10。)

```
x * 5 + y++ * 7 / 4
++x * 5 * y++
x++ * 5 * ++y
++x + 5 * 7 + y++
++y++ % ++x++
x * 7 == 35 || y++ == 0 // super tricky
++x * ++y
x++ * y++
true && x * 7
x * 2 == y || ++y == 10 // super tricky
x * 2 == y || ++y == 10 // super tricky
```
- 仅使用操作符 `<<`, 将 17 转换成 34。
- 1 的十进制值是多少?

17. 展示>>和>>>的不同。对于正数，这两个操作符有区别吗？对于负数呢？
18. 创建一个函数，该函数调用 `Math.random` 生成 1 到 50 之间的随机整数。应该如何测试该方法？您可以完美的测试它吗？您怎样才能确定自己的代码可以工作？
19. 创建一个列表，该列表包含从 1 到 100 之间的数字。使用随机数生成器，随机地交换列表中的元素，一共交换 100 次。编写测试，进行合理的验证：列表的长度不变，而且每次交换过后，都有两个数字被互换。 ◀ 376
20. 演示：两个 `Random`，其中一个的种子为 1，另一个没有种子，它们的下一个 `double` 值不相同。编写测试，证明前面的结论的确为 `true`。
21. 挑战：不用临时变量，使用异或操作符交换两个数字。
22. 挑战：编程求出存储每种整数类型（`char`, `byte`, `short`, `int`, `long`）所需要的位数。提示：使用位移操作。注意，有符号类型的最高位用来表示符号。 ◀ 377



IO（输入/输出）

在本课中，你将学习 Java 的输入-输出（input-output, IO）设施（facility）。IO 指的是，任何向你的应用传入、或从中传出数据的途径。Java 中的 IO 是非常完整、非常复杂、并且相当一致的。你可以使用 Java 的 IO 功能，将报告写入文件，或者从终端读取用户的输入。

在本课中，你将学习以下内容：

- 流（stream）类的组织
- 字符流和字节流
- File 类
- 数据流
- System.in 和 System.out 的重定向
- 对象流
- 随机存取文件
- 嵌套类

组织

Java.io 包中含有许多类，来管理使用数据流、序列化和文件系统进行的输入和输出。理解包的结构以及类的命名策略，有助于你将在 Java 中处理 IO 的不适降至最低。

Java 的 IO 是建立在对流的使用基础之上的。流是数据的一个序列，你可以从中读取，也可以向其写入。流可能有源（例如终端），或者有目标（例如文件系统）。具有源的流是输入流，具有目标的流是输出流。

通过流载送数据格式的不同，Java 进一步对流进行了划分。Java 包括字符流和字节流。你可以使用字符流来处理 2 字节（16 位）的 Unicode 字符数据（译注：Java 使用 Unicode 的 UTF-16

编码)。类名中带有“Reader”和“Writer”的Java类是字符流。通常你使用字符流来处理人类可读的文本。

字节流处理8位的二进制数据。字节流的类名中含有Input或Output的字样。通常你可以使用字节流来处理非文本的数据，例如图像文件或者编译生成的字节码。

底层（low-level）的Java流支持诸如每次读/写一个字节等基本概念。如果只让你和底层的流打交道，那么你将不得不编写许多冗长乏味的重复性代码。Java提供了许多更高级别的流，通过提供聚合的（aggregate）、附加的功能，来简化你的工作。这些更高级别的流被称为包装（wrapper）流。

Wrapper流对象可能内含一个底层流对象的引用（reference），即所谓包装（wrap）了一个底层流对象。你首先和这个wrapper流打交道。然后，它再和所包装的、最终完成脏活儿的底层流进行交互。

让我们看看java.io包，它含有一组接口和类，这些接口和类定义了IO的基石。你会注意到，这个包相当密集——Java的IO设施完整而复杂。包中含有许多特定流的实现，包括实现过滤（filtered）、缓冲（buffered）或管道（piped）的流，以及对象流。

除了这些流的类以外，java.io包还提供了一组类来操作底层（underlying）的文件系统。

字符流

在第3课中，你创建了RosterReporter类来记录学生在一学期的签到报告。你将报告写入到一个字符串，然后使用System.out将它打印到终端上。下面是已有的RosterReporter类：

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;
class RosterReporter {
    static final String ROSTER_REPORT_HEADER =
        "Student" + NEWLINE +
        "-" + NEWLINE;
    static final String ROSTER_REPORT_FOOTER =
        NEWLINE + "# students = ";

    private Session session;

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

    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);
    }
}

```

现在，`RosterReporter` 类构建了一个字符串来表示整个报告。另一个使用了 `RosterReporter` 的类，将获取这个字符串并把它送到预期的目标——终端、文件、甚或 Internet。这样，报告中的每个字符被输出了两次，首先是写入到 `String`，然后再到最终的目标。对于更大规模的报告，报告的接收者在等待整个报告生成时，可能会碰到无法接受的延迟。

更优的方案是，将每个字符直接写入到一个代表最终目标的流中。这还意味着你不需要将整个报告保存在一个巨大的字符串缓冲中，避免产生某些潜在的内存空间问题。

◀ 381

你已经知道，要让学生信息系统保持灵活。首先，系统必须能够将报告写入到终端或者本地文件。为了满足这个要求，你先要更新 `RosterReporter` 类，将报告直接写入到字节流中。首先，你需要更新测试用例。

```

package sis.report;

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

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

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

        Writer writer = new StringWriter();
        new RosterReporter(session).writeReport(writer);
    }
}

```

```
String rosterReport = writer.toString();

assertEquals(
    RosterReporter.ROSTER_REPORT_HEADER +
    "A" + NEWLINE +
    "B" + NEWLINE +
    RosterReporter.ROSTER_REPORT_FOOTER + "2" +
    NEWLINE, rosterReport);
}
}
```

要使用 Java 的 IO 类，你必须引入（import）java.io 包。许多 IO 操作可能会发生异常。你需要声明该测试方法可能会抛出 IOException 异常。

你想要让 RosterReporter 将其报告写入到由客户提供的 Writer 对象中。java.io 包中的 Writer 类是字符流的抽象基类。为测试的目的，你可以创建一个 StringWriter（Writer 的子类）的对象。当你向 StringWriter 发送 toString 消息时（译注：发送消息是 OO 的一个术语，在此即为调用 StringWriter 的 toString 方法。之后的章节，大都将发送消息直译为调用方法），它返回包括所有写入字符的字符串。

注意其中对设计上的改进。不再是被动地向 RosterReporter 对象请求一份报告（getReport），而是主动地告诉它输出一份报告（writeReport）。

在 RosterReporter 中，和 RosterReporterTest 一样，你需要添加 import 语句，以及为执行 IO 操作的方法添加 throws 子句（clause）。（除了信任我之外，更好的方法是，在编写代码时不加 throws 子句，而让编译器告诉你该怎样做。）

```
package sis.report;

import java.util.*;
import sis.studentinfo.*;
import static sis.report.ReportConstant.NEWLINE;
import java.io.*;

class RosterReporter {
    static final String ROSTER_REPORT_HEADER =
        "Student" + NEWLINE +
        "- " + NEWLINE;
    static final String ROSTER_REPORT_FOOTER =
        NEWLINE + "## students = ";

    private Session session;
    private Writer writer;

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

    void writeReport(Writer writer) throws IOException {
        this.writer = writer;
        writeHeader();
        writeBody();
        writeFooter();
    }
}
```

```

private void writeHeader() throws IOException {
    writer.write(ROSTER_REPORT_HEADER);
}

private void writeBody() throws IOException {
    for (Student student: session.getAllStudents())
        writer.write(student.getName() + NEWLINE);
}

private void writeFooter() throws IOException {
    writer.write(
        ROSTER_REPORT_FOOTER + session.getAllStudents().size() +
        NEWLINE);
}
}

```

在 `writeHeader`、`writeBody` 和 `writeFooter` 中，你已经把对 `StringBuilder` 的 `append` 方法调用，替换为调用 `Writer` 的 `write` 方法。而且之前的代码，是将 `StringBuilder` 作为参数在方法之间传递。而现在你使用 `Writer` 的一个实例变量（instance variable）。通过避免将相同的变量在每个方法中蔓延，你可以消除某些重复。

等候你的测试通过后，进行下面的重构以充分利用 Java String 的格式化功能。

```

package sis.report;

import java.util.*;
import sis.studentinfo.*;
import java.io.*;

class RosterReporter {
    static final String ROSTER_REPORT_HEADER = "Student%-n%-n";
    static final String ROSTER_REPORT_FOOTER = "%n# students = %d%-n";

    private Session session;
    private Writer writer;

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

    void writeReport(Writer writer) throws IOException {
        this.writer = writer;
        writeHeader();
        writeBody();
        writeFooter();
    }

    private void writeHeader() throws IOException {
        writer.write(String.format(ROSTER_REPORT_HEADER));
    }

    private void writeBody() throws IOException {
        for (Student student: session.getAllStudents())
            writer.write(String.format(student.getName() + "%n"));
    }
}

```



```
private void writeFooter() throws IOException {
    writer.write(
        String.format(ROSTER_REPORT_FOOTER,
            session.getAllStudents().size()));
}
}
```

384

对测试用例进行必要的更改：

```
package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import java.io.*;

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

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

        Writer writer = new StringWriter();
        new RosterReporter(session).writeReport(writer);
        String rosterReport = writer.toString();

        assertEquals(
            String.format(RosterReporter.ROSTER_REPORT_HEADER +
                "A%n" +
                "B%n" +
                RosterReporter.ROSTER_REPORT_FOOTER, 2),
            rosterReport);
    }
}
```

写入文件

现在你需要更新 `RosterReporter`，使其能够接收文件名作为参数。测试需要确认报告被正确写入到操作系统的文件中了。

首先，你需要重构 `RosterReporterTest`：创建一个 `setUp` 方法，以及一个断言（assert）方法 `assertReportContents`。这样，你将为快速地添加新测试 `testFiledReport` 打好基础，它会使用这些通用方法，因而不会引入重复。

在 `assertReportContents` 中，你可以修改断言，以从学期对象中获得预期的学生数目。你需要将 `Session` 中的 `getNumberOfStudents` 从 `package` 改为 `public`。

385

经过重构的 `RosterReporterTest`：

```

package sis.report;

import junit.framework.TestCase;
import sis.studentinfo.*;
import java.io.*;

public class RosterReporterTest extends TestCase {
    private Session session;

    protected void setUp() {
        session =
            CourseSession.create(
                new Course("ENGL", "101"),
                DateUtil.createDate(2003, 1, 6));

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

    public void testRosterReport() throws IOException {
        Writer writer = new StringWriter();
        new RosterReporter(session).writeReport(writer);
        assertReportContents(writer.toString());
    }

    private void assertReportContents(String rosterReport) {
        assertEquals(
            String.format(RosterReporter.ROSTER_REPORT_HEADER +
                "A%n" +
                "B%n" +
                RosterReporter.ROSTER_REPORT_FOOTER,
                session.getNumberOfStudents()),
            rosterReport);
    }
}

```

新的测试，`testFiledReport`，调用了重载后的 `writeReport` 方法。这个 `writeReport` 方法的第二个版本，使用文件名而不是 `Writer` 作为其参数。

```

public void testFiledReport() throws IOException {
    final String filename = "testFiledReport.txt";
    new RosterReporter(session).writeReport(filename);

    StringBuffer buffer = new StringBuffer();
    String line;

    BufferedReader reader =
        new BufferedReader(new FileReader(filename));
    while ((line = reader.readLine()) != null)
        buffer.append(String.format(line + "%n"));
    reader.close();

    assertReportContents(buffer.toString());
}

```

在调用 `writeReport` 之后，该测试使用 `BufferedReader` 将 `report` 文件的内容读取到一个缓冲中。然后测试将缓冲的内容传给 `assertReportContents` 方法。

`BufferedReader` 是 `Reader` 的子类，它可以包装(wrap)其它的 `reader`。记住你需要使用 `Reader`

对象从字符流中进行读取。该测试使用 `FileReader` 作为参数，构造一个 `BufferedReader` 对象。`FileReader` 是一个基于字符的输入流，可以从文件中读取数据。你需要使用一个文件名来构造 `FileReader`。

你可以直接通过调用 `FileReader` 而不是 `BufferedReader` 来读取文件的内容。不过，`BufferedReader` 更有效率，因为它会将读取的字符缓冲起来。而且，`BufferedReader` 提供了 `readLine` 方法来帮助你简化代码。`readLine` 方法从所包装的流中读取一个逻辑的输入行，它使用系统 property “`line.separator`” 对行进行分隔。

`RosterReporter` 的新方法：

```
void writeReport(String filename) throws IOException {
    Writer bufferedWriter =
        new BufferedWriter(new FileWriter(filename));
    try {
        writeReport(bufferedWriter);
    }
    finally {
        bufferedWriter.close();
    }
}
```

`writeReport` 方法使用传递给它的文件名，创建一个 `FileWriter` 对象。它把 `FileWriter` 包装成一个 `BufferedWriter`。然后该方法将这个 `writer` 传给现有的 `writeReport` 方法（它使用 `PrintWriter` 作为参数）。

`finally` 代码块确保 `PrintWriter` 总被关闭，而不管 `writeReport` 是否会抛出异常。你必须牢记要关闭文件资源，否则你会遇到文件被锁住的问题。而且，被缓冲的信息在你关闭 `Writer` 之前不会被保存到文件中。你还可以使用 `Writer` 的 `flush` 方法，强制 `Writer` 将其内容写入到目标。

需要注意的是，你无需修改现有 `RosterReporter` 类中的任何一行代码，就可以为其添加功能。你只需要简单地添加一个新方法就好。如果你始终坚持让代码保持最优的设计，要达到这种理想的结果并不很困难。在你的代码中更多地使用抽象。

387

java.io.File

`File` 类不是一个基于流的类。除了和流一起使用之外，`File` 类提供给你一个访问底层文件系统中文件和目录结构的接口。它包括许多针对文件的实用方法，例如删除文件和创建临时文件的能力。

在编写和文件打交道的测试用例时，你希望确保有一个干净的环境来进行测试。你还希望确保在测试结束时，保持系统的原貌。对 `testFiledRepor` 来说，这意味着在测试中，首先你需要删除所有现有的报告文件，并在测试的最后一步删除所创建的报告文件。

```

public void testFileReport() throws IOException {
    final String filename = "testFileReport.txt";
    try {
        delete(filename);

        new RosterReporter(session).writeReport(filename);

        StringBuffer buffer = new StringBuffer();
        String line;

        BufferedReader reader =
            new BufferedReader(new FileReader(filename));
        while ((line = reader.readLine()) != null)
            buffer.append(String.format(line + "%n"));
        reader.close();

        assertReportContents(buffer.toString());
    }
    finally {
        delete(filename);
    }
}

private void delete(String filename) {
    File file = new File(filename);
    if (file.exists())
        assertTrue("unable to delete " + filename, file.delete());
}

```

delete 方法使用 File 类来完成它的目标。通过向 File 的构造函数传入一个文件名，它创建一个 File 对象。然后，它调用 exists 方法来判断文件系统中是否存在这个文件。最后，它调用 delete 方法，如果文件系统成功地删除了该文件，它返回 true，否则返回 false。文件系统无法删除文件，可能是由于这个文件是被锁住的、具有只读属性，或者其它原因。

◀ 388

我在表 11.1 中对 java.io.File 类中的功能进行了分类。

表 11.1 java.io.File 方法的分类

类 别	方 法
创建临时文件	createTempFile
创建空白文件	createNewFile
文件操作	delete, deleteOnExit, renameTo
查询文件或路径名	getAbsolutePath, getAbsolutePath, getCanonicalFile, getCanonicalPath, getName, getPath, toURI, toURL
级别	isFile, isDirectory
属性查询/操作	isHidden, lastModified, length, canRead, canWrite, setLastModified, setReadOnly
目录查询/操作	exists, list, listFiles, listRoots, mkdir, mkdirs, getParent, getParentFile

字节流与转换

Java 通过保存在 `System.in` 和 `System.out` 中的流对象，分别表示标准输入 (stdin) 和标准输出 (stdout)。`System.in` 指向一个 `InputStream`，而 `System.out` 指向一个 `PrintStream`。`PrintStream` 是一个专门的 `OutputStream`，来简化各种对象类型的写入。

`System.in` 和 `System.out` 都是字节流，而非字符流。回忆一下 Java 使用多字节字符集 Unicode，而绝大多数操作系统使用单字节字符集。

`java.io` 包提供了在字节流和字符流之间相互转换的方法。`InputStreamReader` 包装了 `InputStream`，并将读入的每个字节转换为适当的字符。转换过程默认使用系统平台的缺省编码方案（可被 Java 理解的）；你还可以提供一个解码 (decoding) 类来定义之间的映射。类似的方式，`OutputStreamWriter` 包装了 `OutputStream`，将字符转换为单字节。

`InputStreamReader` 和 `OutputStreamWriter` 的主要作用是，映射 stdin/stdout 和 Java 字符流。`Reader` 和 `Writer` 的子类，还可以让你以逐行的方式处理输入和输出。



不要直接调用 `System.in` 和 `System.out` 中的方法。

至于 `Reader` 和 `Writer` 抽象类，虽然它们无法快速地重定向到不同的介质（例如文件），但有助于使测试更容易。

学生用户界面



在本节中，你将编写一个非常简单的、基于终端的用户界面 (UI)。UI 允许学生信息系统的最终用户，创建学生对象。你需要向用户显示一个简单的菜单，来驱动新学生的添加，或者让用户终止（退出）应用。

有许多通过 TDD 开发用户界面的方法。下面在 `StudentUITest` 类中实现的测试，提供了一种方案。为清晰起见，我将整个代码列表分为了三部分。第一部分展示了核心的测试方法，`testCreateStudent`。

```
package sis.ui;

import junit.framework.*;
import java.io.*;
import java.util.*;
import sis.studentinfo.*;

public class StudentUITest extends TestCase {
    static private final String name = "Leo Xerces Schmoo";
```

```

public void testCreateStudent() throws IOException {
    StringBuffer expectedOutput = new StringBuffer();
    StringBuffer input = new StringBuffer();
    setup(expectedOutput, input);
    byte[] buffer = input.toString().getBytes();

    InputStream inputStream = new ByteArrayInputStream(buffer);
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(inputStream));

    OutputStream outputStream = new ByteArrayOutputStream();
    BufferedWriter writer =
        new BufferedWriter(new OutputStreamWriter(outputStream));

    StudentUI ui = new StudentUI(reader, writer);
    ui.run();

    assertEquals(
        expectedOutput.toString(),
        outputStream.toString());
    assertEquals(ui.getAddedStudents());
}

private String line(String input) {
    return String.format("%s\n", input);
}
...
}

```

390

该测试创建了两个 `StringBuffer` 对象。我们将使用其中一个来保存终端程序所显示的输出，另一个用来保存用户（在此情况下是测试用例）所键入的输入。`setup` 方法显式地把它们组装起来：

```

private void setup(StringBuffer expectedOutput, StringBuffer input) {
    expectedOutput.append(StudentUI.MENU);
    input.append(line(StudentUI.ADD_OPTION));
    expectedOutput.append(StudentUI.NAME_PROMPT);
    input.append(line(name));
    expectedOutput.append(line(StudentUI.ADDED_MESSAGE));
    expectedOutput.append(StudentUI.MENU);
    input.append(line(StudentUI.QUIT_OPTION));
}

```

可以将 `setup` 方法看作是系统必须遵守的一个剧本（script）。将对字符串缓冲的拼接交织在一起，来模拟系统输出和用户输入之间的交互。以下浅白的流程应当能够帮助你理解这个剧本。首先，用户界面向用户展示一个菜单：

```
expectedOutput.append(StudentUI.MENU);
```

用户通过首先按下添加学生的选项键来响应，然后回车，

```
input.append(line(StudentUI.ADD_OPTION));
```

`line` 方法是一个实用（utility）方法，用来向输入字符串追加一个行结束字符串。`setup` 方法的其余部分，描述了所预期的事件序列：系统呈现的文本（`expectedOutput.append()`），以及用户输入的响应（`input.append()`）。

391

`testCreateStudent` 中的代码，通过将输入 `StringBuffer` 中的字节包装成一个 `ByteArrayInputStream`，来创建一个输入流。然后它创建一个 `InputStreamReader`（包装了 `BufferedReader`），将输入的字节转换为字符。

类似的，测试用例创建了 `ButeArrayOutputStream`，并且包装了 `OutputStreamWriter` 来将字符转换为字节。`BufferedWriter` 包装了 `OutputStreamWriter`。

你可以在 `setup` 方法执行之后，将这两个流（`reader` 和 `writer`）传给 `StudentUI` 的构造函数。`UI` 被通知运行，并应该始终显示选项菜单，直至用户选择退出。

在 `UI` 完成处理之后，使用一个断言来验证应用产生了正确的输出。断言对 `ByteArrayOutputStream` 和 `expectedOutput StringBuffer` 的内容进行比较。

最后，测试用例调用 `assertStudents` 来确保在 `UI` 实例中创建和保存的学生列表，同预期的一致。

```
private void assertStudents(List<Student> students) {
    assertEquals(1, students.size());
    Student student = students.get(0);
    assertEquals(name, student.getName());
}
```

为了验证 `setup` 方法中示例的用户“脚本”，`assertStudents` 必须确保 `StudentUI` 对象只添加了一个学生，并且学生的数据（他的名字）和预期一致。

整个 `StudentUI` 的实现如下。要不断增量地构建这个测试为先（`test-first`）的类，你应该从一些较简单的单元测试集开始。首先，确保菜单被显示出来，并且用户可以立即退出这个应用。然后添加功能来支持添加学生。以用户界面流程的断言作为开始，然后添加断言来确保创建了一个学生对象。一个更完整的测试将会确保多个学生被创建。

```
package sis.ui;

import java.io.*;
import java.util.*;
import sis.studentinfo.*;

public class StudentUI {
    static final String MENU = "(A)dd or (Q)uit?";
    static final String ADD_OPTION = "A";
    static final String QUIT_OPTION = "Q";
    static final String NAME_PROMPT = "Name: ";
    static final String ADDED_MESSAGE = "Added";

    private BufferedReader reader;
    private BufferedWriter writer;
    private List<Student> students = new ArrayList<Student>();
    public StudentUI(BufferedReader reader, BufferedWriter writer) {
        this.reader = reader;
        this.writer = writer;
    }
    public void run() throws IOException {
        String line;
```

392

```

do {
    write(MENU);
    line = reader.readLine();
    if (line.equals(ADD_OPTION))
        addStudent();
    } while (!line.equals(QUIT_OPTION));
}

List<Student> getAddedStudents() {
    return students;
}

private void addStudent() throws IOException {
    write(NAME_PROMPT);
    String name = reader.readLine();

    students.add(new Student(name));
    writeln(ADDED_MESSAGE);
}

private void write(String line) throws IOException {
    writer.write(line, 0, line.length());
    writer.flush();
}

private void writeln (String line) throws IOException {
    write(line);
    writer.newLine();
    writer.flush();
}
}

```

测试应用

学生用户界面展现了客户可以执行并与之交互的实际应用。你需要提供一个 `main` 方法，这样他们可以启动这个应用。目前，要执行这个应用，你需要适当地包装 `System.in` 和 `System.out`，来构造一个 `StudentUI`，然后调用 `run` 方法。

通过将这些操作封装在 `StudentUI` 类本身中，你可以简化 `main` 方法。如果你只有一个用户界面类，这可能是/也可能不是合适的策略，现在还好。但是，如果你有许多相关的 UI 类，而每一个类都控制着用户界面的一部分，那么你最好在单独的类中构造这个终端的 `wrapper`。

对这个示例来说，我们让 `main` 方法尽可能的简单。你可以通过使用 `System` 中的 `setIn` 和 `setOut` 方法对终端的输入和输出进行重定向、而不是把输入和输出流包装在缓冲流中，来达成这一目标。你必须要把 `ByteArrayOutputStream` 包装为 `PrintStream`，以调用 `setOut`。

```

public void testCreateStudent() throws IOException {
    StringBuffer expectedOutput = new StringBuffer();
    StringBuffer input = new StringBuffer();
    setup(expectedOutput, input);
    byte[] buffer = input.toString().getBytes();
}

```

◀ 393


```

InputStream inputStream = new ByteArrayInputStream(buffer);
OutputStream outputStream = new ByteArrayOutputStream();

InputStream consoleIn = System.in;
PrintStream consoleOut = System.out;
System.setIn(inputStream);
System.setOut(new PrintStream(outputStream));
try {
    StudentUI ui = new StudentUI();
    ui.run();

    assertEquals(
        expectedOutput.toString(),
        outputStream.toString());
    assertEquals(ui.getAddedStudents());
}
finally {
    System.setIn(consoleIn);
    System.setOut(consoleOut);
}
}

```

通过使用 try-finally 语句来确保你重置了 System.in 和 System.out。

新的 StudentUI 构造函数必须使用 InputStreamReader 将标准输入 (stdin) 包装成为 Buffered Reader, 并使用 OutputStreamWriter 将标准输出 (stdout) 包装成为 BufferedWriter。

```

public StudentUI() {
    this.reader =
        new BufferedReader(new InputStreamReader(System.in));
    this.writer =
        new BufferedWriter(new OutputStreamWriter(System.out));
}

```

在演示测试通过之后, 你可以编写一个 main 方法来启动应用。

```

public static final void main(String[] args) throws IOException {
    new StudentUI().run();
}

```

这样做有许多考虑。首先, 有可能为 main 方法编写一个测试, 因为你可以像调用其它方法那样调用它 (但是你必须提供一个 String 对象的数组来表示命令行参数):

```
StudentUI.main(new String[] {});
```

测试的另一个原则是, main 方法实质上是牢不可破的。它只有一行代码。你至少要从命令行将这个应用运行一次, 来确保它能够正常工作。只要 main 方法保持不变, 它就不会抛锚 (break), 这样你就不必为 main 方法编写测试。

是否要测试 main 方法, 取决于你自己的选择。不管怎样, 你应该努力将 main 方法减少到只有一行代码。将 main 方法中的代码重构到静态或者实例方法。创建实用类来帮助你管理命令行参数。将你要测试的代码移出 main。

我希望你已经注意到, 测试这个简单的、基于终端的用户界面, 需要大量的代码。如果你

编写的多为终端应用，这些工作将帮助你在为测试和产品编码时，简化构建这些实用方法和类的工作量。

数据流

你可以将 Java 的基本数据类型直接写入到 `DataOutputStream` 中。`DataOutputStream` 是过滤（filtered）流的一个例子。过滤流包装另一个流，来提供附加的功能，或者以这种方式更改数据。基础的 filtered 类包括 `FilteredOutputStream`、`FilteredInputStream`、`FilterWriter` 和 `FilteredReader`。

`DataOutputStream` 中的过滤器提供了对每个 Java 基本类型的输出方法：`writeBoolean`、`writeDouble` 等等。它还提供了 `writeUTF` 方法来输出字符串。

395

CourseCatalog



学生信息系统需要一个 `CourseCatalog` 类来保存所有可参加的课程安排。`CourseCatalog` 负责将基本的课程信息（系、课程号、开始日期和学分）持久化到文件，这样应用在重新启动时不会丢失任何数据。

`CourseCatalog` 提供了一个 `load` 方法，从 `DataOutputStream` 流中读取所有的 `Session` 对象，并将它们保存到一个集合（collection）中。它还提供了一个 `store` 方法来把这个集合写入到 `DataOutputStream`。

```
package sis.studentinfo;

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

public class CourseCatalogTest extends TestCase {
    private CourseCatalog catalog;
    private Session session1;
    private Session session2;
    private Course course1;
    private Course course2;

    protected void setUp() {
        catalog = new CourseCatalog();
        course1 = new Course("a", "1");
        course2 = new Course("a", "1");

        session1 =
```



```

        CourseSession.create(
            course1, DateUtil.createDate(1, 15, 2005));
        session1.setNumberOfCredits(3);

        session2 =
            CourseSession.create(
                course2, DateUtil.createDate(1, 17, 2005));
        session2.setNumberOfCredits(5);

        catalog.add(session1);
        catalog.add(session2);
    }

    public void testStoreAndLoad() throws IOException {
        final String filename = "CourseCatalogTest.testAdd.txt";
        catalog.store(filename);
        catalog.clearAll();
        assertEquals(0, catalog.getSessions().size());
        catalog.load(filename);
        List<Session> sessions = catalog.getSessions();
        assertEquals(2, sessions.size());
        assertSession(session1, sessions.get(0));
        assertSession(session2, sessions.get(1));
    }

    private void assertSession(Session expected, Session retrieved) {
        assertNotSame(expected, retrieved);
        assertEquals(expected.getNumberOfCredits(),
            retrieved.getNumberOfCredits());
        assertEquals(expected.getStartDate(), retrieved.getStartDate());
        assertEquals(expected.getDepartment(), retrieved.getDepartment());
        assertEquals(expected.getNumber(), retrieved.getNumber());
    }
}

```

该测试将课程加载到目录（catalog）中，调用 store 方法，清空目录，然后调用 load 方法。它判断目录是否包含最初插入的两个课程。

在 CourseCatalog 中除 load 和 store 之外的代码比较琐碎：

```

package sis.studentinfo;

import java.util.*;
import java.io.*;

public class CourseCatalog {
    private List<Session> sessions =
        new ArrayList<Session>();

    public void add(Session session) {
        sessions.add(session);
    }

    public List<Session> getSession() {
        return sessions;
    }

    public void clearAll() {
        sessions.clear();
    }
}

```

```

    }
    // ...
}

```

sotre 方法通过包装 `FileOutputStream` 创建了一个 `DataOutputStream`。它首先写入一个 `int` 来表示要保存的课程安排的数目。然后它遍历所有的课程安排，写入每个的开始时间，学分的多少，系 (department) 以及课程编号。

```

public void store(String filename) throws IOException {
    DataOutputStream output = null;
    try {
        output =
            new DataOutputStream(new FileOutputStream(filename));
        output.writeInt(sessions.size());
        for (Session session: sessions) {
            output.writeLong(session.getStartDate().getTime());
            output.writeInt(session.getNumberOfCredits());
            output.writeUTF(session.getDepartment());
            output.writeUTF(session.getNumber());
        }
    } finally {
        output.close();
    }
}

```

397

你需要在 `Session` 中创建 `getter` 方法来返回学分的多少：

```

public int getNumberOfCredits() {
    return numberOfCredits;
}

```

load 方法通过包装 `FileInputStream` 来创建 `DataInputStream`。它读取课程安排的数目，来判断在文件中保存了多少课程安排的信息。这种保存对象数目的方法，相对另外一种方法，即在每次读操作时等待文件结束异常，更为可取。

load 方法假定要读取的课程安排是 `CourseSession` 对象，不是 `SummerCourseSession` 或其它的 `Session` 子类。如果 `CourseCatalog` 需要支持多种类型，你还需要保存 `Session` 的类型。类型信息将可以让 load 方法中的代码，了解在读取每个对象时应该要实例化哪一个类。

```

public void load(String filename) throws IOException {
    DataInputStream input = null;
    try {
        input = new DataInputStream(new FileInputStream(filename));
        int numberOfSessions = input.readInt();
        for (int i = 0; i < numberOfSessions; i++) {
            Date startDate = new Date(input.readLong());
            int credits = input.readInt();
            String department = input.readUTF();
            String number = input.readUTF();
            Course course = new Course(department, number);
            Session session =
                CourseSession.create(course, startDate);
            session.setNumberOfCredits(credits);
            sessions.add(session);
        }
    }
}

```

```
    }  
    finally {  
        input.close();  
    }  
}
```

load 和 store 方法通过使用 finally 代码块，确保相关的数据流最终会被关闭。

高级流

管道 (pipelined) 流

你可以使用管道流在不同的线程间进行安全的基于 IO 的数据通讯。管道流以成对儿的方式工作：被写入输出管道流的数据，会被与之相联的输入管道流读取。管道流的实现包括：PipedInputStream、PipedOutputStream、PipedReader 和 PipedWriter。关于多线程的更多信息，请参考第 13 课。

SequenceInputStream

你可以使用 SequenceInputStream，使一组输入流的行为如同一个单独的输入流一样。资源的集合是有序的：当一个资源被完整地读取后，它就会被关闭，然后集合中的下一个流将被打开和读取。

Pushback 流

Pushback 流 (PushbackInputStream 和 PushbackReader) 的主要用途，是用于词法分析程序 (例如编译器的分词程序)。它们可以将数据回放到流中，就如它没有被流读取一样。

StreamTokenizer

StreamTokenizer 的主要用途，也是用于解析 (parsing) 的应用。它的行为同 StringTokenizer 类似。不过不同于 StringTokenizer 只从底层流返回字符串，StreamTokenizer 不仅返回分词 (token) 的值，还会返回其类型。分词的类型可能是一个单词、数字、行结束标记或文件结束标记。

对象流

399

Java 提供了直接从/向流中读写对象的能力。Java 可以借由对象的序列化（serialization）将对象写入到对象输出流中。Java 通过将对象转换为一个字节的序列，将之序列化。将对象序列化的能力，是 Java RMI（Remote Method Invocation）技术的基础。RMI 能够让对象在同远程系统中的对象交互时，就像访问本地对象一样。RMI 进而是 Java EJB（Enterprise Java Bean）这一基于组件计算技术的基础。

你可以使用 `ObjectOutputStream` 和 `ObjectInputStream` 来写入和读取对象。作为概略的演示，下面的代码重写了 `CourseCatalog` 类中的 `store` 和 `load` 方法。修改后的代码使用对象流而不是数据流。

```
public void store(String filename) throws IOException {
    ObjectOutputStream output = null;
    try {
        output =
            new ObjectOutputStream(new FileOutputStream(filename));
        output.writeObject(sessions);
    }
    finally {
        output.close();
    }
}

public void load(String filename)
    throws IOException, ClassNotFoundException {
    ObjectInputStream input = null;
    try {
        input = new ObjectInputStream(new FileInputStream(filename));
        sessions = (List<Session>)input.readObject();
    }
    finally {
        input.close();
    }
}
```

测试用例基本保持不变。

```
public void testStoreAndLoad() throws Exception {
    final String filename = "CourseCatalogTest.testAdd.txt";
    catalog.store(filename);
    catalog.clearAll();
    assertEquals(0, catalog.getSessions().size());
    catalog.load(filename);

    List<Session> sessions = catalog.getSessions();
    assertEquals(2, sessions.size());
    assertSession(session1, sessions.get(0));
    assertSession(session2, sessions.get(1));
}
```

测试方法原型中的 `throws` 子句必须改变，`load` 方法现在可能会抛出 `ClassNot`

400

FileNotFoundException。当然就这个示例而言，它不大会产生 ClassNotFoundException。你保存了一组 Session 对象，并立刻把它们读取回来，你的代码需要了解 java.util.List 和 Session 类。不过如果另一个应用不能访问到你的 Session 类，而想要读取文件中的对象，异常就会被抛出。

在运行该测试时，你应该收到这样一个异常：

```
java.io.NotSerializableException: studentinfo.CourseSession
```

要将对象写入到对象流中，该对象的类必须是可序列化的。通过声明你的类实现了 java.io.Serializable 接口，可以标明这个类是可序列化的。在 Java 系统类库中，大多数你预计是可序列化的类，都已经这样标注了。包括 String 和 Date 类，以及所有的集合类 (HashMap、ArrayList 等等)。不过你需要标记你在自己应用中的类：

```
abstract public class Session
    implements Comparable<Session>,
        Iterable<Student>,
        java.io.Serializable {...
```

别忘了 Course 类，因为 Session 封装了它：

```
public class Course implements java.io.Serializable {...
```

当你将抽象父类标注为可序列化时，所有它的子类也将能够被序列化。Serializable 接口没有包含任何方法的定义，因此你不需要为 Session 额外做任何事情。

没有什么任何方法的接口被称为标记式接口 (marker interface)。为某种特定用途，开发者可以创建标记式接口以显式地标记一个类。你必须显式地指定一个类有能力被序列化。Serializable 标记被设计作为一种安全机制——出于安全的考虑，你可能需要这种机制来阻止某些特定的对象被序列化。

Transient

课程安排应该支持学生登记。不过，你并不希望课程目录对象被学生对象搞乱。假定，虽然目录还未创建，就有学生登记。举例来说，CourseCatalogTest 中的 setUp 方法登记了一个学生：

```
protected void setUp() {
    catalog = new CourseCatalog();
    course1 = new Course("a", "1");
    course2 = new Course("a", "1");

    session1 =
        CourseSession.create(
            course1, DateUtil.createDate(1, 15, 2005));
    session1.setNumberOfCredits(3);

    session2 =
        CourseSession.create(
            course2, DateUtil.createDate(1, 17, 2005));
    session2.setNumberOfCredits(5);
    session2.enroll(new Student("a"));

    catalog.add(session1);
```

```
catalog.add(session2);
}
```

如果你运行测试，你会再次收到一个 `NotSerializableException`。课程安排现在引用了一个必须被序列化的 `Student` 对象。但是 `Student` 类并没有实现 `java.io.Serializable`。

除了更改 `Student`，你可以通过 `Transient` 修饰符 (modifier) 指示在序列化时跳过 `Session` 中的学生对象列表。

```
abstract public class Session
    implements Comparable<Session>,
               Iterable<Student>,
               java.io.Serializable {
    ...
    private transient List<Student> students = new ArrayList<Student>();
```

在本示例中，学生对象的列表将不会被序列化。你的测试现在可以顺利通过了。

序列化与更动

序列化使得对象持久化非常容易。也许，太容易了。将类声明为 `Serializable`，有许多含意。最显著的结果是，当你持久化一个被序列化的对象时，你同时会导出这个类当前的定义。如果你随后更改了类的定义，再尝试读取已序列化的对象时，会得到一个异常。

作为演示，我们在 `Session.java` 中添加成员变量 `name`。这会创建 `Session` 类的新版本。不要担心测试：这是一个暂时的“刺痛”，或试验。你稍后会删除这行代码。而且，不要运行任何测试，否则将会搞砸这个试验。

402

```
abstract public class Session
    implements Comparable<Session>,
               Iterable<Student>,
               java.io.Serializable {
    private String name;
    ...
```

你上一次执行的测试，将对象流持久化到名为 `CourseCatalogTest.testAdd.txt` 的文件中。在文件中的对象流，包括了使用旧版 `Session` 类定义（没有成员变量 `name`）创建的 `Session` 对象。

然后，创建一个全新的测试，`studentinfo.SerializationTest`：

```
package sis.studentinfo;

import junit.framework.*;

public class SerializationTest extends TestCase {
    public void testLoadToNewVersion() throws Exception {
        CourseCatalog catalog = new CourseCatalog();
        catalog.load("CourseCatalogTest.testAdd.txt");
        assertEquals(2, catalog.getSessions().size());
    }
}
```

该测试尝试装入被持久化的对象流。只执行这个测试。不要执行你的 `AllTests` 套件 (suite)。

你应该会收到类似下面的异常：

```
testLoadToNewVersion(studentinfo.SerializationTest)
java.io.InvalidClassException: studentinfo.Session;
local class incompatible:
stream classdesc serialVersionUID = 5771972560035839399,
local class serialVersionUID = 156127782215802147
```

Java 判断在输出流中保存的对象，和现有（本地）类定义之间的兼容性。它考察类的名称，该类所实现的接口，它的成员变量（field）和方法。改变其中的任何一项，将会导致不兼容。

不过 `transient` 的成员变量会被忽略。如果你把 `Session` 中成员变量 `name` 的声明改为 `transient`：

```
private transient String name;
```

`SerializationTest` 将可以通过。

403

序列版本（Serial Version）UID

你所收到的 `InvalidClassException`，包括了对象流中的类定义、以及本地（当前 Java 中）类定义的 `serialVersionUID`。为了判断类的定义是否发生了改变，Java 会基于类名、实现的接口、成员变量和方法来生成这个 `serialVersionUID`。这个 `serialVersionUID` 是 64 位的长整型，被称为流的唯一标识符（stream unique identifier）。

你可以选择定义自己的 `serialVersionUID`，而不使用 Java 生成的那一个。这为你更好地控制版本管理，提供了一些方法。你可以通过使用命令行工具 `serialver` 来获得初始的 `serialVersionUID`，或者赋给它任意的值。执行 `serialver` 的示例：

```
serialver -classpath classes studentinfo.Session
```

指定 `classpath` 是可选地（optionally），后面跟着你希望生成 `serialVersionUID` 的类的列表。

将成员变量 `name` 从 `Session` 中删除，重新编译并重新运行你整个的测试套件。然后，在 `Session` 中添加一个 `serialVersionUID` 的定义，把成员变量 `name` 添加回来。

```
abstract public class Session
    implements Comparable<Session>,
        Iterable<Student>,
        java.io.Serializable {
    public static final long serialVersionUID = 1L;
    private String name;
    ...
}
```

然后运行 `SerializationTest`。即使你已经添加了新的成员变量，但版本 ID 是相同的。Java 将会把成员变量 `name` 的初始值设为 `null`。如果你把 `serialVersionUID` 的值设为 2L 并运行测试，你将会导致流中的类版本（1）和本地类版本（2）失去同步。

创建一个自定义的序列化格式

在你类中所包含的某些信息，可能需要基于类中的其它数据来重新构造。在对类建模（model）时，你会为之定义若干属性来表示这个类每个对象实例的逻辑状态。除了那些属性之外，你可能需要一些数据结构或其它成员变量来缓存（cache）动态计算的信息。序列化这类动态计算的数据可能非常慢，而且空间的使用效率很低。



假定你不仅需要持久化课程安排，还要持久化每个课程登记的学生信息。学生带有大量的额外数据，而它们已经在其它某处被持久化了。你可以遍历课程安排的集合，并只将学生的唯一标识符持久化到对象流中¹。当你加载这个紧缩的集合时，你可以执行查找来取回完整的学生对象，并将它保存在课程安排中。

404

要完成这一目标，你需要在 `Session` 中定义两个方法，`writeObject` 和 `readObject`。在为每个对象读写对象流时，这些方法钩住（hook）了序列化机制的调用。如果你不提供这些钩子，就会执行缺省的序列化和反序列化。

首先，更改 `CourseCatalogTest` 中的测试，以确保登记的学生能够正确地持久化和恢复。

```
public void testStoreAndLoad() throws Exception {
    final String filename = "CourseCatalogTest.testAdd.txt";
    catalog.store(filename);
    catalog.clearAll();
    assertEquals(0, catalog.getSessions().size());
    catalog.load(filename);

    List<Session> sessions = catalog.getSessions();
    assertEquals(2, sessions.size());
    assertSession(session1, sessions.get(0));
    assertSession(session2, sessions.get(1));

    Session session = sessions.get(1);
    assertSession(session2, session);
    Student student = session.getAllStudents().get(0);
    assertEquals("a", student.getLastName());
}
```

405

确保 `Session` 中的成员变量 `students` 被标记为 `Transient`。然后编写 `Session` 的 `writeObject` 定义：

```
private void writeObject(ObjectOutputStream output)
    throws IOException {
    output.defaultWriteObject();
    output.writeInt(students.size());
    for (Student student: students)
        output.writeObject(student.getLastName());
}
```

`writeObject` 方法的第一行，调用了流的 `defaultWriteObject` 方法。这会如通常一样，

¹ 我们是一个很小的学校。我们并不容许有多个学生具有相同的姓（last name），因此你也可以使用它作为其唯一标识符。

把所有非 `transient` 的成员变量写入到流中。接下来, `writeObject` 中的代码首先将学生的数目写入到流中, 然后遍历学生的列表, 把每个学生的姓 (`last name`) 写入到流中。

```
private void readObject(ObjectInputStream input)
    throws Exception {
    input.defaultReadObject();
    students = new ArrayList<Student>();
    int size = input.readInt();
    for (int i = 0; i < size; i++) {
        String lastName = (String)input.readObject();
        students.add(Student.findByLastName(lastName));
    }
}
```

在另一端, `readObject` 首先调用 `defaultReadObject` 从流中载入所有非 `transient` 的成员变量。它将 `transient` 的成员变量的 `students`, 初始化为一个新的 `Student` 数组。它将学生的数目读出赋值给 `size`, 并作为迭代的次数。每次迭代会从流中提取 (`extract`) 出学生的姓。余后的代码使用学生的姓查找并取回 `Student` 对象, 并把该 `Student` 对象保存在 `students` 集合中。

在现实中, `findByLastName` 方法可能包括: 调用学生字典对象的某个方法, 继而从数据库或其它序列化文件中取回适当的学生对象。作为演示的目的, 你可以提供一个简单的实现, 使其通过测试:

```
public static Student findByLastName(String lastName) {
    return new Student(lastName);
}
```

序列化的提议

对于那些定义可能发生变化的类来说, 处理序列化版本的不兼容性, 可能是最头痛问题。虽然从一个早期版本加载已序列化的对象是有可能的, 但这依然很困难。你的最佳策略包括:

- 将序列化的使用最小化
- 尽可能使用 `transient` 的成员变量
- 使用 `serialVersionUID` 标识版本
- 定义一个自定义的序列化格式

在你序列化对象时, 将会导出它的接口 (译注: 应为定义)。正如你应该尽可能地保持接口的抽象性、并避免对其改动那样, 对可序列化的类你也应如此。

随机存取文件

除了在每次执行应用时将整个课程目录装入和保存, 你可以通过随机存取文件来实现同目录

的动态交互。随机存取文件可以让你快速地定位到文件中的特定位置，并从该位置读取或写入。

在 Java 中使用随机存取文件创建全功能的对象数据库，是有可能的。接下来的示例代码是个起点。

在第 9 课中，你创建了 `StudentDirectory` 类。在这个类中你使用 `HashMap` 来保存学生对象，使用学生的 ID 作为 `key`。

现在，你需要确保你的学生词典可以支持大学内数以万计的学生进行登记。而且，你必须要将词典持久化到文件系统中，以保全数据。从词典中检索学生的操作必须要快速。检索必须在一定时间内完成，访问学生的时间量不应因对象在文件中的存储位置不同而发生变化。

你可以使用简单的索引文件系统（indexed file system）来实现学生字典。你可以把学生记录保存在数据文件（data file）中，然后将每个学生的唯一标识符保存在索引文件中。当你将一个已序列化的学生对象插入到数据文件中时，把它的 `id`、位置和长度记录放在索引文件中。和数据文件大小比较起来，索引文件是非常小的。它可以非常快地被装入到内存中，并在数据文件关闭时快速地回写。²

示例代码是本书迄今为止牵扯最多的代码了。如果你每次只考察一个测试和方法，理解它应该不会有太多麻烦。从头建立测试和代码是有一定挑战的。UML 类图如图 11.1 所示。

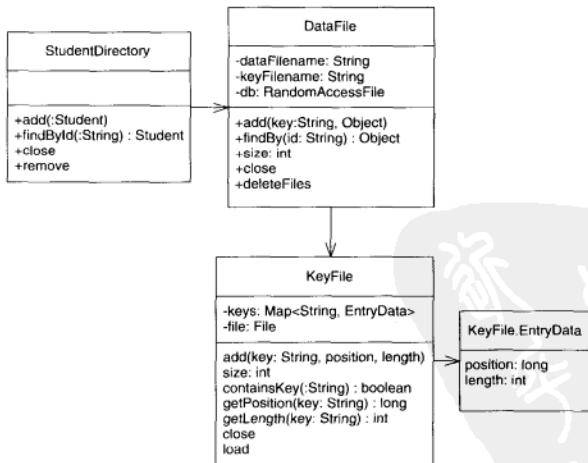


图 11.1 学生字典

² 在你添加 `Student` 时，没有立即将索引数据持久化，是有风险的。你可以通过在数据文件本身中也写入数据的长度来降低这一风险。这样做可以让你通过遍历数据文件来重建索引文件。

下面是代码的部分。我会在列出每个类时解释其中令人感兴趣的部分。

学生字典

```
package sis.studentinfo;

import junit.framework.*;
import java.io.*;

public class StudentDirectoryTest extends TestCase {
    private StudentDirectory dir;

    protected void setUp() throws IOException {
        dir = new StudentDirectory();
    }

    protected void tearDown() throws IOException {
        dir.close();
        dir.remove();
    }

    public void testRandomAccess() throws IOException {
        final int numberOfStudents = 10;
        for (int i = 0; i < numberOfStudents; i++)
            addStudent(dir, i);
        dir.close();

        dir = new StudentDirectory();
        for (int i = 0; i < numberOfStudents; i++)
            verifyStudentLookup(dir, i);
    }

    void addStudent(StudentDirectory directory, int i)
        throws IOException {
        String id = "" + i;
        Student student = new Student(id);
        student.setId(id);
        student.addCredits(i);
        directory.add(student);
    }

    void verifyStudentLookup(StudentDirectory directory, int i)
        throws IOException {
        String id = "" + i;
        Student student = dir.findById(id);
        assertEquals(id, student.getLastName());
        assertEquals(id, student.getId());
        assertEquals(i, student.getCredits());
    }
}
```

`StudentDirectoryTest` 中最显著的新增部分位于 `testRandomAccess` 方法中。在它向学生添加到字典后，测试会将之关闭。然后它会创建一个新的字典实例用来查找学生。通过这样做，

测试至少演示了某些持久化的概念。

为了演示学生对象无论保存在文件的什么地方，查找字典都会花费相同的时间量，一个附加的性能测试是值得的。将学生对象添加到字典中，也应该在常量时间内完成。

```
package sis.studentinfo;

import java.io.*;
import sis.db.*;

public class StudentDirectory {
    private static final String DIR_BASENAME = "studentDir";
    private DataFile db;

    public StudentDirectory() throws IOException {
        db = DataFile.open(DIR_BASENAME);
    }

    public void add(Student student) throws IOException {
        db.add(student.getId(), student);
    }

    public Student findById(String id) throws IOException {
        return (Student)db.findById(id);
    }

    public void close() throws IOException {
        db.close();
    }

    public void remove() {
        db.deleteFiles();
    }
}
```

409

不过，StudentDirectory 类中的大部分都已经改变了。StudentDirectory 现在封装了一个 DataFile 的实例来提供字典的功能。它提供了一些额外的细节，包括要使用的成员变量 key（学生的 id），以及数据文件和 key 文件的基本文件名（base filename）。除此之外，该类只是将调用（message）委托给 DataFile 对象。

sis.db.DataFileTest

```
package sis.db;

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

public class DataFileTest extends TestCase {
    private static final String ID1 = "12345";
    private static final String ID2 = "23456";
    private static final String FILEBASE = "DataFileTest";
}
```

```

private DataFile db;
private TestData testData1;
private TestData testData2;
protected void setUp() throws IOException {
    db = DataFile.create(FILEBASE);
    assertEquals(0, db.size());

    testData1 = new TestData(ID1, "datum1a", 1);
    testData2 = new TestData(ID2, "datum2a", 2);
}

protected void tearDown() throws IOException {
    db.close();
    db.deleteFiles();
}

public void testAdd() throws IOException {
    db.add(ID1, testData1);
    assertEquals(1, db.size());

    db.add(ID2, testData2);
    assertEquals(2, db.size());

    assertTestDataEquals(testData1, (TestData)db.find(ID1));
    assertTestDataEquals(testData2, (TestData)db.find(ID2));
}

public void testPersistence() throws IOException {
    db.add(ID1, testData1);
    db.add(ID2, testData2);
    db.close();

    db = DataFile.open(FILEBASE);
    assertEquals(2, db.size());

    assertTestDataEquals(testData1, (TestData)db.find(ID1));
    assertTestDataEquals(testData2, (TestData)db.find(ID2));

    db = DataFile.create(FILEBASE);
    assertEquals(0, db.size());
}

public void testKeyNotFound() throws IOException {
    assertNull(db.find(ID2));
}

private void assertTestDataEquals(
    TestData expected, TestData actual) {
    assertEquals(expected.id, actual.id);
    assertEquals(expected.field1, actual.field1);
    assertEquals(expected.field2, actual.field2);
}

static class TestData implements Serializable {
    String id;
    String field1;
    int field2;
    TestData(String id, String field1, int field2) {
        this.id = id;
    }
}

```

```

        this.field1 = field1;
        this.field2 = field2;
    }
}

```

411

`DataFileTest` 演示了使用静态工厂 (static factory) 方法 `create`, 创建一个新的 `DataFile`。
`create` 方法以 `DataFile` 的名字作为参数。

测试还演示了通过使用唯一的 `key` 和相关联的对象作为参数, 调用 `add` 方法将对象插入到 `DataFile` 中。要取回对象, 调用 `findBy`, 传入要检索对象的唯一 `id`。如果对象不可获得, `DataFile` 返回 `null`。

持久化的测试, 是通过关闭一个 `DataFile`, 并使用静态工厂方法 `open`³ 创建新实例来进行的。
`open` 和 `create` 之间的区别是, `create` 将会删除数据文件 (如果它已经存在的话), 而 `open` 将会重用现有的数据文件或者在必要时创建新的。

注意 `findBy` 方法返回的对象需要强制转型 (`cast`)! 这暗示了 `DataFile` 是以参数化类型实现的候选者。(关于参数化类型的更多内容, 参见第 14 课。)

静态内嵌 (static nested) 类和内联 (inner) 类

`DataFileTest` 需要证明, `DataFile` 可以持久化对象, 并在稍后把该对象取回。为了编写这样的测试, 最好使用你能确保其他人不会修改的类。你可以使用 Java 系统库中的类, 例如 `String`, 但是它们会随着 Java 的新版本而改变。不过, 更好的方案是创建一个只被 `DataFileTest` 使用的类。

`TestData` 类被定义为 `DataFileTest` 的内嵌类; 也就是说, 它是完全被包含在 `DataFileTest` 之内的。内嵌类有两种类型: 内联 (inner) 类和静态内嵌 (static nested) 类。主要的区别是, 内联类可以访问定义在外围类 (enclosing class) 中的实例变量。而静态内嵌类则不可以。

另一区别是, 内联类是完全被封装在外围类中的。因为内联类可以引用外围类的实例变量, 让其它代码有能力去创建实例内联类 (instance inner class) 的实例是没有意义的。虽然 Java 在技术上允许你从外部代码中引用内联类, 其中有些诡计花招, 我是不会在这里演示给你的, 不要试图这样做!

412

内嵌类的类名

当你编译类或接口时, 使用类的名字添加 `.class` 作为文件扩展名, 来命名编译单元 (Compilation Unit)。例如, `Couse` 类会编译成为 `Course.class`。

³ 严格的说这个测试并不能证实磁盘上的持久化。你可以已经实现了一个方案, 将对象保存在一个静态区 (static-side) 的集合中。不过, 测试的要点并不是防止你不诚实或编写愚蠢的方案, 相反地, 测试演示预期的行为。不过, 如果你还不确信, 没有任何限制让你编写一个确保对象确实被保存在磁盘文件的测试用例。它只是更复杂, 并且可能没什么必要。

当你编译包含内嵌类的类时, Java 会使用外围类的名字、紧随一个\$符号、再加上内联类的名字来命名内联类的编译单元。

例如, `DataFileTest` 类会创建两个编译单元: `DataFileTest.class` 和 `DataFileTest$TestData.class`。

当你出于发布版本的目的, 将类文件拷贝到 JAR 文件内 (参见附加第 3 课) 时, 不要忘记包括所有的内嵌类。

另一方面, 静态内嵌类可以被外部代码所使用, 只要访问限定符不是 `private`。你已经使用了静态内嵌类 `Entry`, 来遍历 `Map` 对象中的 `key-value` 对儿。在 `Map` 之外的代码上下文中, 你需要使用 `Map.Entry` 来引用这个类。

因此, 将内嵌类声明为静态的首要原因是, 让其它的类可以使用它。你可以将类声明为顶层类 (top-level, 也就是非内嵌类), 但是你可能希望把它紧密地绑定在容器类中。例如, `Map.Entry` 被紧密地绑定到 `Map`; 因为如果缺少 `Map`, `Entry` 类就没有存在的意义。

将内嵌类声明为静态的第二个原因是, 允许将其序列化。你不能序列化内联类对象, 因为它们能够访问外围类的实例变量。要让序列化工作, 序列化机制必须处理外围类的成员变量。这实在令人厌恶。

因为你需要持久化 `TestData` 对象, 你必须使这个类是可被序列化的。如果 `TestData` 是内嵌类, 你必须将它声明为 `static`。

sis.db.DataFile

```
package sis.db;

import java.util.*;
import java.io.*;
import sis.util.*;
public class DataFile {
    public static final String DATA_EXT = ".db";
    public static final String KEY_EXT = ".idx";

    private String dataFilename;
    private String keyFilename;

    private RandomAccessFile db;
    private KeyFile keys;

    public static DataFile create(String filebase) throws IOException {
        return new DataFile(filebase, true);
    }

    public static DataFile open(String filebase) throws IOException {
        return new DataFile(filebase, false);
    }

    private DataFile(String filebase, boolean deleteFiles)
```

413

```

        throws IOException {
            dataFilename = filebase + DATA_EXT;
            keyFilename = filebase + KEY_EXT;

            if (deleteFiles)
                deleteFiles();
            openFiles();
        }

        public void add(String key, Object object) throws IOException {
            long position = db.length();

            byte[] bytes = getBytes(object);
            db.seek(position);
            db.write(bytes, 0, bytes.length);

            keys.add(key, position, bytes.length);
        }

        public Object findBy(String id) throws IOException {
            if (!keys.containsKey(id))
                return null;

            long position = keys.getPosition(id);
            db.seek(position);

            int length = keys.getLength(id);
            return read(length);
        }

        public int size() {
            return keys.size();
        }

        public void close() throws IOException {
            keys.close();
            db.close();
        }

        public void deleteFiles() {
            IOUtil.delete(dataFilename, keyFilename);
        }

        private Object read(int length) throws IOException {
            byte[] bytes = new byte[length];
            db.readFully(bytes);
            return readObject(bytes);
        }

        private Object readObject(byte[] bytes) throws IOException {
            ObjectInputStream input =
                new ObjectInputStream(new ByteArrayInputStream(bytes));
            try {
                return input.readObject();
            }
            catch (ClassNotFoundException unlikely) {
                // but write a test for it if you must
                return null;
            }
        }

```

```

    }
    finally {
        input.close();
    }
}

private void openFiles() throws IOException {
    keys = new KeyFile(keyFilename);
    db = new RandomAccessFile(new File(dataFilename), "rw");
}

private byte[] getBytes(Object object) throws IOException {
    ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    ObjectOutputStream outputStream =
        new ObjectOutputStream(byteStream);
    outputStream.writeObject(object);
    outputStream.flush();
    return byteStream.toByteArray();
}
}

```

`DataFile` 类是该方案的核心部分。它演示了保存在实例变量 `db` 中（数据库 `database` 的缩写，我所使用的少数缩写之一）的 `RandomAccessFile` 对象的使用。你通过传入一个 `File` 对象和一个表明 `RandomAccessFile` 访问模式的字符串，来创建 `RandomAccessFile` 对象。

`RandomAccessFile` 提供了四种模式：“r”（只读）、“rw”（读写访问）、“rws”（同步数据/元数据更新的读写）和“rwd”（同步数据更新的写）。同步更新确保对文件的更改被安全地写入到底层的存储设备中。如果不使用同步模式，在系统崩溃时你肯定会丢失数据。“rws”选项确保内容和元数据（例如文件的最近更改时间戳等信息）的持久化，而“rwd”选项只确保内容的更新。

这里我们选择“rw”选项，因为你希望能够从数据文件中读取并写入数据。另两种读写选项会招致额外的开销，但对于维护数据的完整性来说可能是必要的。

你使用 `seek` 方法快速地将一个内部文件指针移动到底层文件中的任何位置。`getFilePointer` 方法（没有在这里使用）返回文件指针的当前位置。`length` 方法返回文件总共的字节数。和其它 `IO` 类一样，`RandomAccessFile` 提供了许多 `read` 和 `write` 方法来取出和保存数据。

`RandomAccessFile` 并不直接支持保存对象。为了将对象持久化到 `RandomAccessFile`，你必须首先将它转换为一个字节数组。要将对象转换为字节，`getBytes` 方法将 `ByteArrayOutputStream` 包装到 `ObjectOutputStream` 中。这意味着，所有写入到 `ObjectOutputStream` 中的对象，都被管接（pipe）到底层的 `ByteArrayOutputStream`。你可以通过调用 `toByteArray` 方法从 `ByteArrayOutputStream` 中取出所有的字节内容。

要从 `RandomAccessFile` 中读取对象，你必须做相反的事情。以适当的长度创建一个字节数组，并使用 `readFully` 方法从 `RandomAccessFile` 中读出并填充字节。将填充的字节数组包装在 `ByteArrayInputStream` 中，然后你再把把这个流包装到 `ObjectInputStream` 中。从 `ObjectInputStream` 中进行读取，就会使用底层的字节数据重建已持久化的对象。

sis.db.KeyFileTest

```
package sis.db;

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

public class KeyFileTest extends TestCase {
    private static final String FILENAME = "keyfiletest.idx";
    private static final String KEY = "key";
    private static final long POSITION = 1;
    private static final int LENGTH = 100;

    private KeyFile keyFile;

    protected void setUp() throws IOException {
        TestUtil.delete(FILENAME);
        keyFile = new KeyFile(FILENAME);
    }

    protected void tearDown() throws IOException {
        TestUtil.delete(FILENAME);
        keyFile.close();
    }

    public void testCreate() {
        assertEquals(0, keyFile.size());
    }

    public void testAddEntry() {
        keyFile.add(KEY, POSITION, LENGTH);

        assertEquals(1, keyFile.size());
        assertTrue(keyFile.containsKey(KEY));
        assertEquals(POSITION, keyFile.getPosition(KEY));
        assertEquals(LENGTH, keyFile.getLength(KEY));
    }

    public void testReopen() throws IOException {
        keyFile.add(KEY, POSITION, LENGTH);
        keyFile.close();

        keyFile = new KeyFile(FILENAME);
        assertEquals(1, keyFile.size());
        assertEquals(POSITION, keyFile.getPosition(KEY));
        assertEquals(LENGTH, keyFile.getLength(KEY));
    }
}
```

416

KeyFileTest 演示了向 **KeyFile** 添加 key (唯一 id) 的功能。**Key** 是通过 **DataFile** 对象中相关数据的位置和长度保存的。通过唯一的 **key**，可以从 **KeyFile** 中取出数据的位置和长度。

第三个测试，**testReopen**，确保你能够使用现有 **key** 文件的文件名创建一个新的 **KeyFile**

417

对象。KeyFile 对象必须装入已经持久化的 key 数据。

sis.db.KeyFile

```

package sis.db;

import java.util.*;
import java.io.*;

class KeyFile {
    private Map<String, EntryData> keys =
        new HashMap<String, EntryData>();
    private File file;

    KeyFile(String filename) throws IOException {
        file = new File(filename);
        if (file.exists())
            load();
    }

    void add(String key, long position, int length) {
        keys.put(key, new EntryData(position, length));
    }

    int size() {
        return keys.size();
    }

    boolean containsKey(String key) {
        return keys.containsKey(key);
    }

    long getPosition(String key) {
        return keys.get(key).getPosition();
    }

    int getLength(String key) {
        return keys.get(key).getLength();
    }

    void close() throws IOException {
        ObjectOutputStream stream =
            new ObjectOutputStream(new FileOutputStream(file));
        stream.writeObject(keys);
        stream.close();
    }

    void load() throws IOException {
        ObjectInputStream input = null;
        try {
            input = new ObjectInputStream(new FileInputStream(file));
            try {
                keys = (Map<String, EntryData>)input.readObject();
            }
        }
    }
}

```

418

```

        catch (ClassNotFoundException e) {
        }
    }
    finally {
        input.close();
    }
}

static class EntryData implements Serializable {
    private long position;
    private int length;

    EntryData(long position, int length) {
        this.position = position;
        this.length = length;
    }

    private long getPosition() {
        return position;
    }

    private int getLength() {
        return length;
    }
}
}

```

KeyFile 使用名为 **keys** 的 **Map** 保存 **key** 信息。这个 **Map** 对象将 **key** 映射为一个可序列化的静态内嵌类，**EntryData**，它保存了数据的位置和长度。在关闭时，**KeyFile** 通过使用 **ObjectOutputStream** 将整个 **Map** 写入到文件中。在打开时，它会装入整个 **Map**。

sis.util.IOUtilTest

```

package sis.util;

import junit.framework.*;
import java.io.*;

public class IOUtilTest extends TestCase {
    static final String FILENAME1 = "IOUtilTest1.txt";
    static final String FILENAME2 = "IOUtilTest2.txt";

    public void testDeleteSingleFile() throws IOException {
        create(FILENAME1);
        assertTrue(IOUtil.delete(FILENAME1));
        TestUtil.assertGone(FILENAME1);
    }

    public void testDeleteMultipleFiles() throws IOException {
        create(FILENAME1, FILENAME2);
        assertTrue(IOUtil.delete(FILENAME1, FILENAME2));
        TestUtil.assertGone(FILENAME1, FILENAME2);
    }
}

```

```

public void testDeleteNoFile() throws IOException {
    TestUtil.delete(FILENAME1);
    assertFalse(IOUtil.delete(FILENAME1));
}

public void testDeletePartiallySuccessful() throws IOException {
    create(FILENAME1);
    TestUtil.delete(FILENAME2);
    assertFalse(IOUtil.delete(FILENAME1, FILENAME2));
    TestUtil.assertGone(FILENAME1);
}

private void create(String... filenames) throws IOException {
    for (String filename: filenames) {
        TestUtil.delete(filename);
        new File(filename).createNewFile();
    }
}
}

```

IOUtilTest 中最有趣的一个方面是，它包含了 4 个测试方法，每个都测试相同的 IOUtil 方法 delete。每个测试提供了一种典型的场景。类似的测试可能有更多种。判断测试的数量是否已经让你对自己的代码有足够的信心，完全取决于你自己。



和测试用例过少比较来说，测试用例过多也非正确的方向。

sis.util.IOUtil

```

package sis.util;

import java.io.*;

public class IOUtil {
    public static boolean delete(String... filenames) {
        boolean deletedAll = true;
        for (String filename: filenames)
            if (!new File(filename).delete())
                deletedAll = false;
        return deletedAll;
    }
}

```

delete 方法使用可变参数，让你可以在一个方法调用中删除多个文件。只有所有文件均被成功删除了，它才返回 true。

sis.util.TestUtil

```
package sis.util;

import junit.framework.*;
import java.io.*;

public class TestUtil {
    public static void assertGone(String... filenames) {
        for (String filename: filenames)
            Assert.assertFalse(new File(filename).exists());
    }

    public static void delete(String filename) {
        File file = new File(filename);
        if (file.exists())
            Assert.assertTrue(file.delete());
    }
}
```

只有测试用例才使用 TestUtil 类中的代码。因为 TestUtil 并非从 junit.framework.TestCase 中派生，它没有任何的 assert 方法可供访问。不过，junit.framework.Assert 将 assert 方法定义为类方法，可以让你在任何地方调用它们。

其它你需要做的事情包括：

- 让 GradingStrategy 的实现类型和 Student 可序列化。
- 在 Student 类中添加成员变量 id 以及相应的 getter/setter 方法。
- 更新 AllTest 套件类。⁴

421

方案的改进

在本示例中，我先是尝试通过你在 StudentDirectoryTest 中所看到的一个测试 testRandomAccess 来实现 StudentDirectory。事如寻常，为 StudentDirectory 构建功能的任务，呈现为一个很复杂的步骤。在我编写 StudentDirectory 中的方法存根（stub）时，它继而衍生出让我构建 DataFile 类的任务。这意味着我在 StudentDirectoryTest 上的工作暂时被搁置起来。类似的，在我开始 DataFile 的工作时，我发现将 key 的功能封装到一个名为 KeyFile 的类中，会更容易。

正如我在开发这个方案时不断改进的那样，我不断地重新评估手中的设计。我的总体策略

⁴ 必须牢记添加类会引入诸多问题。在第 12 课中，你将学习如何使用 Java 的反射能力来动态地构建测试套件。

是，从“外部”开始勾勒一个初始的设计，并不断深入。换言之，我先确定客户端代码希望如何使用一个类，并基于这个接口进行类的设计。如果必要，我会勾勒出概要的结构，并继续完成某些“内部”的细节。无论怎样进行，我发现设计总是处于一种变迁的状态。某些时候内部设计的实现会影响周边的类，反之亦然。

对设计的更改是平常事。最外层的接口保持相当的稳定性，但是细节则经常改变。设计的更改通常不是大范围的改动，更像是将一小段代码从一个地方挪到另一个地方，不断地琢磨和改进设计。

进行初始设计也可以是非常有价值的。只是不要在细节上投入太多的时间。良好设计的重要组成部分是接口，它们指出了你的代码必须在什么地方同系统中的其它代码打交道。除这部分之外，中间的部分是由你控制的。通过以实现指导设计的方法，你可以雕琢出一个更优的系统。

练习

1. 创建一个测试将本练习的文本写入到文件系统中。该测试应该把文件的内容读取回来，并判断内容是否一致。确保你可以多次运行这个测试并使其通过。最后，确保在测试结束时，即使有异常抛出，也没有残留文件。
2. （较难）创建一个计时的测试，来证实使用缓冲类对性能是非常重要的。该测试可以遍历大小不定的文件，以 10 倍为增长因子创建字符数据，调用最基本的、每次一个字符的方法；然后把 `writer` 包装到一个使用缓冲的输出流中，并再次写入它，直至达到 5 倍以上的性能提升。看看使用缓冲的 `writer` 在达到什么阈值时会取得显著的性能提升。
3. 创建并测试实用类 `MyFile`。这个类需要包装一个 `File` 对象，接受一个字符串文件名作为构造函数的参数。它需要实现若干成员方法，将文件的内容读出一个字符串、或者行的列表。读和写操作应该封装打开和关闭文件。客户程序不必自己关闭文件。

确保如果文件不存在，读方法会失败并抛出一个特定的 `unchecked` 异常。类似的，如果文件不存在写方法也会失败。再辅以 `delete` 和 `overwrite` 方法，这样你将得到一个经过充分测试的实用类，可以把它放在你的工具箱中。

4. 深入练习 3 中的冒险：创建一个 `Dir` 类，进而封装一个代表实际文件系统中目录的 `File` 对象。将这个类设计为：只有当它映射到一个已存在的目录时，才有作用。提供一个名为 `ensureExists` 的方法，在目录不存在时创建它。如果目录名和已存在文件名相同时，构造函数应该抛出一个异常。最后，通过一个方法返回该目录内对应的 `MyFile` 对象列表，并在目录未能创建时抛出异常。
5. 编写一个测试来演示使用 `ByteArrayOutputStream` 捕获异常，并将栈跟踪（stack trace）转

储 (dump) 到一个字符串中。分别以使用 `OutputStreamWriter` 和不使用 `OutputStreamWriter` 的方式来编写它。在字符版本和字节版本中, 使用带缓冲的 `reader` 和 `writer`。

6. 修改国际象棋的应用, 以允许你将棋盘位置保存到一个文本文件中, 并可以从中读取回来。提供两个选择: `Board` 类型的序列化对象, 或者在之前练习中演示的文本化表示。
7. 在“附加课III”中, 你将学习在 Java 中克隆或创建一个对象拷贝的首选技术。在学习这项技术之前, 你可以通过使用对象序列化和反序列化来实现克隆以复制对象。你的实现是一种“可怜人 (poor man's)”的克隆实现, 永远不要在你的产品系统中使用。
8. 为 `Dir` 创建一个名为 `Attributes` 的实例内联类, 用以封装两个目录属性: 目录是否为只读, 和目录是否为隐藏目录。`Dir` 类应该在被请求时返回该对象的实例。演示 (通过编译失败) 测试无法实例化该对象。 ◀ 423
9. 将 `Dir.Attributes` 内联类更改为静态内嵌类并修改代码, 并测试这样它们可以工作。这有怎样的含义呢? 演示测试可以实例化一个 `Dir.Attributes` 类。这样的设计是否有意义?
10. 在第 10 课的练习中, 你编写代码来以编程的方式判断每种基本 (primitive) 整数类型的大小。现在, 编写代码通过使用数据流来判断所有基本数据类型的大小 (base size)。 ◀ 424



反射及其他高级主题

如果说要充分学习 Java 的各项技术以应对大部分问题，本课的内容是你不容错过的。本章展示了一些高级的 Java 课题以及其他一些零碎但重要的部分，这些内容还没有应用到你的学生信息系统中。

你将学习到：

- 附加的 *mocking*（模拟）技术
- 匿名内联类
- 反射
- 适配器（*adapter*）
- 实例初始化器（*initializer*）
- *Class* 类
- 动态代理机制

再顾 Mock 对象

在第 10 课中，你曾创建了一个 *mock* 类来强制随机数生成器的行为。你所建立的 *mock* 类派生自 *java.util.Random*。你当时是以内嵌类的方式在测试中建立这个 *mock* 类的。因为这个 *mock* 类只被该测试使用，直接将 *mock* 代码加入到测试类中会更有益。在本课中，你将学习一种更简练的技术：将 *mock* 类完全嵌入到一个测试方法中。

如果你的代码必须同外部 API 打交道，你可能需要使用 *mock* 对象。基本上，你对外部代码或者外部 API 返回的结果没有什么控制权。为这样的代码编写测试，经常出现的局面是，API

返回结果的不断改变，使测试受到破坏。而且，该 API 可能会同某些外部资源通讯，而这些外部资源并不总是可获得的。这样的 API 引入了有害的依赖性，而 `mock` 可以帮助你对其进行管理。¹



在引入 `mock` 之前，确认你有依赖性的问题。



学生的 `Account` 类需要能够处理从相关联的银行账号中转账。如此，它必须要与来自 Jim Bob 的 ACH (automated clearing house, 自动房屋清扫) 公司的 ACH 软件打交道。ACH 软件公布有 API 规范。另一个糟糕的依赖是：我们还没有得到授权，或者没有安装实际的软件！但是你现在就需要开始编写代码了，以便于我们能够在实际代码得到授权和交付之后，尽快地发布可运转的软件。

测试本身非常小。

```
package sis.studentinfo;

import java.math.BigDecimal;
import junit.framework.*;

public class AccountTest extends TestCase {
    static final String ABA = "102000012";
    static final String ACCOUNT_NUMBER = "194431518811";

    private Account account;

    protected void setUp() {
        account = new Account();
        account.setBankAba(ABA);
        account.setBankAccountNumber(ACCOUNT_NUMBER);
        account.setBankAccountType(Account.BankAccountType.CHECKING);
    }
    // ...
    public void testTransferFromBank() {
        account.setAch(new com.jimbob.ach.JimBobAch()); // uh-oh

        final BigDecimal amount = new BigDecimal("50.00");
        account.transferFromBank(amount);

        assertEquals(amount, account.getBalance());
    }
}
```

426



从账号中透支需要银行的 ABA (American Banking Association, 美国银行协会) 路由号 (routing number)、账号，以及账号的类型。你可以使用 `setUp` 方法来填充所需的账号信息。

测试方法 `testTransferFromBank`，创建了一个新的 `JimBobAch` 对象，并将它传递给 `Account` 对象。哎呀：你还没有 `JimBobAch` 对象，因此代码甚至都无法进行编译。你将马上通

¹ [Langr2003].

过 mock 类来解决这个问题。

代码的剩余部分，调用账号的 `TransferFromBank` 方法，然后判定账号的结余增至正确的金额。

那么如何解决 `JimBobAch` 类不存在这一问题呢？你的确已经万事具备只欠东风了。Jim Bob ACH 公司已经将 API 文档提供给你了。

Jim Bob ACH 接口

API 文档已经以 PDF 的格式提供给你了，包括 `JimBobAch` 类所实现的接口定义。它还包括了有关数据类的定义。你可以拿到 Jim Bob ACH 接口的代码，并暂时把它拷贝到你的系统中。把代码加入到你的源代码目录的 `./com/jimbob/ach` 目录结构中。（你可能可以直接从 PDF 文档拷贝粘贴出来。）

当你从 Jim Bob 和公司拿到实际的 API 类库时，在删除你的临时拷贝之前，确保实际的 API 类库同你持有的是匹配的。

```
// com.jimbob.ach.Ach
package com.jimbob.ach;

public interface Ach {
    public AchResponse issueDebit(AchCredentials credentials, AchTransactionData
        data);
    public AchResponse markTransactionAsNSF(AchCredentials credentials,
        AchTransactionData data, String traceCode);
    public AchResponse refundTransaction(AchCredentials credentials,
        AchTransactionData data, String traceCode);
    public AchResponse issueCredit(AchCredentials credentials, AchTransactionData
        data);
    public AchResponse voidSameDayTransaction(AchCredentials credentials,
        AchTransactionData data, String traceCode);
    public AchResponse queryTransactionStatus(AchCredentials credentials,
        AchTransactionData data, String traceCode);
}

// com.jimbob.ach.AchCredentials
package com.jimbob.ach;

public class AchCredentials {
    public String merchantId;
    public String userName;
    public String password;
}

// com.jimbob.ach.AchTransactionData
```

```

package com.jimbob.ach;

import java.math.BigDecimal;

public class AchTransactionData {
    public String description;
    public BigDecimal amount;
    public String aba;
    public String account;
    public String accountType;
}

// com.jimbob.ach.AchResponse
package com.jimbob.ach;

import java.util.*;

public class AchResponse {
    public Date timestamp;
    public String traceCode;
    public AchStatus status;
    public List<String> errorMessages;
}

// com.jimbob.ach.AchStatus
package com.jimbob.ach;

public enum AchStatus {
    SUCCESS, FAILURE;
}

```

现在，你只会对 Ach 接口中的 issueDebit 方法感兴趣。你必须调用 JimBobAch 对象的 issueDebit 方法，来从银行账号进行划账。这个方法以一个 AchCredentialsObject 对象和一个 AchTransactionData 对象作为参数。注意，直接在这两个类中曝露实例变量的作法是值得推敲的。不幸的是，当你同第三方供应商的软件打交道时，就只有认命了。

issueDebit 方法根据完成的状况返回一个 AchResponse 对象。这个响应数据通过使用 AchStatus 枚举值来指示透支是否成功了。

428

Mock 类

你并没有实际的 JimBobAch 类，但是你可以创建一个 mock 类来像 JimBobAch 类那般实现同样的 Ach 接口。通过这个 mock 对象，你的测试可以正常工作。

下面的 MockAch 类实现了 Ach 接口。目前 ACH 服务中唯一让你关注的是 issueDebit 方法；你可以将其它方法实现为存根(stub)，并返回 null 来满足编译器的需要。编写 issueDebit 方法，确保所传入的参数同预期是一样的。如果参数是有效的²，你构造一个 AchResponse 对象，

² 你还应该确保 AchCredentials 对象同预期是一样的；由于篇幅的考虑，我省略了这个检查。

并填入测试预期所见的数据。

```
package sis.studentinfo;

import java.util.*;
import com.jimbob.ach.*;
import junit.framework.Assert;

class MockAch implements Ach {
    public AchResponse issueDebit(
        AchCredentials credentials, AchTransactionData data) {
        Assert.assertTrue(
            data.account.equals(AccountTest.ACCOUNT_NUMBER));
        Assert.assertTrue(data.aba.equals(AccountTest.ABA));

        AchResponse response = new AchResponse();
        response.timestamp = new Date();
        response.traceCode = "1";
        response.status = AchStatus.SUCCESS;
        return response;
    }

    public AchResponse markTransactionAsNSF(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }

    public AchResponse refundTransaction(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }

    public AchResponse issueCredit(AchCredentials credentials,
        AchTransactionData data) {
        return null;
    }

    public AchResponse voidSameDayTransaction(
        AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }

    public AchResponse queryTransactionStatus(AchCredentials credentials,
        AchTransactionData data, String traceCode) {
        return null;
    }
}
```

注意这个方案创建了一个双向的依赖，如图 12.1 所示。AccountTest 类创建 MockAch，意味着测试类依赖于 MockAch。而由于 MockAch 引用了 AccountTest 中的 ACCOUNTNUMBER 和 ABA 类常量，MockAch 也依赖于 AccountTest。双向依赖通常不是好现象，但是在某些情况下，这样紧密的耦合也是可以接受的。测试需要 mock，而 mock 只在与测试协作时才是有用的。

现在回到 testTransferFromBank 方法。在其中，你原本编写代码来初始化不存在的 JimBobAch 类。而现在，你可以创建一个 MockAch 实例，并将它传递给 Account 对象：


```

public void testTransferFromBank() {
    // account.setAch(new com.jimbob.ach.JimBobAch());
    account.setAch(new MockAch());

    final BigDecimal amount = new BigDecimal("50.00");
    account.transferFromBank(amount);

    assertEquals(amount, account.getBalance());
}

```

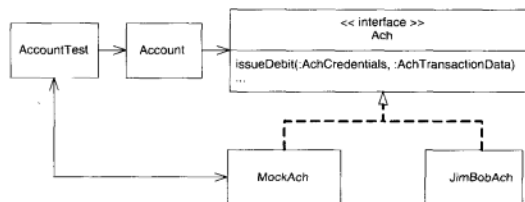


图 12.1 模拟 Ach 接口

Account 类的实现

下面是经过更新的 Account 类实现：

```

package sis.studentinfo;

import java.math.BigDecimal;
import com.jimbob.ach.*;

public class Account {
    private BigDecimal balance = new BigDecimal("0.00");
    private int transactionCount = 0;
    private String bankAba;
    private String bankAccountNumber;
    private BankAccountType bankAccountType;
    private Ach ach;

    public enum BankAccountType {
        CHECKING("ck"), SAVINGS("sv");
        private String value;
        private BankAccountType(String value) {
            this.value = value;
        }
        @Override
        public String toString() {
            return value;
        }
    }
}

```

```

public void credit(BigDecimal amount) {
    balance = balance.add(amount);
    transactionCount++;
}

public BigDecimal getBalance() {
    return balance;
}

public BigDecimal transactionAverage() {
    return balance.divide(
        new BigDecimal(transactionCount), BigDecimal.ROUND_HALF_UP);
}

public void setBankAba(String bankAba) {
    this.bankAba = bankAba;
}

public void setBankAccountNumber(String bankAccountNumber) {
    this.bankAccountNumber = bankAccountNumber;
}

public void setBankAccountType(
    Account.BankAccountType bankAccountType) {
    this.bankAccountType = bankAccountType;
}

public void transferFromBank(BigDecimal amount) {
    AchCredentials credentials = createCredentials();

    AchTransactionData data = createData(amount);

    Ach ach = getAch();
    AchResponse achResponse = ach.issueDebit(credentials, data);

    credit(amount);
}

private AchCredentials createCredentials() {
    AchCredentials credentials = new AchCredentials();
    credentials.merchantId = "12355";
    credentials.userName = "sismerc1920";
    credentials.password = "pitselah411";
    return credentials;
}

private AchTransactionData createData(BigDecimal amount) {
    AchTransactionData data = new AchTransactionData();
    data.description = "transfer from bank";
    data.amount = amount;
    data.aba = bankAba;
    data.account = bankAccountNumber;
    data.accountType = bankAccountType.toString();
    return data;
}

private Ach getAch() {
    return ach;
}

```

```

void setAch(Ach ach) {
    this.ach = ach;
}
}

```

Account 允许客户端将一个 Ach 引用传给它，它会将该引用保存起来。在产品化系统中，构造 Account 对象的客户端代码，会把一个 JimBobAch 对象传递给 Account。而在测试中，你的代码将一个 MockAch 对象传递给 Account。无论哪种情况，Account 都不知道也不用关心它所调用 issueDebit 方法的对象具体是哪种类型。

现在 testTransferFromBank 运行通过了，还无法看到准确的原因。Mock 类在另一个源文件中，因此要推断出发生了什么，你必须辗转于这两个类的定义。这并不是完全不可接受的，但是你可以基于某些技术对其进行改进。一个方案是，将 mock 类定义作为内嵌类包含进来。另一个方案是，使用匿名内联类直接在测试方法内设置 mock 对象。

432

匿名内联类

第11课讨论了静态内嵌类和内联类的区别。第三种类型的内嵌类是匿名内联类。匿名内联类允许你在一个方法体内动态地定义没有命名的类实现。

理解匿名内联类的语法和细微差异，是令人生畏的。但是一旦你理解了匿名内联类，利用它你可以使代码更简练并更容易理解。

将 MockAch 类的代码体（也就是所有的方法定义）插入到 testTransferFromBank 的第一行代码。径直在所有代码之前，加上这一行：

```
Ach mockAch = new Ach() {
```

然后，使用一个右大括号和分号来结束 Ach 模拟方法的定义。转换之后的测试：³

```

public void testTransferFromBank() {

    Ach mockAch = new Ach() {
        public AchResponse issueDebit(
            AchCredentials credentials, AchTransactionData data) {
            Assert.assertTrue(
                data.account.equals(AccountTest.ACCOUNT_NUMBER));
            Assert.assertTrue(data.aba.equals(AccountTest.ABA));

            AchResponse response = new AchResponse();
            response.timestamp = new Date();
            response.traceCode = "1";
            response.status = AchStatus.SUCCESS;
            return response;
        }
    };
}

```

³ 你需要引入 com.jimbob.ach 和 java.util 来使之编译通过。

```

    }
    public AchResponse markTransactionAsNSF(
        AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse refundTransaction(AchCredentials credentials,
        AchTransactionData data,

```

433

实例初始化器 (Instance Initializer)

在第 4 课中，你学习了静态初始化代码块。Java 还允许你使用实例初始化代码块，通常被称为实例初始化器。

在匿名内联类中，你不能创建构造函数。因为匿名内联类没有名称，你无法为它的构造函数提供名称！不过，你可以使用实例初始化段。

```

Expirable t = new Expirable() {
    private long then;
    {
        long now = System.currentTimeMillis();
        then = now + 86400000;
    }

    public boolean isExpired(Date date) {
        return date.getTime() > then;
    }
};

```

实例初始化器在顶级 (top-level) 类中很少使用。可以选择构造函数 (constructor) 或成员变量级的初始化。当然，如果你有多个构造函数必须执行公用的初始化代码，你可以使用实例初始化器来消除重复。

```

        String traceCode) {
            return null;
        }
        public AchResponse issueCredit(AchCredentials credentials,
            AchTransactionData data) {
            return null;
        }
        public AchResponse voidSameDayTransaction(
            AchCredentials credentials,
            AchTransactionData data,
            String traceCode) {
            return null;
        }
        public AchResponse queryTransactionStatus (
            AchCredentials credentials,
            AchTransactionData data, String traceCode) {
            return null;
        }
    };

    account.setAch(mockAch);

    final BigDecimal amount = new BigDecimal("50.00");

```

```

        account.transferFromBank(amount);

        assertEquals(amount, account.getBalance());
    }
}

```

434

下面的这行代码：

```
Ach mockAch = new Ach() {
```

创建了 `Ach` 接口类型的名为 `mockAch` 的引用。右侧的代码使用 `new` 初始化了一个 `Ach` 对象。但是现在 `Ach` 是接口，你怎能初始化接口呢？

Java 允许你动态地提供 `Ach` 接口的实现，同时使用这个实现创建一个实例。你在构造函数的调用（`new Arch()`）之后、且在结束语句的分号之前编写的一段代码，提供接口的实现。

这段代码中的实现提供的是 `Ach` 类型，但是没有具体的类名。它是匿名的。你可以调用这个对象的方法，保存对象，或者将它作为参数传递，就如同你对命名的（named）`Ach` 接口实现对象（例如 `MockAch`）一样，对它进行操作。

确认你的测试可以通过，但是现在先不要删除 `MockAch`。

如你所见的，在为测试目的建立 `mock` 时，匿名内联类是很有用处的。`Swing` 应用程序（参见附加课程 I）以及多线程应用（参见第 13 课）经常大量使用匿名内联类。

诚然，按照 `testTransferFromBank` 中的方式编写代码还是有些困难。测试方法现在太长且太乱。下一节介绍的适配器将演示如何对其进行改进。

适配器（Adapter）

过度使用匿名内联类最终会使得代码非常难于理解。你的目标应该是使方法尽量简短，即使是测试方法也应如此。在 `testTransferFromBank` 中冗长的匿名内联类实现，意味着你可能需要滚屏才可以了解该方法实作的全貌。

你可以创建接口适配器（`adaper`）类，来提供 `Ach` 接口空的实现。修改 `MockAch` 类来为 `issueDebit` 方法提供空的定义。

```

package sis.studentinfo;

import java.util.*;
import com.jimbob.ach.*;
import junit.framework.Assert;

class MockAch implements Ach {
    public AchResponse issueDebit(
        AchCredentials credentials, AchTransactionData data) {
        return null;
    }
    public AchResponse markTransactionAsNSF(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {

```

435

```

        return null;
    }
    public AchResponse refundTransaction(AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse issueCredit(AchCredentials credentials,
        AchTransactionData data) {
        return null;
    }
    public AchResponse voidSameDayTransaction(
        AchCredentials credentials,
        AchTransactionData data,
        String traceCode) {
        return null;
    }
    public AchResponse queryTransactionStatus(AchCredentials credentials,
        AchTransactionData data, String traceCode) {
        return null;
    }
}

```

回到 `AccountTest` 中，将初始化代码从 `new Ach()` 替换为 `new MockAch()`。除了 `issueDebit`，删除所有的方法实现。

```

public void testTransferFromBank() {
    Ach mockAch = new MockAch() {
        public AchResponse issueDebit(
            AchCredentials credentials, AchTransactionData data) {
            Assert.assertTrue(
                data.account.equals(AccountTest.ACCOUNT_NUMBER));
            Assert.assertTrue(data.aba.equals(AccountTest.ABA));

            AchResponse response = new AchResponse();
            response.timestamp = now Date();
            response.traceCode = "1";
            response.status = AchStatus.SUCCESS;
            return response;
        }
    };

    account.setAch(mockAch);
    final BigDecimal amount = new BigDecimal("50.00");
    account.transferFromBank(amount);

    assertEquals(amount, account.getBalance());
}

```

你还是创建了匿名内联类。匿名内联类可以实现接口，或者继承另一个类。上面的代码演示了后者。这里，你使用 `MockAch` 类作为适配器，来隐藏 `mock` 不需要关心的方法。现在，你可以把有关的 `mock` 代码包含在一个相当简短的测试方法中。

访问外围类中的变量



你需要一个额外的测试，`testFailedTransferFromBank`，来定义当银行拒绝透支请求时应该发生什么。这个测试和 `mock`，会和 `testTransferFromBank` 非常相似。当转账失败时，学生账号中的结余应该不变，因此测试的判定必须改变。从 `AchResponseData` 对象的 `mock` 中所返回的状态，必然是 `AchStatus.FAILURE`。

因为这次要实现 `mock` 对象，同之前的几乎是一样的，你希望使用一种方法在不引入重复代码的情况下同时创建它们两个。你希望能够用下面的代码表述新的测试：⁴

```
public void testFailedTransferFromBank() {
    account.setAch(createMockAch(AchStatus.FAILURE));
    final BigDecimal amount = new BigDecimal("50.00");
    account.transferFromBank(amount);
    assertEquals(new BigDecimal("0.00"), account.getBalance());
}
```

现有的测试，`testTransferFromBank`，可以使用相同的结构。

```
public void testTransferFromBank() {
    account.setAch(createMockAch(AchStatus.SUCCESS));
    final BigDecimal amount = new BigDecimal("50.00");
    account.transferFromBank(amount);
    assertEquals(amount, account.getBalance());
}
```

匿名内联类对象同样是对象。它们可以像其他对象那样，在一个单独的方法中被创建并返回。

437

```
// 无法编译通过!
private Ach createMockAch(AchStatus status) {
    return new MockAch() {
        public AchResponse issueDebit(
            AchCredentials credentials, AchTransactionData data) {
            Assert.assertTrue(
                data.account.equals(AccountTest.ACCOUNT_NUMBER));
            Assert.assertTrue(data.aba.equals(AccountTest.ABA));

            AchResponse response = new AchResponse();
            response.timestamp = new Date();
            response.traceCode = "1";
            response.status = status;
            return response;
        }
    };
}
```

⁴ 你还希望如果 ACH 透支失败，Account 代码应该抛出异常。这里为简单起见，我取消了这个异常。

匿名内联类的类名

当 Java 编译器编译包含匿名内联类的类时，它使用同非匿名内嵌类的命名规则相近的规则，来命名这些编译单元。Java 使用美元符 (\$) 来分隔类名。

一个小问题：匿名内联类没有名字！Java 提供了一个简单的解决方案，使用从 1 开始的计数器来对匿名内联类进行编号。然后它使用这个编号作为内联类名中的一部分，来命名编译单元。

例如，AccountTest 类会创建两个编译单元：AccountTest.class 和 AccountTest\$1.class。

我将再次重复我在上一章中的警告：当你把类文件拷贝到一个 JAR 文件中进行分发时，不要忘了包括所有的内嵌类。

不过，这段代码还是无法编译。你会看到下面的错误消息：

```
local variable status is accessed from within inner class; needs to be declared final (局部变量 status 是由一个内联类访问的；需要将它声明为 final)
```

匿名内联类是内联类。通过你在第 11 课中学习的定义，一个内联类可以访问外围类的实例变量和方法。

但匿名内联类不能访问局部变量。status 参数相当于一个局部变量，因为它只为 createMockAch 方法中的代码所访问。

不过，根据这个编译错误，你可以通过把 status 参数声明为 final 来克服这个限制。

```
private Ach createMockAch(final AchStatus status)
```

在你如此声明之后，你的代码应该可以编译通过了，但是你的测试还是无法通过。

◀ 438

为什么你必须要将变量声明为 final 呢？回忆一下如果将变量声明为 final，那就意味着当它的值被设置之后，就无法再更改了。

匿名内联类的实例可以存活在它所声明方法的范围之外。在我们的示例中，createMockAch 方法实例化了一个新的匿名内联类，并把它从这个方法返回，然后被调用 createMockAch 方法的测试方法所用。

一旦方法结束执行，局部变量就不复存在了！但是参数并非如此——方法内的所有参数都是调用代码传入变量的拷贝⁵。如果匿名内联类可以访问局部变量，那么局部变量的值在匿名内联类中的代码开始实际执行时，已经被摧毁了。这当然是一件糟糕的事情。

为了让你的测试通过，在 Account 中插入代码，这样你只认可成功的转账。

```
public void transferFromBank(BigDecimal amount) {
    AchCredentials credentials = createCredentials();
    AchTransactionData data = createData(amount);
    Ach ach = getAch();
```

⁵ 进一步的详述，参见附加课 III，其中讨论了按值调用。


```
AchResponse achResponse = ach.issueDebit(credentials, data);
if (achResponse.status == AchStatus.SUCCESS)
    credit(amount);
}
```

稍稍重构，这个方法可以简化为：

```
public void transferFromBank(BigDecimal amount) {
    AchResponse achResponse =
        getAch().issueDebit(createCredentials(), createData(amount));
    if (achResponse.status == AchStatus.SUCCESS)
        credit(amount);
}
```

折衷

早前，我提到过，能够在测试本身就看到 mock 的行为，是一件很好的事情。然后，我让你把 mock 的定义提取到单独的方法中，这样当为另一个测试方法实现相似的 mock 时，可以消除重复。这对于可表达性来说是一个损耗，尽管它适当地清理了下面代码的行为。

439

```
account.setAch(createMockAch(AchStatus.SUCCESS));
```

哪一个方案更好一些呢？重复或者使代码难于理解？回答并不总是那么清晰的。这个问题常常造成争论。以我的观点来看，重复代码通常比可读性差的代码更棘手。你并不总是需要在两者中优先选择一个，但是当你选择时，消除重复是更安全的路线。通过消除重复代码，你可以减少维护工作量的成本——你可以避免在多个地方对代码进行同一更改。你还可以把改了一处忘了另一处的风险最小化。

反射 (Reflection)

你已经看到如何通过使用 Object 的 getClass 方法，让你能够在运行时判断一个对象的类型。这被称为代码检验自身信息的反射能力；换句话说，就是自我反射。另一个用于反射能力的术语是元数据 (metadata)。

JUnit 大量使用了反射来获得它所需的信息，以运行你的测试用例。你提供给 JUnit 一个要测试的测试类文件。JUnit 使用反射来判断你的类文件是否从 junit.framework.TestCase 中派生而来，如果不是的话则拒绝执行它。JUnit 还使用反射来获取一个测试类中方法的列表。然后它遍历这些方法，只执行那些满足测试方法准则的方法：

- 方法名必须以小写字母的“test”开头
- 方法的返回值类型必须是 void

- 方法不得接收任何参数

目前为止，你已经告诉了 JUnit 希望对哪些类进行测试。这让你能够只运行指定的测试项。这种方法的一个问题是，很容易忘记将你的测试类加入到一个测试套件（suite）中。你极可能会错过关键的测试！

另一个方法是让 Java 扫描你的 classpath 中的所有类，收集那些合法的测试类，然后逐一执行。这样做的好处是，你不需要在代码中维护测试套件。不会有测试被遗漏。唯一不好的地方是，其中的某些测试（例如性能测试），你并不希望每次都运行它们。当然，如果必要，你可以编写额外的代码来跳过这样的测试。

◀ 440

使用 JUnit 代码

幸运的是，JUnit 提供了一种机制，帮助你收集在 classpath 中的测试类。你如何能知道这一点呢？首先，你大概已经运行过 JUnit，并发现它具有收集所有测试类列表的能力。除此之外，你可能已经仔细阅读了随 JUnit 一同发布的代码，并已经看到它有名为 `junit.runner.ClassPathTestCollector` 的类。

你将创建名为 `SuiteBuilder` 的类来收集所有的测试类。对 `SuiteBuilder` 的第一个简单测试将会是，确保 classpath 中找到的测试用例列表中包含 `SuiteBuilderTest` 本身。

```
package sis.testing;

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

public class SuiteBuilderTest extends TestCase {
    public void testGatherTestClassNames() {
        SuiteBuilder builder = new SuiteBuilder();
        List<String> classes = builder.gatherTestClassNames();
        assertTrue(classes.contains("sis.testing.SuiteBuilderTest"));
    }
}
```

这个测试中令人感兴趣的部分是，它将列表绑定为 `String` 类型而不是 `Class` 类型。当你查看 `ClassPathTestCollector` 中的代码时，你会发现它返回 `String` 的 `Enumeration`，其中每个字符串均表示一个类名。我们现在先沿用这种方法，然后看看它之后是否会导致某些问题。

```
package sis.testing;

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

public class SuiteBuilder {
    public List<String> gatherTestClassNames() {
        TestCollector collector = new ClassPathTestCollector();
        public boolean isTestClass(String className) {
```

```

        return super.isTestClass(classFileName);
    }
};
return Collections.list(collector.collectTests());
}
}

```

`ClassPathTestCollector` 被声明为 `abstract`，因此你必须委派生它以对其实例化。有趣的是，类中没有任何抽象的方法，意味着你并不需要覆写（`override`）`ClassPathTestCollector` 中的任何方法。你需要覆写方法 `isTestClass`：

```

protected boolean isTestClass(String classFileName) {
    return
        classFileName.endsWith(".class") &&
        classFileName.indexOf('$') < 0 &&
        classFileName.indexOf("Test") > 0;
}

```

查询方法 `isTestClass` 关心只是类文件的名称，而不是类本身的结构。条件语句表示的是，如果一个文件名以 `.class` 扩展名结尾、不包含 `$`（也就是说它不表示内嵌类），并且包括“`Test`”字样，那么它就是一个测试类。

你大概已经意识到这样的定义会导致问题：如果 `.class` 文件不是有效的类文件会怎样？如果类文件名中包含“`Test`”字样，但不是一个测试类会怎样？最后一种情况是很有可能出现的。你自己的学生信息系统代码中，尚且包括有 `util.TestUtil` 类，但它并不是 `TestCase` 的子类。

你现在不必担心这些问题，因为你的测试还没有涉及到这类情况。眼下，你在 `getTestClassName` 中实现的代码，覆写了 `isTestClass`，并简单地调用父类方法来使用其现有的行为。这实际上没什么必要，但是当你已经准备好处理其它关注（`concern`）时，它可以作为一个占位符（`placeholder`）和提醒（`reminder`）。

`gatherTestClassNames` 的最后一行调用 `ClassPathTestCollector` 的 `collectTests` 方法。在调用的过程中，`collectTests` 间接地调用了 `isTestClass` 方法。`collectTests` 的返回类型是 `java.util.Enumeration`。你可以使用 `java.util.Collections` 的实用方法 `list` 来将 `Enumeration` 转换成一个 `ArrayList`。你会收到一个 `unchecked` 的警告，因为 `ClassPathTestCollector` 使用的是原始（`raw`）集合，而没有绑定到一个特定的类型。

Class 类

让我们着手应对亟待解决的重点：类名中带有“`Test`”字样的类并不是 `junit.framework.TestCase` 的子类。要测试这种情况，可以考虑指定一个类，让 `SuiteBuilder` 尝试收集然后拒绝它。你可以考虑使用一个已经为学生信息系统编写的现有类，例如 `TestUtil`。

然而，你并不希望对现有的类产生存在性和稳定性方面的依赖。有些人可能会对 `TestUtil` 进行有效的更改，甚至删除它，这会无意间破坏 `SuiteBuilderTest` 的测试。相反地，你可以创建

空的 (dummy) 测试类, 特别提供给 `SuiteBuilderTest` 使用。首先在新的包 `sis.testing.testclasses` 中创建类 `NotATestClass`。

```
package sis.testing.testclasses;
public class NotATestClass {}
```

`NotATestClass` 没有继承 `junit.framework.TestCase`, 因此它不应被认为是一个测试类。向你的测试方法添加一个断言。

```
public void testGatherTestClassNames() {
    SuiteBuilder builder = new SuiteBuilder();
    List<String> classes = builder.gatherTestClassNames();
    assertTrue(classes.contains("testing.SuiteBuilderTest"));
    assertFalse(classes.contains("testing.testclasses.NotATestClass"));
}
```

`SuiteBuilder` 中的相关代码:⁶

```
public List<String> gatherTestClassNames() {
    TestCollector collector = new ClassPathTestCollector() {
        public boolean isTestClass(String className) {
            if (!super.isTestClass(className))
                return false;
            String className = classNameFromFile(className); // 1
            Class klass = createClass(className); // 2
            return TestCase.class.isAssignableFrom(klass); // 3
        }
    };
    return Collections.list(collector.collectTests());
}

private Class createClass(String name) {
    try {
        return Class.forName(name);
    }
    catch (ClassNotFoundException e) {
        return null;
    }
}
```

在 `SuiteBuilder` 中实现的附加限制是, 类文件名必须表示一个从 `TestCase` 派生的编译单元。◀ 443
其中涉及三步。下面的描述对应于 `gatherTestClassNames` 代码中的注释行。

1. 将基于目录的文件名转换为类名, 例如, 将 “testing/testclasses/NotATestClass” 转换为 “testing.testclass.NotATestClass”。`ClassPathTestCollector` 中的 `classNameFromFile` 方法完成这一任务。
2. 由类名创建 `Class` 对象。在 `createClass` 中, 调用在 `Class` 中定义的静态方法 `forName` 来完成这一任务。如果你传入的 `String` 对象无法表示一个可被 Java 加载的类, `forName` 方法会抛出 `ClassNotFoundException` 异常。这样, 如果这种情况发生, 你可以从 `createClass` 返回 `null`, 但是你将需要编写一个测试来涵盖这种可能性。

⁶ 注意使用 `klass` 作为变量名来表示一个 `Class` 对象, 你不能使用 `class`, 因为它是 java 保留的关键词, 只用来定义类, 有些开发者使用一个字母 `c` (不提倡), 有些使用 `clazz`。

3. 判断这个类是否为 `TestCase` 的子类。为进行判断，你需要使用 `java.lang.Class` 类中提供类定义的元数据方法。`Class` 的 `isAssignableFrom` 方法接收一个 `Class` 对象作为参数，如果有可能将参数的实例（在上面代码中为 `klass`），赋值给传入类型（`junit.framework.TestCase`）的引用，则该方法返回 `True`。

简略地查看一下 `java.lang.Class` 的 Java API 文档。它包括诸如 `getMethods`、`getConstructors` 以及 `getFields` 等方法，你可以使用它们从已编译的 Java 类中得到大部分所需的信息。

建立测试套件

下一步是构造 `TestSuite` 对象，将之传递给 Swing 的测试运行程序（test runner）。完成这一任务的代码很简单：遍历 `gatherTestClassNames` 的返回结果，为每个测试类的类名创建相应的 `Class` 对象。测试看起来非常直接：确保套件中包含了希望的测试类。

```
public void testCreateSuite() {
    SuiteBuilder builder = new SuiteBuilder() {
        public List<String> gatherTestClassNames() {
            List<String> classNames = new ArrayList<String>();
            classNames.add("testing.SuiteBuilderTest");
            return classNames;
        }
    };

    TestSuite suite = builder.suite();
    assertEquals(1, suite.testCount());
    assertTrue(contains(suite, testing.SuiteBuilderTest.class));
}
```

444

在此我使用了 `mock`，而没有像平常那样让套件方法去收集所有的类。否则，`gatherTestClassNames` 将会返回 `classpath` 中所有类的列表。这个列表将会包括学生信息系统中其它的所有测试。你将没有确定的方式，来检验套件方法是否完成了它的工作。

判断某个类是否包含在 `TestSuite` 中，并不像你想象的那样简单。测试套件可以包括测试用例类，甚至其他的测试套件。图 12.2 展现了这种设计，也就是通常所说的 *composite 模式*⁷。实心的菱形箭头表示 `TestSuite` 和 `Test` 类之间是一种组合（composition）关系。`TestSuite` 是由 `Test` 对象组成的。

要判断类是否包含在测试套件的某个地方，你必须遍历套件的整个层次结构。这里的 `contains` 方法使用递归方式，当它在当前套件中遇到一个子套件时，`contains` 调用其自身。

```
public boolean contains(TestSuite suite, Class testClass) {
    List testClasses = Collections.list(suite.tests());
    for (Object object: testClasses) {
```

⁷ [Gamma1995].

```

    if (object.getClass() == TestSuite.class)
        if (contains((TestSuite)object, testClass))
            return true;
    if (object.getClass() == testClass)
        return true;
}
return false;
}

```

你剩下的工作是创建 `TestRunner` 类。`TestRunner` 将使用 `SuiteBuilder` 来构造一个测试套件。然后它使用该套件来执行 Swing 界面的测试运行程序。但是在连续运行 JUnit 的过程中，并没有测试来检验这部分代码。为这部分代码编写测试是有可能的，但是很困难。而且，你构造 `SuiteBuilder` 所采用的方法，已经暗含了 `TestRunner` 代码非常之简短，它几乎是牢不可破的。

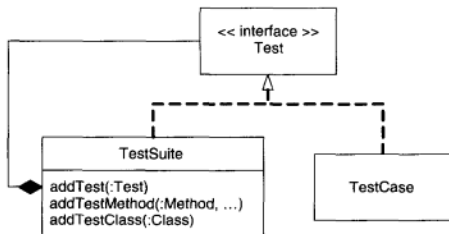


图 12.2 Junit Composite 设计

445

```

package sis.testing;

public class TestRunner {
    public static void main(String[] args) {
        new junit.swingui.TestRunner().run(TestRunner.class);
    }

    public static junit.framework.Test suite() {
        return new SuiteBuilder().suite();
    }
}

```

你需要在构建脚本中加入一个目标来执行 `testing.TestRunner`:

```

<target name="runAllTests" depends="build">
    <java classname="testing.TestRunner" fork="yes">
        <classpath refid="classpath" />
    </java>
</target>

```

不过，当你运行全部测试时，会收到半打的错误。

类修饰符

问题来自 `SessionTest`，一个抽象类，包含有许多测试方法的定义。这些方法作为 `SessionTest` 子类的一部分，也会要求被执行；不过，你之前没有将 `SessionTest` 添加到套件中。你需要修改 `SuiteBuilder` 来忽略抽象类。

创建一个由 `TestCase` 派生的抽象类。

```
package sis.testing.testclasses;

abstract public class AbstractTestClass
    extends junit.framework.TestCase {
    public void testMethod() {}
}
```

然后修改测试来确保它不会被收集。

```
public void testGatherTestClassNames() {
    SuiteBuilder builder = new SuiteBuilder();
    List<String> classes = builder.gatherTestClassNames();
    assertTrue(classes.contains("testing.SuiteBuilderTest"));
    assertFalse(classes.contains("testing.testclasses.NotATestClass"));
    assertFalse(
        classes.contains("testing.testclasses.AbstractTestClass");
);
```

运行测试，并确保它失败了。只有这样，你才应该继续改正 `SuiteBuilder` 中的代码。你将遵循这样一种测试模式：如果你发行的代码，无论在产品系统还是在验收测试（一种用户定义的测试，从最终用户的立场来检验代码）中失败了，这意味着它没有经过充分的单元测试。你的任务是添加遗漏的测试代码，并确保它的失败原因同验收测试失败是一样的⁸。



永远不要只依赖于一种层次的测试。确保你拥有一个在单元测试之上的测试层次，来检索出单元测试的漏洞。

现在，你已经看到测试失败了，可以在 `SuiteBuilder` 中添加代码来改正这一缺陷：

```
public List<String> gatherTestClassNames() {
    TestCollector collector = new ClassPathTestCollector() {
        public boolean isTestClass(String className) {
            if (!super.isTestClass(className))
                return false;
            String classFile = classNameFromFile(className);
            Class klass = createClass(className);
            return
                TestCase.class.isAssignableFrom(klass) &&
```

⁸ [Jeffries2001], 第163页。

```

        isConcrete(klass);
    }
};
return Collections.list(collector.collectTests());
}

private boolean isConcrete(Class klass) {
    if (klass.isInterface())
        return false;
    int modifiers = klass.getModifiers();
    return !Modifier.isAbstract(modifiers);
}

```

如果某个类型（Class 可以表示接口和类两种类型）为接口，它不是具体的（concrete）。你可以通过 Class 的 `isInterface` 方法来进行判断。

`isConcrete` 方法还使用 Class 的 `getModifiers` 方法。`getModifiers` 返回一个 `int`，其中嵌入了一列标志（flag），每个都对应一种可能的类修饰符⁹。你已经看到的某些描述符包括 `abstract`、`static`、`private`、`protected` 和 `public`。`java.lang.reflect.Modifier` 实用类（你^{◀ 447}需要在 `SuiteBuilder` 中添加一个适当的 `import` 语句）包含许多静态查询方法，来帮助你判断在 `int` 值中设置了哪些修饰符。

现在你可以完全删除你的 `AllTests` 类了！

动态代理

J2SE 1.3 版引入了一个新的类，`java.lang.reflect.Proxy`，允许你构造动态的代理类。动态代理类允许你在运行时动态地实现一个或多个接口。

Proxy（代理）模式在《设计模式》¹⁰一书中被认为是最有用的模式之一。代理是一种替身（stand-in）：代理对象代替了实际的对象。在 Proxy 模式的实现中，客户对象认为它们在同实际的对象交互，但实际上是同代理打交道。

图 12.3 展示了在分布式对象通讯环境下的代理模式。`ServiceImplenatation` 类位于同 Client 类不同机器的不同进程空间内。为了让 Client 同 `ServiceImplenatation` 交互，必须进行某些底层的通讯。假定 Client 必须要调用 `ServiceImplenatation` 的 `submitOrder` 方法。这个 Java 的方法调用必须被转换成数据流，以通过网络进行传送。在服务器端，数据流必须被重组为对 `ServiceImplenatation` 对象方法的调用。

你希望在 Client 中的代码，调用 `ServiceImplenatation` 的方法时，就像它在本地同一个 Java VM 进程中执行那样。无论 Client 或 `ServiceImplenatation` 中的代码都不需要关心这种远程通讯的具体细节。

⁹ 方法和成员变量也有修饰符，并使用相同的整型标志常量。

¹⁰ [Gamma1995]。

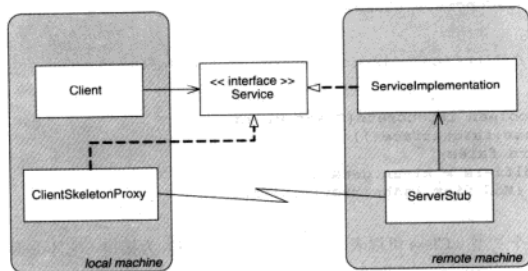


图 12.3 使用代理的分布式通讯

解决方案是，让 `ServiceImplementation` 类实现 `Service` 接口，同时这个接口也被客户端类 `ClientSkeletonProxy` 实现。`Client` 对象继而可以通过这个接口同 `ClientSkeletonProxy` 交互，想象自己正在同真正的 `ServiceImplementation` 打交道。这样，`ClientSkeletonProxy` 是真正 `ServiceImplementation` 的一个代表，或代理。`ClientSkeletonProxy` 接受传入的请求并同远程机器上的 `ServerStub` 协作，来完成底层的通讯。`ServerStub` 接收传入的数据传输并将它们转换为对 `ServerImplementation` 的调用。（你可能已经意识到了，用 Java 实现该方案，将大量依赖反射的使用。）

在 Java 中，这种代理模式是 RMI——远程方法调用（Remote Method Invocation）的基础。RMI，继而基于组件的分布式计算技术——EJB（企业级 Java Bean）的基础。¹¹

Proxy 还有许多应用，包括通常所说的延时加载（lazy load）、写时拷贝（copy on write）、池（pooling）、缓存（caching）以及事务性标记。你将使用代理来编写一个透明的安全机制。

在 Java 的代理实现中，代理需要同真实对象实现相同的接口。代理拦截了发给客户端的消息；在处理消息之后，代理通常将它委托给真实对象、或目标对象。你可以设想，如果代理类必须要实现和目标类相同的接口，维护众多的代理类将会非常繁琐。对你添加到接口中的每个方法，你必须同时提供真实实现和代理实现。幸运的是，Java 中的动态代理消除了你在代理类中显式实现每个接口方法的需要。

安全账号类



你需要能够基于客户端具有的权限，限制对某些 `Account` 方法的访问。客户端可以具有只读或更新的访问权限。具有更新访问的客户端可以使用 `Account` 中定义的所有方法。具有只读

¹¹ EJB 是 J2EE（Java 2 企业版）平台的一部分。

访问的客户端只能使用那些不会更改账号状态的方法。

◀ 449

你可以在 `Account` 类中添加代码，当创建账号时传入用户的分类（classification）。为了限制每个方法，你可以添加代码来检查用户的分类，并当用户不具有适当访问权限时抛出异常。安全相关的代码将会很快地在 `Account` 类中蔓延开来，并使其业务逻辑变得晦涩不明。你正在违反单一职责（Single-Responsibility）原则！¹²

相反，你可以将安全限制剥离到一个代理类中。`Account` 类本身几乎完全没有改变！我们在这里将使用开闭（open-closed）原则¹³，通过添加新的代码来建立新功能，而不是更改现有的代码。

对代理的使用通常会要求工厂类（factory）。图 12.4 中的 UML 图展示了安全代理的完整方案，包括使用 `AccountFactory` 类来返回 `SecureProxy` 的实例。客户端可以认为它在同真实的 `Account` 交互，但它实际上是在同一个代理（能够对同样的调用进行响应）打交道。

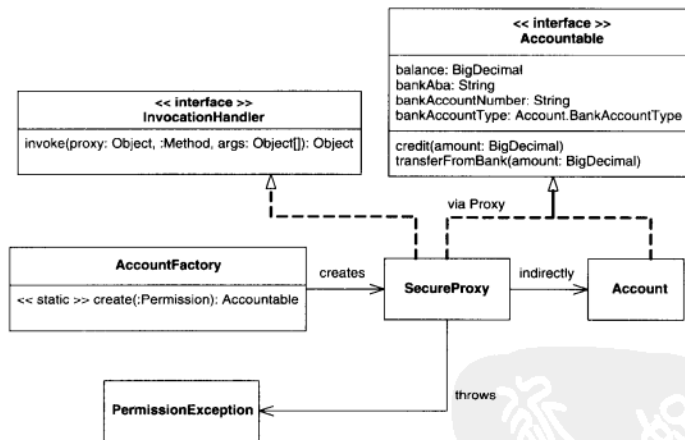


图 12.4 安全代理

◀ 450

`AccountFactory` 类使用动态代理类 `Proxy` 来创建 `SecureProxy` 对象。`SecureProxy` 并不直接实现 `Accountable`；相反地，`Proxy` 类设置 `SecureProxy` 来捕获所有传入的调用，并将每一个调用重定向给 `InvocationHandler` 接口方法 `invoke`。

¹² [Martin2003]，第 95 页。

¹³ [Martin2003]，第 99 页。

建立安全账号方案

起点是 `AccountFactoryTest`，一个检查各种权限和方法组合的测试类。该测试类包括两个测试方法：`testReadOnlyAccess` 确保所有的账号方法都可以被具有更新权限的用户调用，以及 `testUpdateAccess` 确保当具有只读访问的用户试图执行一个安全方法时会抛出异常。

```
package sis.studentinfo;

import java.math.*;
import java.util.*;
import java.lang.reflect.*;
import junit.framework.*;

import sis.security.*;

public class AccountFactoryTest extends TestCase {
    private List<Method> updateMethods;
    private List<Method> readOnlyMethods;

    protected void setUp() throws Exception {
        updateMethods = new ArrayList<Method>();
        addUpdateMethod("setBankAba", String.class);
        addUpdateMethod("setBankAccountNumber", String.class);
        addUpdateMethod("setBankAccountType",
            Account.BankAccountType.class);
        addUpdateMethod("transferFromBank", BigDecimal.class);
        addUpdateMethod("credit", BigDecimal.class);

        readOnlyMethods = new ArrayList<Method>();
        addReadOnlyMethod("getBalance");
        addReadOnlyMethod("transactionAverage");
    }

    private void addUpdateMethod(String name, Class parmClass)
        throws Exception {
        updateMethods.add(
            Accountable.class.getDeclaredMethod(name, parmClass));
    }

    private void addReadOnlyMethod(String name) throws Exception {
        Class[] noParms = new Class[] { };
        readOnlyMethods.add(
            Accountable.class.getDeclaredMethod(name, noParms));
    }

    public void testUpdateAccess() throws Exception {
        Accountable account = AccountFactory.create(Permission.UPDATE);
        for (Method method: readOnlyMethods)
            verifyNoException(method, account);
        for (Method method: updateMethods)
            verifyNoException(method, account);
    }

    public void testReadOnlyAccess() throws Exception {
        Accountable account = AccountFactory.create(Permission.READ_ONLY);
```

```

        for (Method method: updateMethods)
            verifyException(PermissionException.class, method, account);
        for (Method method: readOnlyMethods)
            verifyNoException(method, account);
    }

    private void verifyException(
        Class exceptionType, Method method, Object object)
        throws Exception {
        try {
            method.invoke(object, nullParamsFor(method));
            fail("expected exception");
        }
        catch (InvocationTargetException e) {
            assertEquals("expected exception",
                exceptionType, e.getCause().getClass());
        }
    }

    private void verifyNoException(Method method, Object object)
        throws Exception {
        try {
            method.invoke(object, nullParamsFor(method));
        }
        catch (InvocationTargetException e) {
            assertFalse(
                "unexpected permission exception",
                PermissionException.class == e.getCause().getClass());
        }
    }

    private Object[] nullParamsFor(Method method) {
        return new Object[method.getParameterTypes().length];
    }
}

```

这个测试演示了 Java 反射 API 中的一些附加功能。

452

在该测试中，你创建了两个集合，一个为只读方法，另一个为更新方法。在 `setUp` 方法中你将 `java.lang.reflect.Method` 对象填充到只读和更新集合中。`Method` 对象代表在 Java 类中定义的方法，并含有包括方法名、参数、返回类型以及修饰符（例如 `static` 和 `final`）等信息。最重要的是，一旦你拥有了 `Method` 对象以及定义了该方法的类的对象，你可以动态地执行这个方法。

通过调用 `Class` 对象的几种方法，你可以得到 `Method` 对象。一种方法是通过调用 `Class` 的 `getDeclaredMethods` 方法得到所有方法的一个数组。`Class` 会返回所有在该类中直接定义的方法。你还可以尝试通过调用 `Class` 的 `getDeclaredMethod` 方法来取得具体的方法，传入方法名和参数类型的列表。在上面的代码中，下面这一行：

```
Accountable.class.getDeclaredMethod(name, parmClass)
```

演示了如何使用 `getDeclaredMethod` 方法。

`testReadOnlyAccess` 测试首先使用 `AccountFactory` 来创建对象——实现 `Accountable` 接口的代理。`create` 方法的参数是 `Permission` 枚举：

```
package sis.security;

public enum Permission {
    UPDATE, READ_ONLY
}
```

通过从 `Account` 类中抽取所有公共（`public`）方法的原型，得到 `Accountable` 接口。

```
package sis.studentinfo;
import java.math.*;

public interface Accountable {
    public void credit(BigDecimal amount);
    public BigDecimal getBalance();
    public BigDecimal transactionAverage();
    public void setBankAba(String bankAba);
    public void setBankAccountNumber(String bankAccountNumber);
    public void setBankAccountType(
        Account.BankAccountType bankAccountType);
    public void transferFromBank(BigDecimal amount);
}
```

453

你需要更改 `Account` 类的定义，声明它实现该接口：

```
public class Account implements Accountable {
```

它实际上已经实现了，因此不需要对 `Account` 做进一步的更改。

一旦 `testReadOnlyAccess` 得到 `Accountable` 的引用，它遍历更新方法和只读方法的列表。

```
for (Method method: updateMethods)
    verifyException(PermissionException.class, method, account);
for (Method method: readOnlyMethods)
    verifyNoException(method, account);
```

对每个更新方法，`testReadOnlyAccess` 调用 `verifyException` 来确保当调用 `account` 的 `method` 时，它抛出了 `exception`。该测试还确保每个只读方法都可以被调用，而不会产生安全异常。

`verifyException` 方法负责调用一个方法，并确保抛出了 `SecurityException`。下面的代码行实际调用了这个方法：

```
method.invoke(object, nullParmsFor(method));
```

要执行方法，你需要调用 `Method` 对象的 `invoke` 方法，同时传入要调用方法的对象以及参数对象的数组。这里，你使用了实用方法 `nullParmsFor` 来构造一个值全部为 `null` 的数组。你将什么样的参数传入方法都无关紧要。即使方法会产生 `NullPointerException` 或者其他的此类异常，但测试只关心 `PermissionException`。

```
package sis.security;

public class PermissionException extends RuntimeException {
}
```

`verifyException` 方法期待 `invoke` 调用的方法产生一个 `InvocationTargetException`，而不是 `PermissionException`。如果 `invoke` 调用的底层方法抛出了异常，`invoke` 将它包装成

`InvocationTargetException`。你必须调用 `InvocationTargetException` 的 `getCause` 来提取原来被包装的异常对象。

在这个测试中，使用反射并不是必须的。如果本课不是有关反射的，我可能会让你以其它非动态的方式实现这个测试。

`AccountFactory` 类的工作是创建实现了 `AccountTable` 接口的类的实例。

```
package sis.studentinfo;

import java.lang.reflect.*;
import sis.security.*;

public class AccountFactory {
    public static Accountable create(Permission permission) {
        switch (permission) {
            case UPDATE:
                return new Account();
            case READ_ONLY:
                return createSecuredAccount();
        }
        return null;
    }

    private static Accountable createSecuredAccount() {
        SecureProxy secureAccount =
            new SecureProxy(new Account(),
                "credit",
                "setBankAba",
                "setBankAccountNumber",
                "setBankAccountType",
                "transferFromBank");

        return (Accountable)Proxy.newProxyInstance(
            Accountable.class.getClassLoader(),
            new Class[] {Accountable.class },
            secureAccount);
    }
}
```

454

工厂类的使用意味着客户端被隔绝开来，甚至不知道有安全代理类的存在。客户端可以请求一个账号，基于传入的 `Permission` 枚举，`AccountFactory` 创建一个真正的 `Account` 对象（`Permission.UPDATE`）或者一个动态代理对象（`Permission.READ_ONLY`）。创建动态代理的工作位于 `createSecuredAccount` 方法中。

`SecureProxy` 是你将要构造的动态代理对象。它可以作为任何目标类的安全代理。要构造 `SecureProxy`，你需要将目标对象以及必须要安全化的方法列表传递给它。（这些方法的列表可以很容易地从数据库查找出来。安全管理员可以通过 SIS 应用的另一部分来填充数据库的内容。）

`createSecuredAccount` 的第二个语句是，如何设置 `SecureProxy` 作为一个动态代理。这部分代码有些丑陋，因此我在此重复它们，并附以索引号码：

```
return (Accountable)Proxy.newProxyInstance( // 1
    Accountable.class.getClassLoader(), // 2
    new Class[] {Accountable.class }, // 3
```

455

```
secureAccount); // 4
```

第一行调用 **Proxy** 工厂方法 `newProxyInstance`。这个方法是非参数化的，它返回一个对象。你必须将返回值强制转型为 `Accountalbe` 接口的引用。

`newProxyInstance` 的第一个参数（第二行）需要这个接口的类加载器。类加载器从源地址读取一个代表 Java 编译单元的字节流。Java 包含缺省的类加载器，它从磁盘文件读取类文件，但是你可以创建自定义的类加载器，直接从数据库或者从远程的源（例如 Internet）读取类文件。大多数情况，你希望调用 `Class` 对象本身的 `getClassLoader` 方法；这个方法返回最初加载这个类的类加载器。

第二个参数（第三行）是接口类型的数组，你希望为它创建动态代理。在幕后，Java 会使用这个列表来动态地构造一个实现了所有接口的对象。

最后一个参数（第四行）的类型是 `InvocationHandler`，这个接口只包括一个方法，你的动态代理类必须实现这个方法来截获传入的调用。将你的代理对象作为第三个参数传入。

SecureProxy 类

SecureProxyTest:

```
package sis.security;

import java.lang.reflect.*;
import junit.framework.*;

public class SecureProxyTest extends TestCase {
    private static final String secureMethodName = "secure";
    private static final String insecureMethodName = "insecure";
    private Object object;
    private SecureProxy proxy;
    private boolean secureMethodCalled;
    private boolean insecureMethodCalled;

    protected void setUp() {
        object = new Object() {
            public void secure() {
                secureMethodCalled = true;
            }
            public void insecure() {
                insecureMethodCalled = true;
            }
        };
        proxy = new SecureProxy(object, secureMethodName);
    }

    public void testSecureMethod() throws Throwable {
        Method secureMethod =
            object.getClass().getDeclaredMethod(
                secureMethodName, new Class[]{});
```

```

    try {
        proxy.invoke(proxy, secureMethod, new Object[]{});
        fail("expected PermissionException");
    }
    catch (PermissionException expected) {
        assertFalse(secureMethodCalled);
    }
}

public void testInsecureMethod() throws Throwable {
    Method insecureMethod =
        object.getClass().getDeclaredMethod(
            insecureMethodName, new Class[]{});
    proxy.invoke(proxy, insecureMethod, new Object[]{});
    assertTrue(insecureMethodCalled);
}
}

```

SecureProxyTest 中的部分代码同 AccountFactoryTest 中的类似。对这些通用测试代码进行一些重构应该是好主意。这两个测试之间的主要区别是，AccountFactoryTest 需要确保在 Accountable 接口中定义的所有方法均被覆盖了。相反 SecureProxyTest 同纯粹的测试类定义打交道。

setUp 方法使用匿名内联类构造来定义未命名的新类，其中包括两个方法，secure 和 insecure。这些方法所做的一切就是，如果它们被调用了，就设置一个对应的 boolean 实例变量。setUp 方法中的第二条语句，通过传入它的目标对象（即匿名内联类的实例）、以及为测试目的要安全化的方法名，创建 SecureProxy 对象。

testSecureMethod 测试首先查找匿名类中正确的 Method 对象。然后测试将该 Method 对象、以及代理实例和空的参数列表，传递给这个代理对象的 invoke 方法。你通常不会直接使用 invoke 方法，但是为了测试的目的你完全有理由这样做。

```
proxy.invoke(proxy, secureMethod, new Object[]{});
```

testSecureMethod 测试使用惯用的异常检测手法，等待 PermissionException。它还确保安全方法未被调用。testInsecureMethod 测试确保非安全的方法会被调用。

457

最后，下面是 SecureProxy:

```

package sis.security;

import java.lang.reflect.*;
import java.util.*;

public class SecureProxy implements InvocationHandler {
    private List<String> secureMethods;
    private Object target;

    public SecureProxy(Object target, String... secureMethods) {
        this.target = target;
        this.secureMethods = Arrays.asList(secureMethods);
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        try {

```



```

        if (isSecure(method))
            throw new PermissionException();
        return method.invoke(target, args);
    }
    catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
}

private boolean isSecure(Method method) {
    return secureMethods.contains(method.getName());
}
}

```

SecureProxy 在它的构造函数中保存了目标对象以及要安全化的方法列表。它实现了 InvocationHandler 接口中的唯一方法——invoke。所有对代理对象的调用都被发送到 invoke 方法。正如你在测试中看到的那样，invoke 方法接收三个参数：代理对象本身、要调用的方法、以及方法的参数数组¹⁴。如何解释传入的信息取决于你。

在 SecureProxy 中，你所要做的解释是查找方法名，查看它是否在安全方法的列表中。如果是，抛出 PermissionException。如果不是，通过调用传入的 Method 对象的 invoke 方法，将调用委托给目标对象。

458

搞定！你的测试应该可以通过了。

反射的问题

如你已经看到的，你可以使用反射完成一些很酷的事情。反射对某些应用来说几乎是必需的。JUnit 目前大量地依赖反射。许多其它的 Java 技术也需要使用反射，包括 EJB 和 JavaBean。在 Java 中，如果不使用反射，某些任务根本就不可能完成。但是，你需要谨慎地使用反射。

首先，使用反射的代码难于解释和调试。现代的 IDE（集成开发环境），例如 Eclipse，煞费苦心，帮助你较容易地浏览一个系统。例如，Eclipse 可以让你查找对某个类或方法的所有引用。不过，它并不能有效地找到你所感兴趣的、使用反射创建类或调用方法的代码。反射为代码的可追溯性戳了窟窿。

使用反射的代码，在执行速度上，比不使用反射的等效代码要慢得多。当你频繁地或大规模地引入反射代码时，应该对其进行剖析（profile），以确保它能及时地执行完成。

最后，当你使用反射时，你的代码可能会表现出在编译期无法捕捉的缺陷。你必须确保代码的防御性，例如处理可能由 Class.forName 抛出的 ClassNotFoundException。



不要为了使用反射而反射。

¹⁴ 如果现在还来明确的话，基本数据类型的参数被自动包装为一个对象引用。

这是 Jeff 关于 Statics 的规则（出自第 4 课），对反射同样有效。

练习

1. 使用 `Comparable` 类型的匿名内联类，对之前练习中创建的棋盘位置进行排序。
2. 在你调试时，你经常会发现 `toString` 方法并不能提供给你足够的信息。在这种时候，如果有一个实用方法来暴露或转储（dump）更多内部的信息，是非常便利的。

创建一个 `object-dumper` 的实用类，接收一个对象为参数，列出这个对象每个成员变量的名称，并且使用分层格式以递归的方式，列出这些对象当前的内存映像。不要遍历 `java` 或 `javax` 包中的类。这个实用类应该能够显示 `private` 的成员变量。并对 `static` 成员变量进行标记。为了简化，忽略所有来自父类的成员变量。

◀ 459

3. 又一个克隆的练习：实现可怜人（poor man）克隆的另一个版本，这一次使用反射操作。从原对象的 `Class` 类中得到一个 `Constructor` 对象。调用 `newInstance` 来创建一个新的对象，并将原对象每个成员变量的值拷贝到新对象中。被克隆的类需要提供一个无参数的构造函数。只支持浅拷贝（shallow copy）。
4. 创建一个 `Proxy` 类，将定义的每个方法直接委托给原始对象，不过 `toString` 除外。当代理拦截到 `toString` 调用时，将其委托给对象转储器（`object dumper`）。因为代理只能处理接口，目标类需要实现一个定义了 `toString` 的接口类型。

◀ 460



本章将呈现最难于理解和掌握的核心 (core) Java 技术: 多线程。迄今为止, 你已经编写了许多代码, 它们都是在一个线程中执行的。从开始到结束, 它是串行运行的。不过, 你可能需要多线程, 或者同时执行代码的多个段落。

你将学习到:

- 暂停一个线程的执行
- 通过派生 Thread 来创建并运行线程
- 通过实现 Runnable 来创建并运行线程
- 合作式协作式与可抢占的多任务
- 同步
- BlockingQueue
- 停止线程
- wait 与 notify 方法
- 锁以及条件
- 线程优先级
- 死锁
- ThreadLocal
- 定时器类
- 多线程的基本设计原则

多线程

多线程的许多需求, 都与于建立及时响应的用户界面有关。例如, 大多数的字处理应用都包括“自动保存”的功能。你可以配置字处理软件每隔 x 分钟将文档自动保存。自动保存是按

进度表 (schedule) 自动执行的, 而无需你的干预。每次保存可能花费若干秒才能完成, 但是你可以继续工作而不会被打断。

字处理软件的代码至少管理两个线程。线程, 通常被称为前台线程, 来管理你同字处理软件是直接交互。例如, 任何键盘的键入, 都会被前台线程执行的代码捕获。同时, 第二个, 后台线程不时的检查时钟。一旦已经超过了所配置的分钟数, 第二个线程执行保存功能的代码。

在一台多处理器的机器上, 多线程确实可能同时运行, 每个线程占据单独的处理单元。在一台单处理器的机器上, 每个线程会从处理器得到一小段时间片 (通常在同一时刻只能执行一件事情), 让线程表现出好像在同时运行。

学习如何编写各行其是的多线程代码, 是很简单的。挑战在于多个线程需要共享同一资源时, 如何处理。如果你不够小心, 多线程的方案可能会产生不正确的结果, 或者冻结你的应用。测试多线程的方案同样是复杂的。

搜索 (Search) 服务器

服务器类应该能够处理大量进入的请求。如果它要花费若干毫秒对每个请求进行处理, 那么当服务器处理先期传入的请求时, 其它发出请求的客户端所等待的时间可能会超出可以接受的范围。较好的方案是, 让服务器将每个进入的请求作为一个搜索任务保存在队列中。另一个线程, 可以按请求达到的顺序从队列中得到搜索任务, 并同时执行它们。这被称为活动对象 (Active Object) 模式¹; 它将对方法的请求从方法执行中解耦出来。参见图 13.1。所有相应的搜索相关的所有信息为了后面的执行, 都被翻译转换为命令对象 (command object), 以稍后执行。

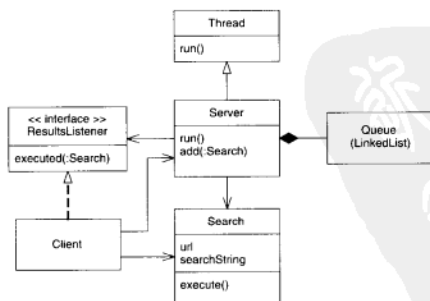


图 13.1 Active Object

¹ [Lavender1996].



你将创建简单的 Web 搜索服务器。这个服务器将接收包括 URL 和搜索字符串的搜索对象作为请求。当它搜索这个 URL 时，服务器将字符串在 URL 所指示的文档中出现的次数，填充到搜索对象中。

注意：如果你没有连接到 Internet，搜索测试可能无法正常执行。这个论断短期内是正确的，在“更少依赖的测试”一节中，将展示如何修改测试以针对本地 URL 执行。

Search 类

创建搜索服务器最简单的方法是，遵循单一职责（Single-Responsibility）原则而先设计一个支持单个搜索的类。一旦它可以正常工作了，你可以关注一下多线程的需求。这样做的好处是，把每个搜索都保持为单独的对象，不必过于考虑共享数据。

SearchTest 类测试可能出现的一些情况：

```
package sis.search;

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

public class SearchTest extends TestCase {
    private static final String URL = "http://www.langrsoft.com";

    public void testCreate() throws IOException {
        Search search = new Search(URL, "x");
        assertEquals(URL, search.getUrl());
        assertEquals("x", search.getText());
    }

    public void testPositiveSearch() throws IOException {
        Search search = new Search(URL, "Jeff Langr");
        search.execute();
        assertTrue(search.matches() >= 1);
        assertFalse(search.errorred());
    }

    public void testNegativeSearch() throws IOException {
        final String unlikelyText = "mama cass elliott";
        Search search = new Search(URL, unlikelyText);
        search.execute();
        assertEquals(0, search.matches());
        assertFalse(search.errorred());
    }

    public void testErrorredSearch() throws IOException {
        final String badUrl = URL + "/z2468.html";
        Search search = new Search(badUrl, "whatever");
        search.execute();
        assertTrue(search.errorred());
        assertEquals(FileNotFoundException.class,
            search.getError().getClass());
    }
}
```

463

}

Search 类的实现之一:

```

package sis.search;

import java.net.*;
import java.io.*;
import sis.util.*;

public class Search {
    private URL url;
    private String searchString;
    private int matches = 0;
    private Exception exception = null;

    public Search(String urlString, String searchString)
        throws IOException {
        this.url = new URL(urlString);
        this.searchString = searchString;
    }

    public String getText() {
        return searchString;
    }

    public String getUrl() {
        return url.toString();
    }

    public int matches() {
        return matches;
    }

    public boolean errored() {
        return exception != null;
    }

    public Exception getError() {
        return exception;
    }

    public void execute() {
        try {
            searchUrl();
        }
        catch (IOException e) {
            exception = e;
        }
    }

    private void searchUrl() throws IOException {
        URLConnection connection = url.openConnection();
        InputStream input = connection.getInputStream();
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(input));
        String line;
        while ((line = reader.readLine()) != null)
            matches += StringUtil.occurrences(line, searchString);
    }
    finally {

```

```

        if (reader != null)
            reader.close();
    }
}

```

Search 中的 `searchUrl` 方法使用 `java.net.URL` 类来获得 `java.net.URLConnection` 对象。`URLConnection` 用来建立在客户端与实际 URL 之间的通讯。只要调用 `URLConnection` 的 `openConnection` 方法, 你就可以调用它的 `getInputStream` 方法来得到 `InputStream` 的引用。这个方法的剩余部分使用 Java 的 IO 代码 (参见第 11 课) 来搜索这个文件。

◀ 465

目前, `searchUrl` 调用 `StringUtil` 的 `occurrences` 方法来计算从输入流读取的每一行中, 匹配字符串的个数。下面的代码是 `StringUtilTest` 以及 `StringUtil`。

```

// StringUtilTest.java
package sis.util;

import junit.framework.*;

public class StringUtilTest extends TestCase {
    private static final String TEXT = "this is it, isn't it";
    public void testOccurrencesOne() {
        assertEquals(1, StringUtil.occurrences(TEXT, "his"));
    }
    public void testOccurrencesNone() {
        assertEquals(0, StringUtil.occurrences(TEXT, "smelt"));
    }
    public void testOccurrencesMany() {
        assertEquals(3, StringUtil.occurrences(TEXT, "is"));
        assertEquals(2, StringUtil.occurrences(TEXT, "it"));
    }
    public void testOccurrencesSearchStringTooLarge() {
        assertEquals(0, StringUtil.occurrences(TEXT, TEXT + "sdfas"));
    }
}

// StringUtil.java
package sis.util;

public class StringUtil {
    static public int occurrences(String string, String substring) {
        int occurrences = 0;
        int length = substring.length();
        final boolean ignoreCase = true;
        for (int i = 0; i < string.length() - substring.length() + 1; i++)
            if (string.regionMatches(ignoreCase, i, substring, 0, length))
                occurrences++;
        return occurrences;
    }
}

```

另一个方案是使用 Java 的正则表达式 (regex) API。参见附加课 III 中关于 regex 的讨论。

更少依赖的测试

如果不幸你无法从计算机连接到 Internet，你可能会因为根本没有能力执行 `ServerTest` 而牢骚满腹。你需要通过将搜索 URL 替换为文件 URL²来解决现状。这还意味着你可以自己编写 HTML 测试文件，来控制要搜索的“Web”页面的内容。

```
package sis.search;

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

public class SearchTest extends TestCase {
    public static final String[] TEST_HTML = {
        "<html>",
        "<body>",
        "Book: Agile Java, by Jeff Langr<br />",
        "Synopsis: Mr Langr teaches you<br />",
        "Java via test-driven development.<br />",
        "</body></html>"};

    public static final String FILE = "/temp/testFileSearch.html";
    public static final String URL = "file:" + FILE;

    protected void setUp() throws IOException {
        TestUtil.delete(FILE);
        LineWriter.write(FILE, TEST_HTML);
    }

    protected void tearDown() throws IOException {
        TestUtil.delete(FILE);
    }
    // ...
}
```

你还需要 `LineWriter` 实用类；下面是测试以及成品（production）代码。

```
// LineWriterTest.java
package sis.util;

import junit.framework.*;
import java.io.*;

public class LineWriterTest extends TestCase {
    public void testMultipleRecords() throws IOException {
        final String file = "LineWriterTest.testCreate.txt";
        try {
            LineWriter.write(file, new String[] {"a", "b"});

            BufferedReader reader = null;
            try {
                reader = new BufferedReader(new FileReader(file));
                assertEquals("a", reader.readLine());
                assertEquals("b", reader.readLine());
                assertNull(reader.readLine());
            }
        }
    }
}
```

² 另一种解决方案可能是安装一个本地的 Web 服务器，例如 Tomcat。

```

    }
    finally {
        if (reader != null)
            reader.close();
    }
}
finally {
    TestUtil.delete(file);
}
}
}

// LineWriter.java
package sis.util;

import java.io.*;

public class LineWriter {
    public static void write(String filename, String[] records)
        throws IOException {
        BufferedWriter writer = null;
        try {
            writer = new BufferedWriter(new FileWriter(filename));
            for (int i = 0; i < records.length; i++) {
                writer.write(records[i]);
                writer.newLine();
            }
        }
        finally {
            if (writer != null)
                writer.close();
        }
    }
}
}

```

其中好的部分是，对 `SearchTest` 更改的侵入性并不是很大。事实上，所有的测试方法都不需要改变。你只是添加了初始化方法，将一段 HTML 写入到本地文件；以及拆卸方法，在每次测试完成之后将之删除。你还需要改变 URL 来使用文件协议而不是 http 协议。

初始化表达式

`searchUrl` 中包括下面的代码：

```

String line;
while ((line = reader.readLine()) != null)

```

圆括号应该能帮助你理解这段代码。首先，`reader.readLine()` 的结果被赋值给引用变量 `line`。引用 `line` 的值同 `null` 进行比较，来判断 `while` 循环是否应该结束了。

468

其中不好的部分是，`Search` 类必须更改。要从文件协议的 URL 获得 `InputStream`，你必须从 URL 中提取出路径信息，并使用它来打开 `FileInputStream` 输入流。这并不是什么显著的变化。稍有瑕疵的是，在你的产品系统中，可能有一小段代码只在测试时才会被使用。

```

private void searchUrl() throws IOException {
    InputStream input = getInputStream(url);
    BufferedReader reader = null;
    try {

```

```

        reader = new BufferedReader(new InputStreamReader(input));
        String line;
        while ((line = reader.readLine()) != null)
            matches += StringUtil.occurrences(line, searchString);
    }
    finally {
        if (reader != null)
            reader.close();
    }
}

private InputStream getInputStream(URL url) throws IOException {
    if (url.getProtocol().startsWith("http")) {
        URLConnection connection = url.openConnection();
        return connection.getInputStream();
    }
    else if (url.getProtocol().equals("file")) {
        return new FileInputStream(url.getPath());
    }
    return null;
}

```

另一件你需要考虑的事情是，你将不会再执行同 http URL 打交道部分的代码。最好的方法是，确保你在验收测试（acceptance test）³级别有足够的测试覆盖。验收测试提供了在单元测试级别之上的一级测试——它们从最终用户的角度测试系统。它们尽可能地针对实际运行的系统而执行，而且不应该使用 mock。对搜索应用来说，你肯定要使用实际的 Web URL 来执行验收测试。

469

服务器

ServerTest:

```

package sis.search;

import junit.framework.*;
import sis.util.*;

public class ServerTest extends TestCase {
    private int numberOfResults = 0;
    private Server server;
    private static final long TIMEOUT = 3000L;
    private static final String[] URLS = {
        SearchTest.URL, SearchTest.URL, SearchTest.URL }; // 1

    protected void setUp() throws Exception {
        TestUtil.delete(SearchTest.FILE);
        LineWriter.write(SearchTest.FILE, SearchTest.TEST_HTML);

        ResultsListener listener = new ResultsListener() { // 2
            public void executed(Search search) {

```

³ 不同类型的测试有不同的命名。验收测试（acceptance test）是为表明系统满足了目标客户验收标准的测试，你可能听说过这些测试也被称为客户测试。

```

        numberOfResults++;
    });

    server = new Server(listener);
}

protected void tearDown() throws Exception {
    TestUtil.delete(SearchTest.FILE);
}

public void testSearch() throws Exception {
    long start = System.currentTimeMillis();
    for (String url: URLS) // 3
        server.add(new Search(url, "xxx"));
    long elapsed = System.currentTimeMillis() - start;
    long averageLatency = elapsed / URLS.length;
    assertTrue(averageLatency < 20); // 4
    assertTrue(waitForResults()); // 5
}

private boolean waitForResults() {
    long start = System.currentTimeMillis();
    while (numberOfResults < URLS.length) {
        try {Thread.sleep(1); }
        catch (InterruptedException e) {}
        if (System.currentTimeMillis() - start > TIMEOUT)
            return false;
    }
    return true;
}
}

```

◀ 470

该测试首先构造一个 URL 字符串的列表（第一行）。

setUp 方法构造一个 ResultsListener 对象（第二行）。当你构造新的 Server 实例时，将这个对象作为参数传入。

你的测试，作为客户端，只是将搜索的请求传递给服务器。为了响应性能的考虑，你需要将服务器设计为，无需让客户端等待每个请求的完成。但是客户端仍然需要获知服务器依次处理每个请求的结果。这种情况下，回调（callback）是一种常用的机制。

回调这个术语是从 C 语言中衍生而来的，它允许你创建函数的指针。当你得到一个函数指针时，你可以将这个指针传递给其它函数，就如传递其它引用一样。接收指针的代码然后可以使用这个函数指针，回调位于发起（originating）代码中的函数。

在 Java 中，实现回调的有效方法是，将匿名内联类的实例作为参数，传递给一个方法。接口 ResultsListener 定义了 executed 方法：

```

package sis.search;

public interface ResultsListener {
    public void executed(Search search);
}

```

在 testSearch 中，你将 ResultsListener 的匿名内联类的实例，传递给 Server 的构造函数。

Server 对象保留了 ResultsListener 的引用，并在搜索执行完成时调用其 `executed` 方法。

在 Java 实现中回调常被称为侦听器 (listener)。侦听器接口定义了你希望当某事发生时要被调用的方法。当有事件发生时，用户界面类常使用侦听器来通知其他代码。例如，你可以使用事件侦听器来设置 Java Swing 类，让你在用户点击按钮关闭窗口时得到通知。这可以让你在窗口实际关闭之前，把所有松散的处理串连在一起。

在该测试中，你简单地使用 ResultsListener 实例来跟踪完成搜索执行的总次数。

当你创建 Server 对象后，你使用循环（第三行）来迭代 URL 的列表。你使用每个 URL 来构造一个 Search 对象（和搜索文本无关）。为了演示 Java 快速分发每个请求的目的，你可以跟踪创建每个搜索以及向服务器添加它们所消耗的执行时间。使用之后的断言（第四行）来展示，平均的延时（响应时间的延迟）绝对小于 20 毫秒甚至更少。

因为服务器在分离的线程中执行多个搜索，JUnit 测试中的代码可能早于搜索的完成。你需要一种机制来挂起处理，直至搜索全部完成，且执行搜索的次数和要搜索的 URL 个数相同。

471

测试中的等待

在 ServerTest 中第五行处的断言调用了 `waitForResults` 方法，它将会挂起当前线程的执行（也就是测试执行所在的线程），直至得到所有的搜索结果。超时值提供了确定的持续时间，超过这个时间之后 `waitForResults` 应该返回 `false` 并导致断言失败。

```
private boolean waitForResults() {
    long start = System.currentTimeMillis();
    while (numberOfResults < URLs.length) {
        try {Thread.sleep(1); }
        catch (InterruptedException e) {}
        if (System.currentTimeMillis() - start > TIMEOUT)
            return false;
    }
    return true;
}
```

`waitForResults` 方法执行简单的循环。循环体的每次迭代会暂停 1 毫秒，然后快速地计算已持续的时间，并检查是否超出了超时的限制。这个暂停是通过调用 `Thread` 的静态方法 `sleep` 取得的。`sleep` 方法以毫秒数为参数，并让当前线程的执行停顿一定的时间。⁴

因为 `sleep` 可能抛出 `InterruptedException` 类型的 checked 异常，你可以选择使用 try-catch 代码块来将它封闭起来。一个线程可能会中断另一个线程，这会产生该异常。但是线程的中断通常是被设计产生的，这里的代码展示了一种罕有的情况，在这里提供空的 catch 块来忽略异常是可以接受的。

⁴ 线程调度器至少会等待指定的时间，可能会更长一些。

`waitForResults` 中使用的循环机制目前是足以胜任的。稍后在本课讨论的 `wait/notify` 技术（参见 `Wait/Notify` 一节），提供了等待某个条件发生的最佳通用机制。

◀ 472

创建并运行线程

`Server` 类需要接收传入的搜索，并同时处理还没有被执行的请求。实际进行搜索的代码将在一个单独的线程中被执行。`Server` 的主线程，也就是其余代码执行的部分，必须衍生（`spawn`）第二个线程。

Java 提供了两种方式让你启动一个单独的线程。第一种需要你继承 `java.lang.Thread` 类并提供一个 `run` 方法的实现。然后你可以调用其 `start` 方法来启动它。从此，`run` 方法中的代码开始在单独的线程中执行。

第二种衍生线程的技术是，创建实现了 `java.lang.Runnable` 接口的类的实例，`Runnable` 接口定义如下：

```
public interface Runnable {
    void run();
}
```

你可以用 `Runnable` 对象作为参数，来构造 `Thread` 对象。调用 `Thread` 对象的 `start` 方法来启动 `run` 方法中的代码。这种技术常常使用实现了 `Runnable` 接口的匿名内联类来完成。

分清线程或者线程的执行与 `Thread` 对象之间的区别，是非常重要的。线程是线程调度器所管理的一段控制流程。`Thread` 是一个管理有关线程执行信息的对象。一个 `Thread` 对象的存在并不意味着一个线程的存在；直至 `Thread` 对象启动，线程才会存在，而且 `Thread` 对象可以在线程结束之后存在更长的时间。

现在，你将使用第一种技术来创建线程，并让 `Server` 类从 `Thread` 中派生。`Thread` 本身实现了 `Runnable` 接口。如果你继承 `Thread`，你需要覆写（`override`）`run` 方法，以进行有意义的处理。

```
package sis.search;

import java.util.*;

public class Server extends Thread {
    private List<Search> queue = new LinkedList<Search>(); // flaw!
    private ResultsListener listener;

    public Server(ResultsListener listener) {
        this.listener = listener;
        start();
    }

    public void run() {
        while (true) {
            if (!queue.isEmpty())
                execute(queue.remove(0));
            Thread.yield();
        }
    }
}
```

◀ 473

```

    }

    public void add(Search search) {
        queue.add(search);
    }

    private void execute(Search search) {
        search.execute();
        listener.executed(search);
    }
}

```

Server 类定义了两个字段成员变量：一个 ResultsListener 引用，以及一个名为 queue，容纳、包含 Search 对象 Search 对象一个名为 queue 的 LinkedList。queue 队列引用的声明是有瑕疵的——它不是“线程安全”的！现在，你的测试很可能会执行成功，但是这样的瑕疵将导致你的应用产生故障。在本章的“同步集合类”一节中，你将学习有关线程安全的问题以及如何改正它。

java.util.LinkedList 类实现了 List 接口。链表与将元素保存在连续内存空间的 ArrayList 不同，它为你添加的每个元素分配一块新的内存。不同的内存块将会散列在内存空间中；每个内存块都含有指向下一个内存块的链接。有关链表如何工作的内存状况概念示意图，参见图 13.2。对于需要频繁在列表尾部之外的其它地方进行删除或插入操作的集合来说，LinkedList 较 ArrayList 会提供更好的性能特性。

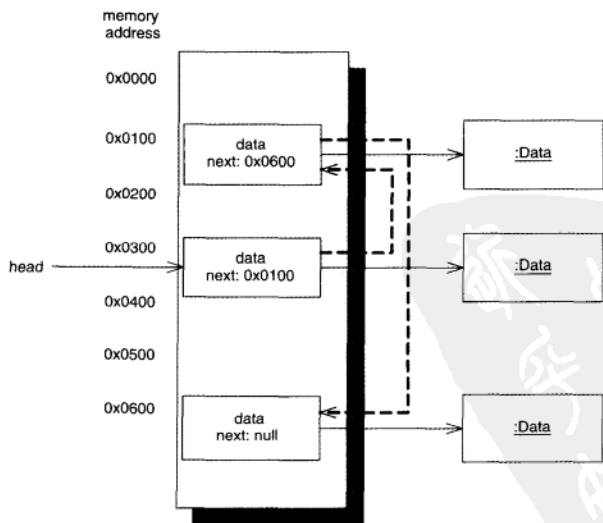


图 13.2 链表的概念示意图

引用 `queue` 保存了传入的搜索请求。链表这种情况下作为队列的数据结构。队列也被称为先进先出的 (FIFO, first-in and first-out) 的列表: 当你要求队列删除元素时, 它会首先删除列表中比较老的项。Server 种的 `add` 方法接收一个 `Search` 对象作为参数, 并把它加入到队列的尾部。而为了让 `LinkedList` 像队列那样工作, 你必须从列表的开始处删除 `Search` 对象。

Server 的构造函数, 通过调用 `start` 启动第二个线程 (通常被称为工人线程或后台线程)。此时 `run` 方法被调用。Server 中的 `run` 方法是一个无限循环, 它将持续运行直至其它代码显式地终止这个线程 (参见本章的“停止线程”一节) 或运行该线程的应用被终止 (参见本章的“关闭”一节)。当你在 JUnit 中运行 `ServerTest` 时, 即使 JUnit 测试已经结束, 后台线程也会保持运行。只有当你关闭 JUnit 窗口时, 它才会停止。

474

循环体首先检查队列是否为空。如果非空, 代码删除并执行队列中的第一个元素。否则, 代码会接下来调用 `Thread` 的 `yield` 方法。`yield` 方法可以让其它线程在后台线程重被调度之前, 有机会得到处理器的运行时间片。在下一节“合作式协作式和可抢占的多任务”中, 我们将讨论 `yield` 为什么是必须的。

`execute` 方法将实际的搜索委托给 `Search` 对象本身。当搜索完成时, `ResultListener` 引用的 `executed` 方法会被调用, 且以 `Search` 对象作为回调的参数。

你可以使用一张 UML 序列图来表示搜索成功的消息流程 (图 13.3)。序列图是你的系统在实际运行时的动态视图。它展示了对象之间有序的消息流程。我发现序列图是交流有关系统如何连接以及如何使用的一种非常有效的方式。⁵

475

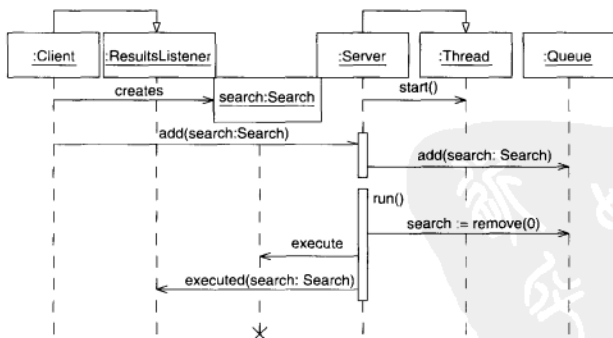


图 13.3 当前活动的 `Search` 对象的序列图

在一幅副序列图中, 你使用的是对象图框 (box) 而不是类图框。从每个图框垂直投下的虚

⁵ 在表示复杂的并行活动时, 序列图有时并不是最好的方式。活动图可能是更适合表示多线程行为的一种 UML 模型。

线表示对象的生命线。在对象生命线底部的“X”示意它的终结。图 13.3 中的 Search 对象，当它被通知搜索已经完成时，便消失不见了；因此生命线以一个“X”结束。

你使用一种有向的线段（从调用对象的生命线指向接收对象）来表示消息的发送（译注：有同步和异步之分）。消息流程的次序是由上至下排列的。在图 13.3 中，一个 Client 对象发送第一个消息，creates。这个特定的消息表示 Client 对象负责创建一个 Search 对象。注意，Search 对象从更低的位置开始；它并不是在一开始就存在的。

在第一个消息发出之后，Client 向 Server 对象发送 add (Search) 消息。这个消息的发送从概念上讲是异步的——Client 不需要等待 Server 对象返回任何信息⁶。你可以使用半箭头来表示异步的消息发送。当 Server 接收到这个 add 方法时，其 add 方法将 Search 对象转交给队列。你可以使用一个活动块（对象生命线上盖住的瘦长矩形）来表示 Server 对象 add 方法执行的生命期。

同时 Server 线程向它的父类（或者，更确切地说，是它自己）发送 start 的消息。在 Server 中覆写的 run 方法开始执行。它的生命期是由 Server 对象生命线上的第二个活动块来指示的。run 方法中的代码，向 Queue 发送消息 remove (0)，从 Queue 获得（并删除）第一个 Search 元素。之后，run 方法调用 Search 对象的 execute 方法，并在 execute 完成时通知 ResultsListener（通过 executed）。

476

合作式协作式(cooperative)与可抢占的(preemptive)多任务

在单处理器的环境中，每个单独的线程都从所执行的处理器获得一个时间片。问题是，一个线程在被另一个接替之前，可以得到多长时间？答案取决于许多情况，包括你运行 Java 所在的操作系统以及 JVM 的具体实现。

大部分现代操作系统（包括 Unix 的各个变体以及 Windows）使用可抢占的多任务，这些操作系统（OS，operating system）可以中断当前执行的线程。关于此线程的信息被保存起来，同时 OS 提供一个时间片给下一个线程。在这种环境中，所有线程最终都会得到线程调度器的一定关照。

另一种可能性是操作系统使用合作式协作式多任务来管理线程。合作式协作式多任务取决于其他线程代码的良好行为，即频繁地出让时间给其他线程。在合作式协作式线程模型中，一个拙劣编写的线程可能会完全占用处理器，阻止其他线程进行任何处理。当线程阻塞在 IO 操作或被挂起时，可以通过显式地调用 yield 方法或进入睡眠（我们已经提及的一些方法），来让

⁶ 我们实现它的方式是，消息的发送是同步的，但是因为操作是即刻完成且没有返回任何信息，你可以把它描绘为异步的。

出时间。

在 `Server` 类中, `run` 方法中的 `while` 循环在每次迭代时会调用 `yield` 方法来让其他线程得到机会执行。⁷

```
public void run() {
    while (true) {
        if (!queue.isEmpty())
            execute(queue.remove(0));
        Thread.yield();
    }
}
```

477

同步

多线程开发中一个最大的陷阱是,不得不面对这样的事实,即线程的运行次序是不确定的。当你每次执行一个多线程应用时,线程可能以一种无法预料的不同次序运行。这种不确定性所带来的最大挑战是,你在编码时的一个缺陷,可能只会在数千次执行后、或者每隔两个月,才表现出一次。⁸

执行代码的两个线程并不必须以同样的速率来遍历代码。一个线程可能执行了五行代码,而之前的线程甚至只执行了一行。线程调度器交错 (interleaves) 每个执行线程的代码片。

而且,线程并不位于 Java 语句的级别,而是位于语句被翻译后由 VM 执行的更底层的级别。你在 Java 中编写的大部分语句都是非原子的:甚至许多诸如增加计数器或给一个引用赋值这样简单的事,Java 编译器都可能会创建若干内部操作来对应这一条语句。假定一个赋值语句会耗费两个内部操作。第二个操作,在线程调度器挂起该线程并执行第二个线程之前,可能还没有完成。

所有这种代码的交错,意味着你可能会遇到同步的问题。如果两个线程均在同一时间访问同一段数据,结果可能是你无法预料的。



你将修改 `sis.studentinfo.Account` 类来支持资金提款。要提款资金,账号的结余至少要等于提款的数额。如果提款的数额太大,你应该什么都不做 (出于简单性和演示的目的)。无论何种情况,你都不希望账号的结余变得小于 0。

```
package sis.studentinfo;
// ...
public class AccountTest extends TestCase {
    // ...
    private Account account;
```

⁷ 当你运行这段代码时,取决于你的环境,可能会体验到明显的 CPU 占用。如果没有其他线程在执行, `yield` 方法可能什么也不做。另一个替代方法可以是引入一毫秒的睡眠。

⁸ 细微的重力引力可能会导致静电潮汐,由此会让处理器变慢百分之一秒。

478

```

protected void setUp() {
    account = new Account();
    // ...
}
// ...
public void testWithdraw() throws Exception {
    account.credit(new BigDecimal("100.00"));
    account.withdraw(new BigDecimal("40.00"));
    assertEquals(new BigDecimal("60.00"), account.getBalance());
}

public void testWithdrawInsufficientFunds() {
    account.credit(new BigDecimal("100.00"));
    account.withdraw(new BigDecimal("140.00"));
    assertEquals(new BigDecimal("100.00"), account.getBalance());
}
// ...
}

    withdraw 方法:

package sis.studentinfo;
// ...
public class Account implements Accountable {
    private BigDecimal balance = new BigDecimal("0.00");
    // ...
    public void withdraw(BigDecimal amount) {
        if (amount.compareTo(balance) > 0)
            return;
        balance = balance.subtract(amount);
    }
    // ...
}

```

withdraw 方法在多线程的环境下有个基本的问题。假定两个线程都同时尝试从一个有 100 美元结余的帐户提款 80 美元。第一个线程应该成功；另一个线程应该失败。最终的结余应该是 20 美元。不过，代码的执行可能被交错如下：

线程 1	线程 2	结余
account.compareTo (balance) > 0		100
	amount.compareTo (balance) > 0	100
balance = balance.subtract (amount)		20
	balance = balance.subtract(amount)	-60

第二个线程对结余的检查，发生在第一个线程得到授权后且在其从结余中减去提款额之前。其它的执行顺序可能导致相同的问题。

479

你可以编写简短的测试来演示这个问题。

```

package sis.studentinfo;

import junit.framework.*;
import java.math.BigDecimal;

public class MultithreadedAccountTest extends TestCase {
    public void testConcurrency() throws Exception {

```

```

final Account account = new Account();
account.credit(new BigDecimal("100.00"));

Thread t1 = new Thread(new Runnable() {
    public void run() {
        account.withdraw(new BigDecimal("80.00"));
    }
});
Thread t2 = new Thread(new Runnable() {
    public void run() {
        account.withdraw(new BigDecimal("80.00"));
    }
});

t1.start();
t2.start();
t1.join();
t2.join();

assertEquals(new BigDecimal("20.00"), account.getBalance());
}
}

```

诚然，这个方法中的许多代码是冗余的。在你理解了这个方法的作用之后，确保你会对其重构以消除重复。

使用 Runnable 创建线程

`testConcurrency` 方法引入了两个新事物。首先，它展示了如何通过传入一个实现 `Runnable` 接口的类的实例来创建一个线程。回忆一下 `Runnable` 接口定义了一个唯一的方法 `run`。这个测试为每个线程构造了一个 `Runnable` 接口的匿名内联类实例。注意，`run` 方法中的代码在 `start` 方法被调用之前，并不会执行。

`testConcurrency` 的第二个新事物是对 `Thread` 的 `join` 方法的使用。当你调用一个线程的 `join` 方法，当前线程的执行会挂起直至该线程完成。在测试可以继续之前，`testConcurrency` 中的代码首先等待 `t1` 完成，然后是 `t2`。

◀ 480

当你执行 JUnit 时，`testConcurrency` 十有八九会通过。对一个小方法而言，所有的执行都非常快速，线程调度器很可能在调度下一个线程之前，已经为一个线程分配了足够的时间来完整地运行其方法。你可以通过在 `withdraw` 方法中插入暂停，来强制使问题出现：

```

public void withdraw(BigDecimal amount) {
    if (amount.compareTo(balance) > 0)
        return;
    try {Thread.sleep(1); }
    catch (InterruptedException e) {}
    balance = balance.subtract(amount);
}

```

现在你应该会看到一个红条。

synchronized

在语义上，你希望 `withdraw` 中所有代码的执行都是原子的。线程应该能够运行整个 `withdraw` 方法，从开始到结束，不会被任何其它线程妨碍。你可以在 Java 中使用 `synchronized` 方法修饰符来达成这一点。

```
public synchronized void withdraw(BigDecimal amount) {
    if (amount.compareTo(balance) > 0)
        return;
    balance = balance.subtract(amount);
}
```

Java 中的同步实现被称为互斥。通过互斥保护代码的另一种称法是临界区。⁹为了保证 `withdraw` 代码的执行是互斥的，Java 在执行线程代码所在的对象上放置了一个锁。当一个线程锁住了某个对象，其他的线程便无法再获得这个对象上的锁。试图获得锁的其他线程将会阻塞，直至锁被释放。锁在方法执行完毕之后会被释放。

Java 使用一种称为监视器（monitor）的概念来保护数据。每个对象都关联有一个监视器：这个监视器保护对象的实例数据。一个监视器同每个类相关联；它保护类的静态数据。当你获得一个锁时，你同时也获得了相关联的监视器；在任何给定的时刻，只有一个线程可以获得一个锁。¹⁰

你应该总是尽可能锁住最小数量的代码，否则当其它线程等待锁的时候，你可能会体验到性能的问题。你如果像我反复建议地那样，创建一个小型的、组合的方法，你会发现方法级的锁已经可以满足大部分的需求。不过，你可以通过创建一个 `synchronized` 代码块来为更小的原子操作、而不是对整个方法加锁。你还必须在 `synchronized` 关键字后面的括号中指定一个对象作为监视器。

下面的 `withdraw` 实现同之前的实现是等价的。

```
public void withdraw(BigDecimal amount) {
    synchronized(this) {
        if (amount.compareTo(balance) > 0)
            return;
        balance = balance.subtract(amount);
    }
}
```

一个 `synchronized` 代码块需要使用大括号，即便它只包括一条语句。

你可以将一个类方法（静态方法）声明为 `synchronized`。当 VM（虚拟机）执行一个类方法时，它会从定义该方法的 Class 对象中得到一个锁。

⁹ [Sun2004].

¹⁰ <http://www.artima.com/insidejvm/ed2/threadsynch2.html>.

同步的集合类

如我早先提示的那样，`Server` 类在将队列声明为 `LinkedList` 时有一个缺陷。

```
public class Server extends Thread {
    private List<Search> queue = new LinkedList<Search>(); // 缺陷!
```

当你同 Java 2 集合类框架（Collection Class Framework）中的诸如 `ArrayList`、`LinkedList` 和 `HashMap` 打交道时，你应该认识到它们不是线程安全的——这些类中的方法不是同步的。早期的集合类，`Vector` 和 `HashTable`，缺省就是同步的。不过，如果你需要使用线程安全的集合，我还是建议你不要使用它们。

相反，你可以使用 `java.util.Collections` 中的实用类，将一个集合的实例封装到一个称为同步包装器（wrapper）的类中。同步包装类获得目标集合所需的锁，然后将所有的调用委托给集合以进行通常的处理。修改 `Server` 类，将 `queue` 的引用包装到一个同步的列表中：

482

```
public class Server extends Thread {
    private List<Search> queue =
        Collections.synchronizedList(new LinkedList<Search>());
    // ...
    public void run() {
        while (true) {
            if (!queue.isEmpty())
                execute(queue.remove(0));
            Thread.yield();
        }
    }

    public void add(Search search) throws Exception {
        queue.add(search);
    }
    // ...
}
```

添加同步包装并不需要你更改使用 `queue` 引用的任何代码。

稍用一点分析，对于队列这里好像没有什么同步问题。`add` 方法向列表的尾部插入对象；`run` 循环只在队列非空时才从队列的起始处删除。没有其它的方法操作队列了。不过，这里仍然有微弱的可能性，一部分 `add` 和 `remove` 操作可能同时执行。这种并发可能会破坏 `LinkedList` 实例的完整性（integrity）。使用同步包装会消除这种缺陷。

BlockingQueue

J2SE 5.0 中添加的一个很有前景的 API 是同步类库，位于 `java.util.concurrent` 包中。它提供了实用类来帮助你解决许多有关多线程的问题。当然你应该首选使用这个库，但是它的存在并

不意味着你不需要学习线程背后的基本原理和概念。¹¹

因为队列在多线程应用中是一个如此有用的概念，同步库定义了 `BlockingQueue` 的接口，以及五个特化的（specialized）队列类实现了该接口。阻塞队列实现了另一个新的接口，`java.util.Queue` 它为集合定义了队列的语义。阻塞队列为队列添加了与同步相关的能力。举例来说，包括在取出下一元素时等候其出现的能力，以及在保存元素时等待有空间空闲的能力。

你可以使用 `LinkedBlockingQueue` 替换 `LinkedList` 来重写 `Server` 类。

```
package sis.search;

import java.util.concurrent.*;

public class Server extends Thread {
    private BlockingQueue<Search> queue =
        new LinkedBlockingQueue<Search>();
    private ResultsListener listener;

    public Server(ResultsListener listener) {
        this.listener = listener;
        start();
    }

    public void run() {
        while (true)
            try {
                execute(queue.take());
            }
            catch (InterruptedException e) {
            }
    }

    public void add(Search search) throws Exception {
        queue.put(search);
    }

    private void execute(Search search) {
        search.execute();
        listener.executed(search);
    }
}
```

`LinkedBlockingQueue` 的 `put` 方法将元素添加到队列的尾部。`take` 方法从队列的开始处删除一个元素。它还会等待，直至队列中有元素存在。和原本的 `Server` 实现比较起来，代码看上去并没有什么不同，但是它现在是线程安全的。

停止线程

在 `ServerTest` 中还存在一个测试的瑕疵。`Server` 类中的线程，执行的是一个无限循环。通常，

¹¹ 其重要性好比在总是使用计算器解决所有问题之前学习好乘法如何工作一样。

循环在停止时 `Server` 本身就停止了。不过，当你的测试在 `JUnit` 中运行时，线程会一直运行直至你关闭 `JUnit` 本身——`Server` 的 `run` 方法将一直盘桓下去。

`Thread` API 中包括 `stop` 方法，看起来好像是停止线程的诀窍。但是不需要花多少时间，在详细的 API 文档可以读到，`stop` 方法已经不建议使用了，并且是“先天性不安全的”。幸运的是，文档接着解释了原因以及你应该如何做。（请阅读它。）推荐的技术是让线程简单地基于某种条件自然死亡。一种方式是使用一个初始化为 `true` 的布尔变量，当你希望线程结束时将之设置为 `false`。`run` 方法中 `while` 循环的条件会检测这个变量的值。

你需要修改 `ServerTest` 的 `tearDown` 方法来关闭线程并验证这种方法是有效的。

```
protected void tearDown() throws Exception {
    assertTrue(server.isAlive());
    server.shutdown();
    server.join(3000);
    assertFalse(server.isAlive());
    TestUtil.delete(SearchTest.FILE);
}
```

需要记住的是，无论测试是否抛出异常，`tearDown` 方法在每次测试方法完成之前，都要被完整地执行。在 `tearDown` 中，你首先确保线程是“活着的”并且正在运行。然后调用服务器的 `shutdown` 方法。你已经选择了用 `shutdown` 作为方法名来通知服务器停止。

然后通过 `join` 方法来等待服务器线程，同时指定超时时间为 3 秒钟。线程中止是要花一些时间的。如果不使用 `join` 方法，`tearDown` 中的剩余代码，很可能会在线程完全中止之前就被执行了。最终，在你调用了 `Server` 的 `shutdown` 命令之后，确保它的线程已经不复存在了。`Thread` 中定义了 `isAlive` 方法，因为 `Server` 是 `Thread` 的子类，你可以调用 `Server` 的 `isAlive` 方法来得到你的答案。

因为 `Server` 类现在使用 `LinkedBlockingQueue`，你不能在 `run` 方法的 `while` 循环中使用布尔标志变量。`LinkedBlockingQueue` 的 `take` 方法会阻塞循环——它会等待直至队列中有一个新对象可用。

◀ 485

让 `LinkedBlockingQueue` 停止等待的一种方法是，将一个特殊的 `Search` 对象放到队列中，这对你如何编码有重要的干系。每次在你从队列中取出对象时，你都需要检查它是否是那个特殊的 `Search` 对象，并且如果是的话就中止 `run` 方法。

另一个将 `LinkedBlockingQueue` 从等待中停止的方法是，通过调用线程的 `interrupt` 方法来中断线程，这将产生一个 `InterruptedException`。如果捕获到一个 `InterruptedException`，你就可以从 `while` 的无限循环中跳出来。下面是这种方法的实现：

```
public class Server extends Thread {
    // ...
    public void run() {
        while (true)
            try {
                execute(queue.take());
            }
            catch (InterruptedException e) {
```



```

        break;
    }
}
// ...
public void shutDown() throws Exception {
    this.interrupt();
}
}

```

Wait/Notify

Java 提供了一种机制来帮助你在两个或更多线程之间协调动作。你常常需要让一个线程等待另一个线程完成其工作。Java 提供了一种称为“等待/通知”的机制。你可以从一个线程内部调用 `wait` 方法来使其停转。另一线程可以通过调用 `notify` 方法来唤醒这个等待的线程。

示例？让我们放松一下！来一个时钟的小玩意儿！



在这个示例中，你将构建适用于时钟应用的 `Clock` 类。该时钟应用的用户界面（User Interface, UI）可以显示数字读数，或者模拟表示——包括小时、分钟和秒的指针。而 UI 忙于创建用户可视的输出，实际的 `Clock` 类可以执行一个单独的线程来不停的跟踪时间。`Clock` 线程将无限循环。每隔一秒钟，它将会唤醒并通知用户界面，时间已经改变了。

`UI` 类实现了 `ClockListener` 接口。你将一个 `ClockListener` 的引用传递给 `Clock` 对象；然后 `Clock` 以更改的时间为参数回调这个接口。使用这种基于侦听器的设计，你可以将一个模拟时钟界面替换为数字时钟界面，且无需更改 `Clock` 类（参见图 13.4）。

比较麻烦的部分是为之编写测试。下面是策略：

- 创建 `mock` 的 `ClockListener` 来保存所有它接收到的消息（调用）
- 创建 `Clock` 实例，传入这个 `mock` 对象
- 启动时钟，并等待直至 `mock` 接收到五个消息
- 确保 `ClockListener` 接收到的消息，是 `Date` 类型的实例，且每个相隔一秒钟

下面的测试依次执行了上述步骤。比较有技巧的一点是，如何让测试等待直至 `mock` 得到了所希望的全部消息。我稍后会解释这部分，但是我会让你推断出测试的结果，包括验证测试侦听器所得到的 `tics`。

```

package sis.clock;

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

public class ClockTest extends TestCase {
    private Clock clock;
    private Object monitor = new Object();
}
// 1

```

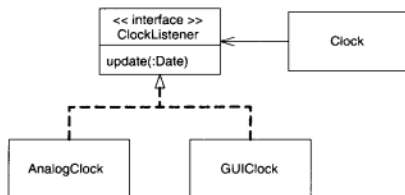


图 13.4 Clock 侦听器

487

```

public void testClock() throws Exception {
    final int seconds = 5;
    final List<Date> tics = new ArrayList<Date>();
    ClockListener listener = new ClockListener() {
        private int count = 0;
        public void update(Date date) {
            tics.add(date);
            if (++count == seconds)
                synchronized(monitor) { // 2
                    monitor.notifyAll(); // 3
                }
        }
    };
    clock = new Clock(listener);
    synchronized(monitor) { // 4
        monitor.wait(); // 5
    }
    clock.stop();
    verify(tics, seconds);
}

private void verify(List<Date> tics, int seconds) {
    assertEquals(seconds, tics.size());
    for (int i = 1; i < seconds; i++)
        assertEquals(1, getSecondsFromLast(tics, i));
}

private long getSecondsFromLast(List<Date> tics, int i) {
    Calendar calendar = new GregorianCalendar();
    calendar.setTime(tics.get(i));
    int now = calendar.get(Calendar.SECOND);
    calendar.setTime(tics.get(i - 1));
    int then = calendar.get(Calendar.SECOND);
    if (now == 0)
        now = 60;
    return now - then;
}
}

```

在测试创建 `Clock` 实例之后，它开始等待（第五行）。你可以调用任何对象的 `wait` 方法——`wait` 是在 `Object` 类中定义的。你必须首先得到同一对象上的锁。可以使用 `synchronized` 代码块（第四行）或者之前提到的 `synchronized` 方法。在这个测试中，你使用保存在 `monitor`

488

字段成员变量（第一行）中的任意对象，作为加锁和等待的对象。

这个测试在等待什么呢？在模拟的侦听器实现中，你可以保存一个 count 的变量，在每次调用 update (Date) 方法时加 1。只要达到了希望的数字，你就可以通知 ClockTest 停止等待。

从实现的观点来看，测试线程会等待直至侦听器（运行在另一个线程）示意可以继续。侦听器通过调用测试所等待的同一对象（第三行）的 notifyAll 方法，来示意可以继续。为了调用 notifyAll，你必须首先得到监视器（monitor）对象上的锁，再一次使用 synchronized 代码块。

不过…等一下！对 wait 的调用是位于 synchronized 代码块中的，这意味着其它代码在 synchronized 代码块退出之前，都无法得到锁——包围了 notifyAll 方法的 synchronized 代码块也不例外。它在等待结束之前不会退出，而等待在 notifyAll 方法被调用之前也不会结束。这似乎是一个令人左右为难的景况。

诀窍诡计是：在幕后，wait 方法将当前进程放到监视器对象的所谓“等待集合”中。然后它会在当前线程进入空闲状态之前，释放所有的锁。这样，当其它线程的代码遇到包围了 notifyAll 调用的 synchronized 代码块时，监视器上并没有锁。然后它可以得到一个锁（第二行）。notifyAll 调用需要这个锁，以便将它的消息发送给监视器。

你可能想要多复习几次有关 wait/notify 的讨论，直到对它有一个透彻的理解。

下面是 ClockListener 接口以及 Clock 实现类：

```
// sis.clock.ClockListener
package sis.clock;

import java.util.*;

public interface ClockListener {
    public void update(Date date);
}

// sis.clock.Clock
package sis.clock;

import java.util.*;

public class Clock implements Runnable {
    private ClockListener listener;
    private boolean run = true;

    public Clock(ClockListener listener) {
        this.listener = listener;
        new Thread(this).start();
    }

    public void stop() {
        run = false;
    }

    public void run() {
        while (run) {
```

489

```

        try {Thread.sleep(1000);}
        catch (InterruptedException e) {}
        listener.update(new Date());
    }
}

```

在 Clock 中的 run 方法，在每次 while 循环开始时，使用非常简单的判断布尔标志 (run) 的技术，来决定是否停止。其 stop 方法，在测试完成等待之后被调用，它将布尔值 run 设置为 false。

Clock 的实现中还有一点缺陷。run 方法至少要睡眠一秒。如我之前提醒你注意的那样，sleep 调用可能会花费额外的纳秒或毫秒。此外，创建一个新的 Date 对象也要花费时间。这意味着时钟可能会跳过一秒钟：例如，如果最近一次报告的时间是 11:55:01.999，同时 sleep 以及额外的执行会花费 1001 毫秒而不是整整 1000 毫秒，那么新报告的时间将会是 11:55:03.000。时钟的 UI 将显示 11:55:01，然后跳到 11:55:03。在模拟显示中，你可能看到秒针多跳动了一些——在大多数应用中不算什么大问题。如果你在测试中运行足够多的迭代次数，它几乎肯定会失败。

下面是改进的实现：

```

public void run() {
    long lastTime = System.currentTimeMillis();
    while (run) {
        try {Thread.sleep(10);}
        catch (InterruptedException e) {}
        long now = System.currentTimeMillis();
        if ((now / 1000) - (lastTime / 1000) >= 1) {
            listener.update(new Date(now));
            lastTime = now;
        }
    }
}

```

你可以编写测试来持续地发现这个缺陷吗？一种选择可能包括创建一个实用类来返回当前的毫秒数。然后你可以锁住这个实用类来强迫一定的时间。

490

对 Clock 的测试是个麻烦事。它为你的单元测试添加了 5 秒钟的执行时间。你怎样将这一时间最小化呢？首先，不管测试是验证五次还是两次，可能都不会有影响。其次，你可以修改时钟类（同时相应地修改测试）来支持可配置的时间间隔。编写测试来演示如何支持百分之一秒（这是运动手表所需要的），而不是一秒。

wait 和 notify 的补充注意事项

Object 类中的 wait 方法有不同的重载 (overload) 版本，包括允许你指定一个超时时间。当你使用这种版本的 wait 方法挂起一个线程时，它会等待直至另一个线程通知它或者超过超时时间段。

在特定的情况下，有可能会发生不合逻辑的唤醒（spurious wakeup）——它并不是由显式通知触发的。这很少见，但是如果你在产品性代码中使用 `wait` 和 `notify`，你需要提防这种情况。为了这么做，你可以将 `wait` 语句套入到一个 `while` 循环中，每次迭代都检测标志执行是否可以继续的条件。`wait` 方法的 Java API 文档，展示了你可以如何实现这个循环。

在这个测试中，你可以创建 `do-while` 循环，在每次迭代时调用 `verify` 方法。如果 `verify` 方法失败了，你需要再次执行 `wait`。我认为这对测试来说引入了不必要的复杂性。没有预防这种很少出现的不合逻辑唤醒，最糟的结果也无外乎测试失败，这时候你只需要再运行它一次。

然而 `notifyAll` 唤醒所有的线程，如果有多个等待线程的话，`notify` 方法选择一个特定的线程来唤醒。在大多数情况下，你会想要使用 `notifyAll`。不过，在某些情况下，你希望只唤醒一个线程。线程池就是一个例子。

搜索服务器现在使用一个线程来处理传入的请求。客户端最终以一种异步的方式得到它们的搜索结果。然而这对较小数量的请求可能是适用的，如果有许多客户端在之前请求了搜索，那么后面的客户端可能需要等待相当长的一段时间。

你可能会考虑在每个搜索请求到达时，为其创建一个新的、单独的线程。不过，你创建并启动的每个线程，都会带来显著的运行开销。为一个搜索中的每一个都创建一个线程，将会导致严重的性能问题。Java 线程调度器在不同线程间切换上下文的时间，可能比每个线程实际处理的时间还要长。

线程池收集了由它创建并启动运行的、较少数量的工人 `Thread` 对象。当搜索进入时，你将它移交给线程池。线程池将这个任务加入到队列的尾部，然后发起一个 `notify` 调用。所有已完成工作的工人线程，检查队列中是否有待处理的任务。如果没有任务可执行，工人线程通过阻塞的 `wait` 调用进入空闲状态。如果有待处理的任务，`notify` 调用将唤醒它们中的一个。然后被唤醒的工人线程得到下一个任务并进行处理。

锁与条件

J2SE 5.0 引入了一种新的，更灵活的机制来获取对象上的锁。`java.util.concurrent.locks.Lock` 接口允许你将一个或多个条件关联到一个 `lock`。存在几种不同类型的锁。例如，`ReadWriteLock` 实现只允许一个线程对共享资源进行写操作，而同时其它线程可以从同一资源中进行读取。你还可以为一个可重入锁添加适当的规则，来完成例如允许等待时间最长的线程最先获得锁的功能。更多细节请参见 API 文档。

`ClockTest` 代码清单，展示了你如何将同步的使用替换为新的锁风格。

```
package sis.clock;

import java.util.*;
import java.util.concurrent.locks.*;
```

```

import junit.framework.*;

public class ClockTest extends TestCase {
    private Clock clock;
    private Lock lock;
    private Condition receivedEnoughTics;

    protected void setUp() {
        lock = new ReentrantLock();
        receivedEnoughTics = lock.newCondition();
    }

    public void testClock() throws Exception {
        final int seconds = 2;
        final List<Date> tics = new ArrayList<Date>();
        ClockListener listener = createClockListener(tics, seconds);

        clock = new Clock(listener);
        lock.lock();
        try {
            receivedEnoughTics.await();
        }
        finally {
            lock.unlock();
        }
        clock.stop();
        verify(tics, seconds);
    }

    private ClockListener createClockListener(
        final List<Date> tics, final int seconds) {
        return new ClockListener() {
            private int count = 0;
            public void update(Date date) {
                tics.add(date);
                if (++count == seconds) {
                    lock.lock();
                    try {
                        receivedEnoughTics.signalAll();
                    }
                    finally {
                        lock.unlock();
                    }
                }
            }
        };
    }
}

```

492

经过修改且稍稍重构后的 ClockTest 实现，使用 Lock 接口的 ReentrantLock 实现。有兴趣访问某个共享资源的线程，调用 ReentrantLock 对象的 lock 方法。一旦线程获得了这个锁，其它线程会一直试图尝试锁住这段代码，直至第一个线程释放了这个锁。一个线程通过调用 Lock 对象的 unlock 方法来释放锁。你始终要确保解锁的操作是发生在 try-finally 代码块中的。

ClockTest 中的共享资源是 Condition 对象。你通过调用 Lock 的 newCondition 方法（参见

setUp 方法)来得到一个 Condition 对象。一旦你获得了一个 Condition 对象,你可以使用 await 来阻塞它。同 wait 方法一样,await 中的代码会释放锁同时挂起当前的线程。¹²和 Condition 对象的使用相结合,另一线程中的代码通过向 Condition 对象发送 signal 或 signalAll 消息,发信号表明条件已经满足。ClockTest 中展示的技巧有效地替代了经典的 wait/notify 模式。

493

线程优先级

线程可以指定不同的优先级。调度器使用线程优先级作为参考,来决定隔多长时间调度该线程。执行的主线程有缺省定义指定的优先级,Thread.NORM_PRIORITY。优先级是一个整型,由低到高的范围是 Thread.MIN_PRIORITY 至 Thread.MAX_PRIORITY。

你可以为在后台执行的线程使用较低的优先级。相反的,你可以为需要及时响应相应的用户界面代码使用较高的优先级。通常,你会希望从 NORM_PRIORITY 偏离较小的量,例如+1 或-1。较大的偏离量可能造成性能糟糕的应用,某些线程专横霸道,而其他的则渴求执行。

每个线程(主线程之外)在开始时衍生它的线程有相同的优先级。你可用调用线程的 setPriority 来指定不同的优先级。

下面是 Clock 类的代码片段,展示了你可以为时钟线程的优先级设置较低的值。

```
public void run() {
    Thread.currentThread().setPriority(Thread.NORM_PRIORITY - 1);
    long lastTime = System.currentTimeMillis();
    while (run) {
        try {Thread.sleep(10); }
        catch (InterruptedException e) {}
        long now = System.currentTimeMillis();
        if ((now / 1000) - (lastTime / 1000) >= 1) {
            listener.update(new Date(now));
            lastTime = now;
        }
    }
}
```

注意静态方法 currentThread 的使用,你随时可以调用该方法得到代表当前执行线程的 Thread 对象。

494

死锁

如果你在编写多线程应用时不够小心,你可能会遇到死锁,把你的系统搞死。假设你有一个 alpha 对象运行在一个线程中,beta 对象运行在另一个线程中。Alpha 执行的一个同步方法(按

¹² 这意味着你可能不需要将 await 包围到一个 try-finally 代码块中。不要冒险——你无法保证 await 中的代码不会有偏差。

定义拥有了 `alpha` 上的锁)调用了在 `beta` 中定义的同步方法。如果同时 `beta` 正在执行一个同步方法,而在这个方法中接下来调用了 `alpha` 中的同步方法,每个线程在继续执行之前,都等待对方线程释放所需的锁。死锁!¹³

解决死锁的方案:

1. 把要上锁的对象排定次序,并保证在获得锁时使用相同的次序。
2. 锁住一个共同的对象。

ThreadLocal

你可能需要为执行的每个线程保存各自独立的信息。Java 提供了一个名为 `ThreadLocal` 的类,来管理对每个线程为每个线程创建和访问任意类型的线程独立的任意类型的实例进行管理。

当你通过 JDBC¹⁴ 同一个数据库交互时,你要得到一个 `Connection` 对象来管理应用同数据库之间的通讯。不过,多个线程同时使用一个连接对象是不安全的。`ThreadLocal` 的典型应用是让每个线程包含一个独立的 `Connection` 对象。

不过,你还没有学习 JDBC。我会为您定义 `ThreadLocal` 的替代应用。

在本例中,你需要为已有的 `Server` 类添加日志的功能。你希望跟踪每个线程的搜索何时开始以及何时结束。你还希望开始和结束消息成对儿一起出现在一个 `log` 文件中。你可以将每个事件(开始和结束)的消息,保存在 `Server` 的一个线程安全的公用列表中。但是如果多个线程要向公共列表中添加消息,消息会是相互交替出现的。你可能首先看到所有的开始日志,然后所有的停止日志事件。

◀ 495

为了解决这个问题,你可以让每个线程将它自己的日志消息保存在 `ThreadLocal` 的变量中。当线程结束时,你可以取得一个锁,并将所有消息一次添加到完整的日志中。

首先,修改 `Server` 类,使其为每个 `Search` 请求创建新的线程。在多数环境下这会推进提高性能,但是让多少个线程同时执行,你还是要谨慎。(你可以试验性地设置测试启动的搜索线程个数,来查看在你环境中的极限。)你可能会考虑“wait 和 notify 的补充注意事项”一节中提到的线程池。

```
private void execute(final Search search) {
    new Thread(new Runnable() {
        public void run() {
            search.execute();
            listener.executed(search);
        }
    }).start();
}
```

¹³ [Arnold2000].

¹⁴ Java DataBase Connectivity API. 进一步信息请参见附加课 III。

确保你依然可以成功地运行测试。下一步，让我们重构 `ServerTest`——更改会是显著的。你希望添加一个新的测试来验证记录下来的日志消息。现在的代码有点乱，而且它有一个相当冗长麻烦的测试。对第二个测试，你希望能重用 `testSearch` 中的大部分非断言（nonassertion）代码。重构后的代码还稍许简化了性能测试。

下面是重构后的代码，包括了一个新的测试，`testLogs`。

```
package sis.search;

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

public class ServerTest extends TestCase {
    // ...
    public void testSearch() throws Exception {
        long start = System.currentTimeMillis();
        executeSearches();
        long elapsed = System.currentTimeMillis() - start;
        assertTrue(elapsed < 20);
        waitForResults();
    }

    public void testLogs() throws Exception {
        executeSearches();
        waitForResults();
        verifyLogs();
    }

    private void executeSearches() throws Exception {
        for (String url: URLS)
            server.add(new Search(url, "xxx"));
    }

    private void waitForResults() {
        long start = System.currentTimeMillis();
        while (numberOfResults < URLS.length) {
            try {Thread.sleep(1); }
            catch (InterruptedException e) {}
            if (System.currentTimeMillis() - start > TIMEOUT)
                fail("timeout");
        }
    }

    private void verifyLogs() {
        List<String> list = server.getLog();
        assertEquals(URLS.length * 2, list.size());
        for (int i = 0; i < URLS.length; i += 2)
            verifySameSearch(list.get(i), list.get(i + 1));
    }

    private void verifySameSearch(
        String startSearchMsg, String endSearchMsg) {
        String startSearch = substring(startSearchMsg, Server.START_MSG);
        String endSearch = substring(endSearchMsg, Server.END_MSG);
        assertEquals(startSearch, endSearch);
    }
}
```

```

private String substring(String string, String upTo) {
    int endIndex = string.indexOf(upTo);
    assertTrue("didn't find " + upTo + " in " + string,
        endIndex != -1);
    return string.substring(0, endIndex);
}
}

```

testLogs 方法执行搜索，等待他们完成，并对日志进行验证。为了验证日志，测试需要从 Server 对象请求完整的日志，然后对其遍历并每次取出两行。它验证两行中的搜索字符串（日志消息的第一部分）是一样的。¹⁵

对 Server 类的修改：

```

package sis.search;

import java.util.concurrent.*;
import java.util.*;

public class Server extends Thread {
    private BlockingQueue<Search> queue =
        new LinkedBlockingQueue<Search>();
    private ResultsListener listener;
    static final String START_MSG = "started";
    static final String END_MSG = "finished";

    private static ThreadLocal<List<String>> threadLog =
        new ThreadLocal<List<String>>() {
            protected List<String> initialValue() {
                return new ArrayList<String>();
            }
        };
    private List<String> completeLog =
        Collections.synchronizedList(new ArrayList<String>());
    // ...
    public List<String> getLog() {
        return completeLog;
    }

    private void execute(final Search search) {
        new Thread(new Runnable() {
            public void run() {
                log(START_MSG, search);
                search.execute();
                log(END_MSG, search);
                listener.executed(search);
                completeLog.addAll(threadLog.get());
            }
        }).start();
    }

    private void log(String message, Search search) {
        threadLog.get().add(
            search + " " + message + " at " + new Date());
    }
}

```

497

¹⁵ 有关 Java 日志的深入讨论，以及是否有必要测试日志代码，参见第 8 课。

```
// ...
}
```

你通过把 `ThreadLocal` 绑定到你希望在每个线程中保存的对象类型, 声明了一个 `ThreadLocal` 对象。这里 `threadLog` 被绑定到 `String` 的 `List` 对象。`threadLog` 这个 `ThreadLocal` 实例, 将会为每个线程在内部管理一个 `String` 对象的 `List`。

你不能简单地给 `threadLog` 赋初始值, 因为它所管理的每个 `List` 对象都需要被初始化。相反, `ThreadLocal` 提供了一个你可以覆写的 `initialValue` 方法。在每个线程第一次通过 `threadLog` 得到它的 `ThreadLocal` 实例时, 在 `ThreadLocal` 类中的代码会调用 `initialValue` 方法。覆写这个方法, 为你提供了机会来一致地初始化这个列表。

`ThreadLocal` 定义了另外三个方法: `get`、`set` 和 `remove`。`get` 和 `set` 方法允许你访问当前线程的 `ThreadLocal` 实例。`remove` 方法允许你删除当前线程的 `ThreadLocal` 实例, 或许为了节约内存空间。

在搜索线程的 `run` 方法中, 你在搜索执行的前后调用 `log` 方法。在 `log` 方法中, 你通过调用 `threadLog` 的 `get` 方法访问当前线程的日志列表拷贝。然后向该列表中添加一个适当的日志字符串。

一旦搜索线程完成, 你希望将线程的日志添加到完整的日志中。因为你已经将 `completeLog` 初始化为一个同步的集合, 你可以调用它的 `addAll` 方法, 来确保所有线程日志中的消息行, 是作为一个整体被加入的。如果没有同步, 另一个线程可能添加它的日志行, 结果可能产生一个相互交错的完整日志。

Timer 类



你希望不停地随时监控一个 Web 站点, 来查看是否它包括了你所指定的搜索词条。你可能有兴趣每隔几分钟或者每隔几秒钟就进行检查。

`SearchSchedulerTest` 中的 `testRepeatedSearch` 测试, 展示了搜索监视器应该如何工作: 创建一个调度器, 并将一个搜索以及代表每隔多长时间运行该搜索的时间间隔传递给它。测试机制会在调度器启动之后耐心等待一段时间, 然后验证在给定时间内执行了预期的搜索数目。

```
package sis.search;

import junit.framework.*;
import sis.util.*;

public class SearchSchedulerTest extends TestCase {
    private int actualResultsCount = 0;

    protected void setUp() throws Exception {
        TestUtil.delete(SearchTest.FILE);
    }
}
```

```

        LineWriter.write(SearchTest.FILE, SearchTest.TEST_HTML);
    }

    protected void tearDown() throws Exception {
        TestUtil.delete(SearchTest.FILE);
    }

    public void testRepeatedSearch() throws Exception {
        final int searchInterval = 3000;
        Search search = new Search(SearchTest.URL, "xxx");

        ResultsListener listener = new ResultsListener() {
            public void executed(Search search) {
                ++actualResultsCount;
            }
        };

        SearchScheduler scheduler = new SearchScheduler(listener);
        scheduler.repeat(search, searchInterval);

        final int expectedResultsCount = 3;
        Thread.sleep((expectedResultsCount - 1) * searchInterval + 1000);

        scheduler.stop();
        assertEquals(expectedResultsCount, actualResultsCount);
    }
}

```

499

SearchScheduler 类，如下所示，使用 `java.util.Timer` 类作为调度和执行搜索的基础。如果你已经创建了 **Timer** 实例，你可以调用其中的方法来调度 **TimerTask**。**TimerTask** 是实现了 **Runnable** 接口的抽象类。你通过子类化 **TimerTask** 并实现 `run` 方法，来提供任务执行的细节。

在 **SearchScheduler** 中使用 `scheduleAtFixedRate` 方法，你可以传入一个 **TimerTask** 和延时启动的毫秒数、以及一个间隔的毫秒数（目前是由测试指定的）。每隔 `interval` 毫秒，**Timer** 对象唤醒并调用 **TimerTask** 的 `run` 方法。这被称为固定频率（fixed-rate）执行。

当你使用固定频率（fixed-rate）执行¹⁶时，**Timer** 尽可能保障任务是每隔 `interval` 毫秒就被运行。如果一个任务的执行比间隔还要长，**Timer** 尝试将比较固定的任务放到一起执行，以确保它们在每次间隔之前完成。你还可以使用延时频率（delayed-rate）执行方法中的一种，在这种情况下使间隔时间从本次搜索结束到下次开始时得以保持一致。

你可以通过 `cancel` 方法取消定时器或者定时任务。

```

package sis.search;

import java.util.*;

public class SearchScheduler {
    private ResultsListener listener;
    private Timer timer;
    public SearchScheduler(ResultsListener listener) {
        this.listener = listener;
    }

    public void repeat(final Search search, long interval) {

```

500

¹⁶ [Sun2004b].)

```

        timer = new Timer();
        TimerTask task = new TimerTask() {
            public void run() {
                search.execute();
                listener.executed(search);
            }
        };
        timer.scheduleAtFixedRate(task, 0, interval);
    }

    public void stop() {
        timer.cancel();
    }
}

```

Thread 的杂项

原子变量和 volatile

Java 编译器试图尽可能地优化代码。例如，如果你在循环之外给某个字段成员变量赋初值：

```

private String prefix;
void show() throws Exception {
    prefix = ":";
    for (int i = 0; i < 100; i++) {
        System.out.print(prefix + i);
        Thread.sleep(1);
    }
}

```

Java 编译器可能认为 prefix 字段成员变量的值在 show 方法期间是固定不变的。它可能会对代码进行优化，在循环内将 prefix 的使用视为常量。Java 可能不会担心在每次循环迭代时去读取 prefix 的值。这是一个恰当的假设，只要没有其他的代码（在多线程执行时）在同时更新 prefix 的值。但是如果其他线程的确更改了 prefix，循环中的代码可能永远不会看到改变的值。¹⁷

你可以通过把对成员变量的访问包围到 synchronized 代码段中，来强制 Java 总是读取更新的新值。Java 基于这种认知：如果代码是在多线程中执行的，确保它总是得到对象内所有字段成员变量的最新值。另一种方式是将字段成员变量声明为 volatile 来让 Java 读取刷新的值。这样做本质上讲是告诉编译器不要对字段成员变量的访问进行优化。

当你使用共享的 boolean 变量作为线程 run 循环的条件时，你可能需要将该 boolean 变量声明为 volatile。这确保 while 循环在每次迭代时都读取更新的 boolean 值。

相关的考量是，Java 如何对待多个线程对字段成员变量的共同访问。Java 保证读写一个变量是原子的——long 和 double 除外。这意味着对变量最小的可能操作，是读取或写入它的整

¹⁷ 实际的行为可能依赖于你的编译器、JVM 以及处理器的配置。

个值。不可能有一个线程写入变量的一部分，而另一个线程写入另一部分，这会搞乱变量。

线程信息

如果我们早先提到的，你可以使用 `Thread` 的静态方法 `currentThread` 来得到当前执行线程对应的 `Thread` 对象。`Thread` 对象包括有许多有用的方法来得到有关线程的信息。关于这些 `getter` 和 `query` 方法的清单，参见 Java API 文档。此外，使用 `setName` 和 `getName`，你可以设置或得到线程的名字。

关闭

在应用执行中的主线程，缺省是一个用户线程。只要还有一个活动的用户线程，你的应用就会继续执行。相反，你可以显式地将一个线程指定为守护（daemon）线程。如果所有的用户线程都已经终止了，应用也会终止，不过守护线程还会继续执行。

`ThreadTest` 实例演示了用户线程的这种行为。`main` 方法中执行的代码运行在一个用户线程中。被衍生的任何线程都会继承父线程的执行模式（`user` 或 `daemon`），因此实例中的线程 `t` 是一个用户线程。

```
public class ThreadTest {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread() {
            public void run() {
                while (true)
                    System.out.print('.');
            }
        };
        t.start();
        Thread.sleep(100);
    }
}
```

502

当你执行 `ThreadTest` 时，在你强迫终止进程（在命令行下 `Ctrl-c` 通常有效）之前，它不会停止。

将一个 `Thread` 对象的执行模式设置为守护线程非常简单，只需要使用参数 `true` 来调用 `Thread` 的 `setDaemon` 方法。你必须在启动线程之前设置其执行模式。一旦线程已经启动了，你就不能改变它的执行模式。

```
public class ThreadTest {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread() {
            public void run() {
                while (true)
                    System.out.print('.');
            }
        };
        t.setDaemon(true);
        t.start();
    }
}
```

```

        Thread.sleep(100);
    }
}

```

你可以通过调用 `System` 的静态方法 `exit`，强制退出一个应用，不管是否还有用户线程存活。（`Runtime` 类中有等效的方法）。

```

public class ThreadTest {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread() {
            public void run() {
                while (true)
                    System.out.print('.');
            }
        };
        t.start();
        Thread.sleep(100);
        Runtime.getRuntime().exit(0);
        //System.exit(0); // this will also work
    }
}

```

503

`exit` 方法需要你传递给它 `int` 值。这个值代表应用的返回值，你可以在 `shell` 脚本或批处理文件中使用这个值来进行流程控制。

在许多应用，例如 `Swing` 应用中，你将不会对由它衍生的其它线程有控制权。`Swing` 本身的代码在线程中执行。通常，这些线程会被初始化为用户线程。相应的，你可能需要使用 `exit` 方法来终止 `Swing` 应用。

管理异常

当 `run` 方法抛出（unchecked¹⁸）异常时，运行它的线程会终止。此外，异常对象本身也会消失。你可能希望捕捉该异常，如 `ServerTest.TestException` 所演示的那样。

```

public void testException() throws Exception {
    final String errorMessage = "problem";
    Search faultySearch = new Search(URLS[0], "") {
        public void execute() {
            throw new RuntimeException(errorMessage);
        }
    };
    server.add(faultySearch);
    waitForResult(1);
    List<String> log = server.getLog();
    assertTrue(log.get(0).indexOf(errorMessage) != -1);
}

private void waitForResult() {
    waitForResult(URLS.length);
}

private void waitForResult(int count) {
    long start = System.currentTimeMillis();
}

```

¹⁸ 因为 `Runnable` 的 `run` 方法原型没有什么异常，你不能覆写它来抛出任何 checked 异常。

```

while (numberOfResults < count) {
    try {Thread.sleep(1); }
    catch (InterruptedException e) {}
    if (System.currentTimeMillis() - start > TIMEOUT)
        fail("timeout");
}
}

```

测试覆写了 `Search` 的 `execute` 方法来模拟抛出一个 `RuntimeException`。你的预期是异常会作为一次失败的搜索而被记录到 `Server` 的日志中。下面是 `Server` 类的实现：

◀ 504

```

private void execute(final Search search) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            log(START_MSG, search);
            search.execute();
            log(END_MSG, search);
            listener.executed(search);
            completeLog.addAll(threadLog.get());
        }
    });
    thread.setUncaughtExceptionHandler(
        new Thread.UncaughtExceptionHandler() {
            public void uncaughtException(Thread th, Throwable thrown) {
                completeLog.add(search + " " + thrown.getMessage());
                listener.executed(search);
            }
        }
    );
    thread.start();
}

```

在创建线程对象之后，且在启动它之前，你可以为线程设置一个未捕获异常的处理器。你可以通过实现 `uncaughtException` 方法来创建未捕获异常处理器。如果线程抛出了异常而没有被捕获，这个 `uncaughtException` 方法会被调用。Java 把 `Thread` 和 `Throwable` 对象作为参数传递给 `uncaughtException` 方法。

线程组

线程组提供了一种方式把线程组织为任意的组。考虑到包含的层次结构，线程组还可以包括其他的线程组，允许包容的层次结构。你可以把线程组中包括的所有线程作为控制单元。例如，你可以调用线程组的 `interrupt` 方法，它继而会调用容器层次中所有线程的 `interrupt` 方法。

在《Effective Java》中，Joshua Bloch 写到线程组时说道，“最好把它看作是失败的试验品，而你可以简单地忽略它们的存在”¹⁹。它们最初被设计用于 Java 中的安全管理，但是从未令人满意地实现这个目的。

在 J2SE 5.0 之前，线程组一个有用的用途是监视线程的未捕获异常。不过，你现在可以通

◀ 505

¹⁹ [Bloch 2001].

过为线程创建一个 `UncaughtExceptionHandler` 来满足这一需要（参见前一节，管理异常）。

Atomic 包装器 (wrapper)

即使是简单的算术操作也不是原子的。从 J2SE 5.0 开始，Sun 提供了 `java.util.concurrent.atomic` 包。这个包中包括了许多 Atomic 包装器类。每个类都包装了一种基本类型（int、long 或 boolean）、引用、或数组。通过使用原子包装器，你可以保证包装后的值是线程安全的，而且总是最新的（与你将字段成员变量声明为 `volatile` 一样）。

下面是一个使用的示例：

```
AtomicInteger i = new AtomicInteger(50);
assertEquals(55, i.addAndGet(5));
assertEquals(55, i.get());
```

有关更多细节参见 Java API 文档。

总结：同步的基本设计原则

- 避免它。仅在必要时引入同步。
- 隔离它。将同步的需要集中到一个职责单一的类中，让它的功能尽可能少而集中。确保尽可能少的客户端代码会与同步类打交道。确保尽可能少的方法会与共享数据打交道。
- 共享的类、或者“服务器”类应该提供同步，而不是完全依赖于客户端来进行同步。
- 如果服务器类没有提供同步，考虑编写一个“包装器”类来添加同步、并将方法调用委托给服务器类。
- 使用 `java.util.concurrent` 包中的同步类库。

练习

- 1 创建一个 `AlarmClock` 类。客户端可以提交一个事件（event）和相应的时间。当时间到达之后，`AlarmClock` 向客户端发送一个闹钟警告。
- 2 使用 `wait` 和 `notify` 来消除 `AlarmClockTest`（源自前一个练习）中的等待循环。
- 3 修改 `AlarmClock` 来支持多个闹钟（如果它还不支持的话）。然后，添加按名字取消闹钟的功能。编写一个测试来演示取消闹钟的功能。然后，分析你的代码并注意潜在的同步问题。在产品代码中引入暂停来迫使问题发生并使你的测试失败。最后，在必要的地方引入同步代码块或将方法改为同步，来修正这个问题。
- 4 修改 `AlarmClock` 中的 `run` 方法，使用定时器每隔半秒钟检查一次闹钟。

我在第 2 课中向你介绍了参数化类型。本课将深入地探讨参数化类型或范型。你将学习到创建参数化类型背后的许多规则。创建参数化类型再搭上多线程，在核心 (core) Java 编程中大概是最复杂的课题了。

你将学习到：

- 创建参数化类型
- 多类型参数
- 擦拭方案
- 上限 (extend)
- 通配符
- 范型方法
- 下限 (super)
- 附加限界
- 原始类型
- checked 集合

参数化类型

一般你会开发一个集合 (collection) 来保存任意类型的对象。不过，在大多数情况下，只有当你限制它们保存单一 (类别) 的类型时 (例如学生的列表、单词到定义的映射等)，集合才是有用的。你可以把它看作是单一职责原则 (Single-Responsibility Principle) 对集合的一种应用：集合应该包含且仅包含一种事物。你很少希望在你的学生列表中夹杂一个教授。而且在你的映射 (map) 中如果包括一个单词到图像的表项，这将会为你的字典应用产生非常头痛的问题。

Sun 原本在 Java 类库中开发的基本集合类，支持保存并返回任意类型的对象（也就是 Object 及其任何子类）。这意味着你可以在一个本打算保存 Student 对象的集合中添加 String 对象。而这听起来未必是错误的，但从集合中取出一个无法预料类型的对象，是应用时常产生缺陷的根源。其原因是，在集合中保存对象的代码，与从集合中取出对象的代码常常相距甚远。

随着 J2SE 5.0 的到来，Sun 引入了参数化类型的概念，也被称为范型。现在你可以将一个集合的实例关联或绑定到某种特定类型。你可以指定 ArrayList 只包括 Student 对象。在早先的 Java 版本中，当你不小心取出 ArrayList 中保存的 String 对象时，会在运行时收到一个 ClassCastException 错误。现在，使用参数化类型，当你试图在列表中插入一个 String 时，这部分代码会收到一个编译错误。

集合框架（Collection Framework）

Sun 已经将集合框架中的类全部参数化了。一个简单的基于语言（language-based）的测试演示了参数化 ArrayList 类型的一个简单使用：

```
final String name = "joe";
List<String> names = new ArrayList<String>(); // 1
names.add(name); // 2
String retrievedName = names.get(0); // 3
assertEquals(name, retrievedName);
```

实现类 ArrayList 和接口 List 都绑定到 String 类（第一行）。你可以向 name（第二行）添加 String 对象。当从 name 中取出首元素并赋值给一个 String 引用（第三行）时，你无需将之转型为 String。

如果你试图插入其他类型的对象：

```
names.add(new Date()); // this won't compile!
```

编译器会呈现给你一个错误：

```
cannot find symbol
symbol : method add(java.util.Date)
location: interface java.util.List<java.lang.String>
names.add(new Date()); // this won't compile!
```

510

多类型参数

如果你查看关于 java.util.List 和 java.util.ArrayList 的 API 文档，你会看到它们分别被声明为

List<E>和 ArrayList<E>。<E>被称为类型参数列表 (type parameter list)。List 接口和 ArrayList 类的声明, 在它们的类型参数列表中只包括一个类型参数。因此, 当你在代码中使用 List 或 ArrayList 时, 必须提供一个绑定类型。¹

HashMap 类表示一个 key-value 对的集合。Key 可以是一种类型, 而 value 则可以是另一种。例如, 你可能在 HashMap 中保存一个约会簿 (appointment book), 其中 key 为 Date, 而 value 是对这一天发生事件的 String 描述。

```
final String event = "today";
final Date date = new Date();
Map<Date, String> events = new HashMap<Date, String>();
events.put(date, event);
String retrievedEvent = events.get(date);
assertEquals(event, retrievedEvent);
```

Map 接口和 HashMap 类分别被声明为 Map<K, V>和 HashMap<K, V>。当你使用它们时必须提供两个绑定类型, 一个是 key (K) 另一个为 value (V)。

创建参数化类型

你将开发一个参数化的 MultiHashMap。MultiHashMap 和 HashMap 类似, 不过它允许你为一个指定的 key 关联多个 value。让我们扩展一下日程表的例子: 有些人过着非常惬意的生活, 某一天他们可能需要做两件事情有两个活动有两个事件。

一些起步的测试会是你成功的开始。记住, 你应该递增地开发, 每次只实现一个测试方法和一个断言。

```
package sis.util;

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

public class MultiHashMapTest extends TestCase {
    private static final Date today = new Date();
    private static final Date tomorrow =
        new Date(today.getTime() + 86400000);
    private static final String eventA = "wake up";
    private static final String eventB = "eat";

    private MultiHashMap<Date,String> events;
    protected void setUp() {
        events = new MultiHashMap<Date,String>();
    }

    public void testCreate() {
        assertEquals(0, events.size());
    }
}
```

¹ 你并不是必须提供任何绑定类型, 不过, 这是不提倡的。参见本课中的“原始类型”一节。

```

    }

    public void testSingleEntry() {
        events.put(today, eventA);
        assertEquals(1, events.size());
        assertEquals(eventA, getSoleEvent(today));
    }

    public void testMultipleEntriesDifferentKey() {
        events.put(today, eventA);
        events.put(tomorrow, eventB);
        assertEquals(2, events.size());
        assertEquals(eventA, getSoleEvent(today));
        assertEquals(eventB, getSoleEvent(tomorrow));
    }

    public void testMultipleEntriesSameKey() {
        events.put(today, eventA);
        events.put(today, eventB);
        assertEquals(1, events.size());
        Collection<String> retrievedEvents = events.get(today);
        assertEquals(2, retrievedEvents.size());
        assertTrue(retrievedEvents.contains(eventA));
        assertTrue(retrievedEvents.contains(eventB));
    }

    private String getSoleEvent(Date date) {
        Collection<String> retrievedEvents = events.get(date);
        assertEquals(1, retrievedEvents.size());
        Iterator<String> it = retrievedEvents.iterator();
        return it.next();
    }
}

```

512

getSoleEvent 方法是我们感兴趣的地方。在使用 Map 的 get 方法，得到保存在数据中的事件集合之后，你必须确保它只包含有一个元素。为了取得这个元素，你可以创建一个迭代器并返回它指向的第一个元素。你必须为 Iterator 对象绑定同集合一样的类型（在本例中是 String）。

从实现的角度来说，有不只一种方式来建立 MultiHashMap。最容易的是使用 HashMap，其中每个 key 都关联一个 value 的集合。例如，你可以通过封装和使用 HashMap 来定义 MultiHashMap。

为了支持现有的测试，MultiHashMap 的实现是简化的。每个方法，例如 size、put 和 get，都委托给封装后的 HashMap：

```

package sis.util;

import java.util.*;

public class MultiHashMap<K,V> {
    private Map<K,List<V>> map = new HashMap<K,List<V>>();

    public int size() {
        return map.size();
    }

    public void put(K key, V value) {

```

```

    List<V> values = map.get(key);
    if (values == null) {
        values = new ArrayList<V>();
        map.put(key, values);
    }
    values.add(value);
}

public List<V> get(K key) {
    return map.get(key);
}
}

```

put 方法首先使用传入的 key 从 map 中提取一个列表。如果 key 没有对应的条目，该方法会构建一个新的、与类型 V 绑定的 ArrayList，并把这个列表放入到 map 中。无论如何，value 被加入到列表中。

和 HashMap 一样，类型参数列表包括两个类型参数，K 和 V。贯穿于 MultiHashMap 的定义，你会在预期见到类型名称（type name）的地方看到这些符号。例如，get 方法返回 V，而不是 Object。在类型声明中使用的每个类型参数符号，被称为裸类型变量（naked type variable）。◀ 513

在编译时，裸类型变量（naked type variable）的每次出现都被替换为对应类型参数的适当类型。在下一节“擦拭法”中，你会看到它实际是如何工作的。

在 map 字段成员变量（加粗显式）的定义中，你构造了一个 HashMap 对象并将它绑定为 <K, List<v>>。map 的 key 和 MultiHashMap 所绑定的 key（V）是相同的。map 的 value 是一个与类型 V（即 MultiHashMap 绑定的类型值）绑定的 List。

绑定参数（bind parameter）对应于类型参数（type parameter）。在测试的 setUp 方法中，包括了对 MultiHashMap 的实例化：

```
events = new MultiHashMap<Date, String>();
```

Date 类型对应于类型参数 K，而 String 类型对应于类型参数 V。这样，嵌入的 map 字段成员变量会被推演为 HashMap<Data, List<String>>——其中 key 为日期，value 为字符串的列表。

擦拭法

要实现对参数化类型的支持，Sun 有不止一种方法可以选择，来实现对参数化类型的支持。一种可能的方法是每种参数化类与类型的绑定，创建一个全新的类型定义。当绑定到一个类型时，源代码中裸类型变量（naked type variable）的每次出现，都会被替换为所绑定的类型。这种技术为 C++ 所使用。

例如，如果 Java 使用这种方法，将 MultiHashMap 绑定为 <Date, String> 将会导致下面的代码在幕后被创建：

```
// THIS ISN'T HOW JAVA TRANSLATES GENERICS:
```

```

package sis.util;

import java.util.*;

public class MultiHashMap<Date,String> {
    private Map<Date,List<String>> map =
        new HashMap<Date,List<String>>();

    public int size() {
        return map.size();
    }

    public void put(Date key, String value) {
        List<String> values = map.get(key);
        if (values == null) {
            values = new ArrayList<String>();
            map.put(key, values);
        }
        values.add(value);
    }

    public List<String> get(Date key) {
        return map.get(key);
    }
}

```

将 `MultiHashMap` 绑定到另一个对儿类型，例如 `<String, String>`，将会使编译器创建 `MultiHashMap` 的另一个版本。使用这种方式潜在地会使编译器创建许多 `MultiHashMap` 的变体类。

Java 使用一种不同方法，叫做“擦拭法”。不同于创建一个独立的类型定义，Java 擦拭了参数化类型的信息，并创建一个单一的等效类型。每个类型参数与一个称为它的“上限（upper bound）”的约束相关联，缺省是 `java.lang.Object`。客户端的绑定信息被擦去，并替换为适当的强制转型（cast）类型。`MultiHashMap` 类会被翻译为：

```

package sis.util;

import java.util.*;

public class MultiHashMap {
    private Map map = new HashMap();

    public int size() {
        return map.size();
    }

    public void put(Object key, Object value) {
        List values = (List)map.get(key);
        if (values == null) {
            values = new ArrayList();
            map.put(key, values);
        }
        values.add(value);
    }

    public List get(Object key) {

```

```

        return (List)map.get(key);
    }
}

```

了解参数化类型在幕后如何工作，对于能够理解并有效地使用它们非常重要。出于擦拭法的原因，Java 对参数化类型的使用有许多限制。我们将在本章中讨论这些限制。每种可能的范型实现方法都有自己的缺陷。Sun 选择擦拭法是为了尽可能地提供向后兼容性。

◀ 515

上限 (Upper Bound)

我们已经提到，每个类型参数都有一个缺省为 `Object` 的上限。你可以将类型参数限制为不同的上限。例如，你可能希望提供一个 `EventMap` 类，其中 `key` 必须被绑定为 `Date` 类型——无论是 `java.util.Date` 或者 `java.sql.Date` (`java.util.Date` 的子类)。一个简单的测试：

```

package sis.util;

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

public class EventMapTest extends TestCase {
    public void testSingleElement() {
        EventMap<java.sql.Date,String> map =
            new EventMap<java.sql.Date,String>();
        final java.sql.Date date =
            new java.sql.Date(new java.util.Date().getTime());
        final String value = "abc";
        map.put(date, value);

        List<String> values = map.get(date);
        assertEquals(value, values.get(0));
    }
}

```

`EventMap` 类本身没有不同的行为，只是额外限制了类型参数 `K`：

```

package sis.util;

public class EventMap<K> extends java.util.Date,V>
    extends MultiHashMap<K,V> {
}

```

你可以使用 `extends` 关键字来指定某个类型参数的上限。在这个实例中，`EventMap` 的类型参数 `K` 有一个 `java.util.Date` 的上限。使用 `EventMap` 的代码必须将 `key` 绑定为 `java.util.Date` 或者 `java.util.Date` 的子类（例如 `java.sql.Date`）。如果你试图违反这一点：

```

EventMap<String,String> map = new EventMap<String,String>();

```

◀ 516

你将会收到编译时的错误:

```
type parameter java.lang.String is not within its bound
EventMap<String,String> map = new EventMap<String,String>();
    ^
type parameter java.lang.String is not within its bound
EventMap<String,String> map = new EventMap<String,String>();
    ^
```

编译器在生成的代码中将裸类型变量 (naked type variable) 替换为上限类型。这给了你在范型类中调用裸类型对象 (naked type object) 更多特定方法的能力。

你需要这种能力, 从 EventMap 中取出某个日期之前的所有事件描述。在 EventMapTest 中编写下面的测试。

```
public void testGetPastEvents() {
    EventMap<Date,String> events = new EventMap<Date,String>();
    final Date today = new java.util.Date();
    final Date yesterday =
        new Date(today.getTime() - 86400000);
    events.put(today, "sleep");
    final String descriptionA = "birthday";
    final String descriptionB = "drink";
    events.put(yesterday, descriptionA);
    events.put(yesterday, descriptionB);
    List<String> descriptions = events.getPastEvents();
    assertTrue(descriptions.contains(descriptionA));
    assertTrue(descriptions.contains(descriptionB));
}
```

在 EventMap 中, 你可以假定类型 K 的对象是 Date 对象:

```
package sis.util;

import java.util.*;

public class EventMap<K extends Date,V>
    extends MultiHashMap<K,V> {
    public List<V> getPastEvents() {
        List<V> events = new ArrayList<V>();
        for (Map.Entry<K,List<V>> entry: entrySet()) {
            K date = entry.getKey();
            if (hasPassed(date))
                events.addAll(entry.getValue());
        }
        return events;
    }

    private boolean hasPassed(K date) {
        Calendar when = new GregorianCalendar();
        when.setTime(date);
        Calendar today = new GregorianCalendar();
        if (when.get(Calendar.YEAR) != today.get(Calendar.YEAR))
            return when.get(Calendar.YEAR) < today.get(Calendar.YEAR);
    }
}
```

```

        return when.get(Calendar.DAY_OF_YEAR) <
            today.get(Calendar.DAY_OF_YEAR);
    }
}

```

为了完成这一点，你必须在 `MultiHashMap` 中添加一个方法，从封装后的 `map` 中返回 `entrySet`：

```

protected Set<Map.Entry<K,List<V>>> entrySet() {
    return map.entrySet();
}

```

`entrySet` 方法返回一个绑定到 `Map.Entry` 类型的集合。`Set` 中的每个 `Map.Entry` 对象，都继而绑定为 `key` 类型 (`K`)、以及一个 `value` 类型的列表 (`List<V>`)。

通配符 (Wildcard)

有些时候，你所编写的方法并不关心同哪种类型参数绑定。假定你需要一个实用方法，通过将元素串连到一个列表中、并用新行将每个元素的可打印字符串表示断开，来创建一个的字符串。`StringUtilTest` 的 `testConcatenateList` 方法演示了这种需要：

```

package sis.util;

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

public class StringUtilTest extends TestCase {
    ...
    public void testConcatenateList() {
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");

        String output = StringUtil.concatenatate(list);

        assertEquals(String.format("a%n b%n"), output);
    }
}

```

在 `StringUtil` 的 `concatenatate` 方法中，你将每个列表元素的字符串表示追加到一个 `StringBuilder` 中。你可以得到任何对象的字符串表示，只要它提供了 `toString` 方法，而不需了解或关心其类型。这样，你希望能够将一个绑定到任意类型的 `List` 作为参数，传递给 `concatenatate`。

你可能会考虑将 `list` 的参数绑定到 `Object`：

```

// this won't work
public static String concatenatate(List<Object> list) {
    StringBuilder builder = new StringBuilder();
    for (Object element: list)

```

```

        builder.append(String.format("%s%n", element));
    }
    return builder.toString();
}

```

通过将 `list` 绑定到 `Object`，你会限制它只能保存 `Object` 类型的对象，而且不能是 `Object` 的任何子类。你不能将一个 `List<String>` 的引用赋值给 `List<Object>` 的引用。如你能够做到的话，客户端代码可以通过这个 `List<Object>` 的引用，将一个 `Object` 加入到 `list` 中（`String` 类型）。然后当代码使用 `List<String>` 引用从 `list` 取出对象时，会意想不到地得到一个 `Object`。

相反地，Java 允许你使用一个通配符（`?`）来表示任意可能的类型：

```

package sis.util;

import java.util.*;

public class StringUtil {
    ...
    public static String concatenate(List<?> list) {
        StringBuilder builder = new StringBuilder();
        for (Object element: list)
            builder.append(String.format("%s%n", element));
        return builder.toString();
    }
}

```

在 `concatenate` 方法体中，你不能直接使用 `?` 作为裸类型变量（`naked type variable`）。但是因为 `list` 可以包含任意类型的对象，你可以在 `for-each` 循环中将它的每个元素赋值给一个 `Object` 引用。

此外，你可以使用 `extends` 子句限制通配符的上限。

对第二个字符串实用方法，你需要能够串连一个数字的列表，无论它们是 `BigDecimal` 对象或者是 `Integer` 对象。一些测试会得出十进制输出和整数输出之间的细微差异。

519

```

public void testConcatenateFormattedDecimals() {
    List<BigDecimal> list = new ArrayList<BigDecimal>();
    list.add(new BigDecimal("3.1416"));
    list.add(new BigDecimal("-1.4142"));

    String output = StringUtil.concatenateNumbers(list, 3);
    assertEquals(String.format("3.142%n-1.414%n"), output);
}

public void testConcatenateFormattedIntegers() {
    List<Integer> list = new ArrayList<Integer>();
    list.add(12);
    list.add(17);

    String output = StringUtil.concatenateNumbers(list, 0);
    assertEquals(String.format("12%n17%n"), output);
}

```

StringUtil 的实现：

```

public static String concatenateNumbers(
    List<? extends Number> list, int decimalPlaces) {
    String decimalFormat = "%." + decimalPlaces + "f";
    StringBuilder builder = new StringBuilder();
    for (Number element: list) {
        double value = element.doubleValue();
        builder.append(String.format(decimalFormat + "%n", value));
    }
    return builder.toString();
}

```

你需要引入 `java.math.*` 来让上面的代码得以编译通过。

`concatenateNumbers` 中的 `list` 参数声明，指定了它是一个绑定到 `Number` 或其子类的 `List.concatenatenumbers` 中的代码可以将 `list` 中的每个元素赋值给一个 `Number` 类型的引用。

使用通配符的隐含问题 (Implication)

对引用使用通配符的缺点是，你不能调用类型参数对象中的方法。例如，假定你要创建一个 `pad` 的实用方法，能够以 `n` 次将一个元素 `n` 次添加到列表的尾部。

```

package sis.util;

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

public class ListUtilTest extends TestCase {
    public void testPad() {
        final int count = 5;
        List<Date> list = new ArrayList<Date>();
        final Date element = new Date();
        ListUtil.pad(list, element, count);
        assertEquals(count, list.size());
        for (int i = 0; i < count; i++)
            assertEquals("unexpected element at " + i,
                element, list.get(i));
    }
}

```

`pad` 方法声明了 `list` 参数是一个可以和任何类型绑定的 `List`：

```

package sis.util;

import java.util.*;

public class ListUtil {
    public static void pad(List<?> list, Object object, int count) {
        for (int i = 0; i < count; i++)
            list.add(object);
    }
}

```

你会收到下面的错误消息，表明编译器无法辨识适当的 `add` 方法。
cannot find symbol

```
symbol : method add(java.lang.Object)
location: interface java.util.List<?>
    list.add(object);
    ^
```

问题是通配符(?)指示了一个未知的类型。假定 List 被绑定到 Date 类型:

```
List<Date> list = new ArrayList<Date>();
```

然后你试图将一个 String 传入到 pad 方法:

```
ListUtil.pad(list, "abc", count);
```

pad 方法对传入对象的特定类型并不知情,而且也不知道列表所绑定的类型。它无法保证客户端代码不会试图破坏列表的类型安全性。Java 就是不让你这么做。

521

即使你为通配符指定一个上限,还是有一个问题。

```
public static void pad(
    List<? extends Date> list, Date date, int count) {
    for (int i = 0; i < count; i++)
        list.add(date); // this won't compile
}
```

问题是,客户端可能将列表绑定到 java.sql.Date:

```
List<java.sql.Date> list = new ArrayList<java.sql.Date>();
```

因为 java.util.Date 是 java.sql.Date 的父类,你无法把它添加到一个限制只能保存 java.sql.Date 或其子类的列表中。由于擦拭法的原因,Java 无法判断某个给定的操作是否是安全的,因此它只有全部禁止它们。一般的经验法则是,只有从数据结构读取时,你可以使用有界的(bounded)通配符。

范型方法

你该如何解决这个问题呢?你可以将 pad 方法声明为一个范型方法。就像你为类指定类型参数那样,你可以指定存在于方法范围的类型参数:

```
public static <T> void pad(List<T> list, T object, int count) {
    for (int i = 0; i < count; i++)
        list.add(object);
}
```

编译器可以根据传递给 pad 的参数,提取或推断出 T 的类型。它使用能够从中推断出的最确切的类型。你的测试现在应该可以通过了。

范型方法类型参数同样可以有上限。

当你的方法参数和其他参数或返回类型存在依赖时,你会需要使用范型方法。否则,你应该更倾向于使用通配符。在 pad 方法的声明中,object 参数的类型依赖于 list 参数的类型。

通配符捕获 (Wildcard Capture)

这种引入范型方法来解决上述问题的技术，被称为通配符捕获 (wildcard capture)。另一个示例是一个将列表中的元素从头至尾交换一遍的简单方法。

522

```
public void testWildcardCapture() {
    List<String> names = new ArrayList<String>();
    names.add("alpha");
    names.add("beta");
    inplaceReverse(names);
    assertEquals("beta", names.get(0));
    assertEquals("alpha", names.get(1));
}

static void inplaceReverse(List<?> list) {
    int size = list.size();
    for (int i = 0; i < size / 2; i++) {
        int opposite = size - 1 - i;
        Object temp = list.get(i);
        list.set(i, list.get(opposite));
        list.set(opposite, temp);
    }
}
```

通配符捕获涉及调用一个新的范型方法，swap：

```
public void testWildcardCapture() {
    List<String> names = new ArrayList<String>();
    names.add("alpha");
    names.add("beta");
    inplaceReverse(names);
    assertEquals("beta", names.get(0));
    assertEquals("alpha", names.get(1));
}

static void inplaceReverse(List<?> list) {
    int size = list.size();
    for (int i = 0; i < size / 2; i++)
        swap(list, i, size - 1 - i);
}

private static <T> void swap(List<T> list, int i, int opposite) {
    T temp = list.get(i);
    list.set(i, list.get(opposite));
    list.set(opposite, temp);
}
```

通过这样做，你可以给通配符一个名字。

Super

上限通配符，是你通过使用 `extends` 来指定的，在从一个数据结构中读取时非常有用。

523

你可以使用下限通配符（lower-bounded wildcard）来支持向数据结构的写入。



你希望能从已有的 `MultiHashMap` 创建一个新的 `MultiHashMap`。新 `map` 是已有 `map` 的一个子集。你可以通过为已有 `MultiHashMap` 中的 `value` 应用一个过滤器来得到这个子集。这里演示的测试创建了一个会议的多重映射（multimap）。一个会议可以是一次性的事件，或者是周期性的。你希望创建一个新的多重映射，其中只包括在周一发生的事件。

再者，原先的会议多重映射包括 `java.sql.Date` 对象。它也许是直接从数据库应用中加载的。你希望新的多重映射包括 `java.sql.Date` 对象。

```
public void testFilter() {
    MultiHashMap<String, java.sql.Date> meetings =
        new MultiHashMap<String, java.sql.Date>();

    meetings.put("iteration start", createSqlDate(2005, 9, 12));
    meetings.put("iteration start", createSqlDate(2005, 9, 26));
    meetings.put("VP blather", createSqlDate(2005, 9, 12));
    meetings.put("brown bags", createSqlDate(2005, 9, 14));

    MultiHashMap<String, java.util.Date> mondayMeetings =
        new MultiHashMap<String, java.util.Date>();
    MultiHashMap.filter(mondayMeetings, meetings,
        new MultiHashMap.Filter<java.util.Date>() {
            public boolean apply(java.util.Date date) {
                return isMonday(date);
            }
        });

    assertEquals(2, mondayMeetings.size());
    assertEquals(2, mondayMeetings.get("iteration start").size());
    assertNull(mondayMeetings.get("brown bags"));
    assertEquals(1, mondayMeetings.get("VP blather").size());
}

private boolean isMonday(Date date) {
    Calendar calendar = GregorianCalendar.getInstance();
    calendar.setTime(date);
    return calendar.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY;
}

private java.sql.Date createSqlDate(int year, int month, int day) {
    java.util.Date date = DateUtil.createDate(year, month, day);
    return new java.sql.Date(date.getTime());
}
```

你可以在 `MultiHashMap` 的 `filter` 方法中使用下限通配符，以允许将 `java.sql.Date` 值转换为 `java.util.Date` 值。在实现中，`filter` 方法的 `target` 参数是 `MultiHashMap`，其 `value` 类型（V）

有一个下限。它被表示为? Super V。这意味着目标 MultiHashMap 的 value 类型 (V)，可以是 V 或者为 V 的父类型。会议的那个示例可以工作，是因为 java.util.Date 是 java.sql.Date 的父类型。

524

```
...
public class MultiHashMap <K,V> {
    private Map<K, List<V>> map = new HashMap<K, List<V>>();
    ...
    public interface Filter<T> {
        boolean apply(T item);
    }

    public static <K,V> void filter(
        final MultiHashMap<K, ? super V> target,
        final MultiHashMap<K, V> source,
        final Filter<? super V> filter) {
        for (K key : source.keys()) {
            final List<V> values = source.get(key);
            for (V value : values)
                if (filter.apply(value))
                    target.put(key, value);
        }
    }
}
```

附加限界

在一个 extends 子句中指定不止一种类型的附加限界，是有可能的。虽然第一个限制可以是一个类或接口，之后的限制必须是接口类型。例如：

```
public static
<T extends Iterable&Comparable<T>> void iterateAndCompare(T t)
```

通过使用附加限界，你限制了传入的参数要实现多于一个接口。在本例中，传入的对象必须实现 Iterable 和 Comparable 接口。这通常并不是你想做的；引入附加限界很大程度上是为了解决向后兼容性（见下文）。

如果一个方法需要一个对象具有两种不同类型的行为，这意味着方法要对象完成两种不同的操作。在大多数情况下，这违反了单一职责原则（Single-Responsibility Principle），这一原则不仅适用于类也适用于方法。当然，凡事总是有例外，但是在出于这个原因使用附加限界之前，看看你能否把这个方法分解。

525

使用附加限界更正统的理由，是由集合类中的代码演示的。集合类包括静态的实用方法来操作集合对象；其中包括名为 max 的方法，返回集合中最大的元素。在 Java 的早期版本中，max 的原型特征是：

```
public static Object max(Collection c)
```


传入的集合应该可以保存任意类型的对象，但是 `max` 方法依赖集合中所保存的对象实现了 `Comparable` 接口。在使用范型的方案中，初步的意图是集合必须保存 `Comparable` 的对象。

```
public static
<T extends Comparable<? super T>> T max(Collection<? extends T> c)
```

解释一下，该原型特征说明的是：`max` 接收一个任意类型对象的集合，但这个类型必须实现了 `Comparable` 接口，且这个类型必须是在集合中存储的元素所绑定的类型或父类。问题是在擦拭之后，结果会产生一个不同的 `max` 方法原型特征：

```
public static Comparable max(Collection c)
```

不同于产生一个 `Object` 引用，`max` 现在将会返回一个 `Comparable` 引用。原型特征的改变，会打破现有使用 `max` 方法编译后的代码，破坏了兼容性。使用附加限界能有效地解决这个问题：

```
public static
<T extends Object&Comparable<? super T>>
T max(Collection<? extends T> c)
```

根据 J2SE 规范，一个类型变量会被擦除为它最左边的限界——在本例中为 `Object`，而不是 `Comparable`。`max` 方法现在返回 `Object` 引用，同时也定义了附加的限制，即收集的对象必须实现适当的 `Comparable` 接口。

原始类型 (Raw Type)

如果你从事于一个旧有的系统，是用 J2SE 1.4 或更早版本编写的，它所使用的集合类并不支持参数化。你会看到许多这样的代码：

```
List list = new ArrayList();
list.add("a");
```

在 J2SE 5.0 下，你依然可以使用这样的代码。当你使用范型类型而不把它绑定到一个类型参数时，我们将这种类型称为原始类型。使用原始类型先天不是类型安全的：你可以将任何类型的对象添加到一个原始集合 (raw collection) 中，在取回一个意外类型的对象时，会遇到运行时异常。这也是 Sun 为什么引入参数化类型的主要原因。

编译器遇到任何这样的非安全性操作，会提出警告。使用缺省的编译器选项，上面向原始 `ArrayList` 添加的两行代码，会产生下面的编译器消息：

```
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

按警告消息所言，使用 VM 的 `-Xlint:unchecked` 开关重新编译：

```
javac -source 1.5 -Xlint:unchecked *.java
```

使用 Ant，可以在 `javac` 任务中提供一个嵌套的额外元素：

```
<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" />
<compilerarg value="-Xlint:unchecked"/>
</javac>
</target>

```

重新编译时，编译器会呈现给你每个 unchecked 警告相关代码更详细的细节。

```

warning: [unchecked] unchecked call to add(E) as a member of the raw type
java.util.List
    list.add("a");
        ^

```

只要 Java 编译器无法保证对某个参数化类型操作的类型安全性，它就会产生 unchecked 警告。如果你正在对现有的 1.4 或更老版本的应用进行开发，你大概会收到大量的警告。而它也许不可能或不值得立刻解决所有警告，你永远都不应轻视警告。如果你修改了遗留代码中的某些部分，尝试引入泛型类型。你应该对由新增代码引起的警告特别留意。

527

Checked 集合

假定你正在从事于工作在忙于遗留的代码上。事实上，你所遇到的大部分系统，都包括许多为老版本 Java 编写的代码。这也是 Java 开发的事实。我依然会碰到弥漫着使用 Vector 和 Hashtable 的系统——即使 Sun 推荐使用集合框架类（List/ArrayList 等），而不是这些 Java 1.2 中的遗留类。它们已经推出有六年之久了！

在未来一段时间，你大概还会遇到原始集合（raw collection）。你可以使用 checked 包装器（wrapper），快速地为你的代码添加某些类型安全性。假定你现有的混乱系统创建了一个列表，用来保存 Integer 包装器对象：

```
List ages = new ArrayList();
```

在代码的其他地方，在远处某个类的一个方法中在嵌套在遥远的、其他类的某个方法中，另一个莽撞的开发者添加了一行新的代码：

```
ages.add("17");
```

但进而，在另一个类中的代码负责从集合中提取出年龄的值：

```
int age = ((Integer)ages.get(0)).intValue();
```

Oops（糟糕）！遗留代码中的这一行会产生一个 ClassCastException，因为 ages 中的第一个元素是 String 而不是 Integer。当然，你可以很快找到这个问题，但是你在在这个过程中已经浪费了宝贵的时间。

最好的方案是，在提到的这三个类中将 ages 的引用参数化。然后，第二个开发者将无法编译插入字符串的代码。但是更改这些类可能超出你的控制，或者受到影响的类有几打儿而远远

不止三个。²

Sun 为 Collections 类引入了一套新的方法来帮助解决这个类型安全性的问题。这些方法创建 checked 包装器对象。一个 checked 包装器对象会将调用委托给实际的集合对象，与禁止修改（unmodifiable）和同步包装器非常类似。一个 checked 包装器会确保传递给集合的对象是适当的类型，并且阻止不适当的对象被插入到集合中。

使用 checked 集合，至少可以限制在前面错误处产生的异常。换言之，添加错误数据的代码会产生异常，而不是取出它的代码。这会使调试的工作更快捷。你可以在一个地方进行改动：

```
List ages = Collections.checkedList(new ArrayList(), Integer.class);
```

当 Java VM 执行这行代码时，你会收到这样的异常：

```
java.lang.ClassCastException: Attempt to insert class java.lang.String element
into collection with element type class java.lang.Integer
```

一个语言的单元测试完整地演示了这一点：

```
public void testCheckedCollections() {
    List ages =
        Collections.checkedList(new ArrayList(), Integer.class);
    try {
        ages.add("17");
        fail("expected ClassCastException on invalid insert");
    }
    catch (ClassCastException success) {
    }
}
```

Checked 集合并非魔术般神奇。checkedList 的第一个参数是你创建的 ArrayList；它被绑定为 Integer 类型。第二个参数是 Integer 类型的 Class 引用。在你每次调用 add 方法时，它使用这个 Class 引用来判断传入的参数是否为这个类型。如果不是，它会抛出一个 ClassCastException。

Checked 包装器除了要指定一个绑定类型（<Integer>），还需要冗余地传入一个类型（Integer.class）的引用。这个需求是由于擦拭法的原因：类型绑定的信息在运行时对列表对象是无法获得的。你可以在自己的参数化类型中封装 unchecked 包装器，但是你需要让客户端传入 Class 引用。

Collection 类为 Connection、List、Map、Set、SortedMap 和 SortedSet 等类型，提供了 checked 集合包装器。

即使你编写 J2SE 5.0 的代码，使用 checked 集合也可以保护你，免受在你影响范围外的、松散且潦草的开发者所带给你的麻烦。某些牛仔开发者非常乐于颠覆规则，无论何时只要他们能够做到的话。就参数化类型的情况而言，Java 给了他们空间。

你可以将一个参数化类型的对象强制转型为一个原始类型。这可以让你把任何类型的对象保存在集合中。你会收到编译警告，但是你可以忽略这些警告。这样做要自己承担风险。结果

² 而且当然，这么多类中没有一个是具有测试用例。遗留系统的绝大多数没有测试用例。

常常是灾难性的。这种策略可以归入“禁止行为”的类别。

◀ 529

在 5.0 的代码中使用 `checked` 集合将帮助你解决这个问题。抛出的异常将会来自于作祟的代码，让你能够精确地定位引起麻烦的部分。

数组（Array）

你不能创建与参数化类型绑定的数组：

```
List<String>[] names = new List<String>[100]; // this does not compile
```

这样的尝试会产生下面的编译错误：

```
arrays of generic types are not allowed
```

同样的，问题在于你可以简单地将参数化类型的引用赋值给另一种类型的引用。然后你可以添加一个不合适的对象，进而在尝试取出时导致一个 `ClassCastException`：

```
// this does not compile
List<String>[] namesTable = new List<String>[100];
Object[] objects = (Object[])namesTable;
List<Integer> numbers = new ArrayList<Integer>();
numbers.add(5);
objects[0] = numbers;
String name = namesTable[0].get(0); // would be a ClassCastException
```

Java 允许你创建无限界的参数化类型的数组（也就是你只使用通配符（?）的地方）：

```
public void testArrays() {
    List<?>[] namesTable = new List<?>[100];
    Object[] objects = (Object[])namesTable;
    List<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    objects[0] = numbers;
    try {
        String name = (String)namesTable[0].get(0);
    }
    catch (ClassCastException expected) {
    }
}
```

主要的差别是，你必须明确可能由强制转型导致的潜在问题，因为 `List` 是一个未知类型对象的集合。

◀ 530

额外的局限

由于擦拭法的原因，参数化类型的对象没有关于绑定类型的信息。这对于你可以和不可以做什么，产生了许多隐含的影响。

你不能使用裸类型变量 (naked type variable) 创建一个新的对象:

```
package util;

import java.util.*;

public class NumericList<T extends Number> {
    private List<T> data = new ArrayList<T>();

    public void initialize(int size) {
        data.clear();
        T zero = new T(0); // does not compile!
        for (int i = 0; i < size; i++)
            data.add(zero);
    }
}
```

这段代码会产生下面的编译错误:

```
unexpected type
found   : type parameter T
required: class
    T zero = new T(0);
           ^
```

擦拭意味着裸类型变量擦去了其上界, 在本例中即为 `Number`。在大多数情况下, 上限是一个抽象类, 例如 `Number` 或 `Object` (缺省的), 因此创建这样类型的对象将毫无用处。Java 简单的禁止了这种操作。

你可以强制转型一个裸类型变量, 但是出于同样的原因, 这么做通常没什么用处。你不能将一个裸类型变量作为 `instanceof` 操作符的目标。

你可以在范型的静态方法中使用类型变量。但是你不能在静态变量、静态方法、或静态内嵌类中使用由外围类定义的类型变量。由于擦拭法, 所有实例都只共享一个类定义, 而无论他们绑定的类型是什么。这个类定义是在编译时创建的。这意味着共享的静态元素将不再工作。不同参数化类型绑定的各个客户端, 都会预期具有同他们指定类型对应的静态成员。

531

反射

反射包也被翻新了, 来为参数化类型和方法提供参数信息。例如, 你可以调用一个 `Class` 类的 `getTypeParameters` 方法来取得一个 `TypeParameter` 对象数组; 每个 `TypeParameter` 对象都为你提供了足够的信息来重新构造这些类型参数。

为了支持这些改变, Sun 改动了 Java 的字节码规范。`Class` 文件现在要保存关于类型参数的附加信息。最重要的是, `Class` 类也被修改为一个参数化的类型, `Class<T>`。下面的赋值是有效的:

```
Class<String> klass = String.class;
```

如果你对如何来使用它有兴趣，查看一下 `CheckedCollection` 类中的代码。它是 `java.util.Collections` 中的一个静态内联类。

反射的修改，为你提供了有关参数化类型和方法声明的信息。你无法从反射中得到的有关绑定类型变量的信息。如果你将一个 `ArrayList` 绑定到 `String`，因为擦拭法，该信息是不为 `ArrayList` 对象所知的。因此，反射没有办法将它提供给你。如果能编写下面的代码，将是一件很爽的事：

```
public class MultiHashMap<K,V> {
    ...
    public Class<V> getKeyType() {
        return V.class; // this will not work!
    }
}
```

但是，它就是无法工作。

最后的注意事项

正如你已经看到的，了解擦拭法如果工作，是能够理解和实现参数化类型的关键。从客户端的角度来说，使用参数化类型相对容易，并且可以得到类型安全的显著益处。从创建参数化类型的开发者角度来说，这可能是—个相当复杂的冒险历程。

如果你发现自己挣扎于如何理解或定义一个参数化类型，可以提取其定义，并将参数化类型的使用简化为它擦拭后的等价物。还有，查看 J2SE 5.0 源代码中对参数化类型的某些使用。集合框架类和接口，例如 `Map` 和 `HashMap`，提供了许多好的示例。如果需要更复杂的示例，参考 `java.util.Collections`。

◀ 532

练习

1. 创建一个新的参数化集合类型——`Ring`。`Ring` 是一个环型的列表，它维护当前元素的信息。客户端可以取得并删除当前元素。`Ring` 必须支持从当前位置单步地前进或后退。`add` 方法在当前元素之后添加元素。`Ring` 类支持客户端使用 `for-each` 从当前位置开始遍历所有元素。任何因环空而导致的无效操作，`Ring` 类支持抛出适当的异常。

不要使用另一个数据结构来实现环（例如，`java.util.LinkedList`）。使用内嵌的节点类，创建你自己的链表结构。每个节点、或表项，应该包括三个内容：被加入的数据元素、环中下一个节点的引用，以及环中前一个节点的引用。

◀ 533

断言与注解

J2SE 5.0 引入了一种被称为注解（annotation）的新功能。注解是一种元编程（meta-programming）的功能，允许你使用定义的任意标记（tag）来标注代码。标记（通常）对 Java 编译器或其运行时本身而言，没有任何意义。然而，其它工具可以解释这些标记。注解可能对下列工具是有用的，包括 IDE、测试工具、剖析工具以及代码生成工具。

在本课中，你将基于 Java 的注解功能，开始建立一个与 JUnit 相似的测试工具。和 JUnit 一样，这个测试工具也需要允许让开发者指定断言。与 JUnit 编写的断言方法不同，你将学习如何使用 Java 内建的断言功能。

你将学习到：

- 断言
- 注解及注解类型
- 注解的保留政策（retention policy）
- 注解目标
- 成员-值对儿（member-value pair）
- 注解成员的缺省值
- 允许的注解成员类型
- 包的注解
- 注解的兼容性考虑

断言

你已经看到了如何使用 JUnit API 中定义的 `assert` 方法。这些 `assert` 方法在 `junit.framework.Assert` 中定义，在适当的时候抛出一个 `AssertFailedError`。

Java 支持类似的断言功能，你可以使用一个 VM 标志来打开或关闭它。断言语句以一个 `assert` 关键字开始。之后跟随一个条件；如果条件失败，Java 抛出一个 `AssertionError` 类型的 `RuntimeException`。可选地，你可以通过在条件后面跟着一个冒号和 `String` 消息，来提供一个要在 `AssertionError` 中保存中的消息。一个示例：

```
assert name != null : "name is required";
```

不含这个消息时：

```
assert name != null;
```

断言缺省是被关闭的。当断言被关闭时，VM 会忽略 `assert` 语句。这会防止 `assert` 语句严重地影响应用的性能。要打开断言：

```
java -ea MainClass
```

或者，更明确地：

```
java -enableassertions MainClass
```

这两种语句都可以为你的代码开启断言，但是不会影响 Java 库中的类。你可以使用 `-da` 或 `-disableassertions` 关闭断言。要打开或关闭系统类的断言，使用 `-enablesystemassertions` (或 `esa`) 或 `-disableassertions` (或 `dsa`)。

Java 允许你以一种更小粒度级别来打开或关闭断言。你可以为任何独立的类、任何包及其子包、或缺省包，打开或关闭断言。

例如，你可以打开一个包的断言，但是只对其中的一个类例外：

```
java -ea:sis.studentinfo... -da:sis.studentinfo.Session SisApplication
```

这里演示的示例，会打开 `sis.studentinfo` 包的断言，但 `Session` 例外。省略号的意思是，`java` 会额外打开 `sis.studentinfo` 所有子包的断言，例如 `sis.studentinfo.ui`。你可以只使用省略号来表示缺省包。

536

断言打开或关闭的顺序，是按照在 `java` 命令行中出现的次序，从左至右。

assert 语句 vs. JUnit 的 Assert 方法

JUnit 是在 Java 断言机制之前，也就是 Sun 引入 J2SE 1.4 版的时候，建立的。今天，JUnit 可以使用断言机制来重写。但是因为 JUnit 的使用已经很普及了，如果重写 JUnit，会为诸多软件厂商带来非常繁重的任务。而且，JUnit 提供的 `junit.framework.Assert` 方法，表达力稍稍丰富些。`assertEquals` 方法内建提供了一种改进的错误消息。

在进行测试驱动开发时，你总需要一个基于断言的框架。TDD 中使用的测试，与按契约设计 (design by contract, 在第 6 课中被称为子契约 (subcontracting) 原则) 相类似。断言提供了前置条件、后置条件以及恒定式 (invariant)。如果你没有进行 TDD，你可能选择通过直接在产品代码引入 `assert` 语句，来加固系统的质量。

Sun 建议使用断言作为应用故障的预防措施。例如，一种可能的应用是检查 `public` 方法的参数，如果客户端代码传入了一个 `null` 引用则会失败。缺点是如果你关闭断言，应用可能无法继续正常运行。出于相似的原因，Sun 建议你不要在断言中包含会产生副作用的代码 (也就是改变系统状态的代码)。

不过，没有什么可以阻止你这样做。如果你已经充分掌控了应用如何执行 (而且你也应该)，那么你可以确保应用在断言打开时总是被正确地初始化了。

Java `assert` 关键字的主要价值似乎是调试的一种辅助手段。适当插播的 `assert` 语句，可以将代码中的问题警示给你。假定某人在调用某个方法时传入了一个 `null` 引用。如果没有防护性断言，你可能于良久之后才能在执行时收到一个 `NullPointerException`，而这时已经离设置引用的地方相距甚远。调试这种情况可能是非常耗时的。

你还可以使用断言来帮助建立一个 TDD 工具来替代 JUnit。在本章的练习中，你将使用断言来开始构建这样一个框架。

注解 (Annotation)

你已经看到了 Java 内建支持的注解的许多示例。在第 2 课中，你学习了如果将一个方法标注为 `@deprecated`，意味着这个方法最终会从这个类的公共接口中删除。`@deprecated` 注解在技术上讲并不是 Java 语言的一部分。不过，它是一个编译器可以解释的标记 (tag)，并基于此适时地打印一条警告消息。¹

537

你已经看到了 `javadoc` 标记的例子了，在 `javadoc` 注释 (comment) 中嵌入的注解，可以用来生成 Java API 文档的 web 页面。`javadoc.exe` 程序会完整读取你的 Java 源文件，分析并解释需要包含在 web 输出中的 `javadoc` 标记。

而且，在第 9 课中，你学习了如何使用 `@override` 标记来表明你要覆写一个父类的方法。

你还可以建立你自己的注解标记，无论出于什么你发现有用的目的。例如，你可能希望能够标注方法的更改注释：

¹ `@deprecated` 在 Java 语言的早期，且 Sun 正式引入注解支持之前就已经存在了。但是它的作用就像其他编译器级别的注解类型一样。

```
@modified("JJL", "12-Feb-2005")
public void cancelReservation() {
    // ...
}
```

然后，你可以建立一个工具（可能是一个 Eclipse 的插件）来让你快速查阅所有的更改注释，可能是以姓名的缩写排序。

当然，你可以通过强制让开发者提供符合某种标准格式的、结构化的注释来达到同样的结果。然后你可以编写代码来解析整个源文件，来查找这样的注释。不过，Java 中对自定义注解类型的支持，具有很大的优势。

首先，Java 会在编译时验证注解。不可能引入拼写错误或格式错误的注解。第二，你可以使用反射的能力来方便地获取注解信息，而不必编写解析代码。第三，你可以限制注解只应用于某种特定的 Java 元素。例如，你可以强制@modified 标记只能用于方法。

建立一个测试工具



在本课中，你将建立一个类似 JUnit 的测试工具。我们将这个工具称为 TestRunner，因为它是主要的负责执行测试的类。这个工具是基于文本的。你可以从 Ant 中执行它。从技术上讲，你不必使用 TDD 方法来建立一个测试工具，因为它并不打算作为产品代码。但是，没有什么条条框框说你不能为此编写测试。我们就会这么做。

538

JUnit 需要测试类从 junit.framework.TestCase 中派生。在 TestRunner 中，你将使用一种与继承不同的机制：使用 Java 注解来标注某个类为测试类。

万事开头难，但是使用 Ant 可以帮助你。你可以设置一个 Ant 目标作为测试的用户界面。如果并非所有的测试都通过，你可以进行某些设置以使 Ant 构建失败，在这种情况下你将会看到“BUILD FAILED”消息。否则，你将看到“BUILD SUCCESSFUL”消息。

TestRunnerTest

在 TestRunnerTest 中的第一个测试，singleMethodTest，衍生了定义在同一源文件中的第二个类 SingleMethodTest。SingleMethodTest 提供了一个最终会通过的、空的测试方法，同在 JUnit 中一样。

目前为止，你还不需要注解。你将测试类 TestRunnerTest.class 的引用，传递给 TestRunner 的一个实例。TestRunner 可以假设这个参数是一个只包括测试方法的测试类。

要产生测试的失败结果，你可以使用 Java 的 `assert` 功能（在本课第一部分描述）。需要牢记的是，你必须在执行 Java VM 时打开断言选项；否则，Java 将会忽略它们。

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
    public void singleMethodTest() {
        TestRunner runner = new TestRunner(SingleMethodTest.class);

        Set<Method> testMethods = runner.getTestMethods();
        assert 1 == testMethods.size() : "expected single test method";

        Iterator<Method> it = testMethods.iterator();
        Method method = it.next();

        final String testMethodName = "testA";
        assert testMethodName.equals(method.getName()) :
            "expected " + testMethodName + " as test method";
        runner.run();
        assert 1 == runner.passed() : "expected 1 pass";
        assert 0 == runner.failed() : "expected no failures";
    }

    public void multipleMethodTest() {
        TestRunner runner = new TestRunner(MultipleMethodTest.class);
        runner.run();

        assert 2 == runner.passed() : "expected 2 pass";
        assert 0 == runner.failed() : "expected no failures";

        Set<Method> testMethods = runner.getTestMethods();
        assert 2 == testMethods.size() : "expected single test method";

        Set<String> methodNames = new HashSet<String>();
        for (Method method: testMethods)
            methodNames.add(method.getName());

        final String testMethodNameA = "testA";
        final String testMethodNameB = "testB";

        assert methodNames.contains(testMethodNameA):
            "expected " + testMethodNameA + " as test method";
        assert methodNames.contains(testMethodNameB):
            "expected " + testMethodNameB + " as test method";
    }
}

class SingleMethodTest {
    public void testA() {}
}

class MultipleMethodTest {
    public void testA() {}
}
```

```

    public void testB() {}
}

```

第二个测试，multipleMethodTest，稍稍有些乱。为了创建它，我复制了第一个测试并修改了部分细节。问题是，只要你试图在 TestRunnerTest 中提取一个公用的实用方法，TestRunner 类就会假定它是一个测试方法而试图执行它。解决方案是引入注解来标注并区别测试方法。

TestRunner

首先，让我们看看 TestRunner 的初始实现。

```

package sis.testing;

import java.util.*;
import java.lang.reflect.*;

class TestRunner {
    private Class testClass;
    private int failed = 0;
    private Set<Method> testMethods = null;

    public static void main(String[] args) throws Exception {
        TestRunner runner = new TestRunner(args[0]);
        runner.run();
        System.out.println(
            "passed: " + runner.passed() + " failed: " + runner.failed());
        if (runner.failed() > 0)
            System.exit(1);
    }

    public TestRunner(Class testClass) {
        this.testClass = testClass;
    }

    public TestRunner(String className) throws Exception {
        this(Class.forName(className));
    }

    public Set<Method> getTestMethods() {
        if (testMethods == null)
            loadTestMethods();
        return testMethods;
    }

    private void loadTestMethods() {
        testMethods = new HashSet<Method>();
        for (Method method: testClass.getDeclaredMethods())
            testMethods.add(method);
    }

    public void run() {
        for (Method method: getTestMethods())
            run(method);
    }
}

```

```

private void run(Method method) {
    try {
        Object testObject = testClass.newInstance();
        method.invoke(testObject, new Object[] {});2
    }
    catch (InvocationTargetException e) {
        Throwable cause = e.getCause();
        if (cause instanceof AssertionError)
            System.out.println(cause.getMessage());
        else
            e.printStackTrace();
        failed++;
    }
    catch (Throwable t) {
        t.printStackTrace();
        failed++;
    }
}

public int passed() {
    return testMethods.size() - failed;
}

public int failed() {
    return failed;
}
}

```

541

如果你对理解 `run (Method)` 方法有些困难，请参见第 12 课中有关反射的讨论。`run` 方法的基本流程是：

- 创建测试类的一个新实例。这个步骤假定测试类有一个无参数的构造函数。
- 使用新的测试类实例和一个空的参数列表，`invoke` 这个方法（作为参数传入的）。
- 如果 `invoke` 调用失败，从抛出的 `InvocationTargetException` 中提取出原因：通常原因应该是 `AssertionError`。Java 在 `assert` 语句失败时抛出一个 `AssertionError`。

`TestRunner` 提供了两个构造函数。一个接收 `Class` 对象，而现在只在 `TestRunnerTest` 中使用。第二个构造函数接收一个 `String` 的类名，然后使用 `Class.forName` 来加载对应的类。你可以从 `main` 方法中调用这个构造函数，`main` 方法稍稍提供了一点用户界面，来显示测试的结果。

`main` 方法接下来从 `Ant` 的目标中得到类名：

```

<target name="runAllTests" depends="build" description="run all tests">
  <java classname="sis.testing.TestRunner" failonerror="true" fork="true">
    <classpath refid="classpath" />
    <jvmarg value="-enableassertions"/>
    <arg value="sis.testing.TestRunnerTest" />
  </java>
</target>

```

² 你可以使用更简明扼要的风格 `new Object[0]` 来替代 `new Object[] {}`。

runAllTest 目标中有几件有趣的事情:

- 在 java 任务中, 你指定了 failonerror=“true”。如果 Java 应用返回了一个非 0 的值, Ant 会认为执行发生了错误。构建脚本会终止并报告一个错误。使用 System.exit 命令 (参见 TestRunner 中的 main 方法) 可以让你立即终止一个应用, 并将传入的参数作为返回值。
- 在 java 任务中, 你指定了 fork=“true”。这是说 Java 应用作为一个单独的进程执行, 与执行 Ant 脚本的 Java 进程不同。不使用 fork 的缺陷 (pitfall) 是, 如果 Java 应用崩溃 (crash), Ant 构建脚本进程本身也会崩溃。
- 使用一个嵌套的 arg 元素将测试类名传递给 TestRunner。
- 使用嵌套的 jvmarg 元素将 enableassertions 参数传递给 Java VM。

@TestMethod 注解

为了要将这些测试重构, 你必须能够将测试方法标注出来, 这样新提取出的其它方法才不会被认为是测试方法。@TestMethod 注解放在每个方法的原型特征之前。在 TestRunnerTest 中使用 @TestMethod 注解两个测试方法 (singleMethodTest 和 multipleMethodTest)。同时注解在迷你类 (SingleMethodTest 和 MultipleMethodTest) 中另外三个被 TestRunnerTest 使用的测试方法。

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
    private TestRunner runner;
    private static final String methodNameA = "testA";
    private static final String methodNameB = "testB";

    @TestMethod
    public void singleMethodTest() {
        runTests(SingleMethodTest.class);
        verifyTests(methodNameA);
    }

    @TestMethod
    public void multipleMethodTest() {
        runTests(MultipleMethodTest.class);
        verifyTests(methodNameA, methodNameB);
    }
}
```

```

private void runTests(Class testClass) {
    runner = new TestRunner(testClass);
    runner.run();
}

private void verifyTests(String... expectedTestMethodNames) {
    verifyNumberOfTests(expectedTestMethodNames);
    verifyMethodNames(expectedTestMethodNames);
    verifyCounts(expectedTestMethodNames);
}

private void verifyCounts(String... testMethodNames) {
    assert testMethodNames.length == runner.passed() :
        "expected " + testMethodNames.length + " passed";
    assert 0 == runner.failed() : "expected no failures";
}

private void verifyNumberOfTests(String... testMethodNames) {
    assert testMethodNames.length == runner.getTestMethods().size() :
        "expected " + testMethodNames.length + " test method(s)";
}

private void verifyMethodNames(String... testMethodNames) {
    Set<String> actualMethodNames = getTestMethodNames();
    for (String methodName: testMethodNames)
        assert actualMethodNames.contains(methodName) :
            "expected " + methodName + " as test method";
}

private Set<String> getTestMethodNames() {
    Set<String> methodNames = new HashSet<String>();
    for (Method method: runner.getTestMethods())
        methodNames.add(method.getName());
    return methodNames;
}

}

class SingleMethodTest {
    @TestMethod public void testA() {}
}

class MultipleMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
}

```

@TestMethod 注解可以出现在任何方法修饰符的后面，例如 public 或 static。但注解必须位于方法原型特征之前（由方法的返回类型开头）。

为了声明 @TestMethod 注解类型，你创建一个看似接口的声明：

```

package sis.testing;
public @interface TestMethod {}

```

接口声明和注解类型声明之间的唯一不同是，你在接口声明的关键字 interface 之前放置了一个 @ 符。在 @ 和 interface 之间可以有空格，但是约定是二者紧邻。

现在代码可以编译，并且你可以运行测试了，但是你会在栈回溯（stack trace）中至少看到

一个 `IllegalAccessException`。`TestRunner` 中的代码依然将每个方法都视为测试方法，包括你刚刚抽取出的私有方法。`TestRunner` 中的反射代码无法调用这些私有方法。是时候在 `TestRunner` 中引入查找 `@TestMethod` 注解的代码了：

```
private void loadTestMethods() {
    testMethods = new HashSet<Method>();
    for (Method method: testClass.getDeclaredMethods())
        if (method.isAnnotationPresent(TestMethod.class))
            testMethods.add(method);
}
```

只需花费一行代码。调用 `Method` 对象的 `isAnnotationPresent` 方法，传入注解的类型 (`TestMethod.class`)。如果 `isAnnotationPresent` 返回 `true`，你便将 `Method` 对象添加到测试方法列表中。

现在测试可以执行了，但是返回的结果不正确：

```
runAllTests:
[java] passed: 0 failed: 0
```

你期待看到两个通过的测试，但是没有任何测试被注册——`isAnnotationPresent` 方法总是返回 `false`。

保留 (Retention)

`java.lang.annotations` 包中含有一种名为 `@Retention` 的元注解 (meta-annotation) 类型。你使用元注解来注解其它注解类型的声明。特别地，你使用 `@Retention` 注解来告诉 Java 编译器，注解信息要保留多久。有三个选择，概括在表 15.1 中。

像表 15.1 中解释的那样，如果你没有指定 `@Retention` 注解，缺省的行为意味着你在运行时可能无法提取注解的信息³。`RetentionPolicy.CLASS` 的一个示例是第 9 课中讨论的 `@Override` 注解。

表 15.1 注解保留策略政策

RetentionPolicy 枚举	注解部署
RetentionPolicy.SOURCE	在编译时被丢弃
RetentionPolicy.CLASS (缺省的)	保存在类文件中；在运行时可以被 VM 丢弃
RetentionPolicy.RUNTIME	保存在类文件中；在运行时由 VM 保留

³ VM 可能会选择保留该信息。

你可能最需要 `RetentionPolicy.RUNTIME`，这样像 `TestRunner` 这样的工具可以从它们的目标类中取出注解信息。如果创建的工具直接同源代码打交道（例如，像 `Eclipse` 这样 IDE 的插件），你可以使用 `RetentionPolicy.SOURCE` 来避免在类文件中保存不必要的注解信息。一个使用的示例可能是 `@Todo` 注解，用来标注代码中的某些片段需要提请注意。

为了让反射的代码认识 `@TestMethod` 注解，你必须修改注解类型的声明：

```
package sis.testing;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface TestMethod {}
```

现在，你的两个 `TestRunnerTest` 测试可以通过了：

```
runAllTests:
[java] passed: 2 failed: 0
```

注解的目标 (Annotation Targets)

你已经设计了 `@TestMethod` 注解，来让开发者标注测试方法。注解可以修饰许多其它的元素类别：类型（类、接口和枚举）、成员变量（field）、参数、构造函数、局部变量和包。缺省地，你可以使用标注修饰任意元素。你还可以选择限制注解类型只修饰一种类型。要达成这一点，你需要在你的注解类型声明中提供一个 `@Target` 元注解（meta-annotation）。 548

因为你没有为 `@TestMethod` 指定 `@Target` 元注解，某个开发者可能使用这个标记来修饰任意元素，例如一个成员变量。通常这样做并没有什么害处，但是这个开发者可能只是错手标注了一个成员变量，而误以为他/她标注的是一个方法。这个测试方法将会被忽略，直至某个人发现这一错误。为一个注解类型添加 `@Target`，则有助于在编译时帮助开发者，而非让他/她解读在运行时出现的问题。

为 `@TestMethod` 添加适当的 `@Target` 元注解：

```
package sis.testing;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface TestMethod {}
```

`@Target` 的参数必须是 `ElementType` 的枚举，在 `java.lang.annotation` 中定义。这个枚举提供了若干常量，分别对应了注解类型可以修饰的元素类别：`TYPE`、`FIELD`、`METHOD`、`PARAMETER`、`CONSTRUCTOR`、`LOCAL_VARIABLE`、`ANNOTATION_TYPE` 和 `PACKAGE`。对目标为

ElementType.PACKAGE 的注解有一些特殊的考量。更多信息参见本课稍后的“包注解”一节。

为了演示@Target 对你注解类型的作用，修改 TestRunnerTest。将一个成员变量而不是一个方法标注为@TestMethod：

```
// ...
public class TestRunnerTest {
    private @TestMethod TestRunner runner;

    @TestMethod
    public void singleMethodTest() {
        // ...
    }
}
```

当你编译代码时，会看到类似下面的错误消息：

```
annotation type not applicable to this kind of declaration
private @TestMethod TestRunner runner;
      ^
```

547

删去不合时宜的注解，重新编译，并再次运行你的测试。

跳过测试方法

有时，你可能希望跳过某些特定测试方法的执行。假定你有若干失败的测试。你可能希望每次只集中解决一个失败的方法，然后才转到下一个。其它测试方法的失败会分散你的注意力。你暂时希望“关闭它们”。

在JUnit中，你可以通过将一个方法注释掉，或者重命名方法使其原型特征不再表示为一个测试方法，来跳过一个测试。跳过一个测试方法的简单途径是，在方法名前添加一个x。例如，你可以将testCreate重命名为XtestCreate。JUnit查找那些名字以test字样开头的方法，因此它不会发现XtestCreate。

你最好不要养成注释掉测试的习惯。将一个方法注释掉、且作用时间超出你当前编程的时段，是一种不好的实践。你应该避免检入（check in）的代码中包括有注释掉的测试。其他开发者将无法理解你的意图。当我看到注释掉的代码时，特别是测试代码，首先的倾向就是删除它。

将测试方法注释掉是很危险的。很容易忘掉你曾经注释掉了一些测试。而且找到被注释掉的测试也很困难。如果JUnit可以警告你遗留了一些注释掉的测试，就太好了。

你的新TestRunner类也存在类似的问题。跳过一个测试最简单的方法是去除它的@TestMethod注解。这样做的问题，同注释掉一个测试所引发的问题一样。很容易在一个有很多测试项的系统中，丢掉一个测试。

作为练习，你需要对TestRunner进行必要的改写，来忽略指定的方法。你要创建一个新的

注解类型, @Ignore, 并修改 TestRunner 中的代码来识别这个注解。@Ignore 注解将允许开发者提供一个文本描述的参数, 解释跳过测试的原因。你还需要修改 TestRunner 来打印这些描述信息。

修改 TestRunner

在 TestRunnerTest 中添加一个新方法, ignoreMethodTest。它将衍生一个新的测试类, IgnoreMethodTest, 它包括三个标注了@TestMethod的方法。其中一个测试方法(testC)还额外标注了@Ignore。你必须验证这个测试方法没有被执行。

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {
    private TestRunner runner;
    // ...
    @TestMethod
    public void ignoreMethodTest() {
        runTests(IgnoreMethodTest.class);
        verifyTests(methodNameA, methodNameB);
    }
    // ...
}

// ...
class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore
    @TestMethod public void testC() {}
}
```

@Ignore 注解的声明同@TestMethod的声明看起来非常类似。

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {}
```

为了让你的测试通过, 对 TestRunner 进行下面的修改:

```

package sis.testing;

import java.util.*;
import java.lang.reflect.*;

class TestRunner {
    ...
    private void loadTestMethods() {
        testMethods = new HashSet<Method>();
        for (Method method: testClass.getDeclaredMethods())
            if (method.isAnnotationPresent(TestMethod.class) &&
                !method.isAnnotationPresent(Ignore.class))
                testMethods.add(method);
    }
    ...
}

```

549

单值 (Single-Value) 注解



@Ignore 注解是一种标注式 (marker) 注解, 标注一个方法是否应该被忽略。你只是需要使用 isAnnotationPresent 来测试该注解是否存在。现在, 你需要开发者来提供一个忽略测试的原因。你要修改 @Ignore 注解, 接收一个表示原因的 String 作为参数。

为了在注解类型中支持单个参数, 你需要提供一个名为 value 的方法, 它具有返回适当的类型且没有任何参数。注解类型的成员方法不能接收任何参数。

```

package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String value();
}

```

将 IgnoreMethodTest 中的一个方法仅标注为 @ignore。不提供任何参数:

```

class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
    @Ignore()
        @TestMethod public void testC() {}
}

```

注意, @Ignore 是 @Ignore () 的简写 (shortcut)。

在你编译时, 你会看到一个错误消息:

```

annotation testing.Ignore is missing value
@Ignore
^

```

编译器使用对应的注解类型声明来确保你提供了适当数目的参数。

修改测试的目标类, IgnoreMethodTest, 为@Ignore 标注提供一个原因。

```

public class TestRunnerTest {
    public static final String IGNORE_REASON1 = "because";
    // ...
}
class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
    @Ignore(TestRunnerTest.IGNORE_REASON1)
    @TestMethod public void testC() {}
}

```

550

重新运行你的测试。它们都通过了, 因此你不会破坏任何事情。但是你还需要打印被忽略方法的列表的能力。相应地修改测试:

```

@TestMethod
public void ignoreMethodTest() {
    runTests(IgnoreMethodTest.class);
    verifyTests(methodNameA, methodNameB);
    assertIgnoreReasons();
}

private void assertIgnoreReasons() {
    Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
    Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
    assert "testC".equals(entry.getKey().getName());
    "unexpected ignore method: " + entry.getKey();
    Ignore ignore = entry.getValue();
    assert IGNORE_REASON1.equals(ignore.value());
}

private <K, V> Map.Entry<K, V> getSoleEntry(Map<K, V> map) {
    assert 1 == map.size(): "expected one entry";
    Iterator<Map.Entry<K, V>> it = map.entrySet().iterator();
    return it.next();
}

```

你将被忽略的方法, 以 Method 对象与忽略原因字符串之间一个映射集合的形式返回。因为你预期只有一个方法被忽略, 你可以引入实用方法 getSoleEntry 来从 Map 中取出唯一的 Method key。因为我还沉浸在找出范型如何使用的激动中 (参见第 14 课), 在这里我走得更激进些, 让 getSoleEntry 成为一个范型方法, 可以为任何集合所使用。当然, 没有任何原因阻止你明确地使用映射的 key 和 value 类型来编写这个方法。

现在, 在 TestRunner 中进行必要的更改, 使之可以保存被忽略的方法, 以供后用:

```

package sis.testing;

import java.util.*;
import java.lang.reflect.*;

class TestRunner {
    // ...
    private Map<Method, Ignore> ignoredMethods = null;
    // ...
    private void loadTestMethods() {
        testMethods = new HashSet<Method>();
        ignoredMethods = new HashMap<Method, Ignore>();
        for (Method method: testClass.getDeclaredMethods()) {
            if (method.isAnnotationPresent(TestMethod.class))
                if (method.isAnnotationPresent(Ignore.class)) {
                    Ignore ignore = method.getAnnotation(Ignore.class);
                    ignoredMethods.put(method, ignore);
                }
            else
                testMethods.add(method);
        }
    }

    public Map<Method, Ignore> getIgnoredMethods() {
        return ignoredMethods;
    }
    // ...
}

```

你可以调用任何被注解元素的 `getAnnotation` 方法，将注解类型的名称（这里是 `Ignore.class`）作为参数传递给它。`getAnnotation` 方法返回一个指向实际注解对象的注解类型引用。一旦你拥有了注解对象引用（`ignore`），你便可以调用在注解类型接口中定义的方法。

我们现在将修改文本用户界面来显示被忽略的方法。

TestRunner 的用户界面类

在此时，`main` 方法不再是潦潦几行简单的代码。是时候把它移到一个单独的类来负责显示用户界面。

正如我早先提到的，因为 `TestRunner` 只是出于测试目的的一个实用工具，测试并不是绝对必须的。对于为测试运行器编写的这一小部分非产品化的用户界面代码，你不必强迫自己也要测试为先（`test first`）。你要这么做我自然无比欢迎，不过我在这里不会这样做。

下面的清单展示了一个重构后的用户界面类，打印被忽略的方法。其中唯一有意思的事情是，我使用 `System.exit` 调用返回失败测试个数的“巧妙”方法。为什么？为什么不是？它比一个 `if` 语句更精练，它不会使代码变得困惑，并且它返回了构建脚本或操作系统可能使用的额外信息。

```

package sis.testing;

import java.lang.reflect.*;
import java.util.*;
public class TestRunnerUI {
    private TestRunner runner;

    public static void main(String[] args) throws Exception {
        TestRunnerUI ui = new TestRunnerUI(args[0]);
        ui.run();
        System.exit(ui.getNumberOfFailedTests());
    }

    public TestRunnerUI(String testClassName) throws Exception {
        runner = new TestRunner(testClassName);
    }

    public void run() {
        runner.run();
        showResults();
        showIgnoredMethods();
    }

    public int getNumberOfFailedTests() {
        return runner.failed();
    }

    private void showResults() {
        System.out.println(
            "passed: " + runner.passed() +
            " failed: " + runner.failed());
    }

    private void showIgnoredMethods() {
        if (runner.getIgnoredMethods().isEmpty())
            return;

        System.out.println("\nIgnored Methods");
        for (Map.Entry<Method, Ignore> entry:
            runner.getIgnoredMethods().entrySet()) {
            Ignore ignore = entry.getValue();
            System.out.println(entry.getKey() + ": " + ignore.value());
        }
    }
}

```

552

数组参数



你可能希望能够让开发者提供多个独立的原因字符串。为了这样做，你可以指定注解类型的 `value` 方法以 `String []` 作为返回类型。然后通过使用看似数组初始化的结构，`@Ignore` 注解可以包含多个原因：

553


```
@Ignore({"why", "just because"})
```

下面是更新的注解类型声明：

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String[] value();
}
```

如果你为一个 `String []` 返回类型的注解类型成员，只提供一个字符串，Java 允许你去掉这种数组式的初始化。下面的这些注解，对现在的 `@Ignore` 定义来说，是等效的：

```
@Ignore("why")
@Ignore({"why"})
```

你还需要修改 `TestRunnerTest` 来支持 `Ignore` 注解中的改动。

```
package sis.testing;

import java.util.*;
import java.lang.reflect.*;

public class TestRunnerTest {

    public static final String IGNORE_REASON1 = "because";
    public static final String IGNORE_REASON2 = "why not";
    ...

    @TestMethod
    public void ignoreMethodTest() {
        runTests(IgnoreMethodTest.class);
        verifyTests(methodNameA, methodNameB);
        assertIgnoreReasons();
    }

    private void assertIgnoreReasons() {
        Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
        Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
        assert "testC".equals(entry.getKey().getName());
        "unexpected ignore method: " + entry.getKey();
        Ignore ignore = entry.getValue();
        String[] ignoreReasons = ignore.value();
        assert 2 == ignoreReasons.length;
        assert IGNORE_REASON1.equals(ignoreReasons[0]);
        assert IGNORE_REASON2.equals(ignoreReasons[1]);
    }
    ...
}

class SingleMethodTest {
    @TestMethod public void testA() {}
}
```

```

class MultipleMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
}

class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore({TestRunnerTest.IGNORE_REASON1,
             TestRunnerTest.IGNORE_REASON2})
    @TestMethod public void testC() {}
}

```

多个参数的注解



你可能希望注解支持多个参数。举例来说，假定你希望开发者在忽略测试时加入他们姓名的缩写。合适的注解可能是：

```
@Ignore(reasons={"just because", "and why not"}, initials="jjl")
```

现在你已经有不止一个注解参数，你必须提供成员-值（member-value）对儿。每个成员-值对儿中，包括成员的名字（必须同注解类型成员相匹配）、跟随其后的等号（=）、最后是该成员对应的常量值。

上面示例中的第二个 member-value 对儿，使用 initials 作为成员名而“jjl”作为其值。为了支持这个注解，你必须修改@Ignore 注解的类型声明，加入 initials 作为额外的成员。你还必须将 value 方法改名为 reasons。注解成员-值对儿中的每个 key 必须同注解类型声明中的成员名相匹配。

```

package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String[] reasons();
    String initials();
}

```

你可以在注解中以任意顺序指定成员-值对儿。其指定顺序不需要匹配注解类型声明中的成员顺序。

下面是 TestRunnerTest 中对应的改动：

```

package sis.testing;
// ...

```

```

public class TestRunnerTest {
    // ...
    public static final String IGNORE_INITIALS = "jjl";
    // ...
    private void assertIgnoreReasons() {
        Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
        Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
        assert "testC".equals(entry.getKey().getName());
        "unexpected ignore method: " + entry.getKey();
        Ignore ignore = entry.getValue();
        String[] ignoreReasons = ignore.reasons();
        assert 2 == ignoreReasons.length;
        assert IGNORE_REASON1.equals(ignoreReasons[0]);
        assert IGNORE_REASON2.equals(ignoreReasons[1]);
        assert IGNORE_INITIALS.equals(ignore.initials());
    }
    // ...
}

class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore(
        reasons={TestRunnerTest.IGNORE_REASON1,
            TestRunnerTest.IGNORE_REASON2},
        initials=TestRunnerTest.IGNORE_INITIALS)
    @TestMethod public void testC() {}
}

```

你不需要对 `TestRunner` 进行更改。你需要稍稍修改 `TestRunnerUI` 来正确地从一个 `Ignore` 对象中提取出原因及姓名缩写。

```

private void showIgnoredMethods() {
    if (runner.getIgnoredMethods().isEmpty())
        return;

    System.out.println("\nIgnored Methods");
    for (Map.Entry<Method, Ignore> entry:
        runner.getIgnoredMethods().entrySet()) {
        Ignore ignore = entry.getValue();
        System.out.printf("%s: %s (by %s)",
            entry.getKey(),
            Arrays.toString(ignore.reasons()),
            ignore.initials());
    }
}

```

556

缺省值



忽略某个测试方法的原因大多数情况下可能是一样的。通常，你只是想要临时地注释掉一个测试，以修正其它有故障的测试。每次都要提供一个原因是很繁琐的，因此你可能希望有个

缺省的忽略原因。TestRunner 中的测试反映出了这种需求：

```
@TestMethod
public void ignoreWithDefaultReason() {
    runTests(DefaultIgnoreMethodTest.class);
    verifyTests(methodNameA, methodNameB);
    Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
    Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
    Ignore ignore = entry.getValue();
    assertEquals("TestRunner.DEFAULT_IGNORE_REASON",
        ignore.reasons()[0]);
}

class DefaultIgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
    @Ignore(initials=TestRunnerTest.IGNORE_INITIALS)
    @TestMethod public void testC() {}
}
```

你需要在 TestRunner 类中定义 DEFAULT_IGNORE_REASON 常量，表示你任意所希望的字符串。

```
class TestRunner {
    public static final String DEFAULT_IGNORE_REASON =
        "temporarily commenting out";
    // ...
}
```

你可以为任何注解类型成员提供一个缺省值。这个缺省值必须是编译时的常量。@Ignore 的新定义包括了 reasons 成员的缺省值。注意使用关键字 default，将成员的原型特征与缺省值分隔开。

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String[] reasons() default TestRunner.DEFAULT_IGNORE_REASON;
    String initials();
}
```

◀ 557

附加返回类型与复式注解类型

除了 String 和 String [] 之外，注解值可以是基本类型（primitive）、枚举、Class 引用、注解类型本身、或者任意这些类型的数组。

稍后的测试（TestRunnerTest）为 @Ignore 注解提出了包括日期的需求。Date 类型将作为一

个注解；它的每个成员返回一个 `int` 的值。

```
@TestMethod
public void dateTest() {
    runTests(IgnoreDateTest.class);
    Map<Method, Ignore> ignoredMethods = runner.getIgnoredMethods();
    Map.Entry<Method, Ignore> entry = getSoleEntry(ignoredMethods);
    Ignore ignore = entry.getValue();
    sis.testing.Date date = ignore.date();
    assert 1 == date.month();
    assert 2 == date.day();
    assert 2005 == date.year();
}

class IgnoreDateTest {
    @Ignore(
        initials=TestRunnerTest.IGNORE_INITIALS,
        date=@Date(month=1, day=2, year=2005))
    @TestMethod public void testC() {}
}
```

`IgnoreDateTest` 中的这种注解被称为复式注解——即一个注解含有另一个注解。`@Ignore` 包括一个成员，`date`，它的值是另一个注解，`@Date`。

`sis.testing.Date` 注解类型的定义：

```
package sis.testing;

public @interface Date {
    int month();
    int day();
    int year();
}
```

558

`@Ignore` 注解类型现在可以定义一个 `date` 成员，它返回一个 `testing.Date` 的实例：

```
package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Ignore {
    String[] reasons() default TestRunner.DEFAULT_IGNORE_REASON;
    String initials();
    Date date();
}
```

因为 `Date` 注解类型只是用作另一个注解的一部分，不需要为它指定保留度（`retention`）或目标（`target`）。

你不能声明一个递归的注解类型——也就是说，某个注解类型的成员以相同的注解作为返回类型。

为了使你的测试能够编译并通过，你还需要修改 `IgnoreMethodTest` 和 `DefaultIgnoreMethodTest`：

```

class IgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}

    @Ignore(
        reasons={TestRunnerTest.IGNORE_REASON1,
            TestRunnerTest.IGNORE_REASON2},
        initials=TestRunnerTest.IGNORE_INITIALS,
        date=@Date(month=1, day=2, year=2005))
    @TestMethod public void testC() {}
}

class DefaultIgnoreMethodTest {
    @TestMethod public void testA() {}
    @TestMethod public void testB() {}
    @Ignore(initials=TestRunnerTest.IGNORE_INITIALS,
        date=@Date(month=1, day=2, year=2005))
    @TestMethod public void testC() {}
}

```

包注解



假定你想要一种将某些包指定为测试包的功能。进一步地，你希望你的测试工具运行“性能有关的”测试，并且同其它测试分离。为了完成这一点，你可以创建一个目标（target）为包（package）的注解。该注解的测试：⁴

559

```

@TestMethod
public void packageAnnotations() {
    Package pkg = this.getClass().getPackage();
    TestPackage testPackage = pkg.getAnnotation(TestPackage.class);
    assert testPackage.isPerformance();
}

```

你从 Package 对象中提取出注解信息，正如你对其它元素对象一样。你可以通过调用任何一个 Class 对象的 getPackage 方法来获得一个 Package 对象。

注解声明是非常直截了当的：

```

package sis.testing;

import java.lang.annotation.*;

@Target(ElementType.PACKAGE)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestPackage {
    boolean isPerformance() default false;
}

```

⁴ 注意变量名为 pkg，因为 package 是一个关键字。

不过，问题是，包注解存在于何处呢？将它放在包中每个源文件的 `package` 声明之前？还是只放在一个源文件的前面？或者保存在其它什么地方？

Java 限制你对每个包至多只能有一个包注解的语句。这意味着你并不能只是把一个注解放在某个源文件的 `package` 声明之前。

答案依赖于你所使用的编译器。Sun 推荐一种基于文件系统的特定方案。其它的编译器供应商可能选择其它方案。当然如果编译环境是基于文件系统的，它们可能别无选择。

Sun 的方案要求你在与包对应的目录中，创建一个名为 `package-info.java` 的源文件。Sun 的 Java 编译器读取这个伪源文件（`pseudosource`），但是不会产生可见的 `class` 文件作为输出。（事实上，在类名中是不可以含有连字符 `[-]` 的。）这个文件应该包含所有的包注解，后面跟随适当的 `package` 语句。你不应该在 `package-info.java` 中包含其它任何东西。

下面是 `sis.testing` 包中 `package-info.java` 文件看上去的样子：

560

```
@TestPackage(isPerformance=true) package sis.testing;
```

兼容性考虑

Sun 已经尽可能谨慎地对注解功能进行设计，以支持在注解改变时对现有代码产生最小的影响。本节将浏览几种不同的修改场景，并解释每种注解类型的更改所带来的影响。

当你为一个注解类型添加一个新成员时，可能的话请提供一个缺省值。使用缺省值可以让代码能够继续使用编译后的注解类型，而不会产生任何问题。但是如果你试图访问新成员，这就会产生问题。假定你从原先编译（使用没有加入新成员的注解类型声明）的注解中访问新成员。如果新成员不存在缺省值，这会产生一个异常。

如果你删除了一个注解类型成员，显然当你重新编译被注解的源代码时，会引起错误。不过，任何使用了被修改的注解类型的现有 `class` 文件会工作得很好，直至它们的源代码被重新编译。

避免删除有缺省值的成员、改变返回类型、或删除现有注解类型的目标元素（`target element`）。所有这些行为会潜在地产生异常。

如果你修改了保留（`retention`）类别，结果通常是你可以预料的。例如，如果你将 `RUNTIME` 改为 `CLASS`，注解在运行时将不再可读。

有疑问时，编写一个测试来演示这些更改的行为。

关于注解的额外注意事项

- 没有指定目标 (target) 的注解可以修饰任何 Java 元素
- 在注解类型声明中, 唯一你可以返回的参数化类型是 Class 类型
- @Document 元注解类型可以让你声明将某个注解类型包含在由 javadoc 等工具产生的 API 文档中。
- @Inherited 元注解类型意味着一个注解类型是被所有子类继承的。如果你调用 Method 或 Class 对象的 getAnnotation 方法而不是 getDeclaredAnnotations 方法, 便可以返回它。
- 你不能将 null 作为注解的值 (value)。
- 你不能用一个给定的注解来重复地修饰某个元素。例如, 你不能为某个测试方法提供两个 @Ignore 注解。
- 为了在内部支持注解类型, Sun 修改了 Array 类, 加入了 toString 和 hashCode 方法的实现。

◀ 561

总结

注解是一种很强大的功能, 可以帮助你放入代码中的注释结构化。一个常用来吹捧的例子是, 对接口声明进行注解, 让工具可以为这些相关的方法生成代码。

使用注解的主要缺点是, 当你的代码使用它时, 便会对注解类型产生依赖关系。尽管 Sun 已经内建了一些方法来帮助维护二进制兼容性, 但对注解类型声明的更改, 依然会对你的代码产生负面影响。你还必须提供注解类型的源文件以进行编译。这不应令人感到惊讶: 一个注解类型实际上就是一个接口类型。

再且, 经验法则是, 谨慎地使用注解类型。注解类型是一个接口, 并且应该表示一种稳定的抽象。同其他接口一样, 确保你已经仔细考虑了在系统中引入注解类型所带来的影响。最糟糕的情况是, 当你需要对某个注解类型进行显著的更改时, 编译之后会牵扯大量的搜索和替换⁵。

⁵ 暂时地, IDE 可能没有对操作和定位注解提供完善的支持。你可能需要诉诸于搜索和替换 (或者编译、标识和替换) 来进行重大的更改。对注解的 IDE 集成支持应该很快就会出现。IDEA 中的支持我认为已经是可以接受的了。

练习

1. 使用上一课中的 `RingTest` 和 `Ring` 类，在 `add` 方法中引入 `assert` 来拒绝 `null` 参数。确保你为此编写了测试！在你运行测试前不要忘记打开断言选项。
2. 创建一个注解 `@Dump`。这个注解可以应用于类中的任何字段成员变量。然后创建一个 `ToStringer` 类，当向它传入一个对象时，它会转储 (`dump`) 每个被标记的字段成员变量（注明了为 `toString` 方法所用）。
3. 修改转储注解，通过添加一个可选的 `order` 参数，以允许对成员变量进行排序。次序必须是一个正整数。如果某个成员变量没有被注解，它应该出现在成员变量列表的最后。
4. 为转储添加另一个参数 `quote`，它是一个布尔值，指示是否将值用引号引起来。这对那些 `toString` 的表示可能为空、或者有前导或结尾空格的对象很有用处。
5. 为 `@Dump` 注解添加一个 `outputMethod` 成员变量。它指定了用于得到成员变量可打印表示的“`toStringer`”方法。它的缺省值应该是 `toString`。当你有一个对象无法更改 `toString` 表示时，例如系统类库中的某个对象，是非常有用的。
6. 将 `outputMethod` 注解更改为 `outputMethods`，并让它支持一个 `String []` 类型的方法名数组。`ToString` 的代码应该通过依次调用该方法名数组中的每个方法，并将结果串联起来，构造一个对象的可打印表示。将每个结果用一个空格隔开。（你可以考虑添加另一个注解来指定分隔字符。）

Swing, 第一部分

Swing 是用来开发跨平台 GUI（图形化用户界面）的 Java 标准类库。Swing API 提供了一套丰富的功能，使你能够构造复杂的用户界面。使用 Swing，你可以构造具有图形化界面的应用，让用户可以通过诸如按钮、输入框和列表框等基本控件¹，或者是诸如表格（table），树（tree）和拖放（drag & drop）等高级控件，与应用交互。

本章介绍 Swing。概述了 Swing 的基础知识：如何使用一些常用部件（widget）构建简单的应用。本章节无法囊括关于 Swing 的一切。事实上，本章节仅触及到 Swing 的表层。关于这个课题，已经有作者投入整本书甚至若干卷的篇幅，来详述它。

不过，学习 Swing 的基本知识，会为你理解它的其余部分如何工作、以及如何找到更多的信息打下基础。例如，一旦你学会如何使用 table model 建立一个 table 部件，学习如何使用一个 tree model 建立一个 tree 部件就很容易了。Java API 文档通常为如何使用一个部件提供了充分的信息。对于更复杂的部件，包括 tree model 以及诸如复杂部件布局（widget layout）的内容，Java API 文档通常会提供一个链接，指向一个有关该课题的 Sun 教程。

更重要的是，你将在本章节中学到如何测试 Swing 应用的各种方法和基本原理。开发者通常认为测试 Swing 应用是件很困难的事情，因而作罢。结果，Swing 代码经常不被测试，并且设计得一塌糊涂。

本章在介绍 Swing 的同时还介绍了 Swing 的测试。测试 Swing 应用是困难的，但不能因此就不去测试它。良好的测试，良好的设计对用户界面代码有极大的好处。

用户界面的设计有两个不同方面。用你设计和编写的 Java 代码去构建用户界面，是你在本章中要关注的一个方面。另一设计方面是用户界面的外观和体验（look and feel）。

外观和体验（look and feel）描绘了这样的需求——一个最终用户是如何与应用交互的？客户，或者一个正与客户合作的用户界面设计专家，应该把这些需求展示给开发小组。许多小作坊也让开发小组成员参与 GUI 的设计。客户将需求展示给开发者的形式，可能包括屏幕快照、未经

¹ 也称为部件或组件，控件组成了图形用户界面，它们或者用来表示信息，或者让最终用户控制他们与用户界面在某些方面的交互。

加工的图画, 或者是各种正式的图表。

- 你将会学到:
- Swing 应用的设计
- 面板 (panel) 和窗体 (frame)
- 按钮和动作侦听器
- list 与 list model
- Swing 布局: BorderLayout、GridLayout、FlowLayout、BoxLayout、GridBagLayout

Swing

Swing 并不是你进行 GUI 开发的唯一选择, 但是对你来说它是标准配置。Eclipse IDE 是使用 SWT (Standard Widget Toolkit, 标准部件工具包) 建立的, 你同样也可以选择这个 API 来建立你的应用。

事实上 Swing 构建在另一个叫做 AWT (Abstract Window Toolkit, 抽象窗口工具包) 的 Java API 基础之上。Sun 在 Java 的第一个版本中引入了 AWT, 作为生成 GUI 的唯一手段。它包括了少量的部件, 确保当时它们在 Java 支持的所有平台上都是存在的。这种“最小公分母”式的设计, 能够让你构建跨平台的 GUI 应用, 但却约束了你对 GUI 设计的表现力。

AWT 被认为是一个重量级的 GUI 框架。Java 将每个 AWT 控件紧密结合到操作系统直接支持的一个对等控件上。操作系统管理着每个 AWT 控件。

Swing, 在 AWT 出现几年后被引入, 相反地, 它是一个轻量级的 GUI 框架。通过创建一个使用重量级控件作为外壳 (shell) 的控件, Swing 消除了最小公分母问题。Swing 组件继承于 AWT 组件。它们增加了自定义的渲染与交互代码, 以使它们的行为和表现比操作系统支持的所有控件都要成熟和完善。

你可以为你的 Swing 应用选择一个外观和体验 (look and feel), 使之模仿一个特定的操作系统。例如, 你可以告诉 Java 将你的 Swing 控件渲染为 Motif 控件或 Windows 控件风格。

起步

你将构建一个简单的 Swing 应用, 向最终用户显示一个课程列表。这个应用也允许用户增新的课程。

当你学习 Swing 时，你可能会从探针方案（spike solution，译注：XP 术语，即由开发人员编写的验证性的小程序，用来探索或解决项目中的某些关键问题）——演示 Swing 如何工作的少量代码——开始。一旦这些探针可以工作了，你可以回退一步，并推断出如何测试 Swing 代码。本章有时会以这种方式呈现其内容：我将首先演示如何使用 Swing 编写代码，然后展示它的测试代码。

大部分应用，无论它们使用 Swing 与否，都从一个窗体（frame）开始。frame 是一个缺省带有标题栏和边框的顶层（top-level）窗口。这个窗口很大程度上是由你的操作系统画出、并由你的操作系统控制的。在 Java 中，Swing 的 frame 类是 `javax.swing.JFrame`。在 Java 中构建一个初始的应用非常简单：构造一个 `JFrame`，然后设置其大小并使之可见。

创建 `Sis` 类（Student Information System，学生信息系统的缩写）。

```
package sis.ui;

import javax.swing.*;

public class Sis {
    static final int WIDTH = 350;
    static final int HEIGHT = 500;

    public static void main(String[] args) {
        new Sis().show();
    }

    public void show() {
        JFrame frame = new JFrame();
        frame.setSize(WIDTH, HEIGHT);
        frame.setVisible(true);
    }
}
```

567

`Sis` 的定义中包括一个 `main` 方法，让你可以从命令行执行这个应用。你在 `main` 方法中唯一要做的就是构造一个新的实例，并只调用它的一个方法。你可能首先需要调用一个方法来解析命令行参数。如果你的 `main` 方法牵扯较多，你需要进行重构。不要在 `main` 中加入任何实际的逻辑——`main` 通常不在测试之列。

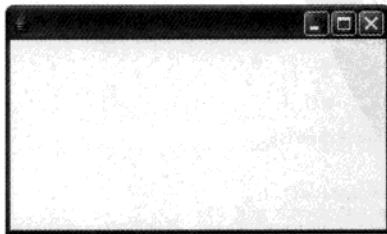


图 1 一个 Frame 窗口

编译并执行 `Sis`。你应该看到一个简单的 `frame` 窗口（如图 1 所示）。体验一下这个窗口，注意你可以像对其他窗口应用一样，设置它的大小、将它最小化或关闭。你还可以试验一下 `setSize` 和 `setVisible` 方法，看看它们对 `frame` 有怎样的影响。尝试忽略一个或全部的方法调用。

唯一的问题是当你关闭窗口时，Java 进程不会终止。在幕后，Swing 创建了用户线程（`user thread`）。在 `main` 方法退出并不会导致这些用户线程关闭。参考第 13 课“线程关闭”一节中对用户线程和守护线程的讨论。

你可以指定让 `JFrame` 在关闭时退出，以终止 Java 进程。

```
package sis.ui;

import javax.swing.*;

public class Sis {
    static final int WIDTH = 300;
    static final int HEIGHT = 200;

    public static void main(String[] args) {
        new Sis().show();
    }

    void show() {
        JFrame frame = new JFrame();
        frame.setSize(WIDTH, HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

568

现在，你已经理解需要哪些步骤来建立一个 `frame`，你可以暂时将这段代码抛在一边，然后编写几个测试。你可以参考 Java 的 API 文档，查看可以对 `JFrame` 对象发送哪些查询消息。下面的测试展示了你可以检视用于初始化 `frame` 的各部分信息。你可以查询 `frame` 是否是可见的。

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;

public class SisTest extends TestCase {
    private Sis sis;
    private JFrame frame;

    protected void setUp() {
        sis = new Sis();
        frame = sis.getFrame();
    }

    public void testCreate() {
        final double tolerance = 0.05;
        assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
        assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
        assertEquals(JFrame.EXIT_ON_CLOSE,
```

```

        frame.getDefaultCloseOperation();
    }

    public void testShow() {
        sis.show();
        assertTrue(frame.isVisible());
    }
}

```

要满足这个测试，原本的探针代码需要稍许改动：

```

package sis.ui;

import javax.swing.*;

public class Sis {
    static final int WIDTH = 300;
    static final int HEIGHT = 200;

    private JFrame frame = new JFrame();
    public static void main(String[] args) {
        new Sis().show();
    }

    Sis() {
        frame.setSize(WIDTH, HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    void show() {
        frame.setVisible(true);
    }

    JFrame getFrame() {
        return frame;
    }
}

```

569

当你运行这个测试时，你会看到 frame 窗口显示在屏幕上，因为测试实例化了 Sis。应该避免嵌入这种弹出窗口的测试。这很容易会分散注意，并会显著地减缓你的测试。稍后你将学习有关技术，以消除在运行测试时对渲染用户界面的需求。现在，一个或两个测试弹出窗口，被认为是可以接受的。

即使测试完成，frame 窗口也不会消失。这是你可以修正的一个更大的困扰。在你测试中的 tearDown 方法中，通知应用关闭：

```

public class SisTest extends TestCase {
    ...
    protected void tearDown() {
        sis.close();
    }
    ...
}

```

通过释放 (dispose) frame 窗口来关闭应用：

```
package sis.ui;

import javax.swing.*;

public class Sis {
    ...
    private JFrame frame = new JFrame();
    ...
    void close() {
        frame.dispose();
    }
}
```

570

View 优先还是 Model 优先

在开发用户界面时，常常引发什么优先的问题：首先建立 model 还是建立 view？没有绝对正确的答案。这取决于你的熟悉程度，以及你通常使用哪种方法可以获得成功。

在本书的 15 节核心课程中，你学习了如何构建没有用户界面的类，或者放入到更大系统中的类。在现实中，你会发现上下文环境常常会改变你的设计。应用可能会极大地影响 model 类的公共接口。

当我以“应用优先（application first）”的方法设计系统时，大都获得了成功。每当我以孤立的方式建立 model 类，然后试图将它们安插到应用中时，我常常被迫显著地改变它们的行为。

即使我把应用的类作为系统开发的中心，在开发时我也会将 view 及其需求放在心上。而且在现实中，我很少上来就开发应用的任何部分。我曾发现，在不同层之间进行跳跃、同时使用测试及其结果作为指导，更有效。

Swing 应用的设计

设计一个 Swing 应用，主要围绕着如何组织职责——那些和任何基于用户界面的应用都密切相关联的职责。这些职责包括显示信息、管理输入、对业务逻辑建模并管理应用流程。

这些职责的术语在一定程度上已经称为标准了。由 view 向最终用户显示信息。在 Sis 中 JFrame 对象是一个 view 对象。Controller 管理最终用户来自键盘、鼠标或其他设备的输入。一个 view 可能同一个或多个 model（也被称为领域对象——domain object）打交道。例如，Sis 应用中的代码需要同 Course 和 Student 这类 model 对象打交道。最终，application 协调 model、view 和 controller。应用负责将用户的体验串联起来。

有很多种方法来建立 Swing 应用。你可能已经听说过这些职责的不同术语。你可能还遇到过其他的某些术语，暗含了各种职责上的重叠与组合。例如，Swing 类本身常常将 view 和 controller 逻辑合并在一起。（事实上，当你在本章中听到我提及 view 时，通常我的意思是指 view 和 controller 的组合。）虽然如此，核心的职责总是存在的。在本课中，我们将看到就设计而言，

571

在什么地方应用 TDD——我没有强迫你在示例的开发中使用任何僵化的设计。

例如，你将首先编写 Sis 应用的 view 部分。按照单一职责原则 (Single-Responsibility Principle)，你在 view 类中编写的代码只用于显示和布置用户界面。按照这种方式组织代码的明显好处是，你可以将面板作为一个单独的应用来运行。这可以让你集中精力开发用户界面的外观、或布局。

相反的，许多 Swing 应用并没有遵循这个规则。这些应用中的 view 类同其他交互纠缠在一起。例如，view 类可能调用一个 model 类的方法，该方法进而从数据库中存取数据。要孤立地显示 view——也就是说并不运行整个应用——事实上是不可能的。通常，你只有经历了该应用的其他若干屏幕 (screen)，才会遇到上面所说的 view。

面板 (Panel)

JFrame 套装 (envelope) 了一个内容窗格 (content pane)——你可以在其中放置可视的内容。一个内容窗格对象是 java.awt.Container 类型的一个容器 (container)。你可以向一个容器中添加其他的组件，包括列表框、文本框或容器本身。在 Swing 中，主干的容器类是 JPanel。像 JFrame 一样，JPanel 类是一个 view 类，只用于显示目的。

你可能已经注意到包及其命名约定。Swing 组件全部都以字母 J 开头 (JFrame、JPanel、JList 等)；它们包含在 javax.swing 包或其子包中。java.awt 包 (或其子包) 中包含了 AWT 类。AWT 的类名没有使用前缀 J。

相关用户界面的第一个部分是，向最终用户显示文本“Courses (课程):”。你将直接把这段文字嵌入到 JFrame 的内容窗格中。一个更好的方法是，构造包含该文本的第二个容器，然后把这个容器嵌入到内容窗格中。为了这样做，你将会创建一个 JPanel 的子类，来向最终用户显示文本。然后，再将这个 JPanel 嵌入到 JFrame 的内容窗格中。

下面是 JPanel 的探针 (spike) 方案：

```
package sis.ui;

import javax.swing.*;
import java.awt.*;

public class CoursesPanel extends JPanel {
    public static void main(String[] args) {
        show(new CoursesPanel());
    }

    private static void show(JPanel panel) {
        JFrame frame = new JFrame();
        frame.setSize(100, 100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



```

        frame.getContentPane().add(panel); // 1
        frame.setVisible(true);
    }

    public CoursesPanel() {
        JLabel label = new JLabel("Courses:");
        add(label);
    }
}

```

这个探针 (spike) 使用了含有一些“驱动 (driver)”代码的 main 方法, 让你可以手工地测试该面板。如果你要编写第二个面板类, 最好把这部分驱动代码重构为一个通用的实用类。

在 CoursesPanel 的构造函数中, 你创建了一个新的 JLabel 对象。JLabel 是一个为最终用户显示文本的控件。你通过 JLabel 的构造函数指定要显示的文本。为了显示 JLabel, 你必须使用 add 方法将它添加到 JPanel 中。

CoursesPanel 并没有定义 add, CoursesPanel 所继承的父类 JPanel 中也没有定义。你必须将继承层次追溯到 java.awt.Container, 才能找到 add 方法的定义。add 方法属于它是说得通的: 你可以向 Container 对象 add 其他组件, 包括 JPanel 对象。

下面是 CoursesPanel 的继承层次:

```

Object
|
| java.awt.Component
|   |
|   | java.awt.Container
|       |
|       | javax.swing.JComponent
|           |
|           | javax.swing.JPanel
|               |
|               | sis.ui.CoursesPanel

```

Swing 的每个部分都是建立在 AWT 框架之上。所有东西都是组件。所有 Swing 组件 (JComponent 对象) 都是容器。因而 JPanel 和 CoursesPanel 也是容器。

为了显示一个面板, 你需要将它添加到 JFrame 的内容窗格 (content pane) 中。在 CoursesPanel 的 show 方法中, 标记为 1 的代码行完成这一操作。

编译并执行 CoursesPanel 探针 (spike) 代码。结果如图 2 所示:

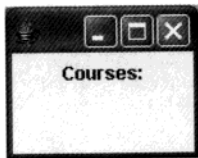


图 2 JPanel

CoursesPanel 的测试应该检验些什么呢？现在，CoursesPanel 所做的唯一有意义的事是，向用户显示一个 label。因而测试应该确保面板包含了显示正确文本的 label：

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;

public class CoursesPanelTest extends TestCase {
    public void testCreate() {
        CoursesPanel panel = new CoursesPanel();
        JLabel label = getLabel(panel);
        assertEquals(CoursesPanel.LABEL_TEXT, label.getText());
    }

    private JLabel getLabel(JPanel panel) {
        for (Component component: panel.getComponents())
            if (component instanceof JLabel)
                return (JLabel)component;
        return null;
    }
}
```

getLabel 方法的工作是，以 JPanel 作为参数，并返回在 JPanel 中遇到的第一个 JLabel 对象。因为 JPanel 是一个容器，你可以使用 getComponents 方法得到它所包含组件的列表。列表中的每个元素都是 java.awt.Component 类型。你必须使用 instanceof 来判断每个元素的类型是否为 JLabel。你必须将匹配的元素强制类型转换并返回为 JLabel 类型。

574

可能有比使用 instanceof 更好的方法来判断某个组件是否存在。不过，在你必须测试第二种类型的组件或第二个 label 之前，这个技术现在已经足够应付了。

这个测试有必要吗？答案是有争议的。即使你要编写千百个自动测试，你总是一次又一次地手工地执行 Swing 应用来进行检验。这样，你可以快速地判断某个特定组件是否显示。

有人也会问相反的问题：这个测试充分吗？你不仅应该可以测试控件是否在窗口中出现，还可以测试它是否使用了合适的字体且处于正确的坐标位置。

我的看法是，虽然编写这样的测试并不困难，但可能还是有些过了。与其历经辛苦在 200 个屏幕中找到某人于某处偶尔地修改了 JLabel 的文本，我宁可把精力放在其他地方。基于定位的布局测试（验证 label 显示在屏幕何处的测试）是另一个问题。它们很难编写，甚至更难维护。些许美感上的不适，通常并不是成功执行应用的障碍。

更重要的是，你要为任何具有动态能力的用户界面编写测试。例如，你可能允许应用在使用户按下一个按钮时更改文本的颜色。你可能希望编写一个测试来确保这个动作和反应是如你所愿的。

CoursesPanel 类同前面探针实现的唯一不同在于，它定义了一个类常量来表示 label 的文本。

```
package sis.ui;

import javax.swing.*;
```

```
import java.awt.*;

public class CoursesPanel extends JPanel {
    static final String LABEL_TEXT = "Courses";
    ...
    public CoursesPanel() {
        JLabel label = new JLabel(LABEL_TEXT);
        add(label);
    }
}
```

在 Sis 应用中, 你希望 CoursesPanel 作为主视图。修改后的 SisTest 使用一个类似在 CoursesPanelTest 所使用的机制, 来检验内容窗格 (content pane) 是否包含一个 CoursesPanel 实例。

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;

public class SisTest extends TestCase {
    ...
    public void testCreate() {
        final double tolerance = 0.05;
        assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
        assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
        assertEquals(JFrame.EXIT_ON_CLOSE,
            frame.getDefaultCloseOperation());
        assertTrue(containsCoursesPanel(frame));
    }

    private boolean containsCoursesPanel(JFrame frame) {
        Container pane = frame.getContentPane();
        for (Component component: pane.getComponents())
            if (component instanceof CoursesPanel)
                return true;
        return false;
    }
    ...
}
```

你已经熟悉了如何得到一个 frame, 来包含并显示一个面板 (panel)。你实际上编写的代码, 同 CoursesPanel 的“驱动 (driver)”代码是一样的。

```
package sis.ui;

import javax.swing.*;

public class Sis {
    ...
    Sis() {
        frame.setSize(WIDTH, HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new CoursesPanel());
    }
    ...
}
```

测试通过。

重构 (Refactory)

在为本书搭建这个示例期间，我所编写的 `Sis` 构造函数中包括了使 `frame` 可视的语句。**Michael Feathers** 提醒我，构造函数只应该用于初始化的。这有点言过其实了，我有不同意见，因为你可以将窗口的显示看作是初始化的一部分。作为对他的回应，使用 **Ron Jeffries** 的至理名言，也就是：构造函数可以大胆地去做那些它没有被要求的事情。

576

我倾听这两种声音，并修正不足。不过，我还是认为 GUI 控件和布局构造是简单的初始化。将面板 (`panel`) 放到 `frame` 中属于布局初始化。当然你有理由将布局初始化同对象初始化分离开来，这样做是有一些价值的。没有初始化内容的 `JPanel` 子类是没什么用处的。强制客户进行额外的调用来初始化其布局是冗余的。

不过，将 `setVisible` 语句从构造函数中分离出来，是有潜在价值的。就 `SisTest` 而言，这样做可以单独地初始化一个测试而不强制渲染 `frame`。在产品系统中，在幕后初始化一个 `frame` 并稍后单独控制其可视性，通常是有价值的。

归根结底，判定是因人而异的，而且是纯理论化的。对 `CoursesPanel` 来说，一个未来的需求可能要求开发者创建 `CoursesPanel` 的一个子类。将初始化代码放在分离的方法中，可以更容易地覆盖 (`override`) 或继承初始化功能。不过你应该避免为未来的假设 (通常永远不会到来) 进行设计。放轻松，只管等待并顺势进行修改 (如有必要)。

一个可接受的折衷是，考虑简单设计规则#3，为可读性的目的，将初始化提取到一个 `private` 方法中：

```
public CoursesPanel() {
    createLayout();
}

private void createLayout() {
    JLabel label = new JLabel(LABEL_TEXT);
    add(label);
}
```

`SisTest.containsCoursesPanel` 同 `CoursesPanelTest.getLabel` 中的代码非常类似。两个方法都是迭代访问容器中的组件，并进行匹配的测试。

重复的代码是一个问题，并且只会变得更糟。添加按钮、列表框、或其他组件类型，每种类型都会需要一个新的方法。并且，假定你在面板中有两个 `label`。当前的代码只能捕捉到第一个 `label`。

一个方案是为每个组件提供一个名字。你可以遍历子组件的列表，直到你找到了匹配的名字。这种方案对你的 `view` 类来说，需要更多的管理开销，但是它使得测试变得容易了。而且如

577

你将要看到的，它还使组件的聚合操作更加简单了。

所有的组件和容器类都继承于 `java.awt.Component`。在这个类中，你可以发现 `setName` 和 `getName` 方法，分别接收并返回一个 `String` 对象。

从 `SisTest` 中的代码开始：

```
public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
        frame.getDefaultCloseOperation());
    assertNotNull(getComponent(frame, CoursesPanel.NAME));
}

private Component getComponent(JFrame frame, String name) {
    Container container = frame.getContentPane();
    for (Component component: container.getComponents())
        if (name.equals(component.getName()))
            return component;
    return null;
}
```

`CoursesPanel` 中对应的改动：

```
package sis.ui;
...
public class CoursesPanel extends JPanel {
    static final String NAME = "coursesPanel";
    ...
    public CoursesPanel() {
        setName(NAME);
        createLayout();
    }
}
```

消除重复部分的秘诀是，推进抽象的使用。例如，新的 `getComponent` 方法处理所谓抽象的组件，而不是具体的 `CoursesPanel` 类型。将 `CoursesPanelTest` 中的代码推向这个方向，会导致：

```
package sis.ui;
...
public class CoursesPanelTest extends TestCase {
    public void testCreate() {
        CoursesPanel panel = new CoursesPanel();
        JLabel label =
            (JLabel) getComponent(panel, CoursesPanel.LABEL_NAME);
        assertEquals(CoursesPanel.LABEL_TEXT, label.getText());
    }

    private Component getComponent(Container container, String name) {
        for (Component component: container.getComponents())
            if (name.equals(component.getName()))
                return component;
        return null;
    }
}
```

```
// in CoursesPanel:
...
public class CoursesPanel extends JPanel {
    static final String LABEL_NAME = "coursesLabel";
    ...
    private void createLayout() {
        JLabel label = new JLabel(LABEL_TEXT);
        label.setName(LABEL_NAME);
        add(label);
    }
}
```

现在, `getComponent` 方法只有第一行是不同的。由于没有更好的地方把它们作为类方法移动到一个新类中, 并加以重构。下面是产生的类的代码, `sis.ui.Util`, 以及修改后的测试类:

```
// sis.ui.Util
package sis.ui;

import java.awt.*;
import javax.swing.*;

class Util {
    static Component getComponent(Container container, String name) {
        for (Component component: container.getComponents())
            if (name.equals(component.getName()))
                return component;
        return null;
    }


    static Component getComponent(JFrame frame, String name) {
        return getComponent(frame.getContentPane(), name);
    }
}

// sis.ui.SisTest
public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
        frame.getDefaultCloseOperation());
    assertNotNull(Util.getComponent(frame, CoursesPanel.NAME));
}

// sis.ui.CoursesPanelTest
public void testCreate() {
    CoursesPanel panel = new CoursesPanel();
    JLabel label =
        (JLabel)Util.getComponent(panel, CoursesPanel.LABEL_NAME);
    assertEquals(CoursesPanel.LABEL_TEXT, label.getText());
}
```

新方法调用更笨重了些。一个原因是对强制类型转换的需要。当强制类型转换的重复变得很明显时, 你可以进行重构。但是, 这个短期的方案已经有极大的提高了——你已经消除了方法级别的重复, 否则这种重复将会在短时间内蔓延开来。

更多的控件

 SIS 应用应该允许用户添加新的课程。为了支持这个功能, CoursesPanel 可以包含一组输入框 (entry field), 让用户输入课程的系与课程编号。用户应该能够通过点击“添加 (Add)”按钮, 把按照输入的系与编号所创建的一个新课程, 加入到列表中。

针对 view 的测试:

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    public void testCreate() {
        CoursesPanel panel = new CoursesPanel();
        JLabel label =
            (JLabel)Util.getComponent(panel, LABEL_NAME);
        assertEquals(LABEL_TEXT, label.getText());

        JList list =
            (JList)Util.getComponent(panel, COURSES_LIST_NAME);
        assertEquals(0, list.getModel().getSize());

        JButton button =
            (JButton)Util.getComponent(panel, ADD_BUTTON_NAME);
        assertEquals(ADD_BUTTON_TEXT, button.getText());

        JLabel departmentLabel =
            (JLabel)Util.getComponent(panel, DEPARTMENT_LABEL_NAME);
        assertEquals(DEPARTMENT_LABEL_TEXT, departmentLabel.getText());

        JTextField departmentField =
            (JTextField)Util.getComponent(panel, DEPARTMENT_FIELD_NAME);
        assertEquals("", departmentField.getText());
        JLabel numberLabel =
            (JLabel)Util.getComponent(panel, NUMBER_LABEL_NAME);
        assertEquals(NUMBER_LABEL_TEXT, numberLabel.getText());

        JTextField numberField =
            (JTextField)Util.getComponent(panel, NUMBER_FIELD_NAME);
        assertEquals("", numberField.getText());
    }
}
```

使用全路径 (qualified) 的类常量变得有些笨拙, 因此我决定静态地引入 CoursesPanel 类。在本例中, 可能会对类常量的来历产生混淆。

新组件类型 JButton、JTextField 和 JList 的角色应该是很明显的。新控件对应的人部分断言

代码，同之前为 label 编写的代码非常相似。

JList 对象的断言是不同的。它检验在新构造的 CoursesPanel 中，JList 是空的。你可以首先得到它的 model，然后请求 model 的大小来判断 JList 中有多少个元素。你将在稍后的“列表 Model”一节中，了解更多关于列表 model 的内容。

修改后的 view 类：

```
package sis.ui;

import javax.swing.*;
import java.awt.*;

public class CoursesPanel extends JPanel {
    static final String NAME = "coursesPanel";
    static final String LABEL_TEXT = "Courses";
    static final String LABEL_NAME = "coursesLabel";
    static final String COURSES_LIST_NAME = "coursesList";
    static final String ADD_BUTTON_TEXT = "Add";
    static final String ADD_BUTTON_NAME = "addButton";
    static final String DEPARTMENT_FIELD_NAME = "deptField";
    static final String NUMBER_FIELD_NAME = "numberField";
    static final String DEPARTMENT_LABEL_NAME = "deptLabel";
    static final String NUMBER_LABEL_NAME = "numberLabel";
    static final String DEPARTMENT_LABEL_TEXT = "Department";
    static final String NUMBER_LABEL_TEXT = "Number";

    public static void main(String[] args) {
        show(new CoursesPanel());
    }

    private static void show(JPanel panel) {
        JFrame frame = new JFrame();
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel); // 1
        frame.setVisible(true);
    }

    public CoursesPanel() {
        setName(NAME);
        createLayout();
    }

    private void createLayout() {
        JLabel label = new JLabel(LABEL_TEXT);
        label.setName(LABEL_NAME);

        JList list = new JList();
        list.setName(COURSES_LIST_NAME);

        JButton addButton = new JButton(ADD_BUTTON_TEXT);
        addButton.setName(ADD_BUTTON_NAME);

        int columns = 20;

        JLabel departmentLabel = new JLabel(DEPARTMENT_LABEL_TEXT);
```



```

departmentLabel.setName(DEPARTMENT_LABEL_NAME);

JTextField departmentField = new JTextField(columns);
departmentField.setName(DEPARTMENT_FIELD_NAME);

JLabel numberLabel = new JLabel(NUMBER_LABEL_TEXT);
numberLabel.setName(NUMBER_LABEL_NAME);

JTextField numberField = new JTextField(columns);
numberField.setName(NUMBER_FIELD_NAME);

add(label);
add(list);
add(addButton);
add(departmentLabel);
add(departmentField);
add(numberLabel);
add(numberField);
}
}

```

这个测试应该可以通过。当你编译并执行 view 类时，应该看到一个与图 3 类似的窗口。

用户界面一团糟。你将在稍后的“布局”一节中对其进行矫正。而且，窗口中好像没有 JList 出现的迹象。一个原因是，你还没有向列表中添加任何元素——它是空的。你也迟早会解决这个问题。

即使只是这七个控件，你已经编写了许多乏味的代码来测试并建立用户界面。让我们看看能做些什么来加强它。

582

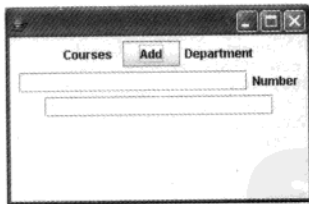


图 3

重构 (Refactory)

调用 `getComponent` 方法需要强制类型转换 (cast)。例如：

```

JTextField numberField =
    (JTextField)Util.getComponent(panel, NUMBER_FIELD_NAME);

```

因为你必须从面板中取得两个文本框，你需要有两行使用相同强制类型转换的代码。这是

一种你可以消除的重复类型,使用便捷的方法在消除强制类型转换的同时还能使代码更清晰。

独立的实用类 Util 包括了 `getComponent` 方法。不过,让 `CoursesPanel` 实现这个职责可能更合适些,因为面板是这些组件的容器。

下面的清单展示了经过大幅重构的 `CoursesPanelTest`。`CoursesPanelTest` 的某些改动,将会需要 `CoursesPanel` 也进行相应的改动——参见下一段代码清单。

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    private CoursesPanel panel;

    protected void setUp() {
        panel = new CoursesPanel();
    }

    public void testCreate() {
        assertLabelText(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);
        assertEmptyList(COURSES_LIST_NAME);
        assertButtonText(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
        assertLabelText(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
        assertEmptyField(DEPARTMENT_FIELD_NAME);
        assertLabelText(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
        assertEmptyField(NUMBER_FIELD_NAME);
    }

    private void assertLabelText(String name, String text) {
        JLabel label = panel.getLabel(name);
        assertEquals(text, label.getText());
    }

    private void assertEmptyField(String name) {
        JTextField field = panel.getField(name);
        assertEquals("", field.getText());
    }

    private void assertEmptyList(String name) {
        JList list = panel.getList(name);
        assertEquals(0, list.getModel().getSize());
    }

    private void assertButtonText(String name, String text) {
        JButton button = panel.getButton(name);
        assertEquals(text, button.getText());
    }
}
```

583

技术上讲,并非所有的重构都旨在消除重复:那些并不旨在消除重复的重构,会帮助提高代码的可读性。首先,即使只需要验证一个列表,创建 `assertEmptyList` 更清晰地表述了意

图。其次,提取 `assertButtonText` 可以让 `testCreate` 的整个方法体,只包括一行简单的、表达一致的断言语句。

每个新的断言方法都包括两个隐含的测试。首先,如果在面板内没有指定名字的组件存在,将会抛出一个 `NullPointerException`,并使测试失败。其次,如果组件不是所预期的类型,将会抛出一个 `ClassCastException`,同样使测试失败。如果这个暗喻困扰了你,只管添加一个额外的断言(例如 `assertNotNull`),但是我并不认为它们是必须的。

新的断言方法是非常通用的。当(或如果)你为 SIS 应用添加第二个面板时,你可以将这些断言方法重构为一个通用的测试实用类。你可以选择将它们移到 `junit.framework.TestCase` 的某个子类中,然后让所有的面板测试类都继承于它。

在大多数系统中,Swing 和相关测试的代码是非常庞大且多有重复的。提前花一些时间来进行重构。你可以显著地降低系统中的冗余和总体代码量。

你可以用类似的方式重构 `CoursesPanel` 中的组件创建。下面的清单除了演示测试所需的 `get` 方法,还演示了新的组件创建方法。

```
package sis.ui;

import javax.swing.*.*;
import java.awt.*.*;

public class CoursesPanel extends JPanel {
    static final String NAME = "coursesPanel";
    static final String COURSES_LABEL_TEXT = "Courses";
    static final String COURSES_LABEL_NAME = "coursesLabel";
    ...
    private void createLayout() {
        JLabel coursesLabel =
            createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

        JList coursesList = createList(COURSES_LIST_NAME);
        JButton addButton =
            createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

        int columns = 20;
        JLabel departmentLabel =
            createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
        JTextField departmentField =
            createField(DEPARTMENT_FIELD_NAME, columns);
        JLabel numberLabel =
            createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
        JTextField numberField =
            createField(NUMBER_FIELD_NAME, columns);

        add(coursesLabel);
        add(coursesList);
        add(addButton);
        add(departmentLabel);
        add(departmentField);
        add(numberLabel);
        add(numberField);
    }
}
```

```

    }

    private JLabel createLabel(String name, String text) {
        JLabel label = new JLabel(text);
        label.setName(name);
        return label;
    }

    private JList createList(String name) {
        JList list = new JList();
        list.setName(name);
        return list;
    }

    private JButton createButton(String name, String text) {
        JButton button = new JButton(text);
        button.setName(name);
        return button;
    }

    private JTextField createField(String name, int columns) {
        JTextField field = new JTextField(columns);
        field.setName(name);
        return field;
    }

    JLabel getLabel(String name) {
        return (JLabel)Util.getComponent(this, name);
    }

    JList getList(String name) {
        return (JList)Util.getComponent(this, name);
    }

    JButton getButton(String name) {
        return (JButton)Util.getComponent(this, name);
    }

    JTextField getField(String name) {
        return (JTextField)Util.getComponent(this, name);
    }
}

```

585

按钮点击与 ActionListener

当你点击一个 Swing 按钮时，应该会发生某些动作。当你点击 CoursesPanel 中的添加 (Add) 按钮时，你希望一个新的课程项出现在课程列表中。完成这一操作的代码需要三个步骤：

1. 读取课程系 (department) 和编号 (number) 文本框的内容。
2. 使用课程系和编号创建一个新的 Course 对象。
3. 将这个 Course 对象放在课程列表的 model 中。

使用这三个步骤添加新课程, 是用户界面职责与业务逻辑的混合。记住, 你或多或少希望 `CoursesPanel` 只是一个向用户显示信息的安静的类。点击一个按钮是一个 controller 事件, 你可以用一个动作 (action) 来响应它。动作的细节——业务逻辑——并不属于 `CoursesPanel`。

现在, 你还需要为 `view` 类编写一个测试。面板类中的代码在用户点击添加 (Add) 按钮时, 还需要告诉“某个人”采取行动。面板测试将检验添加 (Add) 按钮的点击会触发某些动作 (action)——已定义的动作。

你可以使用回调将按钮点击连接到一个动作 (action) 方法。为这个目的, Java 提供了 `java.awt.event.ActionListener` 接口。要实现 `ActionListener`, 你要编写动作方法 `actionPerformed`。`actionPerformed` 中的代码, 应该完成在点击按钮时你所希望发生的任何事情。

在定义了 `ActionListener` 类之后, 你可以把它的一个实例赋值给按钮。当用户点击该按钮时, `JButton` 中的逻辑会回调 `ActionListener` 对象中的 `actionPerformed` 方法。

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    private CoursesPanel panel;
    private boolean wasClicked;

    protected void setUp() {
        panel = new CoursesPanel();
    }
    ...
    public void testAddButtonClick() {
        JButton button = panel.getButton(ADD_BUTTON_NAME);

        wasClicked = false;
        panel.addCourseAddListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                wasClicked = true;
            }
        });

        button.doClick();
        assertTrue(wasClicked);
    }
}
```

你通常希望将侦听器实现为一个匿名的内联类。这里, `ActionListener` 的唯一工作是确保 `actionPerformed` 方法在按钮点击时被调用了。`JButton` 类提供了 `doClick` 方法来模拟用户对按钮的点击。

`CoursesPanel` 必须提供一个新的方法, `addCourseAddListener`。这个方法简单地将 `ActionListener` 回调对象连接到 `JButton` 对象。某些使用 `CoursesPanel` 的产品化客户端, 将负责定义这个回调并将它传递给 `view`。`View` 依然“幸福地”保持着对任何业务逻辑的一无所知。

```
package sis.ui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CoursesPanel extends JPanel {
    ...
    private JButton addButton;
    ...
    private void createLayout() {
        ...
        addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
        ...
    }

    void addCourseAddListener(ActionListener listener) {
        addButton.addActionListener(listener);
    }
    ...
}
```

为了这样做, 你必须更改 `addButton`, 使之成为一个成员变量而不是局部变量。

`addCourseAddListener` 方法只有一行代码。对布局之外的 `view` 代码来说, 这正是你的理想。如果你发现自己将 `while` 循环、`if` 语句、或者其他费解的逻辑放到 `view` 类中, 打住! 这很有可能包含了业务或应用逻辑, 你应该在其他地方表现它们。

列表 Model

你已经得到了 `view` 来表示等式的一个部分: 当用户点击添加 (Add) 按钮时, 通知“某人”做“某事”。你知道点击添加 (Add) 按钮的结果, 应该是课程面板显示一个新的课程。你希望 `view` 类将之视为两个分立的操作; 你需要在其他地方实现逻辑将二者连接起来。

◀ 588

`View` 应该允许代码传入一个 `Course` 对象, 并且作为结果, 显示该课程对象。它并不在乎 `Course` 对象是如何产生的。为 `CoursePanelTest` 添加一个新测试:

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import sis.studentinfo.*;
```

```
import static sis.ui.CoursesPanel.*;

public class CoursesPanelTest extends TestCase {
    ...
    public void testAddCourse() {
        Course course = new Course("ENGL", "101");
        panel.addCourse(course);
        JList list = panel.getList(COURSES_LIST_NAME);
        ListModel model = list.getModel();
        assertEquals("ENGL 101", model.getElementAt(0).toString());
    }
    ...
}
```

测试调用 `CoursePanel` 的 `addCourse` 方法。然后它提取出 `JList` 底层的 `model`。列表 `model` 是一个集合类，当其中的某些元素发生改变时，它会通知 `JList`。

`JList` 的 `view` 需要为 `model` 中保存的对象，显示一个有意义的表示。为了这样做，`JList` 代码调用 `model` 中每个对象的 `toString` 方法。测试的最后一行判断，`model` 中第一个对象的可打印字符串，是否由课程的系和编号串联而成。

```
package sis.ui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import sis.studentinfo.*;

public class CoursesPanel extends JPanel {
    ...
    private DefaultListModel coursesModel = new DefaultListModel();
    ...
    private void createLayout() {
        JLabel coursesLabel =
            createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

        JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
        ...
    }
    ...
    void addCourse(Course course) {
        coursesModel.addElement(course);
    }
    ...
    private JList createList(String name, ListModel model) {
        JList list = new JList(model);
        list.setName(name);
        return list;
    }
}
```

列表 `model` 是一个实现了 `javax.swing.ListModel` 接口的类。对 `JList` 对象来说，你通常使用 `ListModel` 的实现类 `DefaultListModel`。有趣的是，`ListModel` 接口没有声明任何添加元素的接口方法。`DefaultListModel` 则定义了 `(addElement)`。你可以将 `model` 引用 (`courseModel`)，声明为 `DefaultListModel` 类型，而不是 `ListModel` 接口类型。

在 `CoursePanel` 的 `createList` 方法中，在你构造 `JList` 时，将 `ListModel` 引用传递给它。`addCourse` 方法接收传入的 `Course` 对象，并将它塞进 `model` 中。

toString 的使用

我在第 9 课中提到，你不应该在产品代码中依赖 `toString` 方法。`toString` 方法通常对有关开发者调试的活动更有用。开发者可能需要将 `Course` 的输出形式为：

```
ENGL 101
```

更改为类似：

```
[Course:ENGL,101]
```

修改后的字符串可能并不适合 `CoursesPanel` 这个用户界面视图。如果两个不同的 `view` 需要以不同的格式显示 `Course` 信息，则会引起另一个冲突。

无论哪种情况，你都可以并应该引入一个显示适配器（`display adapter`）类，来包装每个课程类并提供所需的 `toString`。你可以把这些适配器对象保存在 `JList model` 中。

用户现在希望看到，在列表的每一行中，系和课程编号之前有一个连字符。修改测试，并要求使用这种新的显示格式：

590

```
public void testAddCourse() {
    Course course = new Course("ENGL", "101");
    panel.addCourse(course);
    JList list = panel.getList(COURSES_LIST_NAME);

    ListModel model = list.getModel();
    assertEquals("ENGL-101", model.getElementAt(0).toString());
}
```

适配器类的一个简单实现是，使之作为 `Course` 的子类，并覆写其 `toString` 的定义。

```
package sis.ui;

import sis.studentinfo.*;

class CourseDisplayAdapter extends Course {
    CourseDisplayAdapter(Course course) {
        super(course.getDepartment(), course.getNumber());
    }

    @Override
    public String toString() {
        return String.format(
            "%s-%s", getDepartment(), getNumber());
    }
}
```

`CoursesPanel` 的 `addCourse` 方法必须更改：


```
void addCourse(Course course) {
    coursesModel.addElement(new CourseDisplayAdapter(course));
}
```

稍后, 当你需要编写代码来取得列表框选中的 Course 对象时, 需要从它的适配器对象中得到目标的 Course。

应用

现在, view 已经就位了, 你可以指定应用如何使用他。现在, 是时候把所有部分连接在一起了²。这里是新的 SisTest 方法, testAddCourse。

591

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import sis.studentinfo.*;

public class SisTest extends TestCase {
    ...
    public void testAddCourse() {
        CoursesPanel panel =
            (CoursesPanel)Util.getComponent(frame, CoursesPanel.NAME);
        panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, "MATH");
        panel.setText(CoursesPanel.NUMBER_FIELD_NAME, "300");

        JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
        button.doClick();

        Course course = panel.getCourse(0);
        assertEquals("MATH", course.getDepartment());
        assertEquals("300", course.getNumber());
    }
}
```

测试从最终用户的立场驱动应用。这基本上是一个验收测试。

首先, 测试设置系和课程编号框的值。然后它使用按钮的 click 方法模拟对添加 (Add) 按钮的一次点击。为了确保应用运转正确, 测试要求嵌入的 CoursesPanel 返回其列表中的第一个 Course 对象。

为了支持这个测试, 你还需要为 CoursePanel 添加两个方法:

```
package sis.ui;
...
public class CoursesPanel extends JPanel {
    ...
}
```

² 取决于你自己的习惯, 你可能发现从建立应用开始继而驱动 view 的开发, 对你来说更简单。甚至在我构造 view 的时候, 我已经对如何连接应用的各个部分了然于胸了。

```

Course getCourse(int index) {
    Course adapter =
        (CourseDisplayAdapter)coursesModel.getElementAt(index);
    return adapter;
}
...
void setText(String textFieldName, String text) {
    getField(textFieldName).setText(text);
}
}

```

因为 CourseDisplayAdapter 派生自 Course，你可以将提取出的适配器对象，作为一个 Course 的引用返回。

592

Sis 中的代码改动（包括了少许重构）：

```

package sis.ui;

import javax.swing.*;
import java.awt.event.*;
import sis.studentinfo.*;

public class Sis {
    ...
    private CoursesPanel panel;
    ...

    public Sis() {
        initialize();
    }

    private void initialize() {
        createCoursesPanel();

        frame.setSize(WIDTH, HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel);
    }
    ...
    void createCoursesPanel() {
        panel = new CoursesPanel();
        panel.addCourseAddListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    addCourse();
                }
            }
        );
    }

    private void addCourse() {
        Course course =
            new Course(
                panel.getText(CoursesPanel.DEPARTMENT_FIELD_NAME),
                panel.getText(CoursesPanel.NUMBER_FIELD_NAME));
        panel.addCourse(course);
    }
}

```

Sis 类将动作侦听器 (action listener)、以及向面板添加课程的能力, 连接在一起。大部分代码应该很眼熟——与你在 CoursesPanel 的诸多测试中所建立的客户端代码类似。

你需要在 CoursesPanel 中添加 getText 方法:

```
String getText(String textFieldName) {
    return getField(textFieldName).getText();
}
```

593

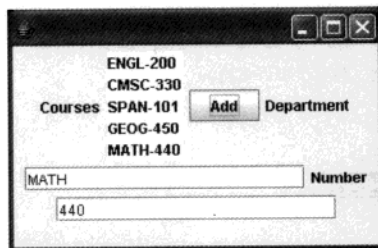


图 4

现在你可以将 Sis 作为一个独立的应用来运行, 并尝试添加课程的功能。图 4 演示了在用户输入五个课程之后, Sis 的屏幕截图。看起来有点乱!

布局

Sis 应用的用户界面布局实在是糟糕, 会让最终用户感到很困惑。症结是, Swing 缺省地将各个组件按照你将它们添加到容器的顺序、从左到右依次排开。当在一行中没有足够的空间放置组件时, Swing 会为其折行, 就如在字处理软件中那样。折行后, 组件会出现在面板当前行下方的左侧。

在图 4 中, 第一行包括四个部件 (widget): “Courses” 标签、课程的列表、Add 按钮和 “Department” 标签。第二行包括系的文本框、“Number” 标签以及课程编号的文本框。

改变窗口的大小使之和你的屏幕一样宽。Swing 会重绘这些部件。如果你的屏幕足够宽, 所有组件就会从左到右排列在一行中。

Swing 提供了几种可替换的布局机制, 每一种都位于一个单独的类中, 帮助你生成赏心悦目的用户界面。Swing 将这些类称为布局管理器 (layout manager)。你可以为每个容器关联不同的布局管理器。缺省的布局管理器是 java.awt.FlowLayout, 如果你希望创建外观专业的用户界面, 它对你没什么帮助。

594

要得到一个看起来“刚刚好”的 view，是一个递增的、繁重的练习。你会发现，对复杂的 view，混合并匹配各种布局，与始终坚持一种布局相比来说，是更容易的策略。

手写布局的一种替代方式是使用工具。许多 IDE 提供了 GUI (view) 编写工具，可以让你可视地编辑一个布局。使用工具可以减少尝试建立完美用户界面时的冗长与乏味。

在让 CoursesPanel 变得好看些的尝试中，你将学习使用一些更有意义的布局机制。这项工作很少需要你测试为先。相反，你应该测试延后。进行少量的改动，编译，运行你的测试，将 CoursesPanel 作为一个单独的应用来执行，查看结果，再修改！

GridLayout

你首先从一个容易理解但不太合用的布局——GridLayout——开始。基于你指定的行与列的数目，GridLayout 将容器划分为大小相等的矩形。当你向容器中添加组件时，GridLayout 将每个组件放到一个单元格（矩形）中，缺省按照从左到右从上到下的顺序。布局管理器会调整每个组件的大小，以适合它的单元格。

在 CoursesPanel 中进行下面的更改：

```
private void createLayout() {
    JLabel coursesLabel =
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
    addButton =
        createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
    int columns = 20;
    JLabel departmentLabel =
        createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    JTextField departmentField =
        createField(DEPARTMENT_FIELD_NAME, columns);
    JLabel numberLabel =
        createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
    JTextField numberField =
        createField(NUMBER_FIELD_NAME, columns);

    int rows = 4;
    int cols = 2;
    setLayout(new GridLayout(rows, cols));

    add(coursesLabel);
    add(coursesList);
    add(addButton);
    add(new JPanel());
    add(departmentLabel);
    add(departmentField);
    add(numberLabel);
    add(numberField);
}
```

595

通过调用面板的 `setLayout` 方法, 将一个布局管理器指派给面板。在 `createLayout` 中, 你调用 `CoursesPanel` 对象的 `setLayout` 方法, 并传入 `GridLayout` 的一个实例。

使用 `GridLayout` 的 `CoursesPanel` 的执行结果, 如图 5 所示:

`coursesLabel` 最终位于左上角的矩形中。添加的第二个组件, `coursesList`, 位于右上角的矩形中。`Add` 按钮顺延第二行的矩形, 并且紧跟这个空的 `JPanel` 来填充下一个矩形。最后两行的每一行, 显示一个标签和相应的文本框。

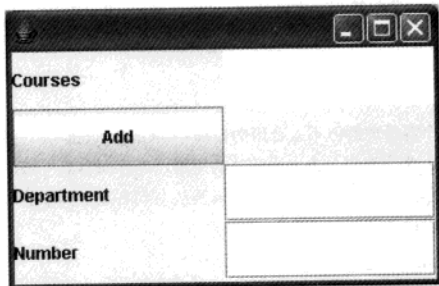


图 5

因为每个矩形都必须有相同的大小, `GridLayout` 并不具备能力去组织“典型”的、含有许多按钮、文本框、列表和标签的界面。当你向最终用户显示一组图标时, 它很合适。`GridLayout` 的确包括一些附加的方法来改进其外观, 但通常你会希望使用一个更完备的布局管理器。

BorderLayout

`BorderLayout` 是一个简单但非常有效的布局管理器。`BorderLayout` 允许你在容器内放置至多五个组件: 其中四个位于罗盘位置: 北 (north)、东 (east)、南 (south) 和西 (west), 另一个填充剩余的部分或中央 (center), 见图 6。

至于 `CoursesPanel`, 你要将 `GridLayout` 替换为 `BorderLayout`。你的 `BorderLayout` 将使用三个可用的区域: `north`, 用来包含“Courses”标签; `center`, 用来包含课程的列表; 以及 `south`, 用来包含剩下的部件。你将要组织南侧, 或者“底部 (bottom)”, 对子面板中的部件使用一个独立的布局管理器。

`createLayout` 方法已经过长了, 接近 30 行。到目前为止, `CoursesPanel` 还是一个简单的界面。设想一个含有几打控件的复杂面板。不幸的是, 我们常常看到 `Swing` 代码将所有必须的初始化和布局放在一个方法中。在这个方法中, 开发者们在面板内创建面板、创建及初始化组件、并将它们放到面板中, 创建布局, 凡此种种。

一个更有效的代码组合方式是，将每个面板的创建提取到一个单独的方法中。这只是为了让代码更可读，不过在重新安排布局时，提供了更多的灵活性。我重构了 `CoursesPanel` 中的代码，反映了这种较整洁的代码组织。

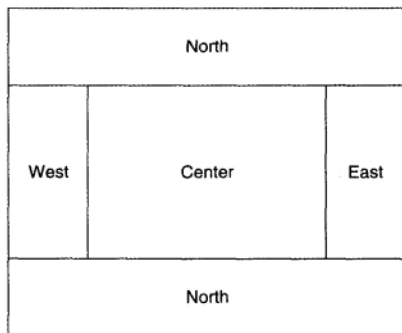


图 6 BorderLayout 的配置

```

private void createLayout() {
    JLabel coursesLabel =
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);

    setLayout(new BorderLayout());

    add(coursesLabel, BorderLayout.NORTH);
    add(coursesList, BorderLayout.CENTER);
    add(createBottomPanel(), BorderLayout.SOUTH);
}

JPanel createBottomPanel() {
    addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());

    panel.add(addButton, BorderLayout.NORTH);
    panel.add(createFieldsPanel(), BorderLayout.SOUTH);

    return panel;
}

JPanel createFieldsPanel() {
    int columns = 20;
    JLabel departmentLabel =
        createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    JTextField departmentField =
        createField(DEPARTMENT_FIELD_NAME, columns);
    JLabel numberLabel =
        createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
  
```

```

    JTextField numberField =
        createField(NUMBER_FIELD_NAME, columns);

    int rows = 2;
    int cols = 2;

    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(rows, cols));
    panel.add(departmentLabel);
    panel.add(departmentField);

    panel.add(numberLabel);
    panel.add(numberField);

    return panel;
}

```

CoursesPanel 构造函数中的代码，将它的布局设置为 BorderLayout 的一个新实例。它将标签放在面板的北边（顶部），课程列表放在面板的中央。它将 createBottomPanel 方法的返回结果，也就是另一个 JPanel，放在面板的南边（底部），参见图 7。将列表放在中央的好处是，它会随着 frame 窗口的扩大而延展。其他部件则保持它们原来的大小。

598

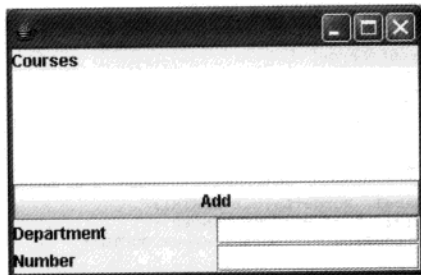


图 7

createBottomPanel 中的代码创建了一个 JPanel，它也使用了 BorderLayout 来组织它的组件。它将添加按钮放在北边，并将 createFieldsPanel 返回的 JPanel 放在南边。createFieldsPanel 使用 GridLayout 来组织系和课程编号的标签与文本框。结果（图 7）有相当大的改进，但是还是不够好。再一次，确保在你体验改变 frame 窗口大小时，留意布局是如何反应的。

一个测试程序

如果你重新运行测试，你将会得到三个 NullPointerException 错误。怎么会这样呢，因为你并没有更改逻辑或添加/删除任何组件？

当你调查 `NullPointerException` 的栈回溯 (stack trace) 时, 你应该发现从容器中取出组件的某些 `get` 方法失败了。问题在于, `Util` 的 `getComponent` 方法只查找直接嵌套在容器中的组件。现在你的代码在容器中又嵌套了容器 (`JPanel` 中嵌套 `JPanel`)。 `getComponent` 中的代码忽略了那些加入到子面板中的组件。

`Util` 类没有相关的任何测试。此时, 为了帮助解决这个问题, 并加强你的测试覆盖, 你需要添加适当的测试。`UtilTest` 包括三个测试, 应该覆盖了大多数的预期情况:

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;

public class UtilTest extends TestCase {
    private JPanel panel;

    protected void setUp() {
        panel = new JPanel();
    }

    public void testNotFound() {
        assertNull(Util.getComponent(panel, "abc"));
    }

    public void testDirectlyEmbeddedComponent() {
        final String name = "a";
        Component component = new JLabel("x");
        component.setName(name);
        panel.add(component);
        assertEquals(component, Util.getComponent(panel, name));
    }

    public void testSubcomponent() {
        final String name = "a";
        Component component = new JLabel("x");
        component.setName(name);

        JPanel subpanel = new JPanel();
        subpanel.add(component);

        panel.add(subpanel);

        assertEquals(component, Util.getComponent(panel, name));
    }
}
```

第三个测试, `testSubcomponent`, 应该会失败, 原因和你其他测试失败的原因一样。为了解决这个问题, 你需要修改 `getComponent`。对容器中的每个组件, 你需要判断它是否是一个容器 (使用 `instanceof`)。如果是, 你需要遍历子容器的所有组件, 为每个子容器重复相同的过程。达到这一目的最有效的方法是, 递归调用 `getComponent`。

```
static Component getComponent(Container container, String name) {
```



```

for (Component component: container.getComponents()) {
    if (name.equals(component.getName()))
        return component;
    if (component instanceof Container) {
        Container subcontainer = (Container) component;
        Component subcomponent = getComponent(subcontainer, name);

        if (subcomponent != null)
            return subcomponent;
    }
}
return null;
}

```

进行这个更改之后，你的测试应该可以全部通过了。

BoxLayout

BoxLayout 类可以让你将组件按照水平或垂直轴排列。当你改变容器的大小时，组件不会换行；而且，组件也不会增长来填充空白区域。底部的面板，必须将添加按钮和文本框子面板垂直放置，一个叠一个，BoxLayout 是理想的候选布局。

```

JPanel createBottomPanel() {
    addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

    JPanel panel = new JPanel();
    panel.setLayout(new BoxLayout(panel, BoxLayout.PAGE_AXIS));

    panel.add(Box.createRigidArea(new Dimension(0, 6)));
    addButton.setAlignmentX(Component.CENTER_ALIGNMENT);
    panel.add(addButton);
    panel.add(Box.createRigidArea(new Dimension(0, 6)));
    panel.add(createFieldsPanel());

    panel.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));

    return panel;
}

```

你必须将面板的一个实例传入到 BoxPanel 的构造函数中，还有一个常量标识组件的布局方向。常量 PAGE_AXIS 缺省地表示从上到下，或垂直方向。另一个选项是 LINE_AXIS，缺省表示水平方向。³

你可以创建一些不可见的“固定区域 (rigid area)”，通过空白把组件隔开。这些固定的区域，即使在容器改变大小时，依然会保持固定的大小。类方法 createRigidArea 接收一个 Dimension 对象作为参数。Dimension 包括 width (宽，本例中为 0) 和 height (高，本例中为 6)。

你可能希望将每个组件与坐标轴对齐。在本例中，通过调用 Add (添加) 按钮的

³ 你可以通过调用容器的 applyComponentOrientation 方法改变方向。BoxLayout 的较早版本只支持显式的 X_AXIS 和 Y_AXIS 常量，新的常量允许动态重组，可能是为了国际化支持的需要。

setAlignmentX 方法，并以 Component.CENTER_ALIGNMENT 为参数，将添加按钮按纵向向中央对齐。

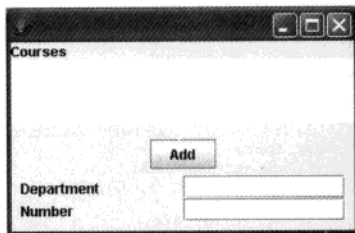


图 8 使用 BoxLayout

最后一个调整是，为整个面板提供一个不可见的边框。BorderFactory 类可以提供若干类型的边框，你可以将它们传递给面板的 setBorder 方法。创建一个空的边框，需要四个参数，每个参数代表和面板外部边框空白的宽度。其他你可以创建的边框，包括斜面边框（beveled border），线边框（line border），复合边框（compound border），风蚀边框（etched border），粗糙面边框（matte border），凸起斜面边框（raised beveled border）以及有标题的边框（titled border）。查看 Java 的 API 文档，并体验使用不同边框所产生的效果。

现在的代码生成了一个有效的、但是并不十分完美的布局。参见图 8。

GridBagLayout

为了更精细地调整布局，你将不得不使用 GridBagLayout，一种高度可配置但更复杂的布局管理器。GridBagLayout 让你可以将组件组织到矩形的栅格中，这一点和 GridLayout 类似。不过，GridBagLayout 给了你更多的控制。首先，每个矩形并不是固定大小的，其大小通常基于缺省的、或者它所包含组件的“推荐”（preferred）设置。组件可以跨越多个行或列。

修改后的 createFieldsPanel 代码演示了如何使用 GridBagLayout 来布局标签和文本框。（方法中的代码假定你已经静态地引入了 java.awt.GridBagConstraints.*）

```

JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
    int columns = 20;
    JLabel departmentLabel =
        createLabel(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    JTextField departmentField =
        createField(DEPARTMENT_FIELD_NAME, columns);
    JLabel numberLabel =

```

◀ 602

```

        createLabel(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
        JTextField numberField =
            createField(NUMBER_FIELD_NAME, columns);

        layout.setConstraints(departmentLabel,
            new GridBagConstraints(
                0, 0, // x, y
                1, 1, // gridwidth, gridheight
                40, 1, // weightx, weighty
                LINE_END, //anchor
                NONE, // fill
                new Insets(3, 3, 3, 3), // top-left-bottom-right
                0, 0)); // padx, ipady
        layout.setConstraints(departmentField,
            new GridBagConstraints(1, 0, 2, 1, 60, 1,
                CENTER, HORIZONTAL,
                new Insets(3, 3, 3, 3), 0, 0));
        layout.setConstraints(numberLabel,
            new GridBagConstraints(0, 1, 1, 1, 40, 1,
                LINE_END, NONE,
                new Insets(3, 3, 3, 3), 0, 0));
        layout.setConstraints(numberField,
            new GridBagConstraints(1, 1, 2, 1, 60, 1,
                CENTER, HORIZONTAL,
                new Insets(3, 3, 3, 3), 0, 0));

        panel.add(departmentLabel);
        panel.add(departmentField);
        panel.add(numberLabel);
        panel.add(numberField);

        return panel;
    }

```

在创建 `GridBagLayout` 并将之设置给面板之后，你需要为添加的每个部件调用布局的 `setConstraints` 方法。`setConstraints` 方法接收两个参数：一个 `Component` 对象和一个 `GridBagConstraints` 对象。`GridBagConstraints` 包括了若干 `Component` 对象的约束。下面的表格非常简要地总结了这些约束（参见 API 文档来查看完整的细节）：

<code>gridx/gridy</code>	组件开始绘制的单元格。左上方的单元格是 0,0
<code>gridwidth/gridheight</code>	组件要跨越的行或列的数目。每个组件的缺省值为 1，意味着一个组件只占据一个单元格。
<code>weightx/weighty</code>	权重（weight）约束被用来决定，如果在一行或一列中排列了所有组件之后还有剩余空间，需要为组件分配多少附加的空间。
<code>anchor</code>	<code>anchor</code> 约束被用来决定，如果组件比它的显示区域要小，如何将它放置到单元格中。缺省为 <code>GridBagConstraints.CENTER</code> 。
<code>fill</code>	<code>fill</code> 约束指定了，如果组件比它的显示区域要小，组件应该如何增长来填充显示区域。它的值包括 <code>NONE</code> （不改变大小）、 <code>HORIZONTAL</code> 、 <code>VERTICAL</code> 和 <code>BOTH</code> 。

insets	使用一个 Insets 对象来指定组件与其显示区域边缘之间的间距。
ipadx/ipady	指定为组件的最小尺寸增加多少间距。

GridBagConstraints 中的每个字段都是 public。你可以构造一个 GridBagConstraints 对象而不使用任何参数，然后按需要设置单个的成员变量。或者，像在 createFieldsPanel 清单中所做的那样，你可以再使用另一个 GridBagConstraints 构造函数来指定所有可能的约束。

最好的策略是在纸或白板上，勾画出一个你希望看到的栅格表示输出。使用 gridx/gridy 和 gridwidth/gridheight 约束来决定组件的相对大小和位置。然后集中精力在每个组件的 anchor 和 fill 方面。为这些规格的草图编写布局代码，并在需要时进行修改。然后你可以体验 insets 和 weightx/weighty 约束（有时也包括 ipadx/ipady 约束）来调整组件之间的间距。

显然，createFieldsPanel 中有大量冗余。下面是经过适当重构后的代码。

604

```

JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
    int columns = 20;

    addField(panel, layout, 0,
        DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT,
        DEPARTMENT_FIELD_NAME, columns);
    addField(panel, layout, 1,
        NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT,
        NUMBER_FIELD_NAME, columns);

    return panel;
}

private void addField(
    JPanel panel, GridBagLayout layout, int row,
    String labelName, String labelText,
    String fieldName, int fieldColumns) {
    JLabel label = createLabel(labelName, labelText);
    JTextField field = createField(fieldName, fieldColumns);

    Insets insets = new Insets(3, 3, 3, 3); // top-left-bottom-right
    layout.setConstraints(label,
        new GridBagConstraints(
            0, row, // x, y
            1, 1, // gridwidth, gridheight
            40, 1, // weightx, weighty
            LINE_END, //anchor
            NONE, // fill
            insets, 0, 0)); // padx, ipady
    layout.setConstraints(field,
        new GridBagConstraints(1, row,
            2, 1, 60, 1, CENTER, HORIZONTAL,
            insets, 0, 0));
}

```

```

panel.add(label);
panel.add(field);
}

```

这些代码繁复的本性，会将你导向极限（extreme）重构的方向。考虑通过使用一个简化的实用构造函数，来替换 GridBagConstraints 对象构造中的重复。如果你需要显示多于一组文本框和相关联的标签，考虑使用一个数据类来表示每个文本框与标签对儿。然后你可以在一个表格中显示整个集合，并遍历这个集合来创建布局。

图 9 展示了一个接近可接受的布局。改变它的大小，来查看文本框组件是如何填充它们的显示区域的。

605

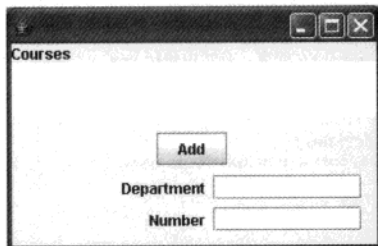


图 9 使用 GridBagLayout

从 Java 1.4 开始，Sun 引入了 SpringLayout 类。这个布局管理器主要设计用于 GUI 编写工具。SpringLayout 的基本概念是，通过使用一种称为 spring 的约束，将组件边缘连接在一起来定义布局。

在 CoursesPanel 中，你可以创建一个 spring，将系文本框的西（左）侧边缘和系标签的东（右）侧边缘连接起来。spring 对象以固定大小的五个像素，将两个组件连接起来。另一个 spring 可能将系文本框的东侧边缘，连接到面板本身的东侧边缘，同样也由固定长度的 spring 分隔开。随着面板在宽度上增加，department 文本框也会相应的增长。

手工创建一个 SpringLayout，对于在一个面板中排列少数组件来说，还算简单。对更复杂的布局来说，它将是难以置信的困难并让人充满挫败感。在大多数情况下，你会希望将这个工作交给布局工具。

继续前进

关于 Swing，到目前你还只是浅尝即止！在下一章中，你将使用一些微调来加强 CoursesPanel 的外观和体验（look and feel）。然后，我们将浏览更多广泛有用的 Swing 课题。

606

II

Swing, Part 2

Swing，第二部分

在上一课中，我们学习了如何使用面板（panel）、标签（label）、按钮（button）、文本框（text field）和列表（list），建立一个基本的 Swing 应用。你还学习了如何使用 Swing 的布局管理器来增强用户界面（UI，User Interface）的外观。

在本章中，你将学习到：

- 滚动面板
- 边框（border）
- 设置标题栏的文本
- 图标
- 键盘支持
- 按钮的助记符（mnemonic）
- 必须填写的文本框（Required Field）
- 键盘侦听器
- Swing 的 Robot 类
- 文本框校订（Field edit）和文档过滤器
- 格式化的文本框
- 表格
- 鼠标侦听器
- 光标
- SwingUtilities 方法：invokeAndWait 和 invokeLater



界面美化的杂项

在本节中，你将学习如何增强 `CoursesPanel` 现有的外观。

JScrollPane

如果你使用 `sis.ui.Sis` 添加了很多门课程，你会发现 `JList` 只能显示其中的几个。为了解决这个问题，你需要将 `JList` 包装 (`wrap`) 到一个滚动面板中。滚动面板起到了作为列表视口 (`viewport`) 的作用。当列表 `model` 所保存的信息，多于当前 `JList` 的大小可以显示的条目时，滚动面板将绘制滚动条。滚动条可以让你水平地或垂直地移动视口，显现出隐藏的信息。

`CoursesPanel` 清单中以粗体显示的代码，添加了滚动面板。你可以使用 `setVerticalScrollBarPolicy` 或 `setHorizontalScrollBarPolicy`，以指定希望滚动条一直显示、还是只在需要时显示。我发现始终显示垂直的滚动条，更为美观。

```
private void createLayout() {
    JLabel coursesLabel =
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
    JScrollPane coursesScroll = new JScrollPane(coursesList);
    coursesScroll.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

    setLayout(new BorderLayout());

    add(coursesLabel, BorderLayout.NORTH);
    add(coursesScroll, BorderLayout.CENTER);
    add(createBottomPanel(), BorderLayout.SOUTH);
}
```

边框 (border)

课程列表和相关联的标签，直接紧邻面板的边缘。你可以使用边框 (`border`)，在面板边缘与其任意子组件之间，创建一个缓冲地带。

```
private void createLayout() {
    JLabel coursesLabel =
        createLabel(COURSES_LABEL_NAME, COURSES_LABEL_TEXT);

    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
    JScrollPane coursesScroll = new JScrollPane(coursesList);
    coursesScroll.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
}
```

```

setLayout(new BorderLayout());

final int pad = 6;
setBorder(BorderFactory.createEmptyBorder(pad, pad, pad, pad));

add(coursesLabel, BorderLayout.NORTH);
add(coursesScroll, BorderLayout.CENTER);
add(createBottomPanel(), BorderLayout.SOUTH);
}

```

你可以使用 `BorderFactory` 类创建几种不同类型的边框。大多数边框只是为了修饰目的；不过空边框是一个例外。你可以使用 `createCompoundBorder` 来创建复合的边框。改写的 `createLayout` 方法演示了如何使用几种不同的边框类型。

```

private void createLayout() {
    JList coursesList = createList(COURSES_LIST_NAME, coursesModel);
    JScrollPane coursesScroll = new JScrollPane(coursesList);
    coursesScroll.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

    setLayout(new BorderLayout());

    final int pad = 6;
    Border emptyBorder =
        BorderFactory.createEmptyBorder(pad, pad, pad, pad);
    Border bevelBorder =
        BorderFactory.createBevelBorder(BevelBorder.RAISED);
    Border titledBorder =
        BorderFactory.createTitledBorder(bevelBorder, COURSES_LABEL_TEXT);
    setBorder(BorderFactory.createCompoundBorder(emptyBorder,
        titledBorder));

    add(coursesScroll, BorderLayout.CENTER);
    add(createBottomPanel(), BorderLayout.SOUTH);
}

```

使用有标题的边框，避免了使用一个单独的 `JLabel` 来显示“Courses:”文本。去除掉的 `JLabel` 将会打断 `CoursesPanelTest` 中的 `testCreate` 测试——还记得在进行这个更改时，你需要测试为先吗？

添加标题

SIS frame 窗口的标题栏上没有任何文本。通过调用 `SisTest` 中的 `testCreate` 来矫正它。

◀ 609

```

public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
        frame.getDefaultCloseOperation());
    assertNotNull(Util.getComponent(frame, CoursesPanel.NAME));
    assertEquals(Sis.COURSES_TITLE, frame.getTitle());
}

```


JFrame 提供了构造函数, 允许你传入一个标题栏文本。Sis 的代码如下:

```
public class Sis {
    ...
    static final String COURSES_TITLE = "Course Listing";
    private JFrame frame = new JFrame(COURSES_TITLE);
    ...
}
```

图标 (icon)

你需要为窗口添加的最后一个美化元素, 是图标。你缺省会得到一个 cup-o-Java 图标 (译注: Java 的商标图案), 当你最小化窗口时, 它作为迷你图标出现在标题栏中。因为图标是标题栏的一部分, 它置于 frame 窗口的控制之下。

测试可以简单地通过调用 frame 的 `getIconImage` 方法来得到图标。这个方法, 在 `java.awt.Frame` 中实现, 返回一个 `java.awt.Image` 类型的对象。`testCreate` 中的代码, 断言 (`assert`) 从 SIS frame 中取得的图标、是否和按名字显式加载的一样。测试和 `sis.ui.Sis` 代码都使用了通用的实用方法来加载图像: `ImageUtil.create`。

```
public void testCreate() {
    final double tolerance = 0.05;
    assertEquals(Sis.HEIGHT, frame.getSize().getHeight(), tolerance);
    assertEquals(Sis.WIDTH, frame.getSize().getWidth(), tolerance);
    assertEquals(JFrame.EXIT_ON_CLOSE,
        frame.getDefaultCloseOperation());
    assertNotNull(Util.getComponent(frame, CoursesPanel.NAME));
    assertEquals(Sis.COURSES_TITLE, frame.getTitle());

    Image image = frame.getIconImage();
    assertEquals(image, ImageUtil.create("/images/courses.gif"));
}
```

在 `sis.util` 包中创建 `ImageUtilTest`。为 `create` 方法编写测试, 有多种方式。最好的方式是, 使用像素的集合动态地生成一副图像, 并把它写入到磁盘中。在 `create` 方法加载图像之后, 你可以断言所加载图像中的像素, 与原来的集合中的像素集合是相同的。不幸的是, 这是一个牵扯甚多的方案, 使用一个简化的图像动态创建图像的 API, 是能做到的最好解决了。

一种更简单的技术是, 让测试假定磁盘中已经存在了正确的图像, 且文件名是人所共知的。然后测试可以简单地通过确保加载后的图像不为 `null`, 来断言图像是否加载成功。图像必须位于 `classpath` 路径中。¹

```
package sis.util;

import junit.framework.*;
```

¹ 我为这一节修改了 Ant 的 `build.xml` 脚本, 在每次编译执行时, 将 `src/images` 中的所有文件拷贝到 `classes/images` 中。这可以让你快速地删除 `classes` 目录中的所有内容, 而不必担心遗留了任何图像。

```
import java.awt.*;

public class ImageUtilTest extends TestCase {
    public void testLoadImage() {
        assertNull(ImageUtil.create("/images/bogusFilename.gif"));
        assertNotNull(ImageUtil.create("/images/courses.gif"));
    }
}
```

这看起来有些脆弱，但是目前它提供了一种有效的方案。你必须确保图像 `courses.gif` 在项目的整个生命期中一直存在。你可以考虑创建一个明确为测试目的使用的图像，而不使用与 SIS 项目有关的图像文件。

Java 提供了几种不同的方法来加载和操作图像。你可以直接使用其中的大部分。

```
package sis.util;

import javax.swing.*;
import java.awt.*;

public class ImageUtil {
    public static Image create(String path) {
        java.net.URL imageURL = ImageUtil.class.getResource(path);
        if (imageURL == null)
            return null;
        return new ImageIcon(imageURL).getImage();
    }
}
```

611

Class 类定义了 `getResource` 方法。这个方法允许你定位资源、(包括文件)，不管应用是从 JAR 文件、单独的类文件、或者诸如 Internet 的其他资源中加载²。调用 `getResource` 的结果是一个 URL——资源的唯一地址。

一旦你得到了正确的 URL，你可以将它传递给 `ImageIcon` 的构造函数，来创建一个 `ImageIcon` 对象。你可以为许多目的使用 `ImageIcon` 对象，例如修饰按钮或标签。因为 `Frame` 类需要一个 `Image` 对象，而不是 `ImageIcon`，你需要使用 `getImage` 方法来处理 `create` 的返回值。

注意传入 `getResource` 的图像文件名是 `/images/courses.gif`——这个路径的起始是一个斜线 (/)。这意味着资源应该按每个 classpath 项的根 (root) 路径进行定位。这样，如果你从 `c:\swing2\classes` 目录中启动这些类，你应该将 `courses.gif` 放到 `c:\swing2\classes\images` 中。如果你通过从 JAR 文件中加载来执行这些类，它应该在相对目录 `images` 中含有 `courses.gif`。

下面是在 `Sis` 类的 `initialize` 方法中的更改。

```
private void initialize() {
    createCoursesPanel();

    Image image = ImageUtil.create("/images/courses.gif");
    frame.setIconImage(image);
}
```

² 从技术角度来说，你通过 Class 对象调用 `GetResource`，图像则是使用其类加载器进行加载的。对于没有使用定制的分类加载器的应用程序来说，使用 `ImageUtil` 类的类加载器是完全可以的。

```
frame.setSize(WIDTH, HEIGHT);  
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
frame.getContentPane().add(panel);  
}
```

至少从视觉的角度来说, 现在你已经有一个可接收的、令人愉悦的界面了。修正后的布局如图 1 所示。

体验 (feel)

界面上的视觉表现是重要的, 但是更重要的是它的“体验 (feel)”。应用的体验是, 用户与界面交互所产生的经历。与应用的体验相关的元素示例包括:

612

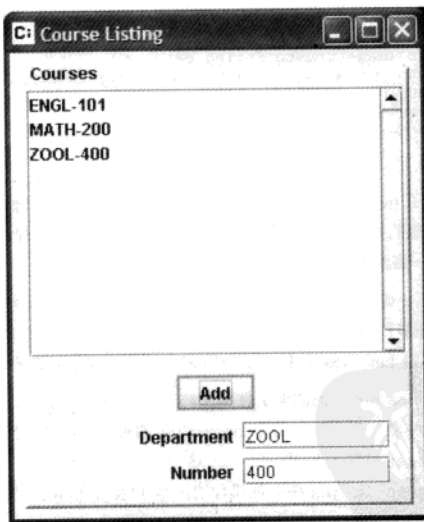


图 1 一个优良的外观

- 是否具备使用键盘或鼠标来实现所有行为的能力?
- 用户是否可以使用 tab 序列, 以适当的次序访问所有的文本框, 并在 tab 序列中略去不相关的项?
- 是否限制用户在文本框中输入过多的字符?

- 是否约束用户在文本框中输入不恰当的数据?
- 已激活的按钮在不适用时是否会被禁用 (deactivate)?

同时处理外观和体验 (look and feel) 是有可能的。在上一节中, 你使用了一个滚动面板来修饰课程的列表。这提高了界面的外观, 同时也提供了让用户在需要时滚动课程列表的“体验”。

◀ 613

键盘支持

GUI 的一个基本原则是, 用户必须能够使用键盘或鼠标对应用进行完全的控制。例外是有的。使用键盘绘制多边形是非常低效的, 正如使用鼠标来输入字符一样。(当然两者都是可以支持的)。在这种情况下, 应用开发者通常选择绕过 (bypass) 这条规则。但是在大多数情况下, 忽略只能使用键盘或只能使用鼠标的用户, 是非常不替他人着想的。

缺省的, Java 为键盘和鼠标的双重控制提供了大部分必需的支持。例如, 你可以通过使用鼠标, 或者用 tab 键定位然后按下空格键, 来激发一个按钮。

按钮助记符 (Mnemonic)

另一种激发按钮的常用方法, 是使用 Alt 组合键。你可以同时按下 Alt 键和其它键。这个键通常是一个包含在按钮文本中的字母或者数字。我们将这种键称为助记符 (mnemonic, 技术上讲, 是帮助你记忆某件事的一个设备; 在 Java 中, 它只是一个单字母的快捷方式)。对添加 (Add) 按钮来说, 字母 A 是一个合适的助记符。

助记符是一个 view 类的元素。View 的外观决定了助记符的规格。因此, 对测试和管理助记符来说, view 类是更合适的地方。

在 CoursesPanelTest 中:

```
public void testCreate() {
    assertEmptyList(COURSES_LIST_NAME);
    assertButtonText(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
    assertLabelText(DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT);
    assertEmptyField(DEPARTMENT_FIELD_NAME);
    assertLabelText(NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT);
    assertEmptyField(NUMBER_FIELD_NAME);

    JButton button = panel.getButton(ADD_BUTTON_NAME);
    assertEquals(ADD_BUTTON_MNEMONIC, button.getMnemonic());
}
```

在 CoursesPanel 中:

```
public class CoursesPanel extends JPanel {
    ...
    static final char ADD_BUTTON_MNEMONIC = 'A';
}
```

```

...
JPanel createBottomPanel() {
    addButton = createButton(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
    addButton.setMnemonic(ADD_BUTTON_MNEMONIC);
    ...
    return panel;
}
...
}

```

当你看到测试通过之后, 以应用的方式运行 `sis.ui.Sis`。输入课程的系与编号, 然后按下 `Alt-A` 来演示助记符的使用。

必须填写的文本框 (Required Field)

一个有效的课程项, 需要同时包括系和课程编号。不过, `sis.ui.Sis` 现在允许你跳过任意一个或者全部, 并且还可以按下 `Add` 按钮。你可以修改应用来禁止这种情况。

一种方案是, 等到用户点击 `Add` 时, 确保系与课程编号文本框中含有非空的字符串。否则, 显示一个弹出的消息, 向用户说明要求。这种方案当然有效, 但是它产生了多余的、恼人的用户体验。用户不希望频繁地被弹出消息所打断。一个较好的方案是, 在用户于两个文本框输入信息之前, 禁用 `Add` 按钮。

你可以监视这两个文本框, 并跟踪用户在其中输入的信息。每当用户按下一个字母时, 你可以测试这个文本框的内容, 并适时地启用或禁用 `Add` 按钮。

管理 `Add` 按钮的启用和禁用, 是应用而非 `view` 所特有的逻辑。它涉及到了业务相关的逻辑, 因为它基于对特定数据的业务需要。同样的, 测试和相关代码不属于面板类, 而应放在其他地方。

另一个表明代码不属于 `view` 类的标识是, 你需要在两个组件之间进行交互。你希望让 `controller` 将事件 (键被按下) 通知给其他类, 然后让这个类告诉 `view` 在特定情况下如何显示 (启用或禁用按钮)。这是两个不同的关注 (concern)。你不希望 `view` 中的逻辑试图仲裁这样的事务。

不过, 为了解决更大的问题, 为 `CoursesPanel` 提供测试来检验这两种情况更容易些。第一个测试确保当键被按下时, 侦听器得到通知了。第二个测试确保 `CoursesPanel` 可以启用和禁用按钮。

从一个测试 (在 `CoursesPanelTest` 中) 开始——启用和禁用按钮。

```

public void testEnableDisable() {
    panel.setEnabled(ADD_BUTTON_NAME, true);
    JButton button = panel.getButton(ADD_BUTTON_NAME);
    assertTrue(button.isEnabled());

    panel.setEnabled(ADD_BUTTON_NAME, false);
    assertFalse(button.isEnabled());
}

```

`CoursesPanel` 中的代码，是只有一行的响应方法——没有任何逻辑：

```
void setEnabled(String name, boolean state) {
    getButton(name).setEnabled(state);
}
```

第二个测试演示了如何为文本框关联一个按键侦听器：

```
public void testAddListener() throws Exception {
    KeyListener listener = new KeyAdapter() {};
    panel.addFieldListener(DEPARTMENT_FIELD_NAME, listener);
    JTextField field = panel.getField(DEPARTMENT_FIELD_NAME);
    KeyListener[] listeners = field.getKeyListeners();
    assertEquals(1, listeners.length);
    assertEquals(listener, listeners[0]);
}
```

`KeyListener` 是一个实现了 `KeyListener` 接口的抽象实现，它什么都不做。测试中的第一行创建了 `KeyListener` 的一个具体子类，没有覆写任何方法。在向面板添加这个侦听器（使用 `addFieldListener`）之后，测试确保面板将侦听器正确地设置给文本框。`CoursesPanel` 中的代码同样很简单：

```
void addFieldListener(String name, KeyListener listener) {
    getField(name).addKeyListener(listener);
}
```

较困难的测试是在应用层。`Sis` 对象中的某个侦听器，应该在用户在系或编号文本框输入时，收到消息。你必须检验侦听器收到了这些消息，并触发了启用/禁用 `Add` 按钮的逻辑。你还必须要检验，在输入框中有无文本的各种组合，是否导致了 `Add` 按钮的适当状态。

在 `SisTest` 中需要特意编写一些代码，为你提供一个测试的骨架：

```
public void testKeyListeners() throws Exception {
    sis.show();

    JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
    assertFalse(button.isEnabled());
    selectField(CoursesPanel.DEPARTMENT_FIELD_NAME);
    type('A');
    selectField(CoursesPanel.NUMBER_FIELD_NAME);
    type('1');
    assertTrue(button.isEnabled());
}
```

测试确保按钮缺省是禁用的。在向系和编号文本框中输入一些值之后，测试验证按钮是启用的。当然，技巧是如何选择一个文本框并模拟在其中按键。`Swing` 提供了若干方案。不幸的是，每种方案都需要你渲染实际的屏幕（使其可见）。因而，测试的第一行调用 `Sis` 的 `show` 方法。

我要展示的方案，涉及了对 `java.awt.Robot` 的使用。`Robot` 类模拟了最终用户使用键盘和/或鼠标的交互。另一个方案需要你创建键盘事件对象，并使用 `java.awt.Component` 中的 `dispatchEvent` 将它们传递给文本框。

你可以在 `SisTest` 的 `setUp` 方法中构造一个 `Robot` 对象。(在建立这个示例之后, 我注意到总是频繁使用 `CoursesPanel` 对象, 因此我还对其进行重构, 将它提取到 `setUp` 中。)

```
public class SisTest extends TestCase {
    ...
    private CoursesPanel panel;
    private Robot robot;

    protected void setUp() throws Exception {
        ...
        panel = (CoursesPanel)Util.getComponent(frame, CoursesPanel.NAME);
        robot = new Robot();
    }
}
```

`selectFiled` 方法也并不困难:

```
private void selectField(String name) throws Exception {
    JTextField field = panel.getField(name);
    Point point = field.getLocationOnScreen();
    robot.mouseMove(point.x, point.y);
    robot.mousePress(InputEvent.BUTTON1_MASK);
    robot.mouseRelease(InputEvent.BUTTON1_MASK);
}
```

在取得一个文本框对象之后, 通过调用它的 `getLocationOnScreen` 方法, 你可以获得它在屏幕上的绝对位置。该方法返回一个 `Point` 对象——由 `x` 和 `y` 偏移表示的笛卡儿空间坐标³。

你可以将这个坐标作为参数, 发送给 `Robot` 的 `mouseMove` 方法。随后, 调用 `Robot` 的 `mousePress` 和 `mouseRelease` 方法, 在这个位置产生一个虚拟的鼠标点击。

`type` 方法相当直接:

```
private void type(int key) throws Exception {
    robot.keyPress(key);
    robot.keyRelease(key);
}
```

`Sis` 中的代码为两个文本框添加了同一个侦听器。这个侦听器等待 `keyReleased` 事件。当它收到一个事件时, 侦听器调用 `setAddButtonState` 方法。`setAddButtonState` 中的代码查看两个文本框中的内容, 决定是否启用 `Add` 按钮。

```
public class Sis {
    ...
    private void initialize() {
        createCoursesPanel();
        createKeyListeners();
    }
    ...
    void createKeyListeners() {
        KeyListener listener = new KeyAdapter() {
```

³ 你的屏幕左上角, 其 `(x,y)` 坐标为 `(0,0)`。当你在屏幕上向下移动时, `y` 的值会增长。例如, 你可以用 `(2,1)` 表示距左侧两个像素并向下一个像素的位置。

```

        public void keyReleased(KeyEvent e) {
            setAddButtonState();
        }
    };
    panel.addFieldListener(CoursesPanel.DEPARTMENT_FIELD_NAME,
        listener);
    panel.addFieldListener(CoursesPanel.NUMBER_FIELD_NAME, listener);

    setAddButtonState();
}

void setAddButtonState() {
    panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME,
        !isEmpty(CoursesPanel.DEPARTMENT_FIELD_NAME) &&
        !isEmpty(CoursesPanel.NUMBER_FIELD_NAME));
}

private boolean isEmpty(String field) {
    String value = panel.getText(field);
    return value.equals("");
}
}

```

注意 createKeyListeners 的最后一行调用了 setAddButtonState，以将 Add 按钮设置为它的缺省（初始）状态。

618

testKeyListeners 中的代码并没有展示所有的可能情况。如果用户什么都不输入，而只输入空格字符该会怎样呢？如果两个文本框中，一个含有数据而另一个没有，按钮会被正确地禁用吗？

你需要加强 testKeyListeners 来应对这些情况。第二个测试使用不同的方法，直接同 setAddButtonState 打交道。这个测试覆盖了更完整的情况集合。

```

public void testSetAddButtonState() throws Exception {
    JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
    assertFalse(button.isEnabled());

    panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, "a");
    sis.setAddButtonState();
    assertFalse(button.isEnabled());

    panel.setText(CoursesPanel.NUMBER_FIELD_NAME, "1");
    sis.setAddButtonState();
    assertTrue(button.isEnabled());

    panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, " ");
    sis.setAddButtonState();
    assertFalse(button.isEnabled());

    panel.setText(CoursesPanel.DEPARTMENT_FIELD_NAME, "a");
    panel.setText(CoursesPanel.NUMBER_FIELD_NAME, " ");
    sis.setAddButtonState();
    assertFalse(button.isEnabled());
}

```

测试失败了。isEmpty 中一个小改动，可以解决这个问题。


```
private boolean isEmpty(String field) {
    String value = panel.getText(field);
    return value.trim().equals("");
}
```

文本框修订

当你提供一个有效的用户界面时，你希望让用户尽可能少的输入无效数据。你已经了解，在遇到必须填写的文本框时，不要使用弹出窗口来打断用户。类似的，你希望避免在用户输入无效数据时，使用弹出窗口来告知他们。

首选的方案，涉及了当用户在文本框中输入数据时，对其进行验证甚至修改。例如，课程系必须只能包括大写的字母。“CMSC”是一个有效的系，而“Cmsc”和“cmsc”则不是。为了让你用户的生活简单些，你可以让系文本框在用户输入时，自动将每个小写字母转换为大写字母。

619

在 Java 的演进中，已经为文本框动态编辑包括了几种尝试方案。现在，至少有半打儿的方法可以用于实现它。你将学习两种首选的技术：使用 `JFormattedTextField` 以及创建自定义的 `DocumentFilter` 类。

通过子类化 `javax.swing.text.DocumentFilter`，你可以创建一个自定义的过滤器。在这个子类中，你需要覆写 `insertString`、`remove` 和 `replace` 三个方法中的任何一个。使用这些方法来限制无效的输入，并/或将无效输入转换为有效输入。

当用户在文本框输入或粘贴时，`insertString` 方法会被间接地调用。当用户在一个文本框中，先选择一段已有的字符，然后键入或粘贴新的字符之前，`replace` 方法会被调用。当用户从文本框中删除字符时，`remove` 方法会被调用。你几乎总需要定义 `insertString` 和 `replace` 的行为，而很少需要涉及 `remove`。

一旦你定义了自定义过滤器的行为，你可以将它关联到一个文本框的文档（document）对象。document 是文本框底层的数据 model：它是 `javax.swing.text.Document` 接口的一个实现类。你可以通过调用 `JTextField` 的 `getDocument` 方法，来获得其 `Document` 对象。然后你可以使用 `setDocumentFilter` 将自定义过滤器关联到 `Document` 上。

测试过滤器

你该如何测试过滤器呢？你可以使用 `Swing Robot`（在“必须填写的文本框”一节中介绍的），在 `CoursesPanel` 中编写一个测试。但是作为单元测试的目的，`Robot` 是在你迫不得已时使用的最后一种技术。在本例中，`DocumentFilter` 的子类是一个你可以直接测试的独立的类。

在某些情况下，你会发现 `Swing` 的设计，有助于更容易的对其进行测试。对自定义过滤器来说，你必须先跑跑腿儿（legwork）解决几个障碍。

UppcaseFilterTest 直接如下所示。在稍许重构之后，独立的测试方法 testInsert，是非常直接并容易阅读的。在 testInsert 中，你直接调用 UppcaseFilter 实例的 insertString 方法。insertString 的第二个参数是开始插入的列。第三个参数是要插入的文本。（眼下，第四个参数是不相关的，而我们将稍后讨论第一个参数）。

在第 0 列插入文本“abc”应该生成文本“ABC”。在位置 1 插入“def”（也就是在第二列之前），应该生成文本“ADEFBC”。

620

```
package sis.ui;

import javax.swing.*;
import javax.swing.text.*;
import junit.framework.*;

public class UppcaseFilterTest extends TestCase {
    private DocumentFilter filter;
    protected DocumentFilter.FilterBypass bypass;
    protected AbstractDocument document;

    protected void setUp() {
        bypass = createBypass();
        document = (AbstractDocument)bypass.getDocument();
        filter = new UppcaseFilter();
    }

    public void testInsert() throws BadLocationException {
        filter.insertString(bypass, 0, "abc", null);
        assertEquals("ABC", document.getText());

        filter.insertString(bypass, 1, "def", null);
        assertEquals("ADEFBC", document.getText());
    }

    protected String documentText() throws BadLocationException {
        return document.getText(0, document.getLength());
    }

    protected DocumentFilter.FilterBypass createBypass() {
        return new DocumentFilter.FilterBypass() {
            private AbstractDocument document = new PlainDocument();
            public Document getDocument() {
                return document;
            }
            public void insertString(
                int offset, String string, AttributeSet attr) {
                try {
                    document.insertString(offset, string, attr);
                }
                catch (BadLocationException e) {}
            }
            public void remove(int offset, int length) {}
            public void replace(int offset,
                int length, String string, AttributeSet attrs) {}
        };
    }
}
```

初始化远比测试本身更为棘手。

如果你查看 `insertString` 的 javadoc, 你会看到它需要 `DocumentFilter.FilterBypass` 类型的一个引用作为第一个参数。过滤器旁路 (filter bypass), 本质上是一个指向忽略所有过滤器的文档的引用。当你在 `insertString` 转换数据之后, 必须调用过滤器旁路的 `insertString`。否则, 你将产生一个无线限循环!

测试的困难在于, Swing 没有提供直接的方法来获得一个过滤器旁路 (filter bypass) 对象。而你为了测试过滤器, 需要这个旁路。

上面提供的方案是, 提供一个 `DocumentFilter.FilterBypass` 的新实现。这个实现保存了一个叫做 `PlainDocument` 的 `AbstractDocument` (实现了 `Document` 接口) 的具体实例。为了充实旁路, 你必须提供 `insertString`、`remove` 和 `replace` 三个方法的实现。目前, 测试只需要你实现 `insertString`。

`insertString` 方法并不需要一个旁路对象作为它的第一个参数, 因为它是在过滤器本身中定义的。它的任务是直接调用文档的 `insertString` 方法 (也就是说, 不需要再回调到 `DocumentFilter`)。注意, 如果起始的位置超出范围, 这个方法可以抛出一个 `BadLocationException`。

一旦你拥有了一个 `DocumentFilter.FilterBypass` 的实例, 初始化余下的工作和测试就简单了。从旁路对象, 你可以得到并保存一个文档的引用。你可以断言这个文档的内容是否被适当更新了。

测试 (`UppcaseFilterTest`) 包括了许多代码。你大概会认为基于 `Robot` 的测试可能更容易编写。事实上, 的确如此。不过, `Robot` 也有它们自己的问题。因为它们需要操纵鼠标和键盘, 你必须小心, 在测试执行时什么都不要做。否则, 你可能会导致 `Robot` 测试失败。单单这一点, 就是不惜任何代价也要避免使用 `Robot` 的原因之一。如果你必须使用 `Robot` 测试, 寻找一种方式将它们隔离开, 或者在你单元测试套件的开始部分执行它们。

而且, 你为第二个过滤器编写的测试代码, 和相应的 `robot` 测试代码相比, 一样简单。这两个过滤器测试都需要 `documentText` 和 `createBypass` 方法, 还有 `setUp` 方法中的大部分。

编写过滤器

建立过滤器的工作你已经进行了大半了。你已经完成了最困难的部分——为其编写一个测试。编写过滤器本身则微不足道。

```
package sis.ui;

import javax.swing.text.*;

public class UppcaseFilter extends DocumentFilter {
    public void insertString(
        DocumentFilter.FilterBypass bypass,
```

```

        int offset,
        String text,
        AttributeSet attr) throws BadLocationException {
    bypass.insertString(offset, text.toUpperCase(), attr);
}
}

```

当过滤器收到 insertString 调用时,在本例中它的任务是将参数 text 转换为大写,并将转换后的数据传给旁路以终止。

在你演示了所有测试均通过之后,你可以编写 replace 方法。下面是对测试的修改:

```

...
public class UppcaseFilterTest extends TestCase {
    ...
    public void testReplace() throws BadLocationException {
        filter.insertString(bypass, 0, "XYZ", null);
        filter.replace(bypass, 1, 2, "tc", null);
        assertEquals("XTC", documentText());

        filter.replace(bypass, 0, 3, "p8A", null);
        assertEquals("P8A", documentText());
    }
    ...
    protected DocumentFilter.FilterBypass createBypass() {
        return new DocumentFilter.FilterBypass() {
            ...
            public void replace(int offset,
                               int length, String string, AttributeSet attrs) {
                try {
                    document.replace(offset, length, string, attrs);
                }
                catch (BadLocationException e) {}
            }
        };
    }
}

```

这个测试展示了 replace 方法还接收额外的参数。第三个参数表示要替换的字符个数,开始位置由第二个参数表示。下面产品系统中的代码:

```

package sis.ui;

import javax.swing.text.*;

public class UppcaseFilter extends DocumentFilter {
    ...
    public void replace(
        DocumentFilter.FilterBypass bypass,
        int offset,
        int length,
        String text,
        AttributeSet attr) throws BadLocationException {
        bypass.replace(offset, length, text.toUpperCase(), attr);
    }
}

```

UppcaseFilter 现在完成了。你不需要担心过滤器在大写输入转换时的删除操作。

关联过滤器

作为一个单独的单元,你已经检验了 `UppcaseFilter` 的功能。为了检验 `CoursesPanel` 中的系文本框已经将其输入转换为大写文本,你只需要证明已经将正确的过滤器关联到文本框上了。

`CoursesPanel` 中的代码应该将过滤器关联到它内部的文本框吗,或者 `Sis` 中的代码应该取得文本框的引用然后将它们同过滤器相关联吗?测试应该属于 `SisTest` 还是 `CoursesPanelTest`?过滤器是业务规则和 `view` 功能的组合。它强制业务约束(例如,“系的缩写是四个大写字母”)。通过让用户更容易地只输入有效信息,过滤器也增强了应用的体验(*feel*)。

记住:保持让 `view` 类尽量简单。尽可能将业务相关的逻辑放在领域(*domain*,即 `Course`)或应用(`Sis`)类中。过滤器表示的业务逻辑非常依赖于 `Swing`。过滤器本质上是 `Swing` 框架中的一个插件。你无法让领域类依赖这样的代码。因此,唯一的选择是放到应用类中。

`SisTest` 中的代码:

```
public void testCreate() {
    ...
    CoursesPanel panel =
        (CoursesPanel)Util.getComponent(frame, CoursesPanel.NAME);
    assertNotNull(panel);
    ...
    verifyFilter(panel);
}

private void verifyFilter(CoursesPanel panel) {
    DocumentFilter filter =
        getFilter(panel, CoursesPanel.DEPARTMENT_FIELD_NAME);
    assertTrue(filter.getClass() == UppcaseFilter.class);
}

private DocumentFilter getFilter( CoursesPanel panel, String fieldName) {
    JTextField field = panel.getField(fieldName);
    AbstractDocument document = (AbstractDocument)field.getDocument();
    return document.getDocumentFilter();
}
...
}
```

`Sis` 中的代码:

```
private void initialize() {
    createCoursesPanel();
    createKeyListeners();
    createInputFilters();
    ...
}
...
private void createInputFilters() {
    JTextField field =
```

```

        panel.getField(CoursesPanel.DEPARTMENT_FIELD_NAME);
        AbstractDocument document = (AbstractDocument)field.getDocument();
        document.setDocumentFilter(new UppcaseFilter());
    }

```

第二个过滤器

你还希望限制系与课程编号文本框中的字符个数。实际上，在大多数需要文本框的应用中，你会想要这种设置文本框界限的能力。你可以创建第二个自定义过滤器，LimitFilter。下面仅演示了产品类中的代码。其测试，LimitFilterTest（参见 <http://www.LangrSoft.com/agileJava/code/>）处的代码）包括了许多和 UppcaseFilterTest 共用的部分，你可以分解出来。

```

package sis.ui;

import javax.swing.text.*;

public class LimitFilter extends DocumentFilter {
    private int limit;

    public LimitFilter(int limit) {
        this.limit = limit;
    }

    public void insertString(
        DocumentFilter.FilterBypass bypass,
        int offset,
        String str,
        AttributeSet attrSet) throws BadLocationException {
        replace(bypass, offset, 0, str, attrSet);
    }

    public void replace(
        DocumentFilter.FilterBypass bypass,
        int offset,
        int length,
        String str,
        AttributeSet attrSet) throws BadLocationException {
        int newLength =
            bypass.getDocument().getLength() - length + str.length();
        if (newLength > limit)
            throw new BadLocationException(
                "New characters exceeds max size of document.", offset);
        bypass.replace(offset, length, str, attrSet);
    }
}

```

625

注意，将 insertString 委托给 replace 方法的技术。代码中另一个有意义的地方在于，当替换字符串过长时抛出 BadLocationException。

建立这样一个过滤器，并将它关联到课程编号文本框非常容易。通过传入字符的长度来构造一个 LimitFilter。例如，代码片段 new LimitFilter(3) 创建一个过滤器，阻止输入三个以上的字符。

问题是,你只能为一个文档设置一个过滤器。你有几个选择。第一个(不好的)选择是为每种组合创建一个单独的过滤器。例如,你可能有 `UppcaseLimitFilter` 和 `NumericOnlyLimitFilter` 的过滤器组合。更好的方案,引入了某些形式的抽象——`ChainableFilter`。`ChainableFilter` 是 `DocumentFilter` 的子类。它包含了一个过滤器类的序列,并管理依次对它们的调用。本课的代码可以在 <http://www.LangrSoft.com/agileJava/code/> 获得,它演示了你如何来建立这样一种结构。⁴

JFormattedTextField

另一种管理文本框修订的机制是,使用 `javax.swing.JFormattedTextField`,它是 `JTextField` 的子类。你可以将格式器(formatter)关联到文本框,以确保其内容符合你的规格。而且,你可以将文本框的内容,作为非文本对象的其他适当类型,从中取出。

你希望为课程提供一个有效的日期文本框。这个日期在系统中代表该课程第一次开课的时间。用户必须按 `mm/dd/yy` 的格式输入日期。例如,04/15/02 是一个有效的日期。

测试得到 `JFormattedTextField` 类型的文本框,然后从 `JFormattedTextField` 中得到格式器(formatter)对象。格式器是 `javax.swing.JFormattedTextField.AbstractFormatter` 的子类。在 `verifyEffectiveDate` 中,你希望格式器是 `DateFormatter`。`DateFormatter` 继而包装了 `java.text.SimpleDateFormat` 实例,并且它的格式模式(pattern)为 `MM/dd/yy`。⁵

测试的最后一部分是,确保文本框保存的是一个日期的实例。当用户点击 `Add` 按钮时, `sis.ui.Sis` 中的代码,以 `java.util.Date` 对象类型得到有效日期输入框中的内容。

```
private void verifyEffectiveDate() {
    assertLabelText(EFFECTIVE_DATE_LABEL_NAME,
        EFFECTIVE_DATE_LABEL_TEXT);

    JFormattedTextField dateField =
        (JFormattedTextField)panel.getField(EFFECTIVE_DATE_FIELD_NAME);
    DateFormatter formatter = (DateFormatter)dateField.getFormatter();
    SimpleDateFormat format = (SimpleDateFormat)formatter.getFormat();
    assertEquals("MM/dd/yy", format.toPattern());
    assertEquals(Date.class, dateField.getValue().getClass());
}
```

`CoursesPanel` 中的代码,通过向 `JFormattedTextField` 的构造函数传入一个 `SimpleDateFormat`,来构造它的一个实例。代码调用 `dateField` 的 `setValue` 方法,提供保存了修订结果的 `Date` 对象。

```
JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();

    JPanel panel = new JPanel(layout);
```

⁴ 由于篇幅的原因,代码清单没有在此列出。

⁵ 大写的字母 M 用来表示月,而小写的字母 m 用来表示分钟。

```

int columns = 20;

addField(panel, layout, 0,
    DEPARTMENT_LABEL_NAME, DEPARTMENT_LABEL_TEXT,
    createField(DEPARTMENT_FIELD_NAME, columns));

addField(panel, layout, 1,
    NUMBER_LABEL_NAME, NUMBER_LABEL_TEXT,
    createField(NUMBER_FIELD_NAME, columns));

Format format = new SimpleDateFormat("MM/dd/yy");
JFormattedTextField dateField = new JFormattedTextField(format);
dateField.setValue(new Date());
dateField.setColumns(columns);
dateField.setName(EFFECTIVE_DATE_FIELD_NAME);

addField(panel, layout, 2,
    EFFECTIVE_DATE_LABEL_NAME, EFFECTIVE_DATE_LABEL_TEXT,
    dateField);

return panel;
}

```

627

如果你执行经过这些改动的应用，你会注意到，有效日期文本框允许你输入无效的值。当你离开文本框时，它将输入的内容转换为有效的值。你可以覆盖这个缺省的行为；关于各种替代方式，参见 `JFormattedTextField` 的 API 文档。

一个设计问题现在出现了。创建格式化文本框的代码在 `CoursesPanel` 中，而相关的测试在 `CoursesPanelTest` 中。这和我之前表述的在应用层级管理修订 (edit) 的目标相违背。

你希望完全分离 view 和应用的关注。方案涉及了单一职责原则 (single responsibility principle)。它还消除了我曾在 `CoursesPanel` 和 `Sis` 中暂时容忍恶化的部分重复和代码混乱。

`Field` 对象是一个数据对象，其属性描述了创建 Swing 文本框所需的必要信息。不过，文本框是实现中立的 (implementation-neutral)，并且无需了解 Swing。`FieldCatalog` 包括了可用文本框的集合。它可以根据指定的名字返回一个 `Field` 对象。

`CoursesPanel` 类只需要保存它必须要渲染的文本框名字列表。`CoursesPanel` 代码可以迭代这个列表，请求 `FieldCatalog` 来得到对应的 `Field` 对象。然后它可以将 `Field` 对象发送给一个工厂，`TextFieldFactory`，其工作是返回一个 `JTextField`。这个工厂将从 `Field` 对象得到信息，并使用它们为 `JTextField` 添加各种约束，例如格式、过滤器和长度限制。

下面是新类中的代码。我还展示了在 `CoursesPanel` 中构造文本框的代码。

```

// FieldCatalogTest.java
package sis.ui;

import junit.framework.*;
import static sis.ui.FieldCatalog.*;

public class FieldCatalogTest extends TestCase {
    public void testAllFields() {
        FieldCatalog catalog = new FieldCatalog();
    }
}

```


628

```

        assertEquals(3, catalog.size());

        Field field = catalog.get(NUMBER_FIELD_NAME);
        assertEquals(DEFAULT_COLUMNS, field.getColumns());
        assertEquals(NUMBER_LABEL_TEXT, field.getLabel());
        assertEquals(NUMBER_FIELD_LIMIT, field.getLimit());

        field = catalog.get(DEPARTMENT_FIELD_NAME);
        assertEquals(DEFAULT_COLUMNS, field.getColumns());
        assertEquals(DEPARTMENT_LABEL_TEXT, field.getLabel());
        assertEquals(DEPARTMENT_FIELD_LIMIT, field.getLimit());
        assertTrue(field.isUppcaseOnly());

        field = catalog.get(EFFECTIVE_DATE_FIELD_NAME);
        assertEquals(DEFAULT_COLUMNS, field.getColumns());
        assertEquals(EFFECTIVE_DATE_LABEL_TEXT, field.getLabel());
        assertEquals(DEFAULT_DATE_FORMAT, field.getFormat());
    }

    // FieldCatalog.java
    package sis.ui;

    import java.util.*;
    import java.text.*;

    public class FieldCatalog {

        public static final DateFormat DEFAULT_DATE_FORMAT =
            new SimpleDateFormat("MM/dd/yy");

        static final String DEPARTMENT_FIELD_NAME = "deptField";
        static final String DEPARTMENT_LABEL_TEXT = "Department";
        static final int DEPARTMENT_FIELD_LIMIT = 4;

        static final String NUMBER_FIELD_NAME = "numberField";
        static final String NUMBER_LABEL_TEXT = "Number";
        static final int NUMBER_FIELD_LIMIT = 3;

        static final String EFFECTIVE_DATE_FIELD_NAME = "effectiveDateField";
        static final String EFFECTIVE_DATE_LABEL_TEXT = "Effective Date";

        static final int DEFAULT_COLUMNS = 20;

        private Map<String,Field> fields;

        public FieldCatalog() {
            loadFields();
        }

        public int size() {
            return fields.size();
        }

        private void loadFields() {
            fields = new HashMap<String,Field>();

            Field fieldSpec = new Field(DEPARTMENT_FIELD_NAME);
            fieldSpec.setLabel(DEPARTMENT_LABEL_TEXT);

```

629

```

        fieldSpec.setLimit(DEPARTMENT_FIELD_LIMIT);
        fieldSpec.setColumns(DEFAULT_COLUMNS);
        fieldSpec.setUpcaseOnly();

        put(fieldSpec);

        fieldSpec = new Field(NUMBER_FIELD_NAME);
        fieldSpec.setLabel(NUMBER_LABEL_TEXT);
        fieldSpec.setLimit(NUMBER_FIELD_LIMIT);
        fieldSpec.setColumns(DEFAULT_COLUMNS);

        put(fieldSpec);

        fieldSpec = new Field(EFFECTIVE_DATE_FIELD_NAME);
        fieldSpec.setLabel(EFFECTIVE_DATE_LABEL_TEXT);
        fieldSpec.setFormat(DEFAULT_DATE_FORMAT);
        fieldSpec.setInitialValue(new Date());
        fieldSpec.setColumns(DEFAULT_COLUMNS);

        put(fieldSpec);
    }

    private void put(Field fieldSpec) {
        fields.put(fieldSpec.getName(), fieldSpec);
    }

    public Field get(String fieldName) {
        return fields.get(fieldName);
    }
}

// TextFieldFactoryTest.java
package sis.ui;

import javax.swing.*;
import javax.swing.text.*;
import java.util.*;
import java.text.*;
import junit.framework.*;
import sis.util.*;

public class TextFieldFactoryTest extends TestCase {
    private Field fieldSpec;
    private static final String FIELD_NAME = "fieldName";
    private static final int COLUMNS = 1;

    protected void setUp() {
        fieldSpec = new Field(FIELD_NAME);
        fieldSpec.setColumns(COLUMNS);
    }

    public void testCreateSimpleField() {
        final String textValue = "value";
        fieldSpec.setInitialValue(textValue);

        JTextField field = TextFieldFactory.create(fieldSpec);

        assertEquals(COLUMNS, field.getColumns());
        assertEquals(FIELD_NAME, field.getName());
    }
}

```

```

        assertEquals(textValue, field.getText());
    }

    public void testLimit() {
        final int limit = 3;
        fieldSpec.setLimit(limit);

        JTextField field = TextFieldFactory.create(fieldSpec);

        AbstractDocument document = (AbstractDocument)field.getDocument();
        ChainableFilter filter =
            (ChainableFilter)document.getDocumentFilter();
        assertEquals(limit, ((LimitFilter)filter).getLimit());
    }

    public void testUppcase() {
        fieldSpec.setUppcaseOnly();

        JTextField field = TextFieldFactory.create(fieldSpec);

        AbstractDocument document = (AbstractDocument)field.getDocument();
        ChainableFilter filter =
            (ChainableFilter)document.getDocumentFilter();
        assertEquals(UppcaseFilter.class, filter.getClass());
    }

    public void testMultipleFilters() {
        fieldSpec.setLimit(3);
        fieldSpec.setUppcaseOnly();

        JTextField field = TextFieldFactory.create(fieldSpec);

        AbstractDocument document = (AbstractDocument)field.getDocument();
        ChainableFilter filter =
            (ChainableFilter)document.getDocumentFilter();

        Set<Class> filters = new HashSet<Class>();
        filters.add(filter.getClass());
        filters.add(filter.getNext().getClass());

        assertTrue(filters.contains(LimitFilter.class));
        assertTrue(filters.contains(UppcaseFilter.class));
    }

    public void testCreateFormattedField() {
        final int year = 2006;
        final int month = 3;
        final int day = 17;
        fieldSpec.setInitialValue(DateUtil.createDate(year, month, day));
        final String pattern = "MM/dd/yy";
        fieldSpec.setFormat(new SimpleDateFormat(pattern));

        JFormattedTextField field =
            (JFormattedTextField)TextFieldFactory.create(fieldSpec);

        assertEquals(1, field.getColumns());
        assertEquals(FIELD_NAME, field.getName());

        DateFormatter formatter = (DateFormatter)field.getFormatter();
    }

```

```

SimpleDateFormat format = (SimpleDateFormat)formatter.getFormat();
assertEquals(pattern, format.toPattern());
assertEquals(Date.class, field.getValue().getClass());
assertEquals("03/17/06", field.getText());

TestUtil.assertDateEquals(year, month, day,
    (Date)field.getValue()); // a new utility method
}
}

// TextFieldFactory.java
package sis.ui;

import javax.swing.*;
import javax.swing.text.*;

public class TextFieldFactory {
    public static JTextField create(Field fieldSpec) {
        JTextField field = null;

        if (fieldSpec.getFormat() != null)
            field = createFormattedTextField(fieldSpec);
        else {
            field = new JTextField();
            if (fieldSpec.getInitialValue() != null)
                field.setText(fieldSpec.getInitialValue().toString());
        }

        if (fieldSpec.getLimit() > 0)
            attachLimitFilter(field, fieldSpec.getLimit());

        if (fieldSpec.isUppcaseOnly())
            attachUppcaseFilter(field);

        field.setColumns(fieldSpec.getColumns());
        field.setName(fieldSpec.getName());
        return field;
    }

    private static void attachLimitFilter(JTextField field, int limit) {
        attachFilter(field, new LimitFilter(limit));
    }

    private static void attachUppcaseFilter(JTextField field) {
        attachFilter(field, new UppcaseFilter());
    }

    private static void attachFilter(
        JTextField field, ChainableFilter filter) {
        AbstractDocument document = (AbstractDocument)field.getDocument();
        ChainableFilter existingFilter =
            (ChainableFilter)document.getDocumentFilter();
        if (existingFilter == null)
            document.setDocumentFilter(filter);
        else
            existingFilter.setNext(filter);
    }

    private static JTextField createFormattedTextField(Field fieldSpec) {

```

```

        JFormattedTextField field =
            new JFormattedTextField(fieldSpec.getFormat());
        field.setValue(fieldSpec.getInitialValue());
        return field;
    }

    // CoursesPanelTest.java
    ...
    public void testCreate() {
        assertEmptyList(COURSES_LIST_NAME);
        assertButtonText(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);

        String[] fields =
            { FieldCatalog.DEPARTMENT_FIELD_NAME,
              FieldCatalog.NUMBER_FIELD_NAME,
              FieldCatalog.EFFECTIVE_DATE_FIELD_NAME };
        assertFields(fields);

        JButton button = panel.getButton(ADD_BUTTON_NAME);
        assertEquals(ADD_BUTTON_MNEMONIC, button.getMnemonic());
    }

    private void assertFields(String[] fieldNames) {
        FieldCatalog catalog = new FieldCatalog();
        for (String fieldName: fieldNames) {
            assertNotNull(panel.getField(fieldName));
            // can't compare two JTextField items for equality,
            // so we must go on faith here that CoursesPanel
            // creates them using TextFieldFactory
            Field fieldSpec = catalog.get(fieldName);

            assertLabelText(fieldSpec.getLabelName(), fieldSpec.getLabel());
        }
    }
    ...

    // CoursesPanel.java
    ...
    JPanel createFieldsPanel() {
        GridBagLayout layout = new GridBagLayout();

        JPanel panel = new JPanel(layout);
        int i = 0;
        FieldCatalog catalog = new FieldCatalog();

        for (String fieldName: getFieldNames()) {
            Field fieldSpec = catalog.get(fieldName);
            addField(panel, layout, i++,
                    createLabel(fieldSpec),
                    TextFieldFactory.create(fieldSpec));
        }

        return panel;
    }

    private String[] getFieldNames() {
        return new String[]
            { FieldCatalog.DEPARTMENT_FIELD_NAME,

```

633

```

        FieldCatalog.NUMBER_FIELD_NAME,
        FieldCatalog.EFFECTIVE_DATE_FIELD_NAME };
    }

    private void addField(
        JPanel panel, GridBagLayout layout, int row,
        JLabel label, JTextField field) {
        ...
        panel.add(label);
        panel.add(field);
    }
    ...

```

注意:

- `TestUtil.assertDateEquals` 是一个新的实用方法, 其实现应该是很明显的。
- 我最终将 `DateUtil` 类从 `sis.studenginfo` 包移到 `sis.util` 包中。这个改变影响了许多现有的类。
- 你必须编辑 `Sis` 和 `CoursesPanel` (及其测试) 来删除为构造过滤器/格式器编写的常量和代码。完整的代码, 参见 <http://www.LangrSoft.com/agileJava/code/>。
- `Field` 类, 没有包括在这些清单中, 它是一个简单的数据类, 实质上没有任何逻辑。
- 你还需要更新 `Course` 类以包括新的属性——有效的日期。

634

表格 (Table)

`CoursesPanel` 中的 `JList` 为每个 `Course` 对象显示一行字符串。这种表示眼下是足够的, 因为你只是显示两段数据: 系与课程编号。不过, 向这个摘要字符串中添加新的有效日期属性, 将很快让列表看起来十分混乱。事实上, 当你在列表的每行中只显示一份数据时, `JList` 控件工作得最好。

`JTable` 是一个非常有效的控件, 让你可以使用顺序的列 (column) 来表示每个对象。`JTable` 看起来有些像电子表格 (spreadsheet)。事实上, `JTable` 的代码和文档也使用和电子表格相同的术语: 行 (row), 列 (column) 和单元格 (cell)。

`JTable` 类在外观和体验 (look and feel) 方面给了你相当大的控制力。例如, 你可以决定是否允许用户编辑表格中的各个单元格, 你可以让用户重新整理列, 并控制每列的宽度。

为了这个练习, 你需要将 `JList` 替换为 `JTable`。最好的着手点是, 创建 `JTable` 底层的数据 model。正如你把 `Course` 对象插入到 `JList` 的列表 model 一样, 你将要吧 `Course` 对象放到与 `JTable` 关联的 model 中。

创建 `JTable` 的 model 稍稍棘手些。`JList` 能够通过调用它所包含的每个对象的 `toString` 方法, 来得到一个可打印的字符串表示。`JTable` 必须分别对待给定对象的每个属性。`JTable` 为它必

须显示的每个单元格, 调用 `model` 的 `getValueAt` 方法。它将当前单元格对应的行和列作为参数传入。`getValueAt` 方法必须返回一个要在该位置显示的字符串。

对 `model` 来说, 推断出你想要在给定的列上显示哪一个属性, 大概没什么容易的办法。而且, 你必须提供一个自己实现的 `model`。你必须在这个实现中提供 `getValueAt` 方法, 以及另外两个方法: `getRowCount` 和 `getColumnCount`。为了加强表格的外观和体验, 你还可以实现其他方法。`CoursesTableModelTest` 测试如下所示, 演示 `model` 实现了 `getColumnName` 方法, 为每一列返回一个文本的表头。

635

```
package sis.ui;

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

public class CoursesTableModelTest extends TestCase {
    private CoursesTableModel model;

    protected void setUp() {
        model = new CoursesTableModel();
    }

    public void testCreate() {
        assertEquals(0, model.getRowCount());
        assertEquals(3, model.getColumnCount());
        FieldCatalog catalog = new FieldCatalog();

        Field department =
            catalog.get(FieldCatalog.DEPARTMENT_FIELD_NAME);
        assertEquals(department.getShortName(), model.getColumnName(0));

        Field number = catalog.get(FieldCatalog.NUMBER_FIELD_NAME);
        assertEquals(number.getShortName(), model.getColumnName(1));

        Field effectiveDate =
            catalog.get(FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
        assertEquals(effectiveDate.getShortName(),
            model.getColumnName(2));
    }

    public void testAddRow() {
        Course course = new Course("CMSC", "110");

        course.setEffectiveDate(DateUtil.createDate(2006, 3, 17));

        model.add(course);

        assertEquals(1, model.getRowCount());
        final int row = 0;
        assertEquals("CMSC", model.getValueAt(row, 0));
        assertEquals("110", model.getValueAt(row, 1));
        assertEquals("03/17/06", model.getValueAt(row, 2));
    }
}
```

测试展示了 `FieldCatalog` 的另一种用法——为每个字段返回一个合适的列头 (column header)。它使用了一个新的 `Field` 属性——更为抽象的概念 “short name”，用于在有限空间显示的缩写名。你需要更新 `Field` 和 `FieldCatalog/FieldCatalogTest` 来提供这些新的信息。

636

建立表格 `model`，最简单的方法是子类化 `javax.swing.table.AbstractTableModel`。然后你只需要提供 `getValueAt`、`getRowCount` 和 `getColumnCount` 的定义。你还希望在 `model` 中保存一个课程的集合。你需要一个方法 (`add`)，来提供向 `model` 中添加 `Course` 对象的功能。

你还可以选择使用 `javax.swing.table.DefaultTableModel`。Sun 提供了这个具体的实现，让你的工作稍稍简单些。不过，`DefaultTableModel` 需要你首先组织你的数据 (以 `Vector` 对象或 `Object` 数组的形式)，然后将它传递给 `model`。这和创建你自己的 `AbstractTableModel` 子类，差不多一样容易，但无疑更有效。

```
package sis.ui;

import javax.swing.table.*;
import java.text.*;
import java.util.*;
import sis.studentinfo.*;

class CoursesTableModel extends AbstractTableModel {
    private List<Course> courses = new ArrayList<Course>();
    private SimpleDateFormat formatter =
        new SimpleDateFormat("MM/dd/yy");
    private FieldCatalog catalog = new FieldCatalog();
    private String[] fields = {
        FieldCatalog.DEPARTMENT_FIELD_NAME,
        FieldCatalog.NUMBER_FIELD_NAME,
        FieldCatalog.EFFECTIVE_DATE_FIELD_NAME };

    void add(Course course) {
        courses.add(course);
        fireTableRowsInserted(courses.size() - 1, courses.size());
    }

    Course get(int index) {
        return courses.get(index);
    }

    public String getColumnName(int column) {
        Field field = catalog.get(fields[column]);
        return field.getShortName();
    }

    // abstract (req'd) methods: getValueAt, getColumnCount, getRowCount
    public Object getValueAt(int row, int column) {
        Course course = courses.get(row);
        String fieldName = fields[column];
        if (fieldName.equals(FieldCatalog.DEPARTMENT_FIELD_NAME))
            return course.getDepartment();
        else if (fieldName.equals(FieldCatalog.NUMBER_FIELD_NAME))
            return course.getNumber();
        else if (fieldName.equals(FieldCatalog.EFFECTIVE_DATE_FIELD_NAME))
            return formatter.format(course.getEffectiveDate());
    }
}
```

637


```

        return "";
    }

    public int getColumnCount() {
        return fields.length;
    }

    public int getRowCount() {
        return courses.size();
    }
}

```

注意周围并不明显的冗余。表格必须包括你感兴趣显示的字段列表。在 `getValueAt` 中, 你通过提供的列索引来得到字段的名字。你在一个伪 `switch` 语句中使用这个名字, 来决定要调用 `Course` 的哪一个 `getter` 方法。下面一段代码包含了一个 `switch` 语句:

```

switch (column) {
    case 0: return course.getDepartment();
    case 1: return course.getNumber();
    case 2: return formatter.format(course.getEffectiveDate());
    default: return "";
}

```

虽然这是可接受的, 但是我不喜欢这种列与属性之间的割裂。改变列或者它们的顺序, 会很容易导致错误。(至少, 你的测试应该捕捉到这个问题。) 不过看看我所展示方案的补充讨论。

当然, 你还需要稍加改动, 以使 `JTable` 可以正确工作。在 `CoursesPanelTest` 中:

```

public void testCreate() {
    assertEmptyTable(COURSES_TABLE_NAME);
    assertButtonText(ADD_BUTTON_NAME, ADD_BUTTON_TEXT);
    ...
}

public void testAddCourse() {
    Course course = new Course("ENGL", "101");
    panel.addCourse(course);
    JTable table = panel.getTable(COURSES_TABLE_NAME);
    CoursesTableModel model = (CoursesTableModel)table.getModel();
    assertEquals(course, model.get(0));
}

private void assertEmptyTable(String name) {
    JTable table = panel.getTable(name);
    assertEquals(0, table.getModel().getRowCount());
}
}

```

638

领域映射 (Domain Map)

`getValueAt` 的实现包含了显著的冗余。针对每个可能的字段名, 你检测所选择的字段名, 以得到 `Course` 对象的相应属性。一种更激进的方案是, 以哈希表的方式实现你自己的领域对象。这个哈希表的 `key` 是属性名。你可以像这样设置 `Course` 中的值:

```
course.set(DEPARTMENT_FIELD_NAME, "CMSC");
```

而你可以像这样取回这个值:

```
String courseName = (String)course.get(DEPARTMENT_FIELD_NAME);
```

或者:

```
String courseName = course.getString(DEPARTMENT_FIELD_NAME);
```

getValueAt 的代码变得非常简明:

```
Course course = courses.get(row);
return course.get(fields[column]);
```

这个方案能够极大地减少你系统中的冗余,但是它引入了其他利害关系。首先,降低了在跟踪或调试时代码的可读性。而且,在直接调用或嵌入一组诸如 getString、getDate 或 getInt 等实用方法的地方,这个方案需要强制类型转换。

在 CoursesPanel 中:

```
static final String COURSES_TABLE_NAME = "coursesTable";
...
private void createLayout() {
    JTable coursesTable = createCoursesTable();
    JScrollPane coursesScroll = new JScrollPane(coursesTable);
    coursesScroll.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    ...
}

private JTable createCoursesTable() {
    JTable table = new JTable(coursesTableModel);
    table.setName(COURSES_TABLE_NAME);
    table.setShowGrid(false);
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    return table;
}

void addCourse(Course course) {
    coursesTableModel.add(course);
}

Course getCourse(int index) {
    return coursesTableModel.get(index);
}
```

你可以删除 CourseDisplayAdapter 和其他指向原有课程列表的引用。

确保你查看了 Java API 文档中有关 JTable 和各种表格 model 的类。JTable 类包括了相当多的定制功能。

反馈 (Feedback)

创建良好用户体验的一部分是要确保你提供了足够的反馈。Sun 已经在 Swing 中建立了许多反馈。例如, 当你在 JButton 上点击鼠标按键时, JButton 重绘以表现出它在物理上被按下的效果。这种信息可以让用户对他或她的动作打消疑虑。

Sis 应用缺少有关反馈的部分: 当用户进入一个过滤或格式化文本框时, 他或她如何知道要怎样进行输入呢? 用户最终会发现他们所受的限制。但是他或她在经历猜测、试探和错误时, 会浪费一些时间。

你可以提前向用户提供帮助性的信息。存在几种选择:

- 将有用的信息放在文本框的标签 (label) 中。不过, 通常没有足够的屏幕让你这样做。
- 提供一个单独的在线帮助窗口, 描述应用如何工作。
- 当用户将鼠标滑至文本框时, 显示一个带有相关信息的弹出矩形。这被称为悬浮帮助或者工具提示 (tool tip)。许多现代的应用, 例如 Internet Explorer, 当你将鼠标悬停在工具栏的按钮上时, 它提供对应的工具提示。
- 当用户将鼠标滑至文本框时, 在窗口底部的状态栏中显示相关的信息。
- 在这个练习中, 你要选择最后一个选项, 创建一个状态栏。

不幸的是, 对基于鼠标的测试, 通常你必须渲染 (显示) 用户界面才能对其进行测试。原因是, 组件在被渲染之前, 还没有既定的大小。你可以再次使用 Swing Robot 来帮助你编写测试。注意测试中的 setUp, 使用了 Swing 的几个实用方法, 我在稍后的清单中会显示它们。

640

```
package sis.ui;

import junit.framework.*;
import javax.swing.*;
import java.awt.*;
import sis.util.*;

public class StatusBarTest extends TestCase {
    private JTextField field1;
    private JTextField field2;
    private StatusBar statusBar;
    private JFrame frame;

    protected void setUp() {
        field1 = new JTextField(10);
        field2 = new JTextField(10);
        statusBar = new StatusBar();

        JPanel panel = SwingUtil.createPanel(field1, field2, statusBar);
        frame = SwingUtil.createFrame(panel);
    }
}
```

```

    }

    protected void tearDown() {
        frame.dispose();
    }

    public void testMouseover() throws Exception {
        final String text1 = "text1";
        final String text2 = "text2";

        statusBar.setInfo(field1, text1);
        statusBar.setInfo(field2, text2);

        Robot robot = new Robot();

        Point field1Location = field1.getLocationOnScreen();

        robot.mouseMove(field1Location.x - 1, field1Location.y - 1);
        assertEquals("", statusBar.getText().trim());

        robot.mouseMove(field1Location.x + 1, field1Location.y + 1);
        assertEquals(text1, statusBar.getText());
        Point field2Location = field2.getLocationOnScreen();

        robot.mouseMove(field2Location.x + 1, field2Location.y + 1);
        assertEquals(text2, statusBar.getText());
    }
}

```

在概念上讲，为你应用的所有窗口提供状态信息，是一个很寻常的需求。你可以将状态概念封装在一个单独的类中，避免将额外的代码混杂在每个窗口中。

641

StatusBar

StatusBar 是一个具有附加功能的 JLabel。你可以通过调用 Status 对象的 setInfo，将每个文本框的信息 String 与之相关联。

测试得到第一个文本框的位置，然后将鼠标移动到该文本框之外的空白位置。状态栏应该什么也不显示；第一个断言对此进行检验。接下来测试将鼠标移动到文本框上，然后确保状态栏中包含了预期的文本。测试最终断言，在鼠标滑至第二个文本框时，状态栏的文本变成了 TEXT2。

SwingUtil 类提取出了为测试创建简单面板和 frame 的通用代码：

```

package sis.util;

import javax.swing.*;

public class SwingUtil {
    public static JPanel createPanel(JComponent... components) {
        JPanel panel = new JPanel();
        for (JComponent component: components)
            panel.add(component);
    }
}

```

```

        return panel;
    }

    public static JFrame createFrame(JPanel panel) {
        JFrame frame = new JFrame();
        frame.getContentPane().add(panel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
        return frame;
    }
}

```

在 `StatusBar` 中, `setInfo` 的任务是为文本框添加一个鼠标侦听器。侦听器响应鼠标的进入和离开事件。当用户将鼠标移动到文本框上时, 侦听器代码显示相关联的信息。当用户将鼠标移出文本框时, 侦听器清除状态栏。

```

package sis.ui;

import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class StatusBar extends JLabel {
    private final static String EMPTY = " ";
    public StatusBar() {
        super(EMPTY);
        setBorder(BorderFactory.createLoweredBevelBorder());
    }

    public void setInfo(final JTextField textField, final String text) {
        textField.addMouseListener(
            new MouseAdapter() {
                public void mouseEntered(MouseEvent event) {
                    setText(text);
                }

                public void mouseExited(MouseEvent event) {
                    setText(EMPTY);
                }
            }
        );
    }
}

```

`StatusBar` 的测试通过了。现在你必须将状态栏附属 (attach) 到 `CoursesPanel`。你怎样测试它呢? 为 `CoursesPanel` 编写的测试, 你可以再次使用 `Robot`。但是, 确保每个文本框都正确地关联到状态栏, 还是比较容易的。

更新 `CoursesPanelTest` 中的测试。断言声明了一个新的意图: `StatusBar` 对象应该能够为指定的文本框返回对应的文本信息。它还建议了信息文本应该来源于文本框特定的对象, 可以从文本框目录 (field catalog) 中得到。

```

private void assertFields(String[] fieldNames) {
    StatusBar statusBar =
        (StatusBar) Util.getComponent(panel, StatusBar.NAME);
}

```

```

FieldCatalog catalog = new FieldCatalog();
for (String fieldName: fieldNames) {
    JTextField field = panel.getField(fieldName);
    Field fieldSpec = catalog.get(fieldName);

    assertEquals(fieldSpec.getInfo(), statusBar.getInfo(field));
    assertLabelText(fieldSpec.getLabelName(), fieldSpec.getLabel());
}
}

```

这还无法通过编译，因为你还没有实现 `getInfo`。注意你还需要将组件名关联到状态栏。
下面是 `StatusBarTest` 和 `StatusBar` 中的改动：

```

// StatusBarTest
...
public void testInfo() {
    statusBar.setInfo(field1, "a");
    assertEquals("a", statusBar.getInfo(field1));
}
...

// StatusBar
package sis.ui;

import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class StatusBar extends JLabel {
    public static final String NAME = "StatusBar";
    private final static String EMPTY = " ";
    private Map<JTextField,String> infos =
        new IdentityHashMap<JTextField,String>();

    public StatusBar() {
        super(EMPTY);
        setName(NAME);
        setBorder(BorderFactory.createLoweredBevelBorder());
    }

    public String getInfo(JTextField textField) {
        return infos.get(textField);
    }

    public void setInfo(final JTextField textField, String text) {
        infos.put(textField, text);
        textField.addMouseListener(
            new MouseAdapter() {
                public void mouseEntered(MouseEvent event) {
                    setText(getInfo(textField));
                }

                public void mouseExited(MouseEvent event) {
                    setText(EMPTY);
                }
            }
        );
    }
}

```

你还必须为 Field 添加一个字段 (field)、及其 getter 和 setter。你还需要修改 FieldCatalog, 使用有关的信息字符串填充每个字段:

```
...
static final String DEPARTMENT_FIELD_INFO =
    "Enter a 4-character department designation.";
static final String NUMBER_FIELD_INFO =
    "The department number should be 3 digits.";
static final String EFFECTIVE_DATE_FIELD_INFO =
    "Effective date should be in mm/dd/yy format.";

private void loadFields() {
    fields = new HashMap<String,Field>();

    Field fieldSpec = new Field(DEPARTMENT_FIELD_NAME);
    ...
    fieldSpec.setInfo(DEPARTMENT_FIELD_INFO);

    put(fieldSpec);

    fieldSpec = new Field(NUMBER_FIELD_NAME);
    ...
    fieldSpec.setInfo(NUMBER_FIELD_INFO);

    put(fieldSpec);

    fieldSpec = new Field(EFFECTIVE_DATE_FIELD_NAME);
    ...
    fieldSpec.setInfo(EFFECTIVE_DATE_FIELD_INFO);

    put(fieldSpec);
}
```

644

最后, CoursesPanel 中的代码, 使学生信息系统能够完整地工作:

```
private Status status;

private void createLayout() {
    ...
    add(coursesScroll, BorderLayout.CENTER);
    add(createBottomPanel(), BorderLayout.SOUTH);
}

JPanel createBottomPanel() {
    JLabel statusBar = new JLabel("");
    statusBar.setBorder(BorderFactory.createLoweredBevelBorder());
    status = new Status(statusBar);

    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    panel.add(statusBar, BorderLayout.SOUTH);
    panel.add(createInputPanel(), BorderLayout.CENTER);
    return panel;
}

JPanel createFieldsPanel() {
    GridBagLayout layout = new GridBagLayout();
```

```

JPanel panel = new JPanel(layout);
int i = 0;
FieldCatalog catalog = new FieldCatalog();
for (String fieldName: getFieldNames()) {
    Field fieldSpec = catalog.get(fieldName);
    JTextField textField = TextFieldFactory.create(fieldSpec);
    status.addText(textField, fieldSpec.getLabel());
    addField(panel, layout, i++,
             createLabel(fieldSpec),
             textField);
}

return panel;
}

```

645

如果你将 `sis.ui.Sis` 作为一个独立的应用来执行，你应该看到像图 2 一样的界面。当我截取这个屏幕图像时，我将鼠标的指针停靠在系 (department) 文本框的上面。

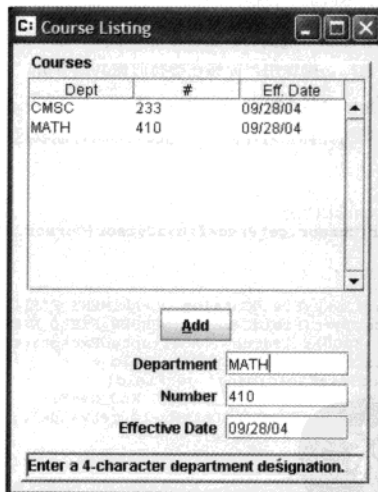


图 2 完成的外观

646

响应性 (Responsiveness)

在 `Sis` 的 `addCourse` 方法中，故意插入一个三秒钟的等待。这个等待模拟了验证和将 `Course` 对象插入到数据库所需的时间。

```
private void addCourse() {
```



```

Course course =
    new Course(
        panel.getText(FieldCatalog.DEPARTMENT_FIELD_NAME),
        panel.getText(FieldCatalog.NUMBER_FIELD_NAME));
try {Thread.sleep(3000);}catch (InterruptedException e) {}
JFormattedTextField effectiveDateField =
    (JFormattedTextField)panel.getField(
        FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
Date date = (Date)effectiveDateField.getValue();
course.setEffectiveDate(date);

panel.addCourse(course);
}

```

执行这个应用。输入课程的系和编号并按下 Add 按钮。你应该体验到三秒钟的延迟。在这段时间中, 你不能对用户界面做任何操作! 对最终用户来说, 这是一个令人沮丧的体验, 因为没有任何关于应用为何不响应的反馈。

作为应用的开发者来说, 对于响应性, 你可以做两件事: 其一, 向用户提供他们需要等待一小段时间的反馈。其二, 确保用户能够在等待时做其他事情。

反馈以“等待”光标的形式提供。Windows 使用一个沙漏来表示等待光标。某些 Unix 桌面使用一个钟表来表示等待光标。对任何一个无法立即将控制权返回给用户的操作, 你应该提供一个沙漏。

```

private void addCourse() {
    frame.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    try {
        Course course =
            new Course(
                panel.getText(FieldCatalog.DEPARTMENT_FIELD_NAME),
                panel.getText(FieldCatalog.NUMBER_FIELD_NAME));
        try {Thread.sleep(3000);}catch (InterruptedException e) {}
        JFormattedTextField effectiveDateField =
            (JFormattedTextField)panel.getField(
                FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
        Date date = (Date)effectiveDateField.getValue();
        course.setEffectiveDate(date);

        panel.addCourse(course);
    }
    finally {
        frame.setCursor(Cursor.getDefaultCursor());
    }
}

```

finally 代码块是必要的。否则, addCourse 任何未被关注或不正常的返回, 都会让用户一直以沙漏作为光标。

提供等待光标, 对任何长时间的等待来说, 都是适当并且必须的响应。但是真正的方案是确保用户不必等待。阈值是二分之一秒: 如果它要花费比较长的时间, 你应该把较慢的操作放到另一个线程中。下面演示的示例中, 我已经把添加课程的幕后操作, 同更新用户界面的代码分隔开来。我还在这个线程结束之前, 禁用了 Add 按钮。

```
// THIS IS AN INADEQUATE SOLUTION!
private void addCourse() {
    Thread thread = new Thread() {
        public void run() {
            panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, false);
            Course course = basicAddCourse();
            panel.addCourse(course);
            panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, true);
        }
    };
    thread.start();
}

private Course basicAddCourse() {
    try { Thread.sleep(3000); } catch (InterruptedException e) {}
    Course course =
        new Course(
            panel.getText(FieldCatalog.DEPARTMENT_FIELD_NAME),
            panel.getText(FieldCatalog.NUMBER_FIELD_NAME));
    JFormattedTextField effectiveDateField =
        (JFormattedTextField)panel.getField(
            FieldCatalog.EFFECTIVE_DATE_FIELD_NAME);
    Date date = (Date)effectiveDateField.getValue();
    course.setEffectiveDate(date);
    return course;
}
```

这段代码中存在一个微妙但现实的问题。它不是线程安全的！因为 Swing 使用了叫做事件分发线程 (event dispatch thread) 的一个单独线程，用户有可能在面板完全更新之前，再点击 Add 按钮。用户可能看到无法预料的结果。

648

通过在事件分发线程中执行更新用户界面的语句，你可以解决这一问题。Javax.swing.SwingUtilities 类包括的两个方法 `invokeLater` 和 `invokeAndWait`，可以让你这样做。每个方法都以一个 `Runnable` 对象作为参数，其中定义了要在事件派发线程中执行的代码。当你允许 `run` 方法异步执行时（你不需要“阻拦”用户界面的动作），你可以使用 `invokeLater`。在我们的示例中，你会希望使用 `invokeAndWait`，使 `run` 方法可以同步执行（译注：`invokeAndWait` 会等事件派发线程执行了指定代码后才返回，而 `invokeLater` 则会立即返回）。

这里演示了 `addCourse` 使用 `invokeAndWait` 的一种可能方式：

```
private void addCourse() {
    Thread thread = new Thread() {
        public void run() {
            panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, false);
            try {
                final Course course = basicAddCourse();
                SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        panel.addCourse(course);
                        panel.setEnabled(CoursesPanel.ADD_BUTTON_NAME, true);
                    }
                });
            }
        }
    };
}
```

```

        catch (Exception e) {}
    }
};
thread.start();
}

```

这个改动的最大劣势是, 将会破坏 `SisTest` 的 `testAddCourse` 方法。测试假定了点击 `Add` 按钮将会阻塞, 直至课程被加入到面板中。作为一个快速的解决方法, 你可以让测试等待直至元素出现在面板的表格中。

```

public void testAddCourse() {
    ...
    JButton button = panel.getButton(CoursesPanel.ADD_BUTTON_NAME);
    button.doClick();

    while (panel.getCourseCount() == 0)
        ;
    Course course = panel.getCourse(0);
    assertEquals("MATH", course.getDepartment());
    assertEquals("300", course.getNumber());
    TestUtil.assertDateEquals(2006, 3, 17, course.getEffectiveDate());
}

```

这个改变需要在 `CoursesPanel` 中添加一点代码:

```

intl getCourseCount() {
    return coursesTableModel.getRowCount();
}

```

余下的任务

在这样一个简单的界面中, 你已经投入了相当可观的代码量。而且它还远未完成。下面罗列了你在完成界面编码时需要考虑的事项:

- 当用户点击 `Add` 按钮, 清空文本框的内容。
- 添加一个限制, 阻止用户输入重复的课程。你可以在 `CourseCatalog` 中编写检查重复的逻辑。
- 添加删除按钮以删除课程。你还可以允许在删除时选择多个行。
- 添加对每列中的数据排序的功能。
- 添加一个数字过滤器来限制用户只能在课程编号中输入数字。
- 按照每列内容的平均或最大宽度, 来设其宽度。
- 添加一个钩子 (hook), 当用户用 `tab` 键定位或用鼠标点击一个文本框时, 选取它的内容。这允许用户通过输入来简单地替换文本框原有的内容。

- 添加鼠标滑过的帮助。当用户将鼠标滑至每个文本框上时，使用状态栏或者弹出式“工具提示”，来显示简要的信息。
- 添加键盘帮助。响应 F1 键以弹出帮助。当然这涉及了（并需要理解）如何建立一个帮助子系统。
- 使用下拉列表（drop-down list，即 JComboBox）来替换系和/或课程编号文本框。
- 添加同首选项子系统的交互（参见附加课程三），允许应用“记住”每个窗口最近显示的位置。

648

毫无疑问，在这个列表中，我还遗漏了许多特性。建立一个鲁棒（robust）、完备的用户界面，是一项很繁重的工作。任何这些功能的缺失，都会严重削弱应用的有效性（effectiveness）。不过，你应该将这些功能都看作是用户的需求，而不是以开发者的角度考虑自身的需要。你的客户小组需要一个合格的专家，来设计并细化用户界面的规格。⁶

最后的注意事项

市面上已经有大量关于 Swing 的书籍。其中许多都很厚重，意味着学习 Swing 有相当多的内容。事实上，也的确如此。在这样两个简短的章节中，你只是接触到了 Swing 的表面。

不过，你已经看到了如何构建 Swing 应用的基本要素。使用这一点点信息，你应该能够建立大方得体的界面。无疑你会想要学习 Swing 中更多更复杂的课题，例如树控件与拖放（drag & drop）。在 Internet 上稍做搜索，便应该可以告诉你需要了解哪些内容。一如既往，Java API 文档是最好的起点。通常，API 文档会指引你有关该课题的 Sun 教程。

我希望你已经在介绍 Swing 的这两章中，学习到更重要的知识：怎样使用 TDD 方法来建立 Swing 应用。Swing 的单元测试常被看作是非常困难的，并且许多小作坊选择完全忘记它。不要对懒惰屈服：不对 Swing 代码进行测试，常常产生这样一种倾向——Swing 代码中充斥了大量原本容易测试的应用或 model 代码。

看看结果产生的 view 类，CoursesPanel，你应该已经注意到它非常的小巧且简单。除了布局之外，几乎没什么复杂性可言；它几乎什么也不做。无论是否采用 TDD，这都是用户界面应用的设计目标：将应用和/或业务逻辑，隔离在 view 之外。

651

使用 TDD 会把你推向这个目标。TDD 的基本原则是，测试所有可能中断（break）的部分。对这条规则的一种解释是，“测试所有你可以测试的内容，并重新设计剩下的部分，使之不可能中断。”TDD 引导你使用短小、具体、容易测试的方法，创建了一个简短的 view 类。你还已经

⁶ 这个人可以是为客户小组担任 UI 专家角色的开发者。但是，不要轻视了这个角色的重要性。大部分的开发者，即使看过一两本关于这个主题的书籍，对如何创建一个有效的用户体验，毫无头绪。

创建了安静的“响应”方法，简单地将它们的工作委托给另一个可信任的类。这些方法是不会中断的。

选择不测试用户界面类的软件公司，常常会为此感到悔恨。业务逻辑蔓延到应用：应用和业务逻辑蔓延到 view。View 中的代码，变成了各种职责的一团混沌。因为测试不是为 view 编写的，相当多的代码没有经过测试。导致缺陷率上升。

更糟糕的是，人总是有懒惰的倾向。因为没有对 view 的测试，开发者常常认为逃避测试最简单的办法就是，将代码塞到 view 中。“是啊，创建一个新类是很痛苦的，而且这样做还要创建一个新的测试类，所以我就把所有代码堆放到 Swing 代码中。”这样做是一种危险的处境，将很快地降低你应用的品质。

在前面的两章中，你已经学习了关于使用 TDD 建立 Swing 应用的一些附加指南和技术：

- 确保你的设计分隔了应用、view 和领域逻辑（domain logic）。
- 考虑单一职责原则（Single-Responsibility Principle），进一步分解设计。
- 消除 Swing 中过度的冗余（例如，使用通用方法来创建和提取文本框。）
- 使用侦听器来测试 view 的抽象响应（例如，确保点击按钮会导致某种动作被触发）。
- 使用 mock 类来避免 Swing 的渲染。
- 不要测试布局。
- 使用 Swing Robot，但只作为最后一种手段。

Swing 在设计时已经考虑了单元级的测试。找出如何测试 Swing，是一个问题解决（problem-solving）的练习。挖掘各种 Swing 类来找到你所需的设施。有时，它们提供了钩子（hook）来帮助你测试；有时则不会。你可能需要在思考时跳出条条框框，以找出如何实现测试。



Java 的杂项

本章提供了 Java 中少量的零碎主题。本章的主要目的有两个：

- 它将为你展示一些牵扯甚众的、需要整本书才能充分涵盖的 API。
- 它还会讨论一些不适合放在本书其他地方的、核心（core）Java 的一些边边角角。

在本课中，你将学习以下内容：

- JAR
- finalize 方法
- 正则表达式（regular expression）
- JDBC
- 国际化
- 按引用调用和按值调用
- Java 的外围：其他各种 Java API 的简略综述，包括指向更多信息的链接

JAR

当你在第 1 课中学习 Java 的 classpath 时，你学习到如何向你的 classpath 添加类文件夹（或目录）。类文件夹（class folder）中包括了 Java 在需要加载时查找的具体类文件。

除了类文件夹，你的 classpath 还可以包括 JAR 文件。JAR（Java ARchive）是一个文件，其中收集了 Java 的类文件和其他在运行时所需的资源。你可以使用在 Java bin 目录中提供的命令行实用工具 jar，来创建 JAR。你还可以使用 Ant 来创建 JAR。

理解 JAR，对能够工作在任何产品化的 Java 环境中，绝对是必要的。JAR 的使用归入到部署分类中，因此本书早先没有合适的地方对 JAR 进行讨论。

实用工具 `jar` 将类文件和其他资源（例如图像或 `property` 文件），合并到一个扩展名为 `.jar` 的文件中。这个实用工具以 `ZIP` 文件的形式创建 `JAR`，你应该很熟悉这种文件格式。一个 `ZIP` 文件使用一种标准的压缩算法来减少总体的空间需求。

使用 `JAR` 有许多原因：

- 部署的速度。只传送或安装几个文件，比传送或安装上千个文件，要显著快得多。
- 压缩。更小的磁盘需求；小的文件能够更快地下载。
- 安全。当工作在安全环境中时，你可以为 `JAR` 附上一个数字签名。
- 可插入性（`Pluggability`）。你可以按照重用的目的来组织 `JAR` 文件。`JAR` 可以是组件、或者完整的 `API` 集合。
- 版本控制（`Versioning`）。你可以为 `JAR` 附上供应商和版本信息。
- 简化执行。你可以将一个 `JAR` 指示为“Java 的可执行程序”。`JAR` 的用户不需要了解入口类的名字。

`JAR` 是向 `Web` 服务器或企业级 `Java` 安装部署类的基础。

要为 `SIS` 应用创建 `JAR`，首先进入类所在的目录。然后执行：

```
jar cvf sis.jar *
```

在上面的 `jar` 命令中，选项 `cvf`（`c`、`v` 和 `f`）告诉 `JAR` 命令 `create`（创建）一个新的 `JAR`，提供 `verbose`（详细的）信息，并且使用 `sis.jar` 作为 `filename`（文件名）。命令结尾处的 `*` 告诉 `jar` 程序归档当前目录和子目录中的所有文件。你可以不带参数执行 `jar` 命令，它会显示 `jar` 命令的一个简要概述、以及它的正确用法。

在执行 `jar cvf` 命令之后，你应该得到一个名为 `sis.jar` 的文件。在 `Windows` 中，你可以使用 `WinZip` 打开这个 `JAR` 文件。在任何平台，你都可以使用 `jar` 命令来列出 `JAR` 的内容。

```
jar tvf sis.jar
```

这个命令中唯一不同的地方是选项 `t`（`table`，列出内容表）而不是 `c`。你应该看到类似下面的输出：

```
0 Mon Aug 02 23:25:36 MDT 2004 META-INF/
74 Mon Aug 02 23:25:36 MDT 2004 META-INF/MANIFEST.MF
553 Sat Jul 24 10:41:28 MDT 2004 A$B.class
541 Sat Jul 24 10:41:28 MDT 2004 A.class
1377 Sat Jul 24 10:41:28 MDT 2004 AtomicLong.class
0 Sat Jul 24 10:41:28 MDT 2004 com/
0 Sat Jul 24 10:41:28 MDT 2004 com/jimbo/
0 Sat Jul 24 10:41:28 MDT 2004 com/jimbo/ach/
486 Sat Jul 24 10:41:28 MDT 2004 com/jimbo/ach/Ach.class
377 Sat Jul 24 10:41:28 MDT 2004 com/jimbo/ach/AchCredentials.class
516 Sat Jul 24 10:41:28 MDT 2004 com/jimbo/ach/AchResponse.class
1164 Sat Jul 24 10:41:28 MDT 2004 com/jimbo/ach/AchStatus.class
...
```

你可以使用命令选项 `x`（也就是 `jar xvf`）从 JAR 文件中解出文件。

在你将类加入到 JAR 之后，你可以将这个 JAR 加入到你的 classpath 中。Java 将深入到这个 JAR 文件内，来查找它需要加载的 class 文件。例如，假定你将 `sis.jar` 部署到 `/usr/sis` 目录中。你可以使用下面的命令执行 SIS 应用：

```
java -cp /usr/sis/sis.jar sis.ui.Sis
```

为了让 Java 从一个 JAR 中查找类，它的路径信息必须匹配包（package）的层次。比如要定位 `com.jimbo.ach.Ach` 这个类，Java 必须能够从 JAR 中找到 `com/jimbo/ach/Ach.class` 的完整路径名。在 `jar tvf` 的列表中，我已经将这一项用粗体显示了。

如果你在上一级目录中创建这个 JAR，那在 JAR 文件中的路径信息会变为：

```
486 Sat Jul 24 10:41:28 MDT 2004 sis/com/jimbo/ach/Ach.class
```

Java 将无法匹配到 `com.jimbo.ach.Ach` 这个类。

`jar` 实用工具在 ZIP 文件中添加了一个清单（manifest）文件，`META-INF/MANIFEST.MF`。这个清单文件包含和 JAR 内容有关的信息（也就是元信息，meta-information）。

清单文件的一个用处是指定一个“main”类。如果你指定了一个 main 类，则只需提供一个 JAR 文件名，就可以启动一个应用。你可以通过首先创建一个单独的清单文件来达成这一点，内容如下：

```
Main-Class: sis.ui.Sis
```

确保在这个文件最后包括一个空行。否则，Java 将不会认可这个清单项。

当你创建 JAR 时，可以使用下面的命令来指定清单文件：

```
jar cvmf main.mf sis.jar *
```

当然，选项 `m` 代表 manifest（清单）。如果你查看一下 `sis.jar` 文件中的 `MANIFEST.MF` 文件，它应该看起来如下所示：

```
Manifest-Version: 1.0
Created-By: 1.5.0 (Sun Microsystems Inc.)
Main-Class: sis.ui.Sis
```

你可以使用简化的命令来启动 SIS 应用：

```
java -jar sis.jar
```

你可以通过编程的方式来操作 JAR（和 ZIP 文件）。Java 在 `java.util.zip` 包中提供了完整的 API，来帮助你完成这一点。`java.util.zip` 包中含有读写 JAR 的工具。它可以让你针对 JAR 的操作编写单元测试，和你编写针对文本文件的测试一样简单。

正则表达式

正则表达式，也被称为 `regex` 或 `regexp`，是一种包含搜索模式（`search pattern`）的字符串。正则表达式语言，对模式匹配规范来说是一种相当标准化的语言。其他例如 `Perl` 和 `Ruby` 的语言，对正则表达式提供了直接的支持。例如 `TextPad` 和 `UltraEdit` 的程序员编辑器，允许你使用正则表达式来搜索文件中的文本。`Java` 提供了一组类库，让你可以利用正则表达式的好处。

你可以使用一个通配符（`''`）来列出目录中的文件。使用该通配符，会告诉 `ls` 或 `dir` 命令来找到所有匹配的文件，假定 `*` 可以扩展为任意字符或字符的序列。例如，`DOS` 命令 `dir *.java` 会列出每个以 `.java` 为扩展名的文件。

656

正则表达式语言在概念上与此类似，但是要强大的多。在本节中，我将使用几个示例来向你介绍正则表达式。我将演示如何在您的 `Java` 代码中利用正则表达式的好处。我还给您建议如何测试使用了正则表达式的 `Java` 代码。但是关于正则表达式的完整讨论，超出了本书的范围。参考本节末尾列出的几个站点，它们提供了正则表达式的教程和信息。

分隔字符串

回到第 7 课，我曾经使用 `String` 的 `split` 方法。将全名（`full name`）分解到具体的姓名部分。你将含有一个空格字符的 `String` 传递给 `split`：

```
for (String name: fullName.split(" "))
```

`split` 方法接收一个正则表达式作为它的唯一参数¹。当 `split` 方法在调用的字符串中、遇到该正则表达式的一个匹配时，将字符串断开。

假如你想要将有三个空格的“`Jeffrey Hyman`”² 字符串，分隔为名（`first name`）和姓（`last name`）。你希望结果是“`Jeffrey`”和“`Hyman`”字符串，但是一个通过的语言测试演示了实际的结果：

```
public void testSplit() {
    String source = "Jeffrey Hyman";
    String expectedSplit[] = {"Jeffrey", "", "", "Hyman"};
    assertTrue(
        java.util.Arrays.equals(expectedSplit, source.split(" ")));
}
```

换言之，按空格隔开了四个子串。其中的两个子串是空字符串（`""`）：第一个空格将“`Jeffrey`”和一个空字符串隔开，第二个空格将第一个空字符串和第二个空字符串隔开，第

¹ `split` 的一个重载版本，可以执行有限次的模式搜索。

² 也就是 `Joey Ramone`，音乐剧 `Gabba Gabba Hey`。

三个空格将第二个空字符串和“Hyman”隔开。

然而你所希望的是，将一组空白字符两侧的姓名分隔开。使用关于 `java.util.regex.Pattern` 的 Java API 文档作为指导，你可以找到结构（construct）`\s` 匹配一个空白字符（包括制表符 `tab`、换行符、换页符、空格、或回车符）。在 API 文档中的“贪心量词（Greedy Quantifiers）”一节中，你会注意到，`x+` 表示在找到匹配时，结构 `x` 至少要出现一次。

◀ 657

组合这两种思路，正则表达式 `\s+` 匹配含有一个或多个空白字符的序列。

我已经将名字分隔的代码从 `Student` 类移到了 `sis.studentinfo.Name` 中。你可以在 <http://www.LangrSoft.com/agileJava/code> 找到 `NameTest` 和 `Name` 的完整代码。下面的列表展示了只与新内容有关的部分。

```
// NameTest.java
public void testExtraneousSpaces() {
    final String fullName = "Jeffrey Hyman";
    Name name = createName(fullName);
    assertEquals("Jeffrey", name.getFirstName());
    assertEquals("Hyman", name.getLastName());
}

private Name createName(String fullName) {
    Name name = new Name(fullName);
    assertEquals(fullName, name.getFullName());
    return name;
}

// Name.java
private List<String> split(String fullName) {
    List<String> results = new ArrayList<String>();
    for (String name: fullName.split(" "))
        results.add(name);
    return results;
}
```

当前的 `Name.split` 实现，无法通过测试方法 `testExtraneousSpaces`。使用你学习到的新模式，你可以更新 `split` 方法来使测试通过：

```
private List<String> split(String fullName) {
    List<String> results = new ArrayList<String>();
    for (String name: fullName.split("\\s+"))
        results.add(name);
    return results;
}
```

反斜线字符（`\`）在 Java 字符串中表示一个转移序列的开始。因此，正则表达式 `\s+` 在 `String` 中以 `\\s+` 的形式出现。

因为 `\s` 匹配任意的空白字符，修改测试来演示额外的空白字符被略掉了。

```
public void testExtraneousWhitespace() {
    final String fullName = "Jeffrey \t\t\n \r\fHyman";
    Name name = createName(fullName);
    assertEquals("Jeffrey", name.getFirstName());
}
```

```

        assertEquals("Hyman", name.getLastName());
    }

```

658

String 中的替换表达式

许多应用都需要得到用户的电话号码。人们输入电话号码时有无数的方式。分隔电话号码中的非数字字符（圆括号、连字符、字母、空白符等等）是最好的起点。使用正则表达式来完成非常简单。以美国的 10 位电话号码为例：

```

public void testStripPhoneNumber() {
    String input = "(719) 555-9353 (home)";
    assertEquals("7195559353", StringUtil.stripToDigits(input));
}

```

产品中相应的代码：

```

public static String stripToDigits(String input) {
    return input.replaceAll("\\D+", "");
}

```

`replaceAll` 方法接收两个参数：要匹配的正则表达式，和替换的字符串。表达式 `\D` 匹配任何非数字的字符。正则表达式语言中的习惯是，小写的字符用于正面（positive）匹配，而大写的字符用于负面（negative）匹配。以另一例子来说，`\w` 匹配一个“单词（word）字符”：任何字母、数字或下划线（‘_’）字符。大写的 `\W` 匹配任何非单词的字符。

Pattern 和 Matcher 类

你可以使用 `JTextPane` 作为基于 `Swing` 的文本编辑器应用的基础。`JTextPane` 可以让你为文本中的一段应用风格（style）。许多文本编辑器提供了搜索和加亮显示匹配模式字符串的功能。你会希望建立一个类，来管理对底层文本的搜索。

下面的测试展示了稍多些的正则表达式语言。

```

package sis.util;

import junit.framework.TestCase;

import java.util.regex.*;

public class RegexTest extends TestCase {
    public void testSearch() {
        String[] textLines = {
            "public class Test {",
            "    public void testMethod() {}",
            "    public void testNotReally(int x) {}",
            "    public void test() {}",
            "    public String testNotReally() {}",
            "}"
        };
        String text = join(textLines);
    }
}

```

659

匹配 0 或多个空白符 `\\s*`

匹配一个左大括号 `\\{`

正则表达式字符串是一个格式自由、未经验证的文本。你必须使用 `Pattern` 类的 `compile` 方法来编译它。一个成功编译的正则表达式字符串返回一个 `Pattern` 对象。从 `Pattern` 对象中，你可以为指定的输入 `String` 得到 `Matcher` 对象。在你得到 `Matcher` 对象之后，你可以调用它的 `find` 方法来得到下一个子串（输入 `String` 中匹配正则表达式的子字符串）。如果发现匹配，`find` 方法返回 `true`，否则返回 `false`。

`Matcher` 实例保存了有关上一个被发现的子串的信息。你可以调用 `Matcher` 对象的 `start` 和 `end` 方法来得到描绘匹配子串的索引。调用 `Matcher` 的 `group` 方法返回匹配的子串文本。

除了调用 `find` 方法，你可以调用 `Matcher` 的 `matches` 方法。如果整个输入字符串匹配了某个正则表达式，它会返回 `true`。你还可以调用 `lookingAt`，当输入字符串的开始处（或者整个字符串）和正则表达式匹配时，它会返回 `true`。

更多的信息

661

如你所见的，测试正则表达式非常简单——和 `String` 对象的断言相比。这一节为你提供的正则表达式的简略信息，足够你进一步发掘 Java 对正则表达式的支持。`java.util.regex.Pattern` 的 API 文档是最好的起点。

正则表达式语言有些棘手。了解如何对文本进行模式匹配，对理解如何编写适当的正则表达式，非常必要。

关于正则表达式的 Sun 教程，可以在 <http://java.sun.com/docs/books/tutorial/extra/regex/> Web 页面找到，它提供了关于正则表达式的一些额外主题。你还可以访问：

<http://www.regular-expressions.info/>

<http://www.javaregex.com/>

克隆（Cloning）与协变（Covariance）

Java 允许你克隆（clone）或拷贝一个对象。你可能会惊讶地发现，克隆通常没什么用处。在我多年专业的 Java 开发中，我只见过少数几个对克隆的需求。因此，我只是在最后一章中简要地介绍它。

不幸的是，克隆常常被误解并拙劣地实现。如果你有较复杂的克隆需求，而不是为了简单

的示例演示, 我建议你参考《Effective Java》⁴, 这本书对与克隆相关的许多问题, 展开了极为出色的探讨。

作为示例, 让我为简单的 `Course` 类提供克隆的能力。测试非常直接:

```
public void testClone() {
    final String department = "CHEM";
    final String number = "400";
    final Date now = new Date();
    Course course = new Course(department, number);
    course.setEffectiveDate(now);
    Course copy = course.clone();
    assertFalse(copy == course);
    assertEquals(department, copy.getDepartment());
    assertEquals(number, copy.getNumber());
    assertEquals(now, copy.getEffectiveDate());
}
```

662

即使 `clone` 已经在 `Object` 类中定义了, 测试还是无法编译通过。`Object` 将 `clone` 方法定义为 `protected`。你必须在子类 `Courses` 中覆写 `clone` 方法, 并使之成为 `public`。`clone` 方法在 `Object` 的定义为:

```
protected native Object clone() throws CloneNotSupportedException;
```

特征原型 (signature) 中的 `native` 关键字表示这个方法是在 JVM 中实现的, 而不是在 Java 中实现的。

为了能够克隆一个对象, 它的类必须实现 `Cloneable` 接口。`Cloneable` 是一个标记 (marker) 接口, 它并不像你预想的那样定义了 `clone`。相反, 它的目的是, 阻止客户对那些没有明确声明为可克隆的类, 进行克隆。

`Course` 中的实现:

```
package sis.studentinfo;
...
public class Course implements java.io.Serializable, Cloneable {
    private String department;
    private String number;
    private Date effectiveDate;
    ...
    @Override
    public Course clone() {
        Course copy = null;
        try {
            copy = (Course)super.clone();
        }
        catch (CloneNotSupportedException impossible) {
            throw new RuntimeException("unable to clone");
        }
        return copy;
    }
}
```

⁴ [Bloch2001].

首先你会注意到的是，`clone` 方法返回一个 `Course` 类型的对象——而不是父类定义的 `Object` 类型。Java 的这种能力被称为协变（*covariance*）。协变指的是这样一种能力：子类方法返回一个对象，其类型为父类定义的返回类型的子类。克隆是 Java 协变的一种典型示例。

克隆方法的核心是调用父类的 `clone` 方法。你总要调用父类的 `clone` 方法。`Object` 中（实际在 VM 中）实现的 `clone` 方法，为对象的内容执行一个浅拷贝（浅拷贝）。这意味着它拷贝了在这个类中定义的每个字段（`field`）。引用同样也被拷贝了，但是它们所指向的对象并不会被拷贝。例如，`Course` 中的 `effectiveDate`，原来的引用和克隆后的引用都指向同一个 `Date` 对象。

在缺省的克隆行为中，拷贝引用但不拷贝它们所指向的对象，被称为浅拷贝（*shallow clone*）。如果你希望进行深拷贝（*deep clone*）——包含的对象也被拷贝——你必须在 `clone` 方法中自己实现这些细节。

调用 `super.clone` 可能抛出一个 `CloneNotSupportedException`。这个异常要求你的类实现 `Cloneable` 标记接口。

JDBC

绝大部分面向业务的应用，需要访问关系数据库，例如 Oracle、MySQL、Sybase、SQL Server 和 DB2。应用要求信息在每次执行时都是可用和可信赖的——要进行持久化。关系数据库⁵提供了与应用无关的信息访问和排序方法。

关系数据库是由表（*table*）组成的。表是一个行（*row*）和列（*column*）的矩阵（*matrix*）。列是一个属性（*attribute*）。行是列数值的一个集合。例如，一个学生表可能包括 ID、name（名字）和 city（城市）列；有两行也就是有两个学生：生活在 Pueblo 市的 Schmoo，和生活在 Laurel 市的 Caboose。

SQL（*Structured Query Language*）语言，可以让你从任何关系数据库中存取信息。SQL，其绝大部分，是跨越各个数据库实现的标准。一条从 Oracle 数据库访问学生记录的 SQL 语句，同样也可以从 Sybase 中访问学生记录（可能需要稍微修改）。在 Java 中，你使用一种叫做 JDBC（*Java Database Connectivity*）的 API 来和数据库打交道。JDBC 可以让你同数据库建立连接，并依靠它来执行 SQL 语句。

表 1 一张有两行数据的学生表

ID	Name	City
221-44-4400	Schmoo	Pueblo
234-41-0001	Caboose	Laurel

⁵ 你可能听到过 DBMS 这个术语——*database management system*——指的是管理数据库的软件系统。RDBMS 是一个关系型的 DBMS。

同数据库打交道的需要太平常了,存在许多 Java 产品来简化它。Enterprise JavaBean (EJB)、Hibernate 和 JDO 是三种比较流行的技术,尝试简化 Java 的持久化。JDBC 是许多这类工具的基础。一般来说,在考虑使用更高层次的工具之前,了解其基础如何工作,可以事半功倍。这样做可以让你对工具如何工作有更好的理解。你甚至可能会发现,一个定制的、高度重构的 JDBC 实现,是更好的方案。

在这简略的一节中,我会演示如何编写简单的 JDBC 测试,以及和数据库交互的代码。和 Swing 一样,关于 JDBC 有许多专著和很多优秀的网站。关于 JDBC 的更多细节,你可以参考这些补充资源。你还可以参考关于 SQL 的补充资源。在本节中,我对 SQL 只做了很少的说明。

本节中的示例和 MySQL 打交道,你可以在 <http://www.mysql.com> 自由地得到这个数据库 (freely, 译注:这里的自由,指的并不是免费,而是说你可以自由地得到它的源代码、并可以按照一定的协议修改后重新发布;软件通常是免费的,而对软件的服务则未必)。大部分的代码也适用于其他数据库。不过,某些连接数据库的代码细节可能不同。而且,少数“起步”的测试,假定数据库中已存在了要测试的数据结构。

你还需要 MySQL 的 JDBC driver(参见 <http://dev.mysql.com/downloads/connector/j/3.0.html>)。驱动是一个满足了 Sun 的 JDBC 接口规范的 Java 类库。你可以一直使用相同的、Sun 提供的 JDBC 接口方法,而不管数据库或数据库驱动的供应商 (vendor) 是谁。数据库驱动的实现,将 JDBC API 的调用适配到数据库本身的具体需求。

连接到数据库

和数据库打交道时最困难的方面,可能是和它们建立连接。安装 MySQL 时总会创建一个名为 test 的数据库可以让你访问。你的第一个测试应该确保,你可以访问 test 数据库。

确保你已经安装了 MySQL JDBC 驱动,并将它加入到你的 classpath 中。注意,在编译时你并不需要驱动位于 classpath 中,但在执行时需要。

```
package sis.db;

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

public class JdbcAccessTest extends TestCase {
    public void testConnection() throws SQLException {
        JdbcAccess access = new JdbcAccess("test");
        Connection connection = null;
        try {
            connection = access.getConnection();
            assertFalse(connection.isClosed());
        }
        finally {
            connection.close();
        }
    }
}
```

测试足够简单。使用数据库名 test 创建一个 JdbcAccess 实例。从访问对象请求一个连接,

并验证这个连接是打开的（不是关闭的）。使用 `finally` 代码块确保连接在最终会被关闭。

几个注解：首先，测试本身可能抛出 `SQLException`。你可以对 `SQLException` 进行简略的封装——希望尽可能少的类知道你在使用 JDBC。



将你系统中对 JDBC 的使用隔绝到一个类中。

其次，你通常不希望客户端代码请求它自己的连接对象。许多系统的共同问题是，客户端代码忘记关闭连接。客户打开越来越多的连接。最终，当系统无法提供更多的连接时，崩溃了。现在编写一个验证建立连接能力的测试，可以让你渐进式地前进。你可能不会想要公开 `getConnection`。

```
package sis.db;

import java.sql.*;

public class JdbcAccess {
    private String database;

    public JdbcAccess(String database) {
        this.database = database;
    }

    Connection getConnection() throws SQLException {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (Exception cause) {
            throw new SQLException(cause.getMessage());
        }
        String url = "jdbc:mysql://localhost/" + database;
        return DriverManager.getConnection(url);
    }
}
```

`JdbcAccess` 中的 `getConnection` 实现，涉及了两个步骤：首先，加载 JDBC 驱动类。然后，通过向 `DriverManager` 类的 `getConnection` 方法传入一个数据库的 URL，来建立一个连接。

加载驱动的推荐技术是，使用反射。你可以将一个表示类名称的 `String`，传给 `Class` 的类方法 `forName`。某些驱动需要你调用 `newInstance` 来实际创建一个驱动的实例⁶。但是，没有任何原因阻止你直接创建一个它的新实例：

```
Connection getConnection() throws SQLException {
    new com.mysql.jdbc.Driver();
    String url = "jdbc:mysql://localhost/" + database;
    return DriverManager.getConnection(url);
}
```

建议使用反射的最初原因是为了灵活性。使用 `Class.forName`，你可以从一个 `property` 文件或者通过 `system property`（参见本章的 `Property` 一节），取得驱动类的名称。然后，你可以得到无需修改代码而改变驱动的灵活性。在现实中，改变驱动并不常发生，而且如果你真的要这

⁶ MySQL 不需要。查看你的驱动文档，或者你可以偷下懒，总调用 `newInstance`。

么做，可能也不得不再次访问你的代码。⁷

当你调用 `Class.forName` 时（或者当你第一次引用并初始化这个类时），驱动中的静态初始化段（static initializer）被调用。静态初始化段中的代码负责将驱动注册到 `DriverManager` 中。`DriverManager` 维护了所有已注册驱动的列表，并基于调用 `getConnection` 时传入的 URL 来选择 一个驱动。

另一个指定一个或多个驱动的方法是，使用 `system` 的“`jdbc.drivers`” property 提供驱动类的名字。关于如何设置这个 property，参见本章稍后的“Property”一节。

数据库 URL 中信息的组织，取决于驱动器的供应商。它通常遵循几个约定，例如以单词 `jdbc` 开头，之后是一个子协议字符串（subprotocol，本例中是 `mysql`；它通常是供应商的名字）。使用冒号（:）来隔开 URL 中的元素。在 MySQL 中，你使用服务器的名字（这里为 `localhost`——你自己的机器）和数据库的名字组成子协议。

667

调用 `getConnection` 最终得到的是一个 `java.sql.Connection` 的对象。你需要先建立连接，才能执行任意一条 SQL 语句。

`Connection` 是稀缺资源。你只能创建数量有限的连接。而且，从性能的角度来看，建立一个新连接是很耗时的。连接池（Connection Pool）让你可以维护始终保持打开的若干连接。客户从这个池中请求连接，使用它，然后释放这个连接并返回给连接池。某些 JDBC 实现直接支持连接池；有些则没有。为了这个简单的、单客户访问的练习，连接池没什么必要。

执行查询

第二个测试演示了如何执行 SQL 语句。

```
package sis.db;

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

public class JdbcAccessTest extends TestCase {
    private JdbcAccess access;

    protected void setUp() {
        access = new JdbcAccess("test");
    }
    ...
    public void testExecute() throws SQLException {
        access.execute("create table testExecute (fieldA char)");
        try {
            assertEquals("fieldA",
                access.getFirstRowFirstColumn("desc testExecute"));
        }
        finally {
            access.execute("drop table testExecute");
        }
    }
}
```

⁷ 当然，支持第二个驱动的需要，已经是消除对类名称硬编码（hard-coded）的充分原因了。

```

    }
}

```

你不能假设 `test` 数据库中有什么表 (table) 存在, 因此 `testExecute` 创建它自己的表。SQL 语句 “`create table testExecute (fieldA char)`” 创建了一个名为 `testExecute` 的表, 有一个 `char` (character, 字符) 类型的列 `fieldA`。

通过 `getFirstRowFirstColumn` 方法证实表被创建了。查询 (query) `desc testExecute` 返回了 `testExecute` 表的描述信息。SQL 命令 `desc` 是 `describe` 的缩写。`desc` 命令为每一列返回一行描述信息。其中第一列是列名。

668

作为最后的措施, 测试通过使用 SQL 命令 `drop table` 删除 `testExecute` 表。

稍加重构的 `JdbcAccess`, 演示了新的方法 `execute` 和 `getFirstRowFirstColumn`。重构消除了一些代码重复。它还通过确保驱动实例只被加载一次, 消除了执行时的重复。`execute` 和 `getFirstRowFirstColumn` 方法, 分别得到并释放自己的连接。这减轻了客户端的职责。

```

package sis.db;

import java.sql.*;

public class JdbcAccess {
    private String url;
    ...
    public void execute(String sql) throws SQLException {
        Connection connection = getConnection();
        try {
            Statement statement = connection.createStatement();
            statement.execute(sql);
        }
        finally {
            close(connection);
        }
    }

    private void close(Connection connection) throws SQLException {
        if (connection != null)
            connection.close();
    }

    Connection getConnection() throws SQLException {
        if (url == null) {
            loadDriver();
            url = "jdbc:mysql://localhost/" + database;
        }
        return DriverManager.getConnection(url);
    }

    private void loadDriver() throws SQLException {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (Exception cause) {
            throw new SQLException(cause.getMessage());
        }
    }
}

```

```

public String getFirstRowFirstColumn(String query)
    throws SQLException {
    Connection connection = getConnection();
    try {
        Statement statement = connection.createStatement();
        ResultSet results = statement.executeQuery(query);
        results.next();
        return results.getString(1);
    }
    finally {
        close(connection);
    }
}
}

```

669

你需要一个 **Statement** 对象来执行 SQL。通过调用 **Connection** 的 `createStatement`，从连接得到一个 **Statement** 对象。一旦你得到了一个 **Statement**，你可以调用 `execute` 方法来执行预期没有返回结果的 SQL 语句，或者使用 `executeQuery` 来执行需要随后得到结果的 SQL 语句。你传入一个 SQL 字符串作为 `execute` 或 `executeQuery` 的参数。

结果以 **ResultSet** 对象的形式返回。**ResultSet** 非常类似于一个 **Iterator**。它维护了一个指向当前行的游标。这个游标最初什么也不指向，你可以通过调用 **ResultSet** 的 `next` 方法来向前移动这个游标。在内部游标越过最后一行之前，`next` 方法返回 `true`。你可以通过列名或列索引（index）来访问列的数据。获得列的值有许多方法，根据每列中保存数据类型的不同而不同。例如，**ResultSet** 的 `getString` 方法返回字符串数据，而 `getInt` 方法返回一个 `int` 值。

`getFirstRowFirstColumn` 中的代码，通过调用 `results.next()`，将游标移动到 `executeQuery` 返回结果的第一行。`getFirstRowFirstColumn` 方法，通过使用索引（index）1 为参数、调用 **ResultSet** 的 `getString` 方法，得到第一列的值。在 JDBC 中，列索引从 1 而不是从 0 开始。

PreparedStatement

你经常想要在一个应用中多次执行同一条语句，而可能仅仅是 **key** 信息不同。例如，SIS 应用需要基于学生的学号（ID）进行快速的查找。

因为你以 **String** 的形式传入一个 SQL 语句，语句在被数据库执行之前必须要编译。对语句进行编译，确保它遵循有效的 SQL 语法。编译会花费很多的执行时间，这会造成性能的问题。（和往常一样，剖析 [profile] 你的代码，肯定会发现性能很差的地方。）为了加速 SQL 语句的执行，你可以使用 SQL 字符串创建一个 **PreparedStatement** 对象。**PreparedStatement** 只一次编译 SQL，并将编译后的版本保存起来，以备之后更快速的使用。

670

你可以用问号的形式在 SQL 字符串中插入占位符（placeholder）。之后，你可以为这些占位符绑定值。一条查找学生的适当 SQL 字符串可以是：

```
select id, name from testQueryBy where id = ?
```

对每次学生查找，你可以使用 `set` 方法为问号绑定一个值。`statement.setString(1,`

"boo")会把值 'boo' 绑定到第一个 (也是唯一一个) 问号参数。

```
public void testQueryBy() throws SQLException {
    drop("testQueryBy");
    access.execute(
        "create table testQueryBy (id varchar(10), name varchar(30))");
    PreparedStatement statement = null;

    try {
        access.execute("insert into testQueryBy values('123', 'schmoe')");
        access.execute(
            "insert into testQueryBy values('234', 'patella')");

        statement =
            access.prepare("select id, name from testQueryBy where id = ?");

        List<String> row = access.getUnique(statement, "234");
        assertEquals("234", row.get(0));
        assertEquals("patella", row.get(1));

        row = access.getUnique(statement, "123");
        assertEquals("123", row.get(0));
        assertEquals("schmoe", row.get(1));
    }
    finally {
        statement.close();
        drop("testQueryBy");
    }
}

private void drop(String tableName) {
    try {
        access.execute("drop table " + tableName);
    }
    catch (SQLException ignore) {
        // exception thrown if table doesn't exist; we don't care
    }
}
```

因为使用 `PreparedStatement` 的着眼点是客户端的效率，你不希望每次在使用 `PreparedStatement` 执行查询之后就关闭连接。在这种情况下，你只得信任让客户端的代码关闭连接。

实现：

```
public PreparedStatement prepare(String sql) throws SQLException {
    Connection connection = getConnection();
    return connection.prepareStatement(sql);
}

public List<String> getUnique(
    PreparedStatement statement, String... values)
    throws SQLException {
    int i = 1;
    for (String value: values)
        statement.setString(i++, value);
    ResultSet results = statement.executeQuery();
}
```

```

results.next();

List<String> row = new ArrayList<String>();
ResultSetMetaData metadata = results.getMetaData();
for (int column = 1; column <= metadata.getColumnCount(); column++)
    row.add(results.getString(column));
return row;
}

```

你可以从 `ResultSet` 中提取到元数据 (metadata)。`ResultSetMetaData` 对象提供了有关返回结果的有用信息：列的个数、每一列的数据类型、每一列的值，等等。`getUnique` 的实现使用 `getColumnCount` 得到列的数目，并按照这个数目迭代所有列的值。

元数据在数据库级别也可以得到。你可以调用 `Connection` 对象的 `getMetaData`。然后，你得到一个 `DatabaseMetaData` 对象，其中填充了有关数据库和驱动的很多信息，包括表和列、版本、驱动局限、和数据库局限。

JDBC 应用设计

在一个应用中，有许多方式使用 JDBC 来实现持久化。如之前提到的那样，你可以使用建立在 JDBC 之上的工具，甚至替代 JDBC。或者，遵循 TDD，使用零重复 (zero duplication) 的哲学，你可以发展一个类似的工具。通过时刻警惕着粉碎重复，你可以快速地建立一个工具来满足你的需求。和第三方工具不同，你自己的实现可以足够灵活，使它可以按照要求快速地改变。

你可以使用上面的示例作为编写你自己工具的起点。`JdbcAccess` 类应该最终成为你唯一同 JDBC API 打交道的地方。应用剩余的大部分，可以愉快地忽略 JDBC 作为持久化机制的事实。如果你选择日后将 JDBC 替换为一个面向对象的数据库，替换的工作量将会很小。

◀ 672

你可以创建在数据库列和领域属性之间进行转换的映射对象。你可以使用代码生成 (code generation，这是首选的方法) 或反射来实现它。甚至还可能将所有内容以 String 的格式保存在数据库中，然后在 `JdbcAccess` 层进行类型转换。这可以保持让 `JdbcAccess` 尽可能简单：它提供了若干访问方法，返回普通的字符串列表的列表 (若干行数据列，每一列都是一个字符串)。

SQL 语句常蔓延着严重的重复。编写一个使用数据库的元数据来构造 SQL 语句的 SQL 生成器，还算容易。

国际化 (Internationalization)

大量的软件产品开始部署到世界上的任何一个角落。过不了多久，你就不再是仅仅为自己的母语部署应用。在某些地方，例如 Quebec (魁北克)，需要所有的软件在两个或更多语言中执行。在国际市场上的成功，取决于你在多个国家以多种不同语言销售软件的能力。

建立支持不同语言和文化的软件，被称为国际化（internationalization），或简写为 *i18n*⁸。为一个语言环境（locale）准备交付软件，被称为本地化（localization）（译注：或简写为 *l10n*）。locale 是一个区域。地理、文化或政治均可定义 locale。

改进一个已有的大型应用来支持多语言，是非常昂贵的。这可能会让你相信，你必须从开发的最初就国际化你的应用。也许。非常执著“无重复”的原则，会让你留下一个能够支持快速过渡到国际化软件的设计。虽然如此，为国际化建立一个基础可以使事情更简单些。你可以检验建立这些基础是值得的，因为它们能帮助消除重复。



673

使用国际化机制帮助消除重复，并让你可以准备好对应用进行本地化。

资源包（Resource Bundle）

直接在代码中嵌入文字的（literal）字符串会导致很多问题。更重要的是，它们会产生重复。如果一个文字字符串出现在产品代码中、并且你要完成所有必要的测试，某些测试肯定会重复相同的文字（literal）。而且，文字的含义并不总是那么清晰。在《Agile Java》中，我已经让你把所有的字面 String 抽取出来，作为类的常量。测试和产品代码都可以引用相同的常量。常量的名字透露了其含义。

即使如此，不得不对一大群类常量进行维护，是非常痛苦的。再考虑到应用的国际化需求。理想的方案是将所有文字抽取到一个公共文件中。当你必须将软件部署到一个不同的 locale 时，你只需要提供一个包含翻译文本的新文件。

Java 中 `java.util.ResourceBundle` 类的实例，管理了特定 locale 资源文件的交互。为了国际化你的软件，你需要更新所有的代码，向一个 `ResourceBundle` 来请求本地化消息 String。

使用 `ResourceBundle` 非常简单。通过传入 bundle 的名字来调用创建方法 `getBundle`，你可以得到一个 `ResourceBundle` 对象。然后你可以将 `ResourceBundle` 作为一个 map 来使用——传入一个 key，它会返回本地化的对象。

测试对 `ResourceBundle` 的使用是另一回事。你希望确保可以从 `ResourceBundle` 中读取一个指定的 property 及其值。但是你无法假定 key 和对应的 value 在资源中总是存在的。最简单的方案是，将一个已知的 key 和 value 写入到一个小文件中。因为，你不希望将应用后面需要的资源文件覆盖掉。

```
package sis.util;

import junit.framework.TestCase;

import java.io.IOException;
```

⁸ internationalization 第一个字母和最后一个字母之间的字母个数是 18 个。

```

public class BundleTest extends TestCase {
    private static final String KEY = "someKey";
    private static final String VALUE = "a value";
    private static final String TEST_APPEND = "test";
    private static final String FILENAME =
        "./classes/sis/util/" + Bundle.getName() + "Test.properties";
    private String existingBundleName;

    protected void setUp() {
        TestUtil.delete(FILENAME);
        existingBundleName = Bundle.getName();
        Bundle.setName(existingBundleName + TEST_APPEND);
    }

    protected void tearDown() {
        Bundle.setName(existingBundleName);
        TestUtil.delete(FILENAME);
    }

    public void testMessage() throws IOException {
        writeBundle();
        assertEquals(VALUE, Bundle.get(KEY));
    }

    private void writeBundle() throws IOException {
        LineWriter writer = new LineWriter();
        String record = String.format("%s=%s", KEY, VALUE);
        writer.write(FILENAME, record);
    }
}

```

674

测试使用 `setUp` 和 `tearDown` 细心地维护了一个测试的资源文件。它和一个你创建的、叫做 `sis.util.Bundle` 的类交互，取得现有 `bundle` 的名字。测试保存现有资源文件的基本名 (`base name`)，然后告诉 `Bundle` 使用一个新的基本名。只有测试知道这个名字。测试本身 (`testMessage`) 输出这个测试的资源文件。资源文件的每一项，是由一个等号 (=) 隔开的 `key` 和 `value` 对儿。

`testMessage` 中的断言调用 `Bundle` 类的 `get` 方法，得到一个本地化的资源。最后，`tearDown` 方法使用原先的基本名重置 `Bundle`。

```

package sis.util;

import java.util.ResourceBundle;

public class Bundle {
    private static String baseName = "Messages";
    private static ResourceBundle bundle;

    static String getName() {
        return baseName;
    }

    static void setName(String name) {
        baseName = name;
        bundle = null;
    }
}

```



```

public static String get(String key) {
    if (bundle == null)
        loadBundle();
    return (String)bundle.getString(key);
}

private static void loadBundle() {
    bundle = ResourceBundle.getBundle("sis.util." + getName());
}
}

```

675

`loadBundle` 方法通过基本名 (base name) 得到一个 `ResourceBundle` 实例。基本名和类的全名 (qualified name) 类似。假设你有一个资源文件，其全路径名是 `./classes/sis/util/Messages.properties`。还假设 `./classes` 目录位于 `classpath` 中。为了让 `getBundle` 定位到这个资源文件，基本名应该是 `"sis.util.Messages"`。

一旦你得到一个 `ResourceBundle` 对象，你可以调用它的几种不同方法，提取出本地化资源。`getString` 方法返回本地化的文本。

一旦你实现了 `Bundle` 类，你可以使用它的 `get` 方法，在应用中替换文字出现的地方。你可能会产生一个很大的 `key-value` 对儿文件。如果文件过大，使得对资源文件的管理很笨重，考虑将它分解到不同的资源包 (resource bundle)。

本地化

假设你需要为墨西哥部署你的应用。你需要将你的资源文件送给翻译者进行本地化。译者的工作是为了返回一个 `key-value` 对儿的新文件，使用西班牙文的翻译替换基础的值 (可能是英文)。

(在现实中，译者可能需要同你或其他理解这个应用的人交互。译者通常需要上下文的信息，因为词在不同上下文有不同的含义。例如，译者需要知道“File”是一个动词还是名词。)

你必须按照特定的 `locale`，命名译者返还给你的文件。`locale` 包括语言、国家 (或地区，可选) 和变体 (也是可选的；它很少被使用)。对墨西哥的部署来说，语言是西班牙语 (Spanish)，以 “es” (español) 表示。国家是 “MX” (Mexico)。为了组成一个资源文件名，你可以使用下划线 (“_”) 作为分隔符，把这部分 `locale` 的信息附加到基本名上。对基本名为 “Message” 的资源文件，为墨西哥本地化后的文件名为 “Messages_es_MX.properties”。

你可以使用 `Locale` 的类方法 `getAvailableLocales`，得到在你平台上支持 `locale` 的完整列表。编写并执行这个有点丑陋的代码：

```

for (Locale locale: Locale.getAvailableLocales())
    System.out.println(String.format("%s %s: use '%s%s'",
        locale.getDisplayLanguage(), locale.getDisplayCountry(),
        locale.getLanguage(),
        (locale.getCountry().equals("")) ? "" : "_" + locale.getCountry())));

```

676

执行这段代码的输出，可以告诉你基本名的适当扩展名 (extension)。

下面是 BundleTest，经过修改后，包括了对墨西哥本地化的一个新测试：

```
package sis.util;

import junit.framework.TestCase;
import java.io.IOException;
import java.util.Locale;

public class BundleTest extends TestCase {
    private static final String KEY = "someKey";
    private static final String TEST_APPEND = "test";
    private String filename;
    private String existingBundleName;

    private void prepare() {
        TestUtil.delete(filename);
        existingBundleName = Bundle.getName();
        Bundle.setName(existingBundleName + TEST_APPEND);
    }

    protected void tearDown() {
        Bundle.setName(existingBundleName);
        TestUtil.delete(filename);
    }

    public void testMessage() throws IOException {
        filename = getFilename();
        prepare();
        final String value = "open the door";
        writeBundle(value);
        assertEquals(value, Bundle.get(KEY));
    }

    public void testLocalizedMessage() throws IOException {
        final String language = "es";
        final String country = "MX";
        filename = getFilename(language, country);
        prepare();

        Locale mexican = new Locale(language, country);
        Locale current = Locale.getDefault();
        try {
            Locale.setDefault(mexican);
            final String value = "abre la puerta";
            writeBundle(value);
            assertEquals(value, Bundle.get(KEY));
        } finally {
            Locale.setDefault(current);
        }
    }

    private void writeBundle(String value) throws IOException {
        LineWriter writer = new LineWriter();
        String record = String.format("%s=%s", KEY, value);
        writer.write(getFilename(), record);
    }
}
```

```

    }

    private String getFilename(String language, String country) {
        StringBuilder builder = new StringBuilder();
        builder.append("./classes/sis/util/");
        builder.append(Bundle.DEFAULT_BASE_NAME);
        builder.append("Test");
        if (language.length() > 0)
            builder.append("_" + language);
        if (country.length() > 0)
            builder.append("_" + country);
        builder.append(".properties");
        return builder.toString();
    }

    private String getFilename() {
        return getFilename("", "");
    }
}

```

BundleTest 需要一些重构来支持资源文件格式的改变。你不能使用 `setUp` 方法来删除这个测试的资源文件，因为它的名字必定改变了。我将 `setUp` 改名为 `prepare` (目前已经足够了)。`prepare` 方法假定 `filename` 字段已经被赋值为适当的资源文件名。

新的测试，`testLocalizedMessage`，将测试数据写入一个名字由语言和国家构成的资源文件中。测试使用相同的语言和国家来创建一个 `Locale` 对象。然后它得到当前的 (缺省) `Locale`，这样在测试完成之后，它可以将 `Locale` 重置为缺省的值。作为测试 `Bundle.get` 方法的最后一项准备，测试将缺省的 `locale` 设置为新创建的墨西哥 `Locale` 对象。

设置 `Locale` 是一个全局的改动。所有可能需要国际化的代码都可以请求当前的 `Locale`，以进行后续的处理。大多数时候，这已经为你处理好了。就 `ResourceBundle` 的情况来说，你什么都不必做。要加载一个 `bundle`，你只需要提供基本名。`ResourceBundle` 中的代码取得缺省的 `Locale`，并组合其中的信息与基本名，创建一个完整的资源文件名。

格式化消息

资源文件中的项可以包括可替换的占位符。例如：

678

```
dependentsMessage=You have {0} dependents, {1}. Is this correct?
```

`dependentsMessage` 的值包括两个格式元素：`{0}` 和 `{1}`。通过 `ResourceBundle` 加载这个字符串的代码，可能使用一个 `MessageFormat` 对象，将格式元素替换为合适的值。下面的语言测试演示了：

```

public void testMessageFormat() {
    String message = "You have {0} dependents, {1}. Is this correct?";

    MessageFormat formatter = new MessageFormat(message);
    assertEquals(

```

```
"You have 5 dependents, Señor Wences. Is this correct?";
formatter.format(message, 5, "Señor Wences"));
}
```

在使用 `MessageFormat` 的早期代码中，你可以看到在 `Object` 数组中包装的参数：

```
formatter.format(new Object[] {new Integer(5), "Señor Wences"})
```

为什么不直接使用 `java.util.Formatter` 类呢？原因是 `MessageFormat` 方式先于 `Formatter` 类（Sun 在 Java 5.0 中引入的）。

使用 `MessageFormat` 方式更令人感兴趣的原因是，利用 `ChoiceFormat` 类的好处。这可以消除编写额外 `if/else` 的逻辑，来提供格式元素的值。典型的示例，演示了如何管理占位符编号消息的格式化。你希望为“Señor Wences”显示一个更友好的消息，并且语法上“one dependents”是不能接受的。⁹

使用 `ChoiceFormat` 是有些负担。你首先要创建数字区间（range）和对应文本的映射。你可以使用一个 `double` 的数组区间作为界限。你以字符串数组的形式，表示对应的文本或格式。

```
double[] dependentLimits = {0, 1, 2 };
String[] dependentFormats =
    {"no dependents", "one dependent", "{0}dependents" };
```

界限数组中的各个数字就是区间。界限数组中的最后一个元素表示区间的低端（low end）——在本例中，表示至多两个依赖。对两个依赖来说，`ChoiceFormat` 会使用对应的格式字符串“{0}dependents”，并用实际的依赖个数替换 {0}。

使用这些界限和格式，你可以创建一个 `ChoiceFormat` 对象：

```
ChoiceFormat formatter =
    new ChoiceFormat(dependentLimits, dependentFormats);
```

679

一个消息字符串可能有多个格式元素。只有某些格式元素可能需要 `ChoiceFormat`。你必须创建 `Format` 对象的一个数组（`Format` 是 `ChoiceFormat` 的父类），如果你不需要格式化程序时，将它替换为 `null`。

```
Format[] formats = {formatter, null };
```

然后你可以创建一个 `MessageFormat` 对象，并为它设置格式数组：

```
MessageFormat messageFormatter = new MessageFormat(message);
messageFormatter.setFormats(formats);
```

你现在已经可以像以前一样使用 `MessageFormat` 了。语言测试 `testChoiceFormat` 演示了整个过程。

```
public void testChoiceFormat() {
    String message = "You have {0}, {1}. Is this correct?";
```

⁹ 毫无疑问，他对此会有愤怒的回应，“Tell it to the hand”。

```

double[] dependentLimits = {0, 1, 2};
String[] dependentFormats =
    {"no dependents", "one dependent", "{0}dependents" };

ChoiceFormat formatter =
    new ChoiceFormat(dependentLimits, dependentFormats);

Format[] formats = {formatter, null };

MessageFormat messageFormatter = new MessageFormat(message);
messageFormatter.setFormats(formats);

assertEquals(
    "You have one dependent, Señor Wences. Is this correct?",
    messageFormatter.format(new Object[] {1, "Señor Wences" }));

assertEquals(
    "You have 10 dependents, Señor Wences. Is this correct?",
    messageFormatter.format(new Object[] {10, "Señor Wences" }));
}

```

同 `ChoiceFormat` 打交道可以更复杂。参考关于这个类的 Java API 文档。

注意格式字符串应该来自于你的 `ResourceBundle`。

国际化的其他领域

数字、日期和货币符号的格式化非常依赖于 `locale`。在显示数字时，每隔三个数字使用分隔符，使它们更可读。在美国，分隔符是逗号 (,)，而在其他某些地方使用句点 (.)。在美国，日期经常以月一日一年的顺序显示，使用斜线或连字符隔开。日期在其他地方可能以不同的顺序显示，例如年一月一日。

`java.text.NumberFormat`、`java.util.Calendar`、`java.text.DateFormat` 以及它们的子类，支持使用 `Locale` 对象。你的代码可以使用工厂方法来请求这些类的实例，然后操作它们。你的代码不应该直接初始化这些类，否则你将无法得到缺省 `Locale` 的支持。例如，要得到一个支持缺省 `locale` 的 `SimpleDateFormat` 实例：

```
DateFormat formatter = SimpleDateFormat.getDateInstance();
```

`Calendar` 类还可以让你管理时区 (time zone)。

Java 中的各种布局管理器，支持不同的坐标轴方向。在某些非西方的文化中，阅读文本的顺序是从上到下，而不是从左到右。例如，`BoxLayout` 类允许你使用 `LINE_AXIS` 组织容器内的组件。每个 `Locale` 都包括一个 `java.awt.ComponentOrientation` 的对象：这个对象定义了文本是从右向左还是从左向右（西方的）读，文本行 (line) 是水平的（西方的）或垂直的。然后，在运行时，`BoxLayout` 得到 `ComponentOrientation`，以决定 `LINE_AXIS` 对应 x 轴还是 y 轴。

其他编程语言需要使用不同的字符集。因为 Java 使用 Unicode（译注：编码为 UTF16），你无需关心不同的字符集。不过，如果你需要对文本元素排序，你将不得不接触到 `collation`（对

照) 序列。java.text.RuleBasedCollator 为此提供了基本的支持。在有关这个类的 API 文档中, 包括了一个对 Java collation 很好的综述。

按引用调用 vs. 按值调用

关于 Java 是使用“按引用调用 (call by reference)”还是“按值调用 (call by value)”, 已经造成了太多的纷扰。这些经典的软件开发用语, 表述了编程语言如何管理向函数 (function, C 中的用语) 或方法传入的参数。术语 (或者人们对术语的解释) 并不重要。重要的是你理解 Java 如何管理参数的传递。

按值调用意味着被调用的函数¹⁰在幕后对参数进行了拷贝。函数中的代码操作这个拷贝。意味着对参数进行的任何改动, 都会在函数执行完毕时被丢弃掉。原因是, 改动实际只作用于局部的拷贝, 而非传入的参数。参数的拷贝只在方法范围中有效。

681

按引用调用意味着函数操作的和传入的参数, 在物理上是相同的。对参数的任何改动都会保持下来。

Java 完全是按值调用的。如果你将一个 int 传给一个方法, 该方法操作这个 int 值的拷贝。如果你将一个对象的引用传给一个方法, 方法所操作的是引用的拷贝——而不是对象本身的拷贝。我将在代码中演示这两种语句。

理解对基本类型 (primitive) 的按值调用, 很容易。下面的测试演示了, 方法中对基本类型的拷贝是如何被丢弃的。

```
public void testCallByValuePrimitives() {
    int count = 1;
    increment(count);
    assertEquals(1, count);
}

private void increment(int count) {
    count++;
}
```

即使 increment 方法改变了 count 的值, 这个改动只作用于局部的拷贝。局部拷贝在 increment 方法退出之后就消失了。testCallByValuePrimitives 中的代码, 对其中的任何改动, 都茫然不知。

回忆一下, 引用是一个指针——对象在内存中的地址。被调用的方法拷贝这个地址, 创建一个新的指针指向同样的内存位置。如果你在被调用方法的内部, 将一个新的内存地址 (也就是一个不同的对象) 赋值给这个引用, 新的地址在方法完成时也会被丢弃。

被调用方法中的代码, 可以调用引用所指向对象中的方法。对这些方法的调用, 会导致对象状态的永久改变。

¹⁰ 我使用术语“函数 (function)”用于这些一般性的概念, 意味讨论并不仅限于 Java。

`testCallByValueReferences` 中的代码（在这一段之后演示），使用 ID 1 创建了一个客户。测试然后调用 `incrementId` 方法。`incrementId` 中的第一行代码，从客户对象中得到现有的 ID，将这个数加 1，并将和的结果置回给客户对象。`incrementId` 中的第二行，创建了一个新的 `Customer` 对象，并将它的地址赋值给 `customer` 参数引用。你不希望这样做！¹¹这个赋值在 `incrementId` 完成时，被丢弃。测试最终验证了客户 ID 是 2，而不是 22。

682

```
public void testCallByValueReferences() {
    Customer customer = new Customer(1);
    incrementId(customer);
    assertEquals(2, customer.getId());
}

private void incrementId(Customer customer) {
    customer.setId(customer.getId() + 1);
    customer = new Customer(22); // don't do this
}

class Customer {
    private int id;
    Customer(int id) {this.id = id; }
    void setId(int id) {this.id = id; }
    public int getId() {return id; }
}
```

Java 的边缘地带

Java 为你提供了许多机制，来帮助你的应用和本地的操作环境相交互。

Property

在你应用中的任意地方，都可以获得一个系统 `property` 的小型哈希表（hash table）。这个表包含了许多有用的信息，包括与你的 Java 环境和操作系统有关的内容。你可以使用 `java.lang.System` 的 `getProperties` 方法，得到 `property` 的整个集合：

```
Properties existingProperties = System.getProperties();
```

`java.util.Properties` 类从旧有的 `java.util.Hashtable` 类派生。它添加了许多管理方法和实用方法。管理方法帮助你完成诸如从一个文件中加载 `property` 集合等。实用方法简化了对哈希表的存取：

```
Java.class.path    Java classpath 的值12
```

¹¹ 你可以考虑将你的所有参数声明为 `final`。尝试对这些参数赋值会导致编译错误。我的固有习惯始终会阻止我这样做。

¹² 你可以将这个 `property` 设置为一个新的值，但是它当前 Java 的执行没有任何影响。Java VM 不会重新读取这个值。如果你认为需要改变这个 `classpath`，你可能需要建立一个自定义的类加载器。

<code>Java.io.tmpdir</code>	缺省的临时文件路径
<code>file.separator</code>	文件分隔符（在 Unix 下是“/”；在 Windows 下是“\”）
<code>path.separator</code>	路径项的分隔符（在 Unix 下是“:”；在 Windows 下是“;”）
<code>line.separator</code>	行分隔符（在 Unix 下是“\n”；在 Windows 下是“\r\n”）
<code>user.home</code>	当前用户的主（home）目录
<code>user.dir</code>	当前的工作目录

使用这些 `property`，你可以编写应用，不加改动就可以运行在不同的操作系统。

通过使用 `System` 的类方法 `getProperty`，你可以得到每个系统 `property` 的值：

```
assertEquals("1.5.0-beta3", System.getProperty("java.version"));
```

系统 `property` 是全局的。你可以从代码中的任何地方存取它们。

你可以将你自己的全局 `property` 添加到系统 `property` 的列表中。一般来说，数据库是存储这类信息更合适的地方。你使用 `property` 的主要原因应该是，为了在应用的不同执行之间动态地更改一小段信息。例如，你所部署的应用，可能需要在大部分时间以“kiosk”模式运行，闭合并占据整个屏幕，没有边框或标题栏。在部署中，你可能希望能够快速地在不同执行模式之间进行切换。

你可以通过许多方式实现这种动态性。你可以将一个模式设置标志放到数据库中，但是必须修改数据库中的数据，会令你停顿下来。你还可以将模式信息作为命令行参数，并在你的 `main` 方法中对它进行管理。但是你可能直到应用的后期才需要这个信息；到时你该怎么办呢？

你可以使用编程的方式，将你的 `property` 添加到系统 `property` 中：

```
System.setProperty("kioskMode", "loopFullscreen");
```

或者，向你在第 8 章日志中看到的那样，将 `property` 设置为 VM 的参数：

```
java DkioskMode=noloopFullscreen sis.Sis
```

如果 `property` 没有设置，`getProperty` 返回 `null`。你可以为 `getProperty` 提供一个缺省值，在 `property` 没有设置时返回。一个语言测试演示了它如何工作：

```
public void testGetPropertyWithDefault() {
    Properties existingProperties = System.getProperties();
    try {
        System.setProperties(new Properties()); // removes all properties
        assertEquals(
            "default", System.getProperty("noSuchProperty", "default"));
    }
    finally {
        System.setProperties(existingProperties);
    }
}
```


Property 文件

你在许多软件公司和应用程序中都会遇到这项使用 `property` 文件的技术。`property` 文件和资源文件非常类似；它是一组以 `key=value` 形式存在的 `key-value` 对儿集合。开发者通常把 `property` 文件作为所有可配置项的一剂低劣的灵丹。诸如驱动的类型、配色方案、各种布尔标志，都一股脑儿地塞到 `property` 文件中。

驱使的力量是，在将软件从产品环境转入到测试环境、或者集成的产品环境时，需要修改某些特定的值。例如，你的代码可能需要连接一个 Web 服务器。你用于开发的 Web 服务器名，和产品中使用的 Web 服务器名可能是不同的。

我建议你将 `property` 文件的使用减至最小，至少将放入其中的配置项减至最小。`Property` 文件中的配置项，和你的应用是脱节的；它们表示了一串随意堆放在一起的全局变量。相反，你应该仔细地可将配置的值和它们所影响的类关联到一起。对这种情况，最好的办法是使用数据库。



使 `property` 文件尽可能小而简单，或者消除它们更好。忍住不要添加新的项。

如果你必须使用 `property` 文件，`Properties` 类包括了若干方法，从一个 `key-value` 文本格式或 XML 格式的输入流中，加载它们。不要试图从你执行的应用中，动态地更新这个文件。

首选项 (Preference)

685

对于你必须要在应用执行之间保存的关键信息，你可以使用数据库或者值得信赖的持久化机制。不过，你经常希望能快速并简单地保存各种少量非关键的信息。通常这是有关用户的一些琐事：首选的颜色，窗口的位置，最近访问的 Web 站点等等。如果任何这类信息丢失了，对用户来说是有些惹人讨厌，但是不会妨碍他或她正地完成使用。

Java Preferences API 为这类数据提供了一个简单的本地持久化方案。`PersistentFrameTest`（在下一段之后演示的）演示了如何使用它。第一个测试，`testCreate`，演示了当 `frame` 第一次创建时，它所使用的是缺省的窗口位置。测试 `testMove` 移动了这个 `frame`，然后关闭它。测试演示了后续创建的 `frame`，会使用 `PersistentFrame` 最近的一次位置。

为了让 `PersistentFrameTest` 可以不止一次的工作，`setUp` 方法调用了 `frame` 的 `clearPreferences`。这个方法确保 `PersistentFrame` 对象使用缺省的窗口位置设置。

```
package sis.ui;

import junit.framework.*;
import java.awt.*;
import java.util.prefs.BackingStoreException;
```

```

import static sis.ui.PersistentFrame.*;

public class PersistentFrameTest extends TestCase {
    private PersistentFrame frame;

    protected void setUp() throws BackingStoreException {
        frame = new PersistentFrame();
        frame.clearPreferences();
        frame.initialize();
        frame.setVisible(true);
    }

    protected void tearDown() {
        frame.dispose();
    }

    public void testCreate() {
        assertEquals(
            new Rectangle(
                DEFAULT_X, DEFAULT_Y, DEFAULT_WIDTH, DEFAULT_HEIGHT),
            frame.getBounds());
    }

    public void testMove() {
        int x = 600;
        int y = 650;
        int width = 150;
        int height = 160;
        frame.setBounds(x, y, width, height);
        frame.dispose();

        PersistentFrame frame2 = new PersistentFrame();
        frame2.initialize();
        frame2.setVisible(true);
        assertEquals(
            new Rectangle(x, y, width, height), frame2.getBounds());
    }
}

```

686

PersistentFrame 中的实现:

```

package sis.ui;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.prefs.*;

public class PersistentFrame extends JFrame {
    static final int DEFAULT_X = 100;
    static final int DEFAULT_Y = 101;
    static final int DEFAULT_WIDTH = 300;
    static final int DEFAULT_HEIGHT = 400;

    private static final String X = "x";
    private static final String Y = "y";
    private static final String WIDTH = "width";
    private static final String HEIGHT = "height";
}

```

```

private Preferences preferences =
    Preferences.userNodeForPackage(this.getClass());           // 1

public void initialize() {
    int x = preferences.getInt(X, DEFAULT_X);                 // 2
    int y = preferences.getInt(Y, DEFAULT_Y);
    int width = preferences.getInt(WIDTH, DEFAULT_WIDTH);
    int height = preferences.getInt(HEIGHT, DEFAULT_HEIGHT);

    setBounds(x, y, width, height);

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    addWindowListener(new WindowAdapter() {
        public void windowClosed(WindowEvent e) {
            saveWindowPosition();
        }
    });
}

private void saveWindowPosition() {
    Rectangle bounds = getBounds();

    preferences.putInt(X, bounds.x);                           // 3
    preferences.putInt(Y, bounds.y);
    preferences.putInt(WIDTH, bounds.width);
    preferences.putInt(HEIGHT, bounds.height);
    try {
        preferences.flush();                                   // 4
    }
    catch (BackingStoreException e) {
        // not crucial; log message
    }
}

// for testing
void clearPreferences() throws BackingStoreException {
    preferences.clear();
    preferences.flush();
}
}

```

通过向 `Preferences` 的两个工厂方法, `userNodeForPackage` 或 `systemNodeForPackage`, 传入一个类名, 你可以创建一个 `Preference` 对象。你可以为特定于应用的信息, 使用系统的首选项树, 例如 `property` 文件的位置。你可以为特定于用户的信息, 使用用户的首选项树, 例如颜色选择。

Java 以类似文件系统目录树的结构, 将首选项保存在树中。取决于平台 (和具体的实现), 首选项的实际存储位置, 可能是操作系统的注册表 (registry)、文件系统、目录服务器或者数据库。

`PersistentFrame` 类使用用户首选项树 (参见第一行)。`Frame` 的初始化代码尝试读取 `frame` 边框¹³的首选项 (从第二行开始)。每次调用 `getInt`, 包括一个缺省值的参数, 在没有对应的

¹³ 指定了窗口边界的一个 `Rectangle` 对象。(x, y)表示它原点, 或者左上角。height 和 width 表示窗口的像素范围。

首选项数据存在时使用它。

当 `frame` 被关闭时（通过客户端代码调用 `dispose` 方法，或其他方式），`saveWindowPosition` 中的代码使用 `Preferences` API 将 `frame` 当前的边界保存起来。你必须对 `Preferences` 对象执行 `flush`（第四行），来强制写入到持久化存储中。除了 `putInt`（第三行），`Preferences` 类还提供了其他方法来支持保存各种类型的数据：`String`、`boolean`、`byte` []、`double`、`float` 和 `long`。

`Preferences` API 提供了从 XML 格式的文件中、读取或写入数据的方法。

688

系统环境

在少数情况下，你还需要了解和你操作系统环境有关、但预定义的系统 `property` 又没有提供的信息。`Unix` 和 `Windows` 操作系统有环境变量，又是一种 `key-value` 对的方式。从命令行提示符，你可以使用命令 `set`¹⁴ 来查看当前的环境变量。在 `Java` 中，`System` 类提供了两个 `getEnv` 方法，来帮助你得到这些环境变量和它们的值。

```
Map<String,String> env = System.getenv();
for (Map.Entry entry: env.entrySet())
    System.out.printf("%s->%s\n", entry.getKey(), entry.getValue());

System.out.println(System.getenv("SystemRoot")); // Unix: try "SHELL"
```

避免依赖于系统的环境变量。这会使得将你的应用很难移植到其他平台。`Windows` 和 `Unix` 之间几乎没有共同的环境变量。如果你觉得必须创建你自己的系统级环境变量，考虑使用 `Java` 的 `property` 机制。

执行其他应用

在少数情况下，你可能需要执行另一个操作系统的进程。例如，你可能希望应用执行系统的计算器程序（在 `Windows` 中是 `calc.exe`）。`Java` 可以允许你执行单独的进程，但是你应该尽量避免这样做。它通常会将你绑定到一个单一的操作系统。多数情况下，如果你必须处理多个平台，那么你需要维护条件逻辑。

在 `Java` 中，有两种方法创建并执行单独的进程。原有的技术需要你使用 `java.lang.Runtime` 的类方法 `getRuntime` 得到它的单体（`singleton`¹⁵）实例。然后你可以调用其 `exec` 方法，传入一个命令行字符串以及其他可选的参数。`JSE 5.0` 引入了一个 `ProcessBuilder` 类来替换这项技术。你可以使用这个命令行字符串创建一个 `ProcessBuilder`，然后通过调用这个 `ProcessBuilder` 对象

¹⁴ 这个命令在 `Windows` 和大多数 `Unix shell` 中都适用。

¹⁵ `Runtime` 类的设计是在一个执行的 `Java` 应用中，`Runtime` 只有唯一一个实例，因此 `Runtime` 是设计模式 `singleton` 的一个例子。

689

的 start 方法来启动这个进程。

在执行新的进程时，你不会看到一个终端窗口。重定向新进程的输入输出，是有可能的，但是稍微有些诡谲。Java 可以将三个标准 IO 流 (stdin, stdout 和 stderr) 重定向到你指定的 Java 流中。通常你只需要截获输出；我们将专注于此。

捕获输出诡异的地方，在于操作系统提供的缓冲可能较小。你必须迅速地管理来自每个输出流的数据。如果你没有这样做，你的应用很可能在执行时挂起。

```
package sis.util;

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

public class CommandTest extends TestCase {
    private static final String TEST_SINGLE_LINE = "testOneLine";
    private static final String TEST_MULTIPLE_LINES = "testManyLines";
    private static final String TEST_LOTS_OF_LINES = "testLotsLines";
    private static Map<String, String> output =
        new HashMap<String, String>();
    private static final String COMMAND = // 1
        "java " +
        "-classpath \"" + System.getProperty("java.class.path") + "\" " +
        "sis.util.CommandTest %s";
    private Command command;

    static { // 2
        output.put(TEST_SINGLE_LINE, "a short line of text");
        output.put(TEST_MULTIPLE_LINES, "line 1\\nline 2\\n");
        output.put(TEST_LOTS_OF_LINES, lotsOfLines());
    }

    static String lotsOfLines() {
        final int lots = 1024;
        StringBuilder lotsBuffer = new StringBuilder();
        for (int i = 0; i < lots; i++)
            lotsBuffer.append(" " + i);
        return lotsBuffer.toString();
    }

    public static void main(String[] args) { // 3
        String methodName = args[0];
        String text = output.get(methodName);
        // redirected output:
        System.out.println(text);
        System.err.println(text);
    }

    public void testSingleLine() throws Exception {
        verifyExecuteCommand(TEST_SINGLE_LINE);
    }

    public void testMultipleLines() throws Exception {
        verifyExecuteCommand(TEST_MULTIPLE_LINES);
    }

    public void testLotsOfLines() throws Exception {
        verifyExecuteCommand(TEST_LOTS_OF_LINES);
    }
}
```

690

```

    }

    private void verifyExecuteCommand(String text) throws Exception {
        command = new Command(String.format(COMMAND, text));
        command.execute();
        assertEquals(output.get(text), command.getOutput());
    }
}

```

CommandTest 将它自己作为要测试的命令。第一行构造了一个 `java` 命令行，将 `sis.util.CommandTest` 作为一个独立的应用来执行。它通过查询系统 `property` 的 `java.class.path`，得到适当的 `classpath`。`main` 方法（第三行）提供了作为独立应用的代码。

`main` 方法简单地向 `stderr` (`System.err`) 和 `stdout` (`System.out`) 输出几行。它使用一个参数，对应测试方法名的任意字符串，以决定要输出的行。输出的行来自于一个使用静态初始化段（`static initializer`）填充的 `map`。

Command 类在下面显示。在第一行中，你使用命令字符串，创建了一个 **ProcessBuilder** 对象。**ProcessBuilder** 的 `start` 方法，启动了对应的操作系统进程。第 4、6 行演示了，你如何得到对应 `stderr` 和 `stdout` 的流。在你得到这些流之后，你可以使用它们，并通过 **BufferedReader**（参见第 7 行中的 `collectOutput` 方法）来捕获输出。让所有这些都可以工作的技巧是，确保收集输出的过程是在并发线程中执行的，否则你将有阻塞或丢失输出的危险。这样，`collectErrorOutput` 中的代码（第 3 行）和 `collectOutput`（第 5 行）创建并启动新的 **Thread** 对象。在启动线程之后，`execute` 中代码会使用 **Process** 的 `waitFor` 方法（第 2 行），等待直至进程结束。

```

package sis.util;

import java.io.*;

public class Command {
    private String command;
    private Process process;
    private StringBuilder output = new StringBuilder();
    private StringBuilder errorOutput = new StringBuilder();

    public Command(String command) {
        this.command = command;
    }

    public void execute() throws Exception {
        process = new ProcessBuilder(command).start(); // 1
        collectOutput();
        collectErrorOutput();
        process.waitFor(); // 2
    }

    private void collectErrorOutput() { // 3
        Runnable runnable = new Runnable() {
            public void run() {
                try {
                    collectOutput(process.getErrorStream(), // 4

```

```

        errorOutput);
    } catch (IOException e) {
        errorOutput.append(e.getMessage());
    }
}
};
new Thread(runnable).start();
}

private void collectOutput() { // 5
    Runnable runnable = new Runnable() {
        public void run() {
            try {
                collectOutput(process.getInputStream(), // 6
                    output);
            } catch (IOException e) {
                output.append(e.getMessage());
            }
        }
    };
    new Thread(runnable).start();
}

private void collectOutput( // 7
    InputStream inputStream, StringBuilder collector)
    throws IOException {
    BufferedReader reader = null;
    try {
        reader =
            new BufferedReader(new InputStreamReader(inputStream));
        String line;
        while ((line = reader.readLine()) != null)
            collector.append(line);
    }
    finally {
        reader.close();
    }
}

public String getOutput() throws IOException {
    return output.toString();
}

public String getErrorOutput() throws IOException {
    return errorOutput.toString();
}

public String toString() {
    return command;
}
}

```

避免使用命令行的重定向(>和<)或管道(|)——它们会被认为是命令行输入字符串的一部分，可能不会被操作系统正确地处理。

ProcessBuilder 类允许你为新进程传入新的环境变量和对应的值。你还可以更改进程执行所在的工作目录。

ProcessBuilder 提供了一个 Runtime.exec 技术所不具备的能力，是将 stdout 和 stderr 流合并（merge）。终端应用常常交汇这两个流：你可以得到一个提示符，或者来自 stdout 的其他输出消息，然后是来自 stderr 的错误消息，接下去又是一个 stdout 的提示符，凡此以往。如果 stdout 和 stderr 被分隔到两个流，就无从判断原本混合输出的时间顺序。



避免使用 getEnv 和 ProcessBuilder 这样的功能，它们会引入操作系统的相关性。

还有哪些内容

Java 包括了许多更高级的课题，而大部分开发者很少会用到。还有许许多多额外的 API，你可能需要考量。本节简略地提及了其中的一些课题和 API。

自定义类加载器

当你第一次访问一个类时，Java VM 使用三个缺省类加载器中的一个，将这个类装入使用。Java 使用的第一个类加载器是自举(bootstrap)类加载器。自举类加载器在系统 JAR 文件 rt.jar 和 i18n.jar 中查找，从核心 Java API 库中找到指定的 Java 类。如果 Java 没有找到这个类，它会在 Java 的扩展目录，JRE 安装目录下的 ./lib/ext 中，进行查找。如果必要，Java 最后会查找你的 classpath 目录来加载类。

你可以创建附加的自定义类加载器。自定义加载器可以从其他地方加载类。它可能会从数据库、Web 站点、FTP 站点、或者动态创建的字节码中，加载类。

693

关于进一步的信息，请参考 Ken McCrary 撰写的文章“Create a Custom Java 1.2-Style Classloader”（创建 Java 1.2 风格的自定义类加载器），位于 <http://www.javaworld.com/javaworld/jw-03-2000/jw-03-classload.html>。对于最新的信息，可以从 java.lang. ClassLoader 的 API 文档开始。

弱引用（Weak Reference）

Java 的垃圾收集机制，用于管理归还对象占用的内存。但是如果你希望编写一个监控内存使用的应用会怎样呢？为了监控一个对象，你必须保存它的一个引用。只要你的监控应用保存这个引用，垃圾收集器就不会回收这个对象。

你可能还希望实现一种缓存（caching）机制。缓存用来载入经常被访问的对象。但是，因为缓存数据结构必须（按定义）引用被缓存的对象，垃圾回收器将永远不会认为它们需要回收。有时，你必须编写复杂的额外代码来管理从缓存中删除对象。否则，你的缓存将会持续增长，直至得到一个内存溢出的错误。

为了帮助解决缓存和其他问题，Java 引入了弱引用（weak reference）。对象的一个弱引用，不会被计入垃圾收集的考量。垃圾收集器将回收所有使用弱引用的对象，但强引用（strong reference）的对象则不会。

Java 提供了三种级别的弱引用，从最弱到最强¹⁶的顺序分别为：phantom（幻像），weak 和 soft。Phantom 引用可以用于在对象结束（finalization）之后的特殊清除处理。Weak 引用可以导致被引用的对象在垃圾收集执行时被删除。Soft 引用导致被引用的对象，只有在垃圾收集器认为这块内存的确需要时，才会被删除。

java.util.WeakHashMap 是一个使用弱引用 key 的 Map 实现。更多细节请参见其 API 文档。

关于各种弱引用类型的更多信息，请参见 java.lang.ref 包的 Java API 文档。

finalize 方法

在第 4 课中，你了解了一些关于垃圾收集的内容。Java 自动收集不再使用的对象——即不再有其他对象引用它。有时，你可能发现，在一个对象被垃圾收集时，需要做某些事情。为此，你可以考虑定义一个 finalize 方法：

694

```
protected void finalize() throws Throwable {
    try {
        // do some cleanup work
    }
    finally {
        super.finalize();
    }
}
```

垃圾收集器会调用它所收集对象的 finalize 方法。

问题是，你无法保证 finalize 最后是否会被调用。甚至在 Java VM 关闭时——当你的应用退出时——finalize 可能也不会被调用。这意味着你永远都不应该在 finalize 中编写必须要执行的代码。在 finalize 方法中关闭文件或数据库连接不是一个好主意——有可能这些资源永远不会被释放。

如果你认为你需要 finalize 方法，尝试找到一种方法来重新设计你的代码。或者考虑使用 phantom 引用作为一种替代方案。保证你总是调用父类的 finalize 方法是一个好主意（像上面代码中演示的那样）。

Instrumentation（指令改写）

使用 Java 的 instrumentation 功能，你可以向 Java 类文件中添加字节码信息。这个信息可以用来辅助日志消息、剖析（profile）方法执行的次数，或者建立代码覆盖测试的工具。它的目标

¹⁶ [JavaGloss2004b].

是，为了让你以一种不会对现有应用的功能产生更改的方式，插入这种信息。¹⁷

为了使用 instrumentation，你必须实现 `ClassFileTransformer` 接口。你需要在 `java` 命令行中使用 `-javaagent` 开关来注册其实现。当类加载器试图装入一个类时，它会调用你的 `ClassFileTransformer` 实现中的 `transform` 方法。`transform` 方法的目的是返回一个表示替换类文件的字节数组。

更多信息请参见 `java.lang.instrument` 包的 Java API 文档。

管理 (Management)

管理 (Management) API 可以让你监控和管理 Java VM。它还可以让你对 VM 执行所在的操作系统进行某些管理。管理 API 让你可以从外部的角度，完成诸如比较垃圾收集器的性能特性、判断可用的处理器数目，监控内存的使用、监控线程、或监控类的载入等工作。

◀ 695

更多信息请参见 `java.lang.management` 包的 API 文档。

网络

`java.net` 包提供了几个类来帮助你建立网络应用。这个包提供了对底层通讯的支持，基于一种被称为 socket (套接字) 的双向通讯机制的通用概念。`Socket` 可以让你访问 TCP/IP 协议。`Socket` 是大多数主机到主机通讯的基础。

`java.net` 还提供了许多围绕 Web 编程和 URL (Universal Resource Locator) 的高层 API。

关于更多信息，请参见 `java.net` 包的 API 文档。

NIO

Java NIO (“New I/O”) 是输入输出操作的一个高级的、高性能的设施。如果你需要为海量数据传输提高性能，你可以考虑 NIO。虽然 Sun 已经将 `java.io` 中的许多流用 NIO 重新实现了，你还是可以直接和 NIO API 交互以得到尽可能快的数据传输。

NIO 并不是基于流 (stream) 的。相反你使用 channel (通道)，和流类似，它们都是数据的源和管道 (sink)。缓冲包含了 channel 读取和写入的大块数据。从 NIO 获得的主要速度提升，来自与对直接缓冲的使用。通常，数据会在 Java 数组和 VM 缓冲之间拷贝。直接缓冲是直接分配在 VM 中的，可以让你的代码直接访问它们。这避免了昂贵的拷贝操作。¹⁸

NIO 是非阻塞的。通常，当你执行一个 I/O 操作时，调用的代码是阻塞的——它必须等待

¹⁷ Java 的 instrumentation API 可以认为是一种被称为面向方面编程 (aspect-oriented programming) 方法的一个例子。

¹⁸ [Travis 2002], 23 页。

696

直至操作完成。使用非阻塞 I/O，即使读写操作正在进行，你的代码依然可以继续执行。你可以使用非阻塞的 socket 通道作为 socket 服务器的基础。这提高了 socket 服务器的可伸缩性（scalability），并简化了对进入的连接和请求的管理。阻塞 I/O 需要你审慎地使用多线程，而 NIO 可以让你在一个线程中管理所有进入的请求。

概念上讲，NIO 比标准的 `java.io` 库更复杂。如果优先考虑性能的话，这个折衷（tradeoff）是值得的。就 socket 服务器的情况来说，NIO 是必须要采用的方法。对其他的大多数需求来说，标准的 I/O 库就足以应对了。

关于更多信息，参见 Sun 的 NIO 文档，位于 <http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html>。

JNI

Java Native Interface (JNI) 是你对外部世界的钩子（hook）。你可能需要同用 C 编写的硬件 API 打交道，或者你可能需要调用 Java 能力之外的、操作系统的例程（routine）。使用 JNI，你可以调用由其他语言（包括 C++ 和 C）编写的函数库（Windows 下的 DLL，和 Unix 下的 so——shared object）。

关于更多信息，Sun 的教程是一个很好的起点（但是可能有些过时）：

<http://java.sun.com/docs/books/tutorial/native/1.1/stepbystep/index.html>。

尽量使用任何可以让你同外部资源交互的 Java 技术，将 JNI 的使用降至最低，以将跨平台部署的潜力最大化。

RMI

Remote Method Invocation（远程方法调用，RMI），允许你调用在另一个 Java VM 上下文中执行的 Java 对象。另外的 VM 可以在同一台机器或远程机器上运行。RMI 是在 EJB（Enterprise Java Beans）中基于组件的分布式计算的基础。

RMI 使用了 proxy（代理）设计模式。通过其公共接口，客户端对象可以和另一台机器上的服务器方法打交道。实际上，客户端是和客户端的存根（stub）打交道。Stub 接收了 Java 的调用，并将它转换为一个可以通过网络发送的、序列化的对象。存根和服务器的骨架（skeleton）打交道，skeleton 的任务是接收这些对象流，并将它们转换为对实际服务器类的方法调用。

你可以使用 RMI 编译器 `rmic`（在 Java 的 bin 目录中），来生成 stub 和 skeleton 的 Java 源代码。

697

使用 RMI 让你的客户端不必编写远程通讯的代码。你的客户端代码现在只知道，它在与同

虚拟机中的另一个 Java 对象打交道。不过，网络的局限会天然地减慢你的应用，因此你必须在设计应用时，将 RMI 的高额开销谨记于心。分布式应用的首要设计目标是，将分布式处理的数量减至最低——万不得已，不要分布开你的对象。¹⁹

Bean

你使用 JavaBean 来建立可插入的 GUI 组件，不要同 Enterprise Java Bean (EJB) 相混淆。你可能建立一个新的 Java 信号灯 GUI 控件，希望卖给 Java 的 GUI 开发者。信号灯 JavaBean 无外乎是一个 Java 类。但是为了让开发者可以加载、以及让例如 JBuilder 这样的工具可以使用你的信号灯控件，它必须符合 JavaBean 规范。

要成为一个可视的 JavaBean，这个类必须从 `java.awt.Component` 中派生，并且必须是可序列化的。它必须使用存取方法 (accessor) 暴露其信息，这些方法要符合标准的 Java 命名约定 (例如 `getName` 返回字段 `name` 的值)。工具基于这些约定得到 Bean 中信息的这种能力，被称为自省。Bean 有一个相关的 `BeanInfo` 类，提供有关 Bean 的元数据。Bean 使用事件机制同其他 Bean 通讯。

关于更多信息，Sun 的教程是最好的起点，位于 <http://java.sun.com/docs/books/tutorial/javabeans/>。

安全

Java 中的安全，毫不费力就可以成为整本书的课题。J2SE 的安全 API，位于名字以 `java.security` 开头的众多包中，包括支持证书管理、密钥存储管理、策略文件、加密/解密算法和控制访问列表等内容的类。

Java 安全模型是基于一种 sandbox (沙箱) 的概念，一个可定制的虚拟空间。在沙箱中，Java 程序可以执行，而不会对底层的系统或用户产生不利的影响。

核心的 J2SE 安全 API，随着 Java 的每次发布而增长。关于可用的完整信息，请参见 Java 的 API 和发行文档。

J2EE

Java 2 Platform Enterprise Edition (J2EE) 是各种企业级 API 的一个集合。J2EE 可以让你创建可伸缩的基于组件的应用。

通常，当人们谈及 J2EE 时，他们会具体地认为是 Enterprise Java Bean (EJB)。EJB 是组件。

¹⁹ [Fowler 2003a], 89 页。

EJB 的目的是，消除应用开发者重新创建部署、安全、事务和持久化等服务的需要。一个 EJB 应用服务器，例如 BEA 的 WebLogic，IBM 的 WebSphere 或者 JBoss，都是一个为 EJB 提供上述服务的容器。

J2EE 非常倚重 XML 作为通用的数据语言。有 4 个 API 以 XML 为中心：Java API for XML Processing (JAXP)、Java API for XML-based RPC (remote procedure calls，远程方法调用) (JAX-RPC)、SOAP with Attachments API for Java (SAAJ) 和 Java API for XML Registries (JAXR)。

另一组 J2EE API 围绕 Web 开发。Java Servlet API 是 Web 应用的基础。JavaServer Pages (JSPs) 简化了 Web 应用的表示层。JSP Standard Tag Library (JSTL) 提供了常用的自定义标记，来简化 JSP 的编写。JavaServer Faces 为你提供了一个框架，用来简化 Web 应用中 Model 和应用方面的开发。

其他众多的 J2EE API 支持事务、资源连接和安全。Java Message Service (JMS) 提供了异步消息处理的接口。

关于 J2EE 有许多书籍。位于 <http://java.sun.com/j2ee/download.html> 的 J2EE 教程，是一个很好的起点。在你跃入超级复杂的 J2EE 世界之前，确保你已经从功能需求、开发成本、复杂性需求、性能要求和伸缩性要求等角度，对你的决定进行了论证。许多软件厂商在 J2EE 投入巨大，结果只是发现他们并不需要它所提供的大部分功能。通常你会发现，你自己的实现提供了所需的全部功能，它极其简单，而且是一个更灵活的方案。

More

还存在许多的 Java API。它们支持各种功用，例如图像处理、语音、applet、密码系统、共享数据、电话系统、辅助功能 (accessibility，用以支持残障人士)，以及自动化的 Web 安装。

Java 是一项确立的技术，绝不会在短期内消逝。证据是，事实上每一种的计算需要，都有对应的 Java API 存在。Sun 的 Java 站点，<http://java.sun.com>，是最好的起点。除此之外，还有许多珍宝散落在凡间。用 Google 找到它们！

699 ►



就到这儿吧，休息，休息一下。(Go fish!)

A

An Agile Java Glossary

敏捷 Java 的术语表

这个附录列出了关键的术语。在本书正文中出现的大多数术语，都以斜体形式显示。

每个定义都很简短，但是应该能够让你充分理解指定术语的含义。参考索引（index）来定位这些术语在本书中出现的位置。具体章节中的上下文，应该提供给你有关特定术语更坚实的理解。

abstract: 抽象 表示和具体、特定细节相反的概念

acceptance test: 验收测试 演示功能性（functional）行为的测试；也被称为功能性测试或客户测试（customer test）

access modifier: 访问修饰符 指示某个 Java 元素向其他代码暴露程度的关键字

action method: 动作方法 一个引发动作的方法，通常会更改一个对象的状态

activation: 活动 在序列图中表示一个方法生命期的 UML 结构（construct）

actual value: 实际值 你在 assertEquals 语句中要测试的表达式或值。实际值应该直接符合所预期的值

agile: 敏捷 表示轻量级过程的术语；敏捷过程核心理念是：需求会随着项目的进展而变化。

annotation: 注解 一种允许你在源代码中为 Java 元素嵌入描述性信息的结构。编译器按照接口（interface）的规格来对注解进行验证。

anonymous inner class: 匿名内联类 在一个方法体内动态定义的非命名类

API 参考 application programming interface

application: 应用 一个基于计算机，为人类或其他系统提供价值的系统

application programming interface: 应用编程接口 其他类公布的规格，你的代码可以按照它与这些类交互

argument: 参数 传入方法的一个元素（element）。对这个术语非正式的使用，可以表示传入方法调用的实际参数（argument）列表，或者表示方法特征原型中列出的形式参数（formal

parameter)。

argument list: 参数列表 传入一个方法的值或引用的列表

array initializer: 数组初始化段 为数组中的元素提供一组初始值的初始化结构

assignment: 赋值 将表达式的结果保存在一个字段 (field) 或变量的行为

assignment operator: 赋值操作符 等号 (=); 作为赋值的一部分使用

atomic: 原子的 在执行时作为一个不可分隔的单元的操作

attribute: 属性 一个一般性的面向对象术语, 表示对象的一个特性 (characteristic)。你可以为一个类定义多个 attribute

autoboxing: 自动打包 自动地将基本类型套封 (enclose) 为对应的包装 (wrapper) 类型

autounboxing: 自动解包 自动地将基本类型从它对应的包装 (wrapper) 类型中提取出来

bind: 绑定 为参数化类 (parameterized class) 提供一个类型

boolean: 布尔类型 一种表示两种可能值的类型: true 或 false

bound: 界限 为参数化类提供一个界限

block: 代码段、阻塞 【名】由大括号包围的一段代码; 【动】等待直至其他操作完成

border: 边框 在容器边界及其子组件之间的缓冲地带

byte codes: 字节码 在 class 文件中的程序操作和其他相关信息

callback: 回调 一种将函数 (function) 作为参数传递给接收者、以待接收者调用的技术

camel case: 驼峰式命名 Java 标识符首选的一种大小写混合的命名机制

cast: 强制类型转换 告诉编译器你希望转换一个引用的类型; 将一个数字值缩窄到更小的类型

catch: 捕获 捕获一个被抛出的异常

checked exception: 检查异常 一个必须由你的代码显式确认的异常

checked wrapper: 检查包装器 集合类的一种包装器 (wrapper), 确保保存在集合中的对象是适当的类型

class: 类 一个用于创建对象的模板 (template); 一个定义了行为和属性的类

class constant: 类常量 一个不管是否创建对象实例都可以访问的、固定值的引用

class diagram: 类图 一种 UML 模型, 表示类的细节、以及类与类之间的静态关系

class file: 类文件、class 文件 一个包含有源文件编译后生成代码的文件

class library: 类库 若干有用类的集合, 由某个厂商或其他开发者以 API 的形式提供

class literal: 类字面量 一个表示类的唯一实例的值

class loader: 类加载器 从代表 Java 编译单元的某个源位置 (source location), 动态地读取一

个字节流，并将之转换为一个可使用类的代码

class variable: 类变量 在一个类中定义、无论是否创建类的实例都可以访问的变量

classpath: Java 用来编译或执行的一组目录或 JAR 文件

client: 客户端 和某个特定目标类交互的一个类、或者一般意义上代码。目标类可被称为一个服务器类 (server class)。

clone: 克隆 创建某个对象的一份拷贝

close: 关闭 设计一个类，不管未来是否改变，都不需要进一步的修改

coding: 编码 编写程序的行为

collective code ownership: 集体代码所有制 一种团队哲学 (philosophy)，允许任何成员修改系统的任何部分

collision: 冲突 在一个哈希 (散列) 表中，两个元素被散列到相同的位置

compiler: 编译器 一个将源代码转换为平台 (platform) 可以理解和执行的二进制形式的应用

command object: command 对象、命令对象 一个对象，包括了稍后能引发特定行为的信息

comment: 注释 被编译器忽略的、代码中一种自由形式的注解

complex annotation: 复式注解 一个包括了其他注解的注解

compound assignment: 复合赋值 一种赋值形式的简称，它假定要赋值的变量同样也是右侧 (right-hand) 表达式第一个操作符的目标

703

composition: 组合 类之间的一种关联关系，一个类的对象是由其他类的对象组成的

concatenate: 串连 附加 (append)

connection pool: 连接池 一组共享的数据库连接

constructor: 构造函数 一段可以让你提供对象初始化语句的代码。使用 new 操作符调用一个构造函数，会导致一个对象的创建

constructor chaining: 构造函数链 从一个构造函数调用另一个构造函数

container: 容器 一个可以包含其他组件的组件

controller: 控制器 负责管理最终用户输入的一类

cooperative multitasking: 协作式多线程 一种由每个线程负责让出 (yield) 时间来让其他线程执行的多线程模式

couple: 耦合 增强类之间的依赖

covariance: 协变 子类能够改变重载方法返回类型的能力

critical section: 临界区 参见 [mutual exclusion]

customer test: 客户测试 参见 [acceptance test]

daemon thread: 守护线程 次要 (secondary) 线程; 无论守护线程是否处于活动状态, 应用都可以终止

declaration: 声明 定义类型或方法的地方

decouple: 解耦 将类之间的依赖最小化

decrement: 减量 减小一个值 (通常是 1)

decrement operation: 减量操作符 --; 一个单目操作符, 将操作数减 1

default package: 缺省包 未命名的包; 没有使用 package 语句的类, 最终都属于缺省包

dependency: 依赖 两个类 (或任意两个实体) 之间的关系, 其中一个需要另一个的存在

dependency inversion: 依赖反转 设计类的关联, 使其依赖于抽象而不是具体实现

deprecated: 不赞成使用 最终打算从类库中删除的 Java 代码

dereference: 解引用、提领 基于引用中保存的内存地址, 定位内存的位置

derived class: 导出类 参见 [subclass]

deserialize: 反串行化 从字节序列中重新构造 (reconstitute) 一个对象

design by contract: 按契约设计 一种设计技术, 要求程序员对他们的软件预期如何运转, 创建一个契约。参见 [precondition]、[postcondition] 以及 [invariant]

driver: 驱动 对 JDBC 而言, 满足 Sun 数据库连接规范的 API

dynamic proxy class: 动态代理类 可以让你在运行时实现一个或多个接口的类

element: 元素 注解能够标注的 Java 结构: 类、枚举、接口、方法、字段 (field)、参数 (parameter)、构造函数、或局部变量

encapsulation: 封装 向客户端隐藏实现细节; 面向对象的一个核心概念

erasure: 擦除法 Sun 选择用来实现参数化类型的机制; 它所指的是, 参数化类型的信息会最后被其上限 (upper bound) 擦拭掉

escape sequence: 转义序列 字符串或字符串面值中的一个字符序列, 映射到一个字符。一个转义序列以字符 \ 开始。转义序列可以让你表示无法直接键入的字符

exception: 异常 一个封装了错误信息的对象

expected value: 预期值 在 assertEquals 语句中, 你希望同实际结果 (actual result) 进行比较的值。预期值就位于实际值的前面

factory method: 工厂方法 一个负责构造和返回对象的方法

field: 字段、域 attribute 的 Java 实现。字段最低程度地为类的一个特性指定了类型和名称

- field initializer:** 字段初始化代码 用来为一个字段或局部变量提供初始值的代码表达式
- formal parameter:** 形式参数 在方法或构造函数特征原型中的命名参数
- format elements:** 格式化元素 在一个资源包 (resource bundle) 中定义的替换参数
- format specifier:** 格式指示符 格式字符串中用以值替换的占位符; 定义从值到输出字符串的转换
- frame:** 窗体 一个顶层 (top-level) 窗口
- fully qualified:** 全限定名 表明类名称中含有其完整的包信息
- garbage collection:** 垃圾收集 VM 回收不再使用内存的过程
- garbage collector:** 垃圾收集器 负责垃圾收集的代码
- generic method:** 范型方法 具有一个或多个类型参数的参数化方法
- generics:** 范型 参数化的类型
- global:** 全局 对所有感兴趣的客户端都可获得
- guard clause:** 守卫子句 定义了当异常条件发生时, 为方法定义了提早返回值 (early return) 的代码结构
- hacking:** 非正式的; 以狂热的态度创造软件。本来, hacking 是一个正面的术语; 现在对这个术语的使用 (包括在 “Agile Java” 中) 通常是有轻蔑的成分。
- hash code:** 散列码、哈希码 一个通过哈希 (散列) 算法生成的整数, 用来定位哈希表 (散列表) 中的元素
- hash table:** 哈希表、散列表 固定大小的集合, 使用散列码决定其元素位置的数据结构
- heavyweight:** 重量级 一种强调要提前编写文档、并尽力避免偏离初始计划的过程
- i18n:** 国际化 Internationalization
- identifier:** 标识符 为 Java 元素指定的名称, 例如字段、方法或类
- immutable:** 不可变的 不可更改的。如一个类的属性不能更改, 它就是不可变的。不可变的对象也被称为数值对象 (value object)
- implement:** 实现 提供代码的细节。当你实现一个接口时, 你提供了方法所声明的具体定义
- increment:** 增量 增加一个值 (通常是 1)
- increment operator ++:** 增量操作符 对操作数加 1 的单目操作符
- inherit:** 继承 从基类扩展
- inheritance:** 继承关系 两个类之间的一种关系, 其中一个类 (也被称为子类或派生类) 特化了 (specialize) 另一个类 (也被称为父类或基类) 的行为
- initial value:** 初始值 在字段或局部变量声明时具有的值

inline: 内联 在一个方法体内嵌入原本属于外部的代码; 也就是说, 通过将目标方法代码移到调用方法中, 消除了调用该方法的需要

706

instance variable: 实例变量 参见 [field]

instantiate: 实例化 创建一个对象

interface: 接口 一般地讲: 在两个事物 (例如两个系统或两个类) 之间的通讯点; 在 Java 中: 让你可以定义方法规格、而不是实现细节的结构

interface adapter class: 接口适配器类 一个类, 将外部定义的接口、转换为更适用于当前系统上下文中的接口

internationalize: 国际化 编写一个系统, 使其支持为不同地区进行部署

interpreter: 解释器 读取并执行编译代码的应用程序; 参见 [virtual machine (虚拟机)]

invariant: 不变式 当执行代码时总保持为 true 的一个断言; 参见 [design by contract (按契约设计)]

invert: 反转 上下颠倒; 在依赖反转上下文中使用

jagged array: 一个数组的数组, 每个子数组都可以改变其大小

JAR Java ARchive

Java archive: Java 归档 类的容器; 使用 ZIP 文件格式实现

layout: 布局 组件在用户界面中的排列

layout manager: 布局管理器 一个在用户界面中辅助组件排列的类

lazy initialization: 延时初始化 一种编程技术, 直到字段被第一次使用时、才提供其初始值

lightweight: 轻量级 指的是一种强调要顺应计划改变、而非忠实于计划的过程

linked list: 链表 一种串行的数据结构, 通过动态分配内存来添加新的元素。链表的元素必须通过其中保存的一个或多个引用来指向其他元素

local variable: 局部变量 在一个方法体内定义的变量

locale: 语言环境 可能出于地理、文化或政治原因定义的区域

localize: 本地化 将一个国际化应用程序转为某个特定区域的应用程序

log: 日志 记录

logical operator: 逻辑操作符 返回布尔值的操作符

lower bound: 下限 一种约束、要求通配符是指定类的父类型 (supertype)

707

marker annotation: 标记注解 没有提供参数的注解

marker interface: 标记接口 没有声明方法的接口; 允许程序员示意类的使用意图

member: 成员 在类的实例端定义的构造函数、成员变量 (field)、或方法

- member-value pair:** 成员-值对儿 在注解中提供的 key 以及对应的 value
- memory equivalence:** 内存等价性 两个对象引用相等——也就是说，它们指向内存中的同一个对象
- metadata:** 元数据 关于数据的数据
- method:** 方法 帮助定义类的行为的代码单元。返回类型、名称和参数列表，唯一地定义了一个方法。方法由若干语句组成
- methodology:** 方法学 建立软件的一种过程
- mnemonic:** 助记符 用于激活按钮的一个单字符快捷方式
- mock object:** 模拟对象 一个对象，用来创建上下文环境以帮助验证针对目标代码的断言。宽泛地讲，mock 是一个替换产品代码以辅助测试的任意对象
- model:** 模型 包括应用逻辑或业务逻辑的类
- modulus:** 模 返回除法余数的操作符
- multiline comment:** 多行注释 可以跨越多个代码行的注释；你可以使用/*来开始一个多行注释，然后使用*/来结束。
- multithreaded:** 多线程的 在同一时间执行多个代码段的能力
- mutual exclusion:** 互斥 一种同步形式；当 Java VM 执行一个代码段时，具有排它性
- naked typed variable:** 裸类型变量 在类型声明中使用的类型变量符号
- namespace:** 名字空间 名字空间隐含了定义唯一名称的能力，这样具有相同名称的实体(entity)不会彼此冲突。在 Java 中，这是通过使用包名称来达成的
- nested:** 嵌套的 被嵌入其中的
- nested class:** 嵌套类 定义在另一个类中的类
- nondeterministic:** 非确定的 在每次给定相同的输入和初始状态时、返回不同结果的潜在可能
- numeric literal:** 数字类型 数字（例如整数或浮点数）的一个代码实例
- object:** 对象 具有标识、类型、状态的实体(entity)；对象是从类创建的
- object-oriented:** 面向对象 基于类和对象的概念
- operator:** 操作符 编译器认可的一种特殊分词(token)或关键字。操作符作用于一个或多个值及表达式
- overload:** 重载 提供一个方法或构造函数的多个定义。你可以通过它们的参数列表（或者也可能通过返回值），来变换定义。参见 [covariance]
- overloaded operator:** 重载操作符 依赖于使用上下文而有不同含义的操作符
- override:** 覆写 为已经在父类中定义的方法，提供一个替代定义

package: 包 出于部署或组织类的目的, 所定义的类的集合

package import: 包引入、包汇入 引入语句的一种形式, 指示了特定包中的所有类都可以在一个类中使用

package structure: 包结构 包组织的当前状态

pair programming: 结对编程、成对编程 两个开发者在一台机器前积极编程的一种开发技术

parameter: 参数 argument 非正式的另一个名字

parameterized type: 参数化类型 和一个或多个类相绑定的类; 这样为类交互的对象类型提供了一个附加约束

persistent: 持久的 从应用的每次执行都存在

platform: 平台 应用程序可以在其上运行的底层系统

polymorphism: 多态性 对象在响应同一个消息(调用)时, 基于它们的类型, 提供不同的行为。从相反的方向看来, 是客户端向一个对象发送消息而不必知道或关心实际由哪个对象接收消息

postcondition: 后置条件 一个在代码执行后必须为 true 的断言

参见 [design by contract (按契约设计)]

postfix operator: 后缀操作符 出现在目标操作数之后的单目操作符

precondition: 前置条件 一个在代码执行前必须为 true 的断言

参见 [design by contract (按契约设计)]

preemptive multitasking: 抢占式多任务 由操作系统(或虚拟机)负责中断和调度线程的一种多线程模式

prefix operator: 前置操作符 出现在目标操作数之前的单目操作符

primitive type: 基本类型 在 Java 中非对象的类型。数字类型以及 boolean 类型是基本类型

process framework: 过程框架 按你的需要创建和定制过程裁剪的基础

process instance: 过程实例 过程框架的一种具体定制

program: 程序 构成应用的代码体; 通常可以和术语“application”互换使用

programmer test: 程序员测试 参见 [unit test]

programming: 编程 参见 [coding]

programming by intention: 按意图编程 一种编程技术, 它在对一个方案进行实现(也就是提供细节的代码)之前, 先确定你的意图(也就是所涉及的一般步骤)

project: 项目 构成工作量(effort)的有关事宜

proxy: 代理 一个替身(stand-in)。代理对象可能过滤或控制所替代对象的访问。对客户端来

说，它无法区分代理和真实对象

public interface: 公共接口 类的方法，暴露给其他的客户端类

query method: 查询方法 向客户端返回信息的方法，但是不会更改对象的状态

queue: 队列 一种串行的数据结构，当客户端从中请求一个对象时，返回最早加入的对象。这也被称为先入先出（first-in-first-out, FIFO）数据结构

random access file: 随机访问文件 文件的一种逻辑表示，可以让你快速地定位到文件的特定位置，然后从这个位置读取或写入

raw type: 原始类型 没有参数化绑定的类型

realizes: 实现

receiver: 接收者 消息发送的目标对象，直接地或间接地，以方法调用的形式

710

recursive: 递归的 调用自身；要调用的方法正是当前执行的同一方法

refactor: 重构 改变代码；按照定义，重构（refactoring）不应该改变代码的行为

refactoring: 重构 【名】代码的变换；【动】对代码进行变换的动作

reference: 引用 内存地址；指向一个对象的变量

reflective: 自省 处理反射

reflection: 反射 在 Java 中，可以让你在运行时动态地取得有关类型和定义信息的一种能力

regression test: 回归测试 确保在代码改动之后、现有应用依然可以工作的测试

regular expression: 正则表达式 用以匹配文本的一组语法元素和符号

resource bundle: 资源包 诸如消息字符串的资源集合；允许应用动态地按名字加载前述的资源。资源包通常是一个外部文件

rethrow: 重新抛出 捕获异常、紧接着抛出同一个或另一个异常对象

return type: 返回类型 指示了方法必须向调用者提供的对象类型

runtime: 运行时 指的是程序执行的范畴；也就是当它运行的时候

sandbox: 沙箱 一个可以定制的虚拟 Java 空间，为安全目的定义了边界

scale: 刻度 在一个十进制实数中，小数点后边的数字位数

seed: 种子 用来决定一个伪随机数序列的数字

semantic equality: 语义等同性 当两个对象，按照程序员提供的某种标准、被认为是相等时。在 Java 中，你使用 equals 方法来定义语义等同性

serializable: 可序列化 一个对象可以被序列化的类；是由一个程序员任意指派的

serialize: 序列化 将一个对象转换为字节序列

711

server: 服务器 一个或一组类, 为客户端提供服务

singleton: 单体 一个类, 禁止创建多于一个的实例

signature: 特征原型 唯一表示一个方法的信息: 它的返回类型、名字、以及参数类型的列表

simple design: 简单设计 一种设计方法, 注重测试、消除重复、以及代码的可表达性。简单设计规则会导致代码中正面的突发行为

single-line comment: 单行注释 以//开头的注释; 行尾字符结束这个注释

software development kit: 软件开发工具包 一套应用和实用工具, 设计用来帮助开发者建立和执行客户应用程序

source file: 源文件 一个包含了程序员键入代码的文件

spurious wakeup: 不合逻辑的唤醒 未经预料地激活一个空闲线程

stack trace: 栈回溯、栈追踪 异常产生的执行状态总结。栈回溯以逆序显示了在发生问题时的方法调用链

stack walkback: 栈回溯 参见 [stack trace]

state: 状态 对象在当前时间点的一个快照, 由其属性的值表示

statement: 语句 Java 的一行代码, 由分号结束

static import: 静态引入, 静态汇入 一种引入方式, 可以让你使用在其他类中定义的方法或变量, 就像它们是在本地定义的那样

static initialization block: 静态初始化代码段 当类第一次被虚拟机加载时、所执行的一段代码

static method: 静态方法 不需要首先初始化定义了该方法的类, 就可以调用它

static scope: 静态范围 在类的生命周期中都存在

story: 需求 的一种非正式表述; 是进一步讨论的保证

strategy: 策略 算法; 完成任务的不同方法

stream: 流 一个数据序列, 你可以从中读取或写入

string literal: 字符串类型 String 对象的一个代码实例, 由双引号括起来的文本表示

strongly typed: 强类型的 编程语言的一种特性 (characteristic), 其中变量和常量必须和一个特定的数据类型相关联

subclass: 子类 继承关系中被特化的类

subscript: 下标 表示数组中某个位置的索引

suite: 套件 单元测试的一个集合

synchronization: 同步 在同时执行的线程之间的协调, 以避免数据竞争

target: 目标 希望达到的目的; 要处理的对象; 在 Ant 中, 一个你想要执行的处理

TDDT: test-driven truth table, 测试驱动的真值表

temp variable: 临时变量 参见 [local variable]

temporary variable: 临时变量 参见 [local variable]

ternary operator: 三目操作符 可以让你在一个表达式中表示 if-else 语句的操作符

test-driven: 测试驱动 在编写相应的实现之前, 依靠编写测试规格来进行设计

test-driven truth table (TDDT): 测试驱动真值表 演示所有可能的位和逻辑操作符组合结果的一串断言

token: 分词 由特别指定的文本字符描述的单个文本元件

tolerance: 容差 当比较两个浮点数时可接受的误差幅度

thread: 线程 Java 的执行单元。一个 Java 进程可能包括多个线程, 每个线程可以和其他线程同时执行

thread pool: 线程池 一组可重用的线程

throw: 抛出 产生一个异常; 一种传输控制机制

trace statements: 追踪语句 在应用中插入的代码, 可以让你在它运行时监控其行为

two's complement: 二进制补码 在内部表示负整数的一种机制。为了得到一个数的二进制补码, 首先取得它正数的二进制表示, 将所有二进制数字反转, 然后加 1

type parameter list: 类型参数列表 在一个参数化类型中, 你必须绑定参数列表

uncaught exception handler: 未捕获异常处理器 一个钩子 (hook), 可以让你俘获 (trap) 从线程中抛出的异常

upper camel case: Java 标识符的一种命名机制: 标识符的第一个字母是大写

unary operator: 单目操作符、一元操作符 只有一个目标的操作符

unchecked exception: 非检查性异常 不需要代码显式确认的异常

unit test: 单元测试 验证目标类 (也就是一个单元) 行为的一段代码; 也被称为程序员测试 (programmer test)

upper bound: 上限 和类型参数有关的限制

user interface: 用户界面 人类同应用程序打交道所使用的方式

user thread: 应用程序执行中的主线程; 活动用户线程的存在, 支撑应用程序的执行

varargs: 可变个数的参数

view: 视图 负责为最终用户显示的类

VM: virtual machine, 虚拟机

virtual machine: 虚拟机 执行并管理客户程序的一个应用程序。虚拟机在需要时同底层的操作

系统打交道；也被称为解释器

walkback: 回溯 参见 [stack trace]

weak reference: 弱引用 一种对垃圾收集没有作用的引用

whitespace: 空白符 空格、tab（制表符）、换行、换页、以及回车符（统称空白符）

wildcard: 通配符 类型参数中的一个替换字符（?）（用于参数化类型），表示所有可能的类型

wrapper: 包装 包含另一个对象或数值的对象。在 Java 中，包装对象被用来包含基本类型，这样它们可以被当作对象来使用

B

Java Operator Precedence Rules

Java 操作符的优先规则

本节中的表格，列出了操作符的完整集合。¹在同一行中分组的操作符具有相同的优先级。表格按照从高到低的顺序排列优先级分组。与二元操作符+和-相比，单目操作符+和-同一个数字或表达式连接，以表示它是正值或负值。

后缀	<code>[] . () ++ --</code>
单目	<code>++ - + -</code>
创建，强制类型转换	<code>new (class)reference</code>
乘（除）法	<code>* / %</code>
加（减）法	<code>+ -</code>
移位	<code><< >> >>></code>
关系	<code>< > >= <= instanceof</code>
相等	<code>== !=</code>
位与	<code>&</code>
异或	<code>^</code>
位或	<code> </code>
条件与	<code>&&</code>
条件或	<code> </code>
三元（条件）	<code>?:</code>
赋值	<code>= += -= *= /= %= >>= <<= >>>= &= ^= =</code>

¹ [Arnold 2000].



Getting Started with IDEA

IDEA 入门

本附录演示了如何使用 IntelliJ IDEA 集成开发环境来建立并执行“Hello World”应用。在你尝试使用像 IDEA 这样完备的 IDE 之前，你应该首先学习如何进行命令行的编译、执行和测试 Java 代码。学习这些基本知识，将确保你可以在遇到的任何平台上使用 Java。

IDEA

IDEA 是一个叫做 JetBrains 的捷克公司建立的 Java IDE。在发展特定 Java 的 IDE 智能特性的浪潮中，IDEA 一直都是背后的驱动力之一。许多在其他 Java IDE 中支持的很有价值的功能，都是首先在 IDEA 中出现的。IDEA 也是一个开放式的工具，意味着第三方厂商或独立的开发者可以通过为它编写插件来增强其能力。

我已经使用 IDEA 有几年了，它与 Eclipse 相互交替，是我主要使用的 IDE（我的客户经常进行这样的选择）。关于 IDEA，我注意到一件事，它提升了最少惊讶（least surprise）的概念：一般来说，如果你希望做某件事，IDEA 就会完成它、并和你预期或希望的方式一样。

当我在最终期限之前编写本附录的时候，IDEA 还先于 Eclipse，非常好地支持了 J2SE 5.0 中全部的新特性。

你可以从 JetBrains 的 Web 站点 <http://www.jetbrains.com> 得到 IDEA 的最新版本。在编写本附录时，我所使用的版本是 IntelliJ IDEA 4.5.1（译注：目前最新的版本是 5.1）。按照站点上的指示下载和安装 IDEA。你应该在操心安装 IDEA 之前，已经先安装了 Java。

安装程序会询问你关于如何配置 IDEA 的若干问题。一般来说，你可以使用缺省的配置，而只管点击“Next”按钮。

下面的指导，只是一个关于如何使用 IDEA 的简略介绍，而不是一个全面的用户指导。如果你需要更多的信息，参考 IDEA 提供的联机帮助或联系 JetBrains 的支持部门。本附录中演示的指导，预计在短期内应该不会有戏剧性的变化，除非 JetBrains 有显著的改变。

Hello 项目

当你安装好 IDEA 之后，启动它。IDEA 会向你显示如图 C.1 所示的对话框。

“Hello world”项目最终将会被废弃。将 Name 文本框中的 untitled 改为 hello。当你输入项目名称时，注意指定 Project 文件位置的目录文本框也会自动地改变。你可能希望记下项目的位，这样你可以了解它在文件系统的什么地方。¹ 点击 Next（下一步）按钮。你将看到如图 C.2 所示的对话框。

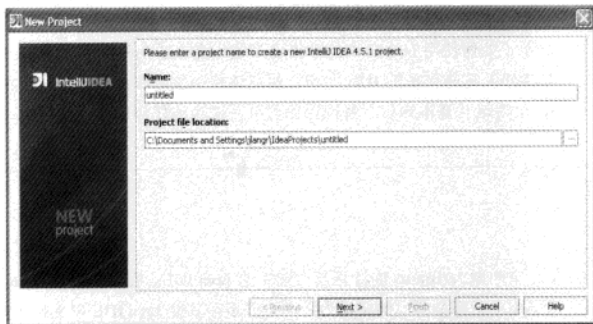


图 C.1

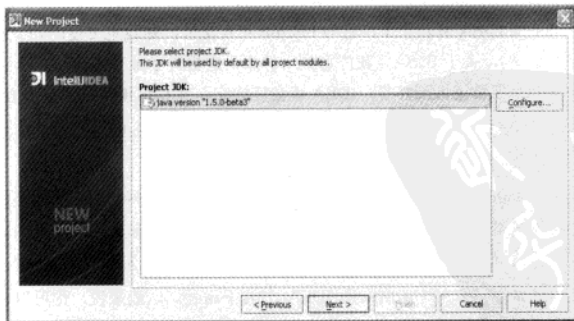


图 C.2

如果你在 Project JDK 选项框没有看到任何选项，或者并非 Java 的 1.5（5.0）或更高版本，

¹ IDEA 保存了最近打开项目的信息，因此通常你不需要关心它的位置。

点击 **Configure** (配置) 按钮。使用 **Configure JDK** (配置 JDK) 来 **Add** (添加) 新的 JDK (如有必要)。你需要知道 J2SE 5.0 的安装位置。

确保你选择了 **J2SE 5.0 JDK** 并点击 **Next**。继续点击 **Next**, 接受每个面板的缺省选项。当 IDEA 第一次将 **Finish** (完成) 按钮显示给你时, 点击它。

初始化的项目屏幕, 如图 C.3 所示, 是很简练的。一个标为 **Project** 的 tab (附签) 出现在左上角, 垂直显示。点击这个 **Tab** 来打开 **Project** 工具窗口 (参见图 C.4)。

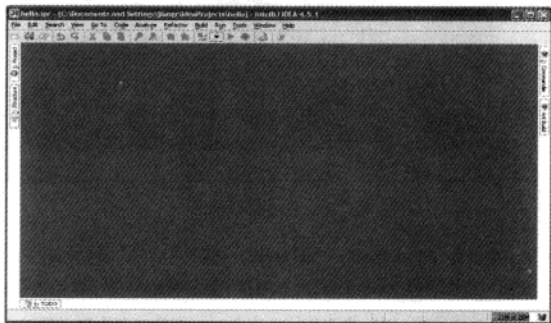


图 C.3

719

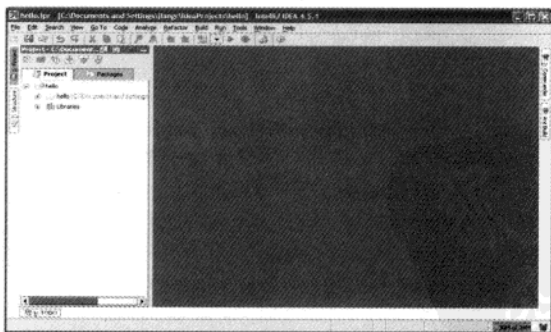


图 C.4

选择树结构顶部的节点——叫做 **hello** 的第一项 (entry)。紧接着的节点也是 **hello**, 不是模块 (module)。你所建立的简单项目, 通常只需要包括一个模块。

当你选择 **hello** 项目后, 点击鼠标右键, 弹出项目的上下文菜单 (context menu)。菜单中的第一项是 **New** (新建); 选择它。你应该看到如图 C.5 所示的屏幕。

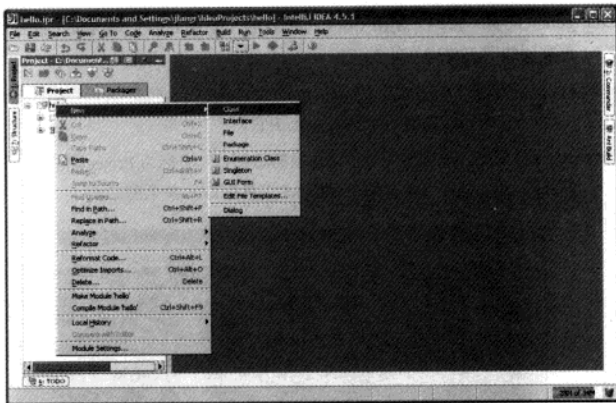


图 C.5

点击 Class 菜单项。当提示输入一个类名时，输入 **Hello**。确保你输入的类名有正确的大小写！你应该看到一个编辑器（editor），含有 Hello 类的初始定义（参见图 C.6）。

在 Hello.java 编辑器窗口中，更改代码来表现原本在第二章中展示的 Hello 代码，设置停当：

```
class Hello {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

左侧的 Project 工具窗口显示了展开后的项目层次。在 src 项之下，你可以看到新的类 **Hello** 被列出了。右键点击这个类节点，弹出上下文菜单。参见图 C.7。

点击菜单项 **Run "Hello.main()"**，图中菜单从下到上的第三项。注意你可以按下组合键 **Ctrl+Shift+F10**，作为使用菜单的替代方案。在尝试运行程序之前，你需要保存你的源文件——IDEA 会为你自动保存。

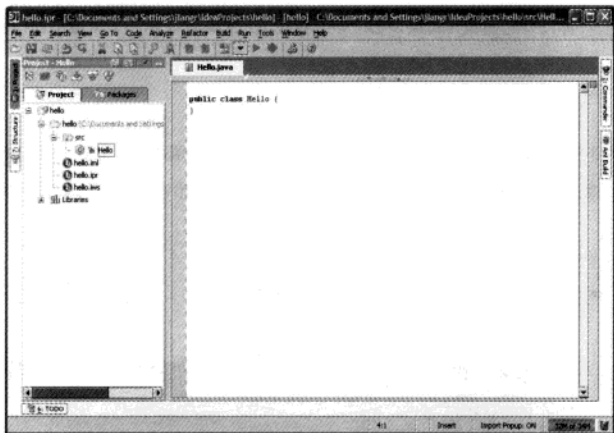


图 C.6

721

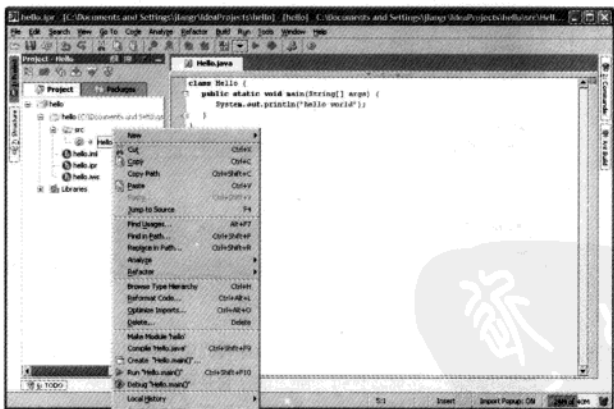


图 C.7

IDEA 可能会向你显示一个对话框，询问你是否想要创建一个输出目录。如果是，简单地点击 Yes。

一个编译进度对话框应该会短暂地显示。在 IDEA 能够执行你的程序之前，它必须编译源代码确保不会有任何问题。当你第一次编译它时，可能会多花一点时间；之后运行应该会更快。

些。²假设你没有遇到编译错误，你应该看到在 IDEA 底部显示的一个输出窗口（参见图 C.8）。

成功

接下来，你可以修改源代码来故意安插一些错误，看看 IDEA 如何处理它们。

修改 Hello.java 编辑器中的源代码，使其会产生一个错误。下面是一种可能性：

```
class Hello {
    public static void main(String[] args) {
        xxxSystem.out.println("hello world");
    }
}
```

722

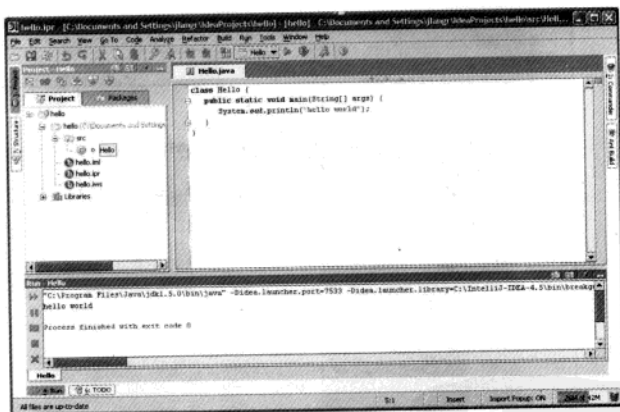


图 C.8

观察编辑器右侧，有一组可视的指示器告诉你代码中存在问题。在编辑器的右上角，出现一个小方块：它是红色的。在你将代码弄脏之前，它是绿色的。在这个方块之下，你将看到一条或更多细红色的线。这些红线指示了代码中的问题所在。你可以将鼠标悬停在某条红线上，来查看错误消息。你可以点击它来定位到问题的所在。

再次尝试运行 Hello，并查看会发生什么。因为你已经执行过 Hello 一次了，你可以使用至少两种方法来快速地再次执行它：按下 Shift-F10 组合键或者点击 Run 箭头（指向右方的一个绿色箭头，好像 CD 播放器上的播放按钮）。Run 箭头位于菜单下方和 Hello.java 编辑器上方的工具条中。

²在你每次启动 IDEA 后，初次编译会比较慢。

你会看到 Messages (消息) 工具窗口 (参见图 C.9), 而不是含有正确输出的 Run (运行) 工具窗口。这个窗口向你显示了错误消息的完整列表。左侧的各种图标帮助你管理这些消息。将你的鼠标在这些图标上悬停大约一秒钟, 可以看到解释图标用途的工具提示 (tool tip)。

如果你双击 Messages 工具窗口中的某条特定错误消息时, IDEA 会把文本光标放到 Hello.java 中出现错误的具体位置。修正这个错误并重新运行 Hello。

723

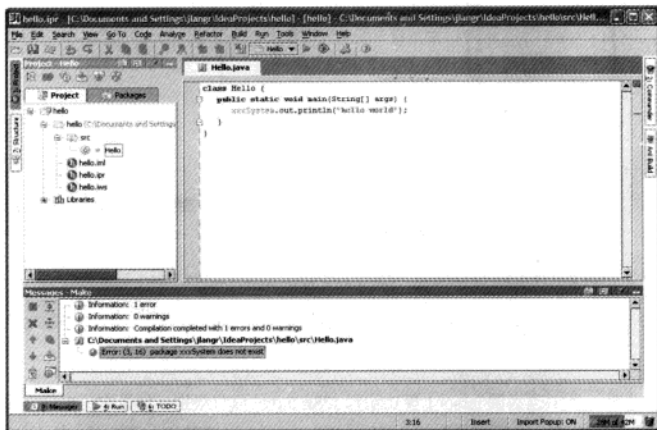


图 C.9

运行测试

本节将带领你重温第 1 课中的第一部分, 其间你创建了 `StudentTest` 类和对应的 `Student` 类。本节的目的, 为你演示如何在 IDEA 中有效地进行测试驱动开发。

首先创建一个新的项目。点击 **File** (文件) 菜单, 然后点击 **New Project** (新建项目)。IDEA 将为你显示和在创建 `hello` 项目时同样的一系列对话框。这一次, 输入项目名 `lesson1`。然后, 遵循创建 `hello` 项目时的相同步骤——连续点击 **Next**。当你点击 **Finish** 时, IDEA 给你提供了机会将项目打开在一个新的 **Frame** (窗框) 中。选择 **Yes** (是)。

在你输入代码之前, 你需要修改某些项目设置。从 **Project** 工具窗口, 选择 **Project (lesson1)**。右键点击以弹出上下文菜单, 并点击 **Module Settings** (模块设置)。

你应该看到 **Paths** (路径) 对话框, 如图 C.10 所示。

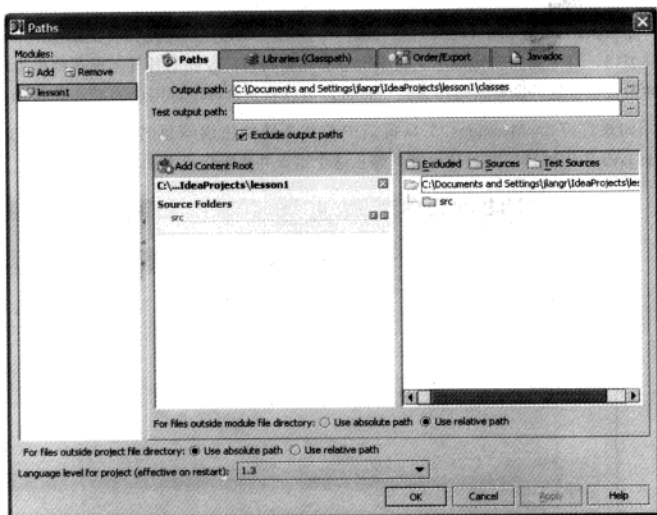


图 C.10

首先，你必须设置 lesson1 模块识别 J2SE 5.0 的语言级别（如果它还未识别的话）。在接近对话框底部，有一个标记为 Language level for project（项目的语言级别）的下拉列表（重新启动 IDEA 才会生效）。括号中的注释建议你退出 IDEA 并重新启动它来使之生效。³现在，点击该下拉列表来显示选择列表。选取表示 5.0 版本的项。

接下来，你必须指定模块于何处查找 JUnit JAR 文件。（如果你不确定其含义，参考 1 课以及第 2 课中有关设置的部分。）点击标记 Libraries（Classpath）（它出现在 Paths 对话框的上方）。你的 Paths（路径）对话框，应该看起来如图 C.11 中的屏幕截图所示。

当前的 Paths 对话框，显示了在你的 classpath 中没有任何库（JAR 文件）。点击 Add Jar/Directory（添加 Jar/目录），你将看到一个 Select Path（选择路径）对话框。定位到你安装了 JUnit 的目录，选择 junit.jar，然后点击 OK。你应该会回到 Paths 对话框。

点击 OK 来关闭 Paths 对话框。退出 IDEA，然后重启 IDEA。确保已完全退出了 IDEA——关闭 hello 项目窗框，如果它还保持打开的话。

³ 虽然某些 J2SE 5.0 的新特性可以工作，其他的则需要你重新启动 IDEA。

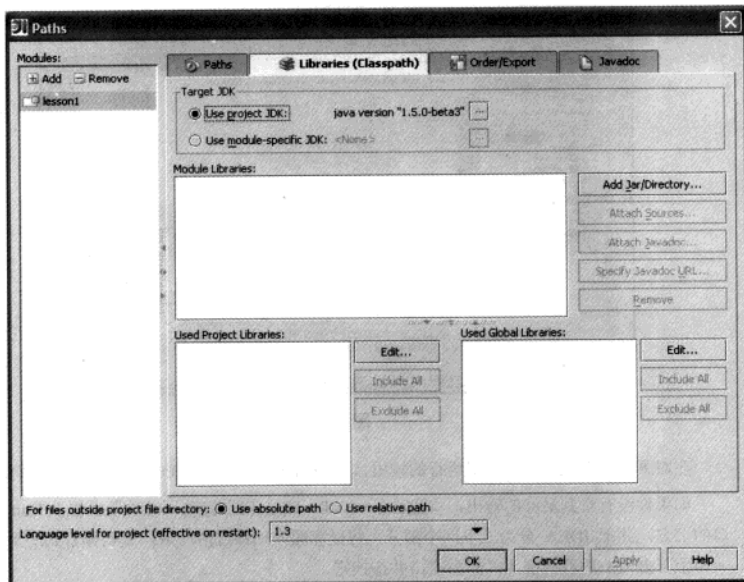


图 C.11

IDEA 可能会重新打开 hello 项目、而不是 lesson1 项目。如果是这样，选择菜单中的 File→Close Project（文件→关闭项目）。然后点击 File→Re open（文件→重新打开），然后从最近打开项目的列表中选择 lesson1 项目。你还需要定位到创建项目的目录。如果你接受了缺省设置，在 Windows 这个目录可能是 C:\Documents and Settings\userName\IdeaProjects\lesson1，其中 userName 是你的 Windows 登录名。

在你已经重新打开了 Lesson1 之后，右键点击 Project 工具窗口。选择上下文菜单中的 New→Class（新建→类）。输入 StudentTest 作为类名。在 StudentTest.java 编辑器中，为 StudentTest 类输入出自第 1 课的初始代码：

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

从 Project 工具窗口的树中，右键点击并选择 Run “StudentTest”。如果它要求，创建一个输出目录。你应该看到如图 C.12 所示的窗口。

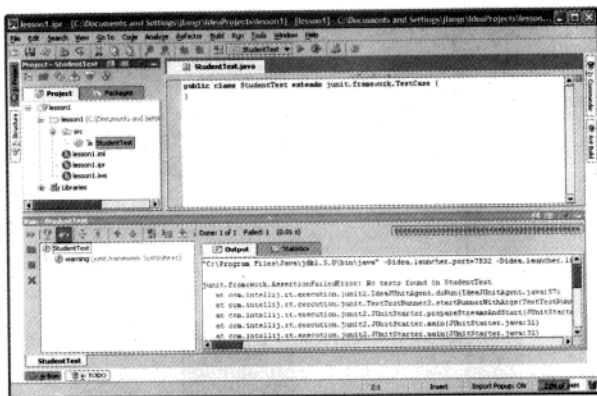


图 C.12

在右下侧的面板显示了测试运行的输出,这是JUnit的输出——IDEA已经直接集成了JUnit。

如果你没有看到足够的输出,将滑动条(slider bar)向上拽动。滑动条就位于Run工具窗口的上方,并把IDEA分为上下两个部分。将鼠标缓慢地移过滑动条,直到你看到一个南北向(即上下方向)的双向箭头,然后点击并拖拽它。

确认在你继续下一步操作时,遵循了第一课中的指导。和预期的一样,Output(输出)窗口显示一个失败,因为你还没有在StudentTest中定义任何测试。修改StudentTest的代码:

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

重新运行(Shift-F10)。现在,你应该看到一个绿条和两行输出,而不是一个红条和输出窗口中的错误。第一行显示了传递给操作系统的实际java命令。第二行说的是“进程结束且退出代码为0。”好的。你演示了失败、修改问题、然后演示了测试成功。

727

下一步,有趣的部分。再次修改StudentTest的代码:

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

你应该看到右侧的红色指示器。类名Student显示为红色。你可以将鼠标悬停在红色的位置上来得到更多信息。如果你点击类名Student,稍微延迟,你会在代码行的左侧看到一个小电灯泡。参见图C.13。

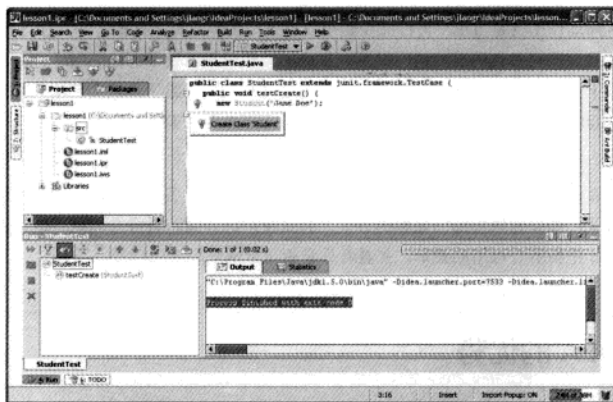


图 C.13

728

这个电灯泡表示“意向动作 (intention action)”工具。它演示了你可以采用的动作列表。作为点击电灯泡的替代方法，你可以使用 Alt-Enter 组合键来弹出动作列表。IDEA 基于它认为你想做的操作，建立这个意向动作列表。本例中，你可以选择列表中的唯一一项，Create Class “Student” (创建类 “Student”)。当要求你输入一个包名时，点击 OK。

图 C.14 展示了你现在有两个编辑区域，StudentTest.java 和 Student.java。你可以通过点击它的 tab (附签) 将其中一个调到上面。

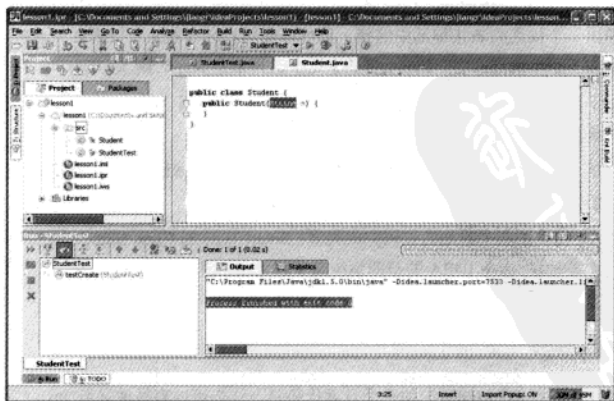


图 C.14

意向动作工具是 IDEA 为你减少必须要键入代码量的许多方式之一。你已经有一个 Student 类，并且许多有关代码已经填写好了。生成的类代码当前处于“template（模板）”模式——你可以按下 **tab** 键，在各个相关的域（Field）之间移动。这给了你机会使用具体细节来替换模板代码。Tab 定位到参数域，将参数名 `s` 替换为 `name`。再次 Tab，或按下 **Esc** 键来退出模板模式。

你应该看到不再有红色指示器了。再次运行并确保你看到了 JUnit 的绿色结果。

继续第 1 课中余下的练习。利用 IDEA 所展示给你的各种意向动作工具。体验其他有效的替代方式。如果你倾向面向鼠标的方式，尝试使用组合键，看看它们是否能够让你更快捷。反之亦然！挖掘所有可用的菜单，并看看你是否可以从它们产生动作。

利用 IDEA 的优势

IDEA，和大部分现代的 IDE 一样，有许多省时（time-saving）的特性。有必要让你学习如何掌握你的 IDE。一个只使用基本功能的新手、和充分利用 IDE 的专家之间的区别，在一个生命期为一年的项目中，可以相差出几周的时间。

729

代码完成（Code Completion）

一个有价值的功能是代码完成，我称它为自动完成（auto-completion）。当你键入 IDEA 能够识别的内容时，例如类名、包名和关键字，按下 **Ctrl-Space**。IDEA 会完成下面三件事情之一：它可能会说“No suggestions（没有建议）”，它也可能会自动地完成你开始键入的文本，或者它可能弹出一个候选列表让你从中选择。你可以使用光标（箭头）键来定位选择，并在想要的选择项上按下回车键或点击它。IDEA 将基于这个选择来完成你的输入。

按下回车键的结果是，新的文本被插入到当前位置中。如果你转而按下 **Tab** 键，新的文本将会替换直到右侧的所有文本。

尝试习惯在按下两或三个主要字符后，按下 **Ctrl-Space** 键。我总是按下它。你可能会惊奇，IDEA 能够推测出你的意图。

IDEA 提供了两个附加的代码完成类型：Smart Type 和 Class Name。还有许多更复杂的工具由不同按键触发。参考 IDEA 的帮助系统来了解更多信息（Help→Help Topics，帮助→帮助主题）。

导航

使用 IDE（例如 IDEA）的一个显著优势是，让你可以容易地导航 Java 代码。点击 Go To 菜单。这个菜单的长度，显现了这个功能是如何的灵活。点击 Go To 菜单的 Class selection（选

择类)。输入三个字符 `Stu`。稍停。你会看到一个下拉列表，其中包括可选择的适当类。

类导航是基本但重要的。更多价值在于从代码中直接导航的能力。打开 `StudentTest` 类。从 `new Student ("Jane Doe")` 的代码行，点击类名 `Student`。按下 `Ctrl-b` 组合键（我推测 `b` 代表“browse”——浏览）。按下 `Ctrl-b` 和选择菜单选项 `Go To→Declaration`（转到→声明）是等效的。

IDEA 将直接定位到 `Student` 的构造函数。你可以使用组合键 `Ctrl-Shift-Backspace` 回到上一次编辑的地方。学习所有这些导航快捷键，会为你节省可观的时间。我曾经，甚至最近，看到许多程序员，还在使用很低级的工具。他们每个小时都会浪费若干分钟，以手动地在一组类中定位，或者在较长的代码文件中滚屏。

搜索是导航的另一种形式。IDEA 理解 Java 以及它是如何组合在一起的，因此它的搜索非常有效。假定你要查找所有调用了某个特定方法的代码。使用一个编辑器（例如 `vi` 或 `ultraedit`），你只能对代码进行文本搜索。假设你想要查找所有调用了 `getId` 方法的客户端类。可能也有其他类实现了 `getId`；文本搜索也会报告使用了这些类的代码。

730

但是 IDEA 可以使用其 Java 智能，来帮助你只找出感兴趣的类使用代码。例如：在 `Student` 类中，点击构造函数。点击右键并从上下文菜单选择 `Find Usages`（查找使用）。当 `Find Usages` 对话框显现时，点击 `OK`。`Find`（查找）工具窗口将出现在 IDEA 的底部。和 `Messages` 工具窗口一样，你可以双击查找结果中的一项来直接导航到代码。

重构

另一个重要功能是内建的重构支持。IDEA 可以自动进行许多重构，允许你快速和安全地变换你的代码。查看 `Refactor`（重构）菜单中的所有菜单项，了解你都可以完成什么。

最简单且可能最有价值的重构首先大概是 `Rename`，也可以用 `Shift-F6` 触发。稍稍按几个键，你就可以安全地更改一个方法的名称。这意味着 IDEA 还会修改所有调用了这个方法的代码，来反映新的名称。在对你的代码实施重构之前，它还会向你显示建议的重命名。如果你意识到要进行的重构有问题，这可以让你回退。

我经常使用 `Rename`（重命名）重构。与其浪费许多时间来找出完美的方法或类名，我经常使用自知粗劣的名字。这可以让我继续前进。在编写一点代码之后，更好的名字会自己显现出来。快速地使用 `Shift-F6`，我就步入正轨了。有时，我可能会多次地更改一个方法名，直到满意为止。

代码分析

IDEA 为你提供了一个代码分析功能，称为代码检视器（`Analyze→Inspect Code`，分析→检

视代码)。你可以根据需要、针对一个源文件或整个项目执行代码检视。检视器查找代码中的问题地点，包括未使用的方法或变量、或者空的捕获代码块。选项列表包括超过 300 个分类的项。你可挑选感兴趣的：单独的项、一组相关的项、或者二者皆有。

执行检视会花费一点时间。当检视完成时，检视器向你显示一个展示了所有结果的树。每个结果或是问题的详细解释、或是代码中潜在的问题。从详细解释中，你可以通过点击导航到对应故障处的代码。在可能时，IDEA 会给你选项来自动地修正该问题。

IDEA 还可能会报告几打儿或上百个、你可能不以为然的问题项。这取决于你和你的团队来决定，哪些项是相干的。有些则根本不是问题。但是列表中的所有问题项，事实上都指向在某特定环境或情况下可能发生的潜在问题。

TDD 实践可以缓解绝大部分这类担忧。在 Agile Java 中，你已经学习了增量式地构建系统，而不总想要一蹴而就。你可以不理睬它们，不过在缺少良好测试时，这无疑是不可接受的。

放轻松。其他人可能认为 Agile Java 中的风格，应该是快捷和松散的。当然，我的代码总有提升的空间。但是当我采用 TDD 时，并不会过于担心它。

我认为自己遇到的大多数代码都难于理解和维护。闭着眼睛固守本位似乎是个好主意，但是它常常会让你浪费可观的时间但收获甚少。如果你转而考虑：以可测试性为目标的简单设计、消除重复、和代码可表达性，你将很少会步入歧途。



Agile Java References

- [Arnold2000] Arnold, K.; Gosling, J.; Holmes, D. *The Java Programming Language (3e)*. Sun Microsystems, 2000.
- [Astels2003] Astels, Dave. *Test-Driven Development: A Practical Guide*. Pearson Education, 2003.
- [Astels2004] Astels, Dave. "One Assertion Per Test." <http://www.artima.com/weblogs/viewpost.jsp?thread=35578>.
- [Beck1998] Beck, Kent; Gamma, Erich. "Test Infected: Programmers Love Writing Tests." <http://members.pingnet.ch/gamma/junit.htm>.
- [Bloch2001] *Effective Java Programming Language Guide*. Addison-Wesley, 2001.
- [Fowler2000] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- [Fowler2003] Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Addison-Wesley, 2003.
- [Fowler2003a] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [Gamma1995] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns*. Addison-Wesley, 1995.
- [George2002] George, Eric. "Testing Interface Compliance with Abstract Test" <http://www.placebosoft.com/abstract-test.html>.
- [Hatcher2002] Hatcher, Eric; Loughran, Steve. *Java Development with Ant*. Manning Publications Company, August 2002.
- [Heller1961] Heller, Joseph. *Catch-22*. Dell Publishing, 1961.
- [JavaGloss2004a] Green, Roedy. "Java Glossary: float." <http://mindprod.com/jgloss/float.html>.
- [JavaGloss2004b] Green, Roedy. "Java Glossary: weak references." <http://mindprod.com/jgloss/weak.html>.
- [Jeffries2001] Jeffries, R.; Anderson, A.; Hendrickson, C. *Extreme Programming Installed*. Addison-Wesley, 2001.
- [Kerievsky2004] Kerievsky, Joshua. *Refactoring to Patterns*. Addison-Wesley, 2004.
- [Langr2000] Langr, Jeff. *Essential Java Style*. Prentice Hall PTR, 2000.

- [Langr2001] Langr, Jeff. "Evolution of Test and Code Via Test-First Design." <http://www.objectmentor.com/resources/articles/tfd.pdf>.
- [Langr2003] Langr, Jeff. "Don't Mock Me." <http://www.LangrSoft.com/articles/mocking.html>.
- [Lavender1996] Lavender, R. Greg; Schmidt, Douglas C. "An Object Behavioral Pattern for Concurrent Programming." <http://citeseer.ist.psu.edu/lavender96active.html>.
- [Link2003] Link, Johannes. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann, 2003.
- [Martin2003] Martin, Robert. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.
- [Massol2004] Massol, Vincent. *JUnit in Action*. Manning Publications, 2004.
- [McBreen2000] McBreen, Pete. *Software Craftmanship*. Addison-Wesley, 2001.
- [Rainsberger2005] Rainsberger, J. B. *JUnit Recipes*. Manning Publications, 2005.
- [Sun2004] [Java] Reference Glossary. <http://java.sun.com/docs/glossary.html>.
- [Travis2002] Travis, Gregory M. *JDK 1.4 Tutorial*. Manning Publications, 2002.
- [Venners2003] Venners, Bill; Eckel, Bruce. "The Trouble with Checked Exceptions: A Conversation With Anders Hejlsberg, Part II." <http://www.artima.com/intv/handcuffs.html>.
- [Vermeulen2000] Vermeulen, Allan, et al. *The Elements of Java Style*. Cambridge University Press, 2000.
- [WhatIs2004] "platform." <http://www.whatis.com>.
- [Wiki2004] "CodeSmell." <http://c2.com/cgi/wiki?CodeSmell>.
- [Wiki2004a] "EmptyCatchClause." <http://c2.com/cgi/wiki?EmptyCatchClause>.
- [Wiki2004b] "SimpleDesign" and "XpSimplicityRules," <http://c2.com/cgi/wiki?SimpleDesign> and <http://c2.com/cgi/wiki?XpSimplicityRules>.
- [Wikipedia2004] http://en.wikipedia.org/wiki/java_programming_language.

本索引所标页码为英文版页码, 中译本边码

A

^ (exclusive-or operator), 312–313
^ (xor (exclusive-or) operator), 362
| (bit-or operator), 362, 364
| (non-short-circuited or operator), 313
|| (logical or operator), 312
~ (logical negation operator), 362, 365
! (not operator), 312
% (modulus operator), 355
& (bit-and operator), 362, 364
&& (and operator), 312
+ (add operator), 67
+ (String concatenation), 106, 344
++ (increment operator), 144–145
— (decrement operator), 144
; (statement terminator), 39
<< (bit shift left operator), 367
= (assignment operator), 42–43
== (reference comparison operator), 346
>> (bit shift right operator), 367
>>> (unsigned bit shift right operator), 367
? (wildcard character), 519
?: (ternary operator), 245–246
\n (line feed escape sequence), 105, 109
^ (bit-xor operator), 362, 365–367
^ (logical exclusive or operator), 312
@deprecated, 537–538
@Override annotation 216, 217, 325, 345, 538

@param, 99
@Retention annotation, 545
@return, 99
@Target annotation, 546–547
abbreviations, 60
abstract, 204–205
abstract classes, 204–205
abstract test pattern, 221–223
abstraction, 13, 188
AbstractTableModel
 (javax.swing.table), 637
access modifiers, 122–124, 212–213
action methods, 133
ActionListener (java.awt.event), 587
actionPerformed (java.awt.event.ActionListener method), 587
activation, 476
active object pattern, 462
actual value, 48
adapter, 435–438, 590–591
additional bounds, parameterized types, 525–526
advanced streams, 399
agile, 9–11
algorithms, hash code, 333–336
American Standard Code for Information Interchange (ASCII), 105
and operator (&&), 312
annotations, 537–538
 @Retention, 545
 @Target, 546–547
 @TestMethod, 543–545, 548
array parameters, 553–554
compatibility, 561

- annotations (*cont.*)
 - complex, 558
 - default values, 557
 - member-value pairs, 555
 - multiple parameter support, 555–556
 - package, 559–560
 - single-value, 550–552
 - types, 558–559
 - anonymous inner classes, 433–440, 471
 - Ant, 25, 41, 128–131, 527, 542–543
 - API (application programming interface), 3, 11, 426
 - documentation, 72
 - Java API Library, 368
 - application, 11
 - exceptions 276
 - application programming interface. *See* API
 - argument, 36, 38
 - argument list, 39
 - arithmetic
 - BigDecimal (java.math), 350–353
 - bit manipulation, 360–367
 - expression evaluation
 - order, 356–357
 - functions, 368–369
 - infinity, 358–359
 - integers, 354
 - NaN, 357–358
 - numeric casting, 355–356
 - numeric overflow, 359
 - random numbers, 371–374
 - ArithmeticException, 358
 - array initialization, 258–259, 262
 - ArrayList (java.util), 71–73, 189, 246, 510–511
 - arrays, 255–262, 264–266
 - annotations, 553–554
 - parameterized types, 530
 - references, 263
 - Arrays.equals, 263
 - ASCII, 105
 - asList (java.util.Arrays method), 266
 - Assert (junit.framework), 421, 535, 537
 - assert keyword, 537
 - assertEquals (junit.framework.Assert method), 46–48, 175, 324, 537
 - assertFalse (junit.framework.Assert method), 150
 - AssertionError, 536
 - assertions, 46–48, 535–536
 - failure messages, 152
 - JUnit, 537
 - assertTrue (junit.framework.Assert method), 336
 - assignment, 42–43
 - atomic variables, 501–502
 - atomic wrappers, 506
 - attributes, 15, 51, 65
 - autoboxing, 254
 - autounboxing, 255
 - avoiding numeric overflow, 359
 - await (java.util.concurrent.locks.Condition method), 493
 - AWT, 35, 566, 572
- ## B
- background threads, 474
 - base class, 203
 - base class references, 220
 - batch files, 40
 - beans. *See* JavaBeans
 - Beck, Kent, 19
 - BigDecimal (java.math), 350–353

- binary numbers, bit manipulation, 360–361
- binding parameterized types, 72
 - additional bounds, 525–526
 - arrays, 530
 - checked collections, 528, 530
 - generic methods, 522
 - limitations of, 531
 - raw types, 526–527
 - reflection, 532
 - support, 514–515
 - upper bound constraints, 516–518
 - wildcards, 518–524
- bit manipulation, 360–367
- bit shift left operator. *See* <<
- bit shift right operator. *See* >>
- bit shifting, 367–368
- bit-and operator. *See* &
- bit-or operator. *See* |
- bit-xor operator. *See* ^
- BitSet (java.util), 246, 368
- bitwise multiplication, 362
- Bloch, Joshua, 505
- BlockingQueue (java.util.concurrent), 483–484
- blocks
 - catch
 - exceptions, 271
 - rethrowing exceptions, 282–284
 - finally, 285–287
 - try-catch exceptions, 271
- Boolean, 252
- boolean expressions, 312. *See also*
 - logical operators
- boolean type, 150–154, 312, 362
- border, 608
- BorderFactory (javax.swing), 602, 609
- BorderLayout (java.awt), 597–599
- bounds
 - arrays, 530
 - parameterized types, 525–526
- BoxLayout (javax.swing), 601, 681
- branch, 172
- break statements, 197–198, 242, 243, 244–245
- bridge icon, 8, 53, 54, 85, 91, 95, 96, 112, 117, 120, 122, 134, 141, 149, 155, 168, 183, 187, 207, 234, 260, 279, 324, 330, 346, 420, 426, 447, 459, 666, 673, 693, 699
- BufferedReader (java.io), 387, 392, 394, 691
- BufferedWriter (java.io), 387, 394
- build targets, 131
- build.xml, 128–131
- byte type, 354
- Byte, 252
- byte codes, 12
- byte streams, 380
- ByteArrayInputStream, 391, 416
- ByteArrayOutputStream, 392, 394, 416

C

- C language, 2, 232, 471
- Calendar (java.util), 87–90, 681
- call by reference, 681–683
- call by value, 681–683
- callbacks, 471–472, 587
- calling superclass constructors, 211–215
- camel case, 59
- case labels, 196–198

- casting
 - numeric, 355–356
 - references, 230–231
- catch blocks, 271, 282–284
- catches, exceptions, 270, 280–282
- causes, exception, 283
- char type, 103–106, 353–354
- character, 103–105, 252
 - literals, 104–105
 - streams, 380–385
 - strings, 106. *See also* strings
 - system properties, 109–110
- checked collections, parameterized
 - types, 528–530
- checked exceptions, 273–278
- checked wrappers, 528
- chess, 62
- ChoiceFormat (java.text), 679–690
- class, 14, 73
 - constants, 84–86, 284
 - diagram, 16
 - file, 12, 27
 - library, 11–12
 - loader, 456, 693–694
 - method, 133–136, 138–141
 - variables, 137, 138–141
- Class class, 442–444, 453, 667
 - modifiers, 447
- class keyword, 33, 167
- ClassCastException, 327–328, 530, 584
- ClassNotFoundException, 401, 459
- classpath attribute, 131, 441–448
- ClassPathTestCollector (junit.runner), 441–442
- client, 83
- clone (Object method), 663–664
- Cloneable, 663, 664
- CloneNotSupportedException, 664
- cloning, 662–664
- closing a solution. *See* open-closed principle
- code smells, 94
- coding standards, 61
- Collection, 339, 482, 509–510
- Collections Framework, 73, 510
- Collections.sort, 164–166
- collective code ownership, 98
- collisions, 332–333
- command objects, 462
- comment, 96–100
- comment, multiline, 97
- comment, single-line, 96
- Comparable, 166–168, 169–171
- Comparator objects, sorting, 342
- compareTo (Comparable method), 167, 169–171
- compilation, 81
 - Ant, 128–131
- compile errors, 35, 40, 41, 45, 76, 117
- compiler, 11, 26
- compiler warnings, 91–92
- complex annotations, 558
- Component (java.awt), 574, 578, 617, 698
- ComponentOrientation (java.awt), 681
- composition, 445
- compound assignment, 144, 362
- concatenation of strings, 106–107
- Condition (java.util.concurrent.locks), 493
- conditional branching, 172. *See also* if statement
- conditional expressions, 245
- conditionals, boolean values, 153

- Connection (java.sql), 668
 - connection pool, 668
 - console, 118
 - ConsoleHandler (java.util.logging), 294, 298
 - class variables, 137–141
 - classes, 84–86
 - enum, 364–365
 - constraints, upper bound, 515–518
 - constructors, 41–42, 576
 - assertions, 46–48
 - chaining, 170
 - default, 69–70
 - enums, 209
 - exceptions, 300. *See also* exceptions
 - fields (initializing), 68–69
 - inheritance, 218–219
 - overloaded, 86–87
 - superclasses, 211–214
 - Container (java.awt), 573
 - continue statement, 243–244
 - contract, design by, 220–221
 - contract, equality, 326
 - control statements, 242
 - break statement, 243
 - continue statement, 243–244
 - labeled break statement, 244–245
 - labeled continue statement, 244–245
 - return statement, 45–46, 173, 178
 - controller, 571
 - conventions used in book, 6
 - conventions, naming, 59–60
 - cooperative multitasking, 477
 - covariance, 663
 - creation method, 146
 - creation test, 64
 - critical section, 481
 - currentThread (Thread method), 494, 502
 - custom exceptions, 276–278
- D
- daemon thread, 502
 - data streams (IO), 395–399
 - character streams, 380–385
 - management of, 379–380
 - writing to files, 385–387
 - DataInputStream (java.io), 398
 - DataOutputStream (java.io), 395–397
 - Date (java.util), 86–91
 - DateFormat (java.text), 681
 - dates, 86–91
 - deadlocks, 495
 - debugger, 22
 - debugging (toString method), 343–344
 - decode (Integer method), 371
 - decrement operator, 144
 - deep clone, 664
 - default access, 122
 - default constructor, 69–70, 218–219
 - default package, 79–80
 - default value, annotations, 557
 - DefaultListModel (javax.swing), 590
 - DefaultTableModel (javax.swing.table), 637
 - dependency, 16, 188, 426
 - dependency inversion principle, 188
 - deployment (Ant), 128–131
 - deprecated, 87–88
 - deprecation warnings, 91–92
 - design, 10, 32, 53, 115, 147
 - synchronization, 506

Design Patterns, 268, 448
do keyword, 239
do loop, 239–240, 241
doClick (javax.swing.JButton), 588
Document (javax.swing.text), 620
DocumentFilter (javax.swing.text),
620–621
domain map, 639
double type, 173–174, 177,
355–356
Double, 252, 357, 358
downloading code, 6
driver, JDBC, 665
DriverManager (java.sql), 667
duplication, 53, 85, 111, 113–118,
147, 168, 185, 195–196, 440,
673
dynamic proxy class, 448–449,
455–458

E

Eclipse, 21, 35, 717
EJBs, 3, 400, 449, 665, 697, 699
element types, 546–547
empty catch clause, 279, 301
encapsulation, 14, 83–84, 140, 272
enhanced for loop, 111
enterprise application development,
3
enum keyword, 179–181, 209–210
enumerated types, 179–181
Enumeration (java.util), 248–249
EnumMap (java.util), 199–201, 342
EnumSet (java.util), 342
environment. *See* system environ-
ment
epoch, 86
equality, 323–330
String, 345–346

equals (java.util.Arrays method),
263
equals (Object method), 324–330,
335, 346
erasure, 514–515, 531
Error, 276
escape sequence, 105
Exception, 276
exceptions, 73, 158–159, 270–287
causes, 283
checked, 273–278
finally block, 285–287
hierarchies, 275–276
logging, 290–292, 294–305
messages, 279–280
multiple, 280–282
refactoring, 287–289
rethrowing, 282–284
stack traces, 285
types, 276–278
exclusive-or (xor) operator (^),
312–313
exercises, 6, 62, 100, 131, 160
exit (System method), 503–504,
552
expected value, 48
expression evaluation order,
356–357
expressions, boolean, 150–154,
312
extending methods, 206–207
extends keyword, 33, 202, 516
extreme programming, 9

F

factory, 450
factory method, 93–94, 145–146
Feathers, Michael, 576
Fibonacci sequence, 239, 242

field, 51
 access modifiers, 122–127
 edits, 619–620
 initializers, 67
 primitive type initialization,
 159–160
 FIFO (first-in, first-out), 474
 File (java.io), 388–389
 files
 logging, 298–299
 random access, 407–410
 redirecting, 118–119
 writing to (IO), 385–387
 FileHandler (java.util.logging), 294,
 298–299, 302
 FileInputStream (java.io), 398
 FileOutputStream (java.io), 397
 FileReader (java.io), 387
 FileWriter (java.io), 387
 filter, field, 620–626
 filtered streams, 395
 final keyword, 54, 84, 438–439
 finalize method, 694–695
 finally blocks, 285–287, 399
 float type, 173–174, 355–356
 Float, 252, 357, 358
 floating point numbers. *See* floats
 floats, 173–175, 350
 FlowLayout (java.awt), 595
 flyweight pattern, 345
 for loop, 236–239, 241
 for-each loop, 111
 iterators, 249–250
 testing, 340
 formal parameter, 56–57
 format (String method), 287–289
 format specifier, 287–288
 Formatter (java.util), 679
 Formatter (java.util.logging), 299
 forName (Class method), 667
 Fowler, Martin, 16–17, 94, 322

Frame (java.awt), 610
 frame, 567
 fully qualified class name, 78

G

garbage collection, 149, 694–695
 generic methods, 522
 generics. *See* parameterized types
 getClass (Object method), 284, 328
 getEnv (System method), 689
 getProperties (System method),
 683–684
 getRuntime (Runtime method),
 689
 getter method, 65
 global, 134
 Goldeberg, Adele, 2
 green bar, 38–39
 GregorianCalendar (java.util), 86,
 95
 GridBagConstraints (java.awt),
 603–605
 GridBagLayout (java.awt), 602–605
 GridLayout (java.awt), 595–596
 guard clause, 177, 328

H

hacking, 10
 Handler (java.util.logging), 294–296
 hash code, 330–337
 hash tables, 313–319, 321–331
 algorithms, 333–336
 collisions, 332–333
 HashMap class, 337–341
 implementation, 341–343
 hashCode (Object method),
 331–337

- HashMap (java.util), 314–315, 330, 332, 333, 339, 511
- HashSet (java.util), 335, 339
- Hashtable (java.util), 246–247, 482, 683
- heavyweight process, 10
- hello world, 2, 26–27, 718
- hexadecimal, 43, 353
- Hibernate, 665
- hourglass, 647
- I
- i18n. *See* internationalization
- icon, 610
- IDE, 21–22
- IDEA. *See* IntelliJ IDEA
- identifier, 59
- IdentityHashMap (java.util), 343
- if statement, 172–173, 176–177, 311–312
- ignoring test methods, 557
- IllegalArgumentException, 277
- Image (java.awt), 610
- ImageIcon (javax.swing), 612
- immutable, 58, 107
- implementing interfaces, 167, 169–171, 184, 205
- implements keyword, 168, 202
- import statement, 78–81, 93
 - static, 141–143
- increment operator. *See* ++
- incremental refactoring, 75
- incrementing, 67–68, 144–145
- infinite loop, 239
- infinity, 358–359
- inherit, 33
- inheritance, 17–18, 74, 201–204
- initial value, 67
- initialization, 68–69, 157–160, 434, 468
- inlining code, 186
- inner classes, 412–413, 438
 - anonymous, 433–435, 440
- InputStream (java.io), 389, 391
- InputStreamReader (java.io), 389–390, 392, 394
- instance initializers, 434
- instance variable. *See* field
- instanceof operator, 328–329, 574–575, 600
- instantiate, 16
- instrumentation, 695
- int type, 65–68, 354
- Integer, 252, 359, 370
- integers
 - bit manipulation, 360–367
 - math, 354–355
- integrated development environment. *See* IDE
- IntelliJ IDEA, 21, 22, 717–732
- interface keyword, 167
- interface references, 187–188
- interfaces, 73, 166–169, 184
- InternalError, 276
- internationalization, 85, 673–681
- interpreter, 12
- interrupt (Thread method), 486
- InterruptedException, 472, 486
- inverting dependences, 188
- InvocationHandler (java.lang.reflect), 451, 456, 458
- InvocationTargetException (java.lang.reflect), 454
- invokeAndWait
 - (javax.swing.SwingUtilities method), 649
- invokeLater (javax.swing.SwingUtilities method), 649

- IO (input-output), 379
 - advanced streams, 399
 - application testing, 393–395
 - byte streams, 389
 - character streams, 380–385
 - data streams, 395–399
 - files
 - File (java.io), 388–389
 - writing to, 385–387
 - functionality, 422
 - interfaces, 390–393
 - management, 379–380
 - object streams, 399–406
 - random access files, 407–410
 - IOException (java.io), 382
 - isAlive (Thread method), 485
 - Iterable (java.util), 249–250
 - iteration, map, 337–341
 - Iterator (java.util), 247–249
 - iterator, 247–249
- J**
- J2EE, 3, 698
 - J2ME, 3
 - J2SE, 3
 - jar command, 654–655
 - JAR, 34, 653–656
 - java command, 23–24, 27–28, 655, 656
 - Java, 2, 11–13
 - API documentation, 72–73
 - language specification, 4
 - platform, 11
 - SDK, 23
 - version, 12, 23–24
 - virtual machine. *See* virtual machine
 - Java Archive. *See* JAR
 - java.awt package, 572
 - java.io package, 379–380, 389
 - java.lang.instrument package, 695
 - java.lang.management package, 696
 - java.lang.ref package, 694
 - java.net package, 696
 - java.security package, 698
 - java.util package, 73
 - java.util.concurrent package, 483
 - java.util.concurrent.atomic package, 506
 - JAVA_HOME, 23
 - JavaBeans, 698
 - javac command, 26–27, 35, 41, 91, 527
 - javadoc command, 99
 - javadoc comment, 97–100
 - JavaRanch, 29
 - JavaServer Faces, 699
 - javax.swing package, 572
 - JAX_RPC, 699
 - JAXP, 699
 - JAXR, 699
 - JButton (javax.swing), 581, 587
 - JComponent (javax.swing), 573
 - JDBC, 3, 495, 664–673
 - JDO, 665
 - Jeff's Rule of Statics, 149–150
 - Jeffries, Ron, 576
 - JetBrains, 21, 717
 - JFormattedTextField (javax.swing), 620, 626–628
 - JFrame (javax.swing), 567–570, 572, 594, 610
 - JLabel, 573
 - JList (javax.swing), 581–582, 589–590, 635
 - JLS. *See* Java language specification
 - JMS, 3, 699
 - JNI, 697

- join (Thread method), 485
- JPanel, 572–574
- JRE, 23
- JScrollPane (javax.swing), 608
- JSPs, 699
- JSTL, 699
- JTable (javax.swing), 635, 640
- TextField (javax.swing), 581, 620
- TextPane (javax.swing), 659
- Unit, 25, 33–38, 47–49, 50, 329, 336, 343, 440–446, 537, 548
- Unit setUp method, 81–82
- Unit suite, 70–71
- UnitPerf, 336
- JVM. *See* VM

- K

- Kerievsky, Joshua, 146
- KeyAdapter (java.awt.event), 616
- KeyListener (java.awt.event), 616

- L

- labeled break, 244–245
- labeled continue, 244–245
- layout managers, 594
- lazy initialization, 201
- leaks, memory, 43
- length (array field), 262
- Level (java.util.logging), 293
- lightweight process, 10
- line feed. *See* \n
- line.separator system property, 109
- LinkedBlockingQueue (java.util.concurrent), 484–486
- LinkedHashMap (java.util), 343
- LinkedHashSet (java.util), 343
- LinkedList (java.util), 474
- List (java.util), 189, 249, 510–511
- listener, 471
- ListModel (javax.swing), 590
- literals
 - boolean, 150–154
 - characters, 105
 - classes, 70, 84–86
 - numeric, 65
 - strings, 39
- local variable, 42–44
- Locale (java.util), 676, 678, 681
- locale, 673
- localization, 673, 676–678
- Lock (java.util.concurrent.locks), 492–493
- locking monitors, 481–482
- Log4J, 291
- Logger (java.util.logging), 292–293, 304
- logging, 290–305
- logging handlers, 294
- logging hierarchies, 304
- logging levels, 302–304
- logical bit operators, 361–367
- logical negation. *See* ~
- logical operators, 311–313
- long type, 354
- Long, 232
- loops
 - control statements, 242
 - break statements, 243
 - continue statements, 243–244
 - labeled break statements, 244–245
 - labeled continue statements, 244–245
- constructs, 232–233
- comparing, 240–241
- do loops, 239–240
- for loops, 236–239
- for-each loops, 249–250

while loop, 234–235
 for-each, 111, 340
 lower bounds, 523–524

M

main method, 261, 393–395, 568
 make command, 41
 MalformedURLException (java.net),
 273–274
 manifest, JAR, 655–656
 Map (java.util), 199–201, 341, 343,
 511
 map, 199–201
 Map.Entry (java.util), 340
 marker annotation, 550
 marker interfaces, 401, 663
 masks, defining, 363
 Matcher (java.util.regex), 661
 Math, 135, 368–370
 mathematics
 BigDecimal class, 350–353
 bit manipulation, 360–367
 expression evaluation order,
 356–357
 functions, 368–369
 infinity, 358–359
 integers, 354
 NaN, 357–358
 numeric casting, 355–356
 numeric overflow, 359
 primitive numerics, 353–356
 random numbers, 371–374
 wrapper classes, 370–371
 member, 140
 member-value pairs, 555
 memory leak, 43–44
 merge sort, 165
 MessageFormat (java.text), 679,
 680

Method (java.lang.reflect), 453,
 454, 458
 method, 37–38
 static, 71
 methodology, 9–11
 mixed case, 59
 mnemonic, button, 614–615
 mock objects. *See* mocking
 mocking, 372–373, 425–426,
 429–430, 504–505
 model, 571
 model-view-controller, 571
 Modifier (java.lang.reflect),
 447–448
 modifiers, class, 446–448
 modulus operator. *See* %
 monitor, 481–482
 mouse listener, 642
 multi-line comment, 97
 multidimensional arrays, 261–262
 multiple exceptions, catching,
 280–281
 multiple return statements, 173
 multiplication, bitwise, 362
 multithreading, 109, 462
 atomic variables, 501
 atomic wrappers, 506
 BlockingQueue interface,
 483–484
 cooperative/preemptive multi-
 tasking, 477
 deadlocks, 495
 groups, 505
 objects, 492–494
 priority levels, 494
 Runnable interface, 480
 shutting down, 502–504
 stopping, 485–486
 synchronization, 478–483, 506
 ThreadLocal class, 495–498
 Timer class, 499–500

mutual exclusion, 481
MySQL, 665, 667

N

naked type variable, 513–514
namespaces, 78–81
naming conventions, 59–61
NaN, 357–358
natural sort order, 169
navigable association, 16
nested class, 377, 412–413
networking, 696
new operator, 39–40, 42
NIO, 696–697
non-short-circuited logical operators, 313
not operator (!), 312
notify (Object method), 491–492
notifyAll (Object method), 489, 491
NotSerializableException, 402
NullPointerException, 82, 159, 257, 327, 357, 599
NumberFormat (java.text), 681
numbers, 66
 BigDecimal class, 350–353
 characters, 103–105
 dates, 86–91
 floating-point, 173–174
 random, 371–374
 sorting, 171–172
 strings, 370
numeric casting, 355–356
numeric literal, 65
numeric overflow, 359
numeric wrapper classes, 370–371
numerics, primitive numeric types, 353–356

O

Object, 73–74
Object Mentor, 1
object streams, 399–407
object-oriented, 2, 12, 13–17
objects, mock, 425–432
ObjectInputStream (java.io), 400, 416
ObjectOutputStream (java.io), 400, 416, 419
octal, 105, 354
one-to-many, 75, 294
open-closed principle, 182–183, 186, 450
operating system, 11
operators, 107
 and (&&), 312
 bit-and (&), 362
 bit-or (|), 362
 exclusive-or (^), 312–313
 increment, 144
 instanceof, 328
 logical, 311–313
 logical negation (~), 362
 new, 39
 not (!), 312
 or (||), 312
 postfix, 145
 prefix, 144
 ternary, 245
 xor (^), 362
or operator (||), 312
OutOfMemoryError, 276
OutputStream (java.io), 389
OutputStreamWriter (java.io), 389–390
OutputStreamWriter (java.io), 394
overflow, numeric, 359–360
overloaded constructor, 86–87

overloaded operator, 107
overriding, 216

P

package (keyword), 72, 78–81
package access, 122–123
package import, 93
package structure, 121
packages
 java.awt, 572
 java.io, 379–380, 389
 java.lang.instrument, 695
 java.lang.management, 696
 java.lang.rcf, 694
 java.net, 696
 java.security, 698
 java.util, 73
 java.util.concurrent, 483
 java.util.concurrent.atomic, 506
pair programming, 72
parameter, 38, 56–57
parameterized types, 71–72, 74–75, 509–510
 additional bounds, 525–526
 arrays, 530
 checked collections, 528–530
 Collections Framework, 510–511
 creating, 511–513
 generic methods, 542
 limitations, 531
 raw types, 526–527
 reflection, 532
 specifying, 509
 support, 514–515
 upper bound constraints, 516–518
 wildcards, 518–524

parameters
 annotations, 555–556
 arrays, 553–554
 interface references, 187–189
parity checking, 365
parseInt (Integer method), 370–371
path, 24
Pattern (java.util.regex), 657, 661–662
performance testing, 336–337
PermissionException (java.lang.reflect), 454, 457
persistence, 664
phantom references, 694
pi, 368
piped streams, 399
platform, 11
Point (java.awt), 617
polymorphism, 14, 181–186, 199, 220
postcondition, 220
postfix, 145
precedence rules, 715
preemptive multitasking, 477
Preference (java.util.prefs), 688
preferences, 685–689
prefix operator, 144–145
PreparedStatement (java.sql), 670–672
primitive types, 66
 casting, 251
 field initialization, 159–160
 wrapper classes, 252–255
printStackTrace (Throwable method), 285
PrintStream (java.io), 389, 394
PrintWriter (java.io), 387
priority levels, threads, 494
private keyword, 57–59, 122
private constructor, 135

- Process, 691
- process framework, 10
- process instance, 10
- ProcessBuilder, 689, 691–693
- processes, creating, 689
- profiling, Java, 345
- program, 11
- programmer's editor, 21–22
- programming, 11
- programming by intention, 213–214, 292
- project, Ant, 129
- Properties (java.util), 683
- properties, 667, 683
 - Ant, 130
 - system, 109–110
- property files, 685
- property, setting on command line, 684
- protected keyword, 213–215
- proxy, 448–449, 697
- public keyword, 33, 122–124, 184
- public interface, 66
- pushback streams (java.io), 399

Q

- query method, 133
- Queue (java.util), 484
- queue, 462, 474, 483–484

R

- Random (java.util), 371–374
- random access file, 407
- random numbers, 371–375
- RandomAccessFile (java.util), 415–416
- raw types, 527

- Reader (java.io), 390
- ReadWriteLock (java.util.concurrent.locks), 492
- realizes association, UML, 183
- receiver, 14, 44
- recursion, 229–230, 242
- red bar, 36–37
- ReentrantLock (java.util.concurrent.locks), 493
- refactoring, 53, 94–95. *See also* duplication
- references, 42
 - interface, 187–189
- reflection, 284, 440–459, 532, 542, 667
- regression test suite, 301
- regular expressions, 266, 656–662
- relational database, 664
- remote method invocation. *See* RMI
- replaceAll (String method), 659
- required fields, 615
- requirements. *See* stories
- resource bundle, 85, 674–676
- ResourceBundle (java.util), 674, 676, 678, 679
- ResultSet (java.sql), 670, 672
- ResultSetMetaData (java.sql), 672
- rethrowing exceptions, 282–283
- return statement, 45–46, 173, 178, 287
- return type, 38, 45
 - annotations, 558–559
- return within finally, 287
- RMI, 400, 449, 697–698
- Robot (java.awt), 617
- rounding, BigDecimal, 352
- RuleBasedCollator (java.text), 681
- run (Thread method), 473, 477, 485, 490, 504, 649
- Runnable, 473, 480, 659

Runtime, 503, 689
 RuntimeException, 276
 RUP, 9, 10

S

SAAJ, 699
 SBCS. *See* single-byte character set
 scaling (BigDecimal class), 352
 scope, static 137
 Scrum, 9
 SDK, 11, 23
 SecureRandom (java.util), 374
 sending messages, 14
 sequence diagram, 476
 SequenceInputStream (java.io), 399
 Serializable (java.io), 401, 402
 serialization, 399–407
 serialver command, 404
 serialVersionUID, 404
 Set (java.util), 339
 setDaemon (Thread method), 503
 setProperty (System method), 694
 setUp (junit.framework.TestCase method), 81–82
 shallow clone, 664
 Short, 252
 short type, 354
 short-circuited logical operators, 313
 shutting down threads, 502–504
 signal (java.util.concurrent.locks.Condition method), 493
 signalAll (java.util.concurrent.locks.Condition method), 493
 signature, 166
 simple design, 147, 577

SimpleFormatter (java.util.logging), 299–300
 Single Responsibility Principle, 60, 112, 115, 450, 463, 509–510, 525, 572
 single-byte character set, 105
 single-line comment, 96
 small steps, 5
 soft references, 694
 software development kit, 11
 sorting, 163–172, 263
 source file, 12
 special characters, 105
 split (String method), 265–266, 347, 657
 SpringLayout (javax.swing), 606
 spurious wakeup, 491
 SQL, 664–665, 668–671, 673
 SQLException, 666
 SRP. *See* Single Responsibility Principle
 Stack (java.util), 246
 stack trace, 48, 159, 285
 stack walkback, 159
 start (Thread method), 477
 state, 67, 133–134
 Statement (java.sql), 670
 statement, 39
 static import, 141–143
 static initialization block, 136
 static keyword, 109., 135, 137
 static method, 71, 127
 static nested class, 412–413
 static scope, 137
 statics, use of, 147–150
 stderr, 118, 680, 691
 stdin, 389, 690
 stdout, 118, 389, 690, 691
 stereotype, 136
 stop (Thread method), 485
 story, 33

strategy, 183–184
 stream, 379–380
 stream unique identifier. *See* serial
 VersionUID
 StreamTokenizer (java.io), 399
 StrictMath, 368
 String, 39, 168
 string concatenation, 106
 string constant, 54
 string literal, 39, 54
 StringBuffer, 109
 StringBuilder, 107–109
 strings, 103
 concatenation, 106–107
 literals, 39
 splitting, 264–266
 StringTokenizer, 264
 strongly typed, 74
 student information system, 6, 31
 student information system stories,
 32
 style
 naming conventions, 60
 whitespace, 61
 subclass, 33, 90, 202
 subcontracting, 220–228, 537
 subdirectories
 classes, 81
 creating, 121
 subscribing, array, 257
 suite, 70–71
 Sun, 28
 super keyword, 207, 212
 superclass, 203
 Swing, 3, 5, 35, 504, 565–567
 Swing layout, 594
 SwingUtilities (javax.swing), 649
 switch statement, 195–199
 synchronization, 478–480, 506
 synchronization wrappers, 482–483

synchronized keyword, 481–482,
 488–489, 502
 System, 109, 389–390, 503, 689
 system environment, 689
 system properties, 109
 System.err, 118
 System.out, 118, 119–120

T

tags. *See* annotations
 target, Ant build, 129
 TDD. *See* test-driven development
 TDDT (test-driven truth table),
 312
 tearDown (junit.framework.Test-
 Case method), 485
 temp variable. *See* local variable
 template method, 217
 temporary variable. *See* local vari-
 able
 ternary operator, 245–246
 test class, 32
 test package, 126
 test suite, 70–71
 test-driven development, 1–2, 4–5,
 11, 18–19, 31–32, 154–157,
 651–652
 test-driven development cycle, 55
 TestCase (junit.framework), 33–34,
 46–47, 421
 TestRunner (junit.awtui,
 junit.swingui, junit.textui),
 35–36
 tests as documentation, 154–155
 TextPad, 21, 656
 this keyword, 55–57, 170
 Thread, 472, 473
 thread groups, 505–506

thread pool, 491–492, 496
 thread priorities, 494
 ThreadLocal, 495–499
 throw statement, 278
 Throwable, 276
 throws clause, 274–275
 Timer (java.util), 500–501
 TimerTask (java.util), 500
 timestamps, 86–91, 95
 title bar, 609–610
 tokens, 264
 tolerance, float, 175
 toString (Object method), 107,
 343–345, 590
 trace statement, 119
 transient (keyword), 402–403,
 405
 trapping exceptions, 271
 TreeMap (java.util), 342–343
 TreeSet (java.util), 342–343
 trim (String method), 243
 try (keyword), 271
 try-catch block, 271–272
 two's complement, 361
 type parameter list, 511

U

UML, 16–17
 abstract test, 223
 activation, 476
 activity diagram, 476
 class box, 15
 class dependency, 16
 composition, 445
 inheritance, 18, 34, 204
 interface, 167, 184, 211
 message send, 14
 multiplicity, 294

 one-to-many relationship, 75
 protected methods, 217
 realizes, 183
 sequence diagram, 475–476
 stereotype, 136
 strategy pattern, 183
 UncaughtExceptionHandler, 505,
 506
 unchecked exception, 274, 276,
 278–279
 unchecked, javac lint switch, 527
 Unicode, 103–105
 Unified Modeling Language. *See*
 UML
 unit test, 18–19
 Unix, 11, 24, 40–41, 50, 109, 299,
 477, 689
 unsigned bit shift right operator.
 See >>>
 upper bounds, 515, 516–518,
 519
 upper camel case, 60
 URL (java.net), 273, 465
 URL, 273, 612, 696
 URL, database, 667
 URLConnection (java.net), 465
 user interface, 121
 user thread, 502, 504
 utility class, 136
 utility method, 134

V

varargs, 260–261
 variables
 atomic, 501
 boolean, 150–154
 classes, 137–143
 enums, 209

variables (*cont.*)
 incrementing, 67
 instance, 49–52, 187–189
 local, 42–43
 naked type, 513, 531
 numeric overflow, 359
 settings, 364
 statics
 applying, 148
 rules, 149–150
 troubleshooting, 147
 switch statements, 196
 Vector (java.util), 246–247, 482
 view, 571
 virtual machine. *See* VM
 VM, 11, 23
 void keyword, 38, 45, 68
 volatile keyword, 502

W

wait (Object method), 489, 491–492
 wait cursor, 647–648
 wait/notify, 472, 486–492
 walkback, 48
 warnings, deprecation, 91–92
 waterfall, 9
 weak references, 694
 WeakHashMap (java.util), 694

while loops, 234–235, 241
 whitespace, 61, 243
 wildcard capture, 522–523
 wildcards, 518–523
 Windows, 11, 23, 40, 50, 109, 299,
 477, 689
 worker threads, 474
 wrapper classes, 252–255
 wrapper streams, 380
 wrappers, checked, 528
 Writer (java.io), 390

X

XML, 319, 719. *See also* Ant
 XMLFormatter (java.util.logging),
 300
 xor (^) operator, 312–313, 362–365
 XP, 9, 10

Y

yield (Thread method), 475, 477

Z

ZIP file, 654

