Machine Translated by Google

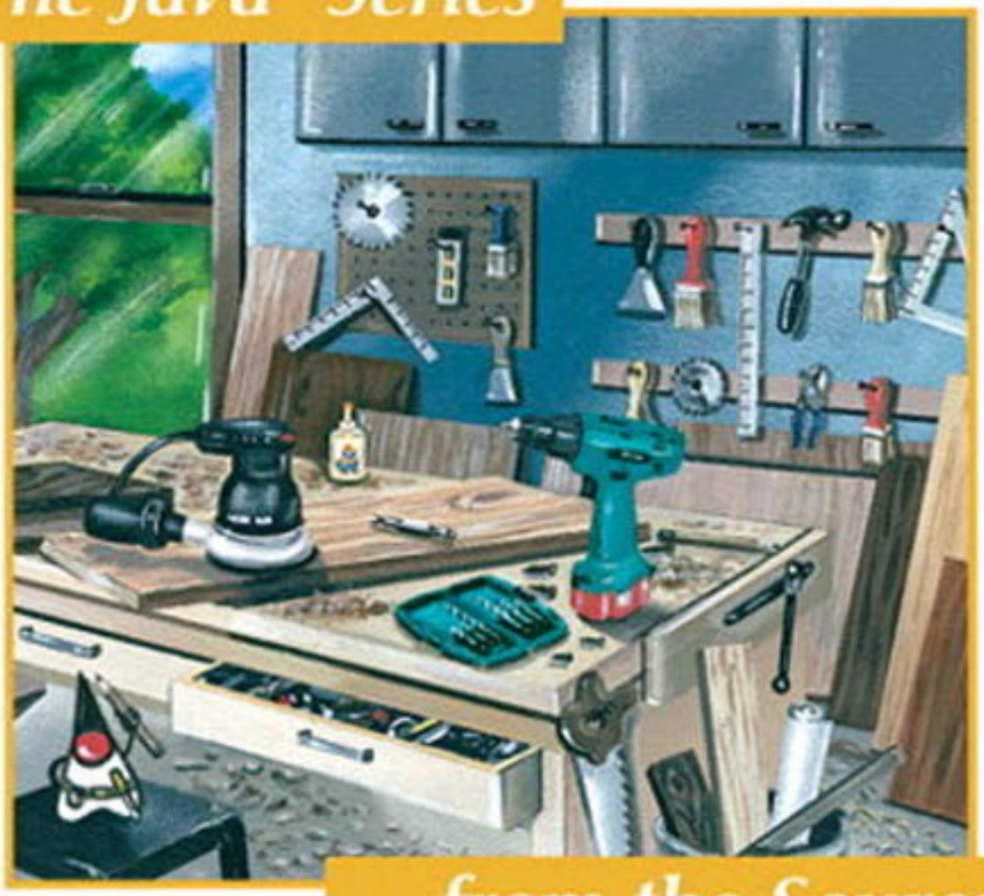
Joshua Bloch

Revised and Updated for Java SE 6

Effective Java™

Second Edition

The Java™ Series



...from the Source

Sun microsystems

Java Sun Microsystems

Machine Translated by Google

Khen ngợi cho Ấn bản đầu tiên

"Tôi chắc chắn ư ớc tôi có cuốn sách này mư ời năm trư ớc. Một số ngư ời có thể nghĩ rằng tôi không cần bất kỳ cuốn sách Java nào, như ng tôi cần cuốn sách này ".

—James Gosling, đồng nghiệp và phó chủ tịch, Sun Microsystems, Inc., và
ngư ời phát minh ra ngôn ngữ lập trình Java

"Một cuốn sách xuất sắc, chứa đầy những lời khuyên hữu ích về việc sử dụng ngôn ngữ ghép chư ơ ng trình Java và lập trình hư ớng đối tư ợng nói chung."

—Gilad Bracha, kỹ sư xuất sắc, Cadence Design Systems,
và đồng tác giả của Đặc tả ngôn ngữ Java ™,
Ấn bản thứ ba (Addison-Wesley, 2005)

"10/10 — bất kỳ ai muốn viết mã Java tốt mà ngư ời khác sẽ đánh giá cao việc đọc và duy trì đều phải sở hữu một bản sao của cuốn sách này. Đây là một trong những cuốn sách hiếm hoi mà thông tin sẽ không trở nên lỗi thời với
các bản phát hành tiếp theo của thư viện JDK. "

—Peter Tran, bartender, JavaRanch.com

"Cuốn sách Java hay nhất đư ợc viết .... Thực sự tuyệt vời; rất dễ đọc và đặc biệt hữu ích. Tôi không thể nói đủ những điều tốt đẹp về cuốn sách này. Tại JavaOne 2001, James Gosling nói, 'Hãy mua cuốn sách này!' Tôi rất vui vì tôi đã làm, và tôi không thể đồng ý hơ n ".

—Keith Edwards, thành viên cấp cao của nhân viên nghiên cứu,
Phòng thí nghiệm Khoa học Máy tính tại Trung tâm Nghiên cứu Palo Alto (PARC),
và tác giả của Core JINI (Prentice Hall, 2000)

"Đây là một cuốn sách thực sự xuất sắc đư ợc thực hiện bởi ngư ời đã thiết kế một số API nền tảng Java tốt hơ n gần đây (bao gồm cả API Bộ sư u tập)."

—James Clark, trư ởng nhóm kỹ thuật của Nhóm công tác XML
trong quá trình tạo Đề xuất XML 1.0;
ngư ời biên tập các Khuyến nghị XPath và XSLT

Machine Translated by Google

"Nội dung tuyệt vời. Tư ơ ng tự với C ++ hiệu quả cổ điển của Scott Meyers. Nếu bạn biết những kiến thức cơ bản về Java, đây phải là cuốn sách tiếp theo của bạn. "

—Gary K. Evans, cố vấn và cố vấn OO, Evanetics, Inc.

"Josh Bloch cung cấp cho bạn cái nhìn sâu sắc về các phư ơ ng pháp hay nhất thực sự chỉ có thể đư ợc khám phá sau nhiều năm nghiên cứu và trải nghiệm."

—Mark Mascolino, kỹ sư phần mềm

"Đây là một cuốn sách tuyệt vời. Nó rõ ràng bao hàm nhiều sự tinh tế của ngôn ngữ / nền tảng và những thủ thuật mà bạn cần học để trở thành một chuyên gia Java thực sự. "

—Victor Wiewiorowski, phó chủ tịch phát triển và quản lý chất lư ợng mã,
ValueCommerce Co., Tokyo, Japan

"Tôi thích những cuốn sách thiếu hứa hẹn về tiêu đề và nội dung quá tải.
Cuốn sách này có 57 mục lời khuyên lập trình đư ợc lựa chọn tốt. Mỗi mục đều cho thấy khả năng nắm bắt ngôn ngữ rõ ràng, sâu sắc. Mỗi thứ đều minh họa bằng những thuật ngữ đơ n giản, thực tế về giới hạn của việc lập trình chỉ dựa trên trực giác, hoặc đi theo con đư ờng trực tiếp nhất dẫn đến giải pháp mà không cần hiểu đầy đủ những gì ngôn ngữ đó mang lại. "

—Michael Ernest, Inkling Research, Inc.

"Tôi không tìm thấy nhiều sách lập trình khiến tôi muốn đọc từng trang—
đây là một trong chúng."

—Matt Tucker, giám đốc kỹ thuật, Jive Software

"Tài nguyên hư ớng dẫn tuyệt vời dành cho nhà phát triển có kinh nghiệm."

—John Zukowski, tác giả của nhiều sách Java

"Tôi đã chọn cuốn sách này cách đây hai tuần và có thể nói rằng tôi đã học đư ợc nhiều hơ n về ngôn ngữ Java trong ba ngày đọc so với ba tháng học! Một cuốn sách xuất sắc và là một bổ sung đáng hoan nghênh cho thư viện Java của tôi. "

—Jane Griscti, chuyên gia tư vấn I / T

Machine Translated by Google

Java ™ hiệu quả

Phiên bản thứ hai

Machine Translated by Google

Dòng Java ™

Ken Arnold, James Gosling, David Holmes
Ngôn ngữ lập trình Java ™, Ấn bản thứ tư

Joshua Bloch
Hư ớng dẫn ngôn ngữ lập trình Java ™ hiệu quả

Joshua Bloch
Java, ™ phiên bản thứ hai hiệu quả

Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock
Hư ớng dẫn sử dụng J2EE ™, Phiên bản thứ hai

Mary Campione, Kathy Walrath, Alison Huml
Hư ớng dẫn Java ™, Ấn bản thứ ba: Một khóa học ngắn về kiến thức cơ bản

Mary Campione, Kathy Walrath, Alison Huml, The Nhóm hư ớng dẫn
Hư ớng dẫn Java ™ Tiếp theo: Phần còn lại của JDK ™

Patrick Chan
Nhật ký nhà phát triển Java ™ 1.4, Tập 1

Patrick Chan
Nhật ký các nhà phát triển Java ™ 1.4, Tập 2

Patrick Chan, Rosanna Lee
Thư viện lớp Java ™, Phiên bản thứ hai, Tập 2: java.applet, java.awt, java.beans

Patrick Chan, Rosanna Lee, Doug Kramer Thư viện lớp Java ™ , Phiên bản thứ hai, Tập 1: Phần bổ sung cho Nền tảng Java ™ 2, Phiên bản tiêu chuẩn, v1.2

Kirk Chen, Li Gong
Lập trình các cổng dịch vụ mở với Java ™ Máy chủ nhúng

Zhiqun Chen
Công nghệ Java Card ™ dành cho Thẻ thông minh: Hư ớng dẫn về Kiến trúc và Lập trình viên

Maydene Fisher, Jon Ellis, Jonathan Bruce
Tài liệu tham khảo và hư ớng dẫn API JDBC ™, Ấn bản thứ ba

Eric Freeman, Susanne Hupfer, Ken Arnold
Nguyên tắc, Mẫu và Thực hành JavaSpaces ™

Li Gong, Gary Ellison, Mary Dageforde
Inside Java ™ 2 Platform Security, Second Edition: Kiến trúc, Thiết kế API và Triển khai

James Gosling, Bill Joy, Guy Steele, Gilad Bracha
Đặc tả ngôn ngữ Java ™, Phiên bản thứ ba

Chet Haase, Romain Guy
Khách hàng giàu có bẩn thỉu: Phát triển hoạt hình và đồ họa Hiệu ứng cho các ứng dụng Java ™ trên máy tính để bàn

Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, Kim Haase
Hư ớng dẫn và Tham khảo API Dịch vụ Tin nhắn Java ™: Nhắn tin cho Nền tảng J2EE ™

Eric Jendrock, Jennifer Ball
Java ™ Hư ớng dẫn EE 5, Ấn bản thứ ba

Jonni Kanerva
Câu hỏi thư ờng gặp về Java ™

Jonathan Knudsen
Kicking Butt với MIDP và MSA: Tạo ra điều tuyệt vời Ứng dụng di động

David Lambert
Bán hàng thông minh hơ n: Các chiến lư ợc bán hàng tư vấn cần đáp ứng Nhu cầu của ngư ời mua của bạn mọi lúc

Doug Lea
Lập trình đồng thời trong Java ™, Phiên bản thứ hai: Nguyên tắc và Mẫu thiết kế

Rosanna Lee, Scott Seligman
Hư ớng dẫn và Tham khảo API JNDI: Thư mục Tòa nhà Các ứng dụng Java ™ đã bật

Sheng Liang
Giao diện gốc Java ™: Hư ớng dẫn dành cho ngư ời lập trình và Sự chỉ rõ

Tim Lindholm, Frank Yellin
Đặc điểm kỹ thuật máy ảo Java ™, Phiên bản thứ hai

Roger Riggs, Antero Taivalsaari, Jim Van Peursem, Jyri Huopaniemi, Mark Patel, Aleksi Uotila
Lập trình thiết bị không dây với Java ™ 2 Nền tảng, Phiên bản vi mô, Phiên bản thứ hai

Rahul Sharma, Beth Stearns, Tony Ng
Kiến trúc trình kết nối J2EE ™ và Doanh nghiệp Tích hợp ứng dụng

Inderjeet Singh, Beth Stearns, Mark Johnson, Doanh nghiệp Đội
Thiết kế các ứng dụng doanh nghiệp với J2EE ™ Nền tảng, Phiên bản thứ hai

Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, Beth Stearns Thiết kế Dịch vụ Web với Nền tảng J2EE ™ 1.4: Công nghệ JAX-RPC, SOAP và XML

Kathy Walrath, Mary Campione, Alison Huml, Sharon Zakhour
Hư ớng dẫn sử dụng JFC Swing, Ấn bản thứ hai: Hư ớng dẫn về Xây dựng GUI

Steve Wilson, Jeff Kesselman
Hiệu suất nền tảng Java ™: Chiến lư ợc và chiến thuật

Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Tom Risser, Mark Hoeber
Java ™ Hư ớng dẫn, Ấn bản thứ tư : Một khóa học ngắn về kiến thức cơ bản

Machine Translated by Google

Java ™ hiệu quả

Phiên bản thứ hai

Joshua Bloch

Addison-Wesley

Sông Upper Saddle, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Thành phố Mexico

Machine Translated by Google

Nhiều ký hiệu đư ợc các nhà sản xuất và ngư ời bán sử dụng để phân biệt sản phẩm của họ đư ợc coi là nhãn hiệu. Khi các ký hiệu đó xuất hiện trong cuốn sách này và nhà xuất bản đã biết về khiếu nại về nhãn hiệu, các ký hiệu đó đã đư ợc in bằng các chữ cái viết hoa đầu tiên hoặc bằng tất cả các chữ hoa.

Sun Microsystems, Inc. có quyền sở hữu trí tuệ liên quan đến việc triển khai công nghệ đư ợc mô tả trong pub lication này. Đặc biệt, và không giới hạn, các quyền sở hữu trí tuệ này có thể bao gồm một hoặc nhiều bằng sáng chế của Hoa Kỳ, đối với bằng sáng chế cấp hoặc các ứng dụng đang chờ xử lý.

Sun, Sun Microsystems, logo Sun, J2ME, J2EE, Java Card, và tất cả các nhãn hiệu và biểu tư ợng dựa trên Sun và Java là nhãn hiệu thư ơ ng mại hoặc nhãn hiệu đã đăng ký của Sun Microsystems, Inc., tại Hoa Kỳ và các quốc gia khác. UNIX là nhãn hiệu đã đăng ký tại Hoa Kỳ và các quốc gia khác, đư ợc cấp phép độc quyền thông qua X / Open Company, Ltd. ĐIỀU KHOẢN PUBLICA NÀY ĐƯ ỢC CUNG CẤP "NGUYÊN TRẠNG" MÀ KHÔNG CÓ BẢO HÀNH BẤT KỲ HÌNH THỨC NÀO, BẤT CỨ RÕ RÀNG HOẶC NGỤ Ý, BAO GỒM ING, NHƯ NG KHÔNG GIỚI HẠN , CÁC BẢO ĐẢM NGẪU NHIÊN VỀ KHẢ NĂNG LINH HOẠT, PHÙ HỢP VỚI MỤC ĐÍCH CỤ THỂ HOẶC KHÔNG VI PHẠM. CÔNG BỐ NÀY BAO GỒM NHỮNG SAI LẦM VỀ KỸ THUẬT HOẶC CÁC LỖI TIÊU BIỂU. NHỮNG THAY ĐỔI ĐƯ ỢC ĐỊNH KỲ THÊM VÀO NHỮNG THÔNG TIN TRONG TÀI LIỆU NÀY; NHỮNG THAY ĐỔI NÀY SẼ ĐƯ ỢC Ư U ĐÃI TRONG CÁC BẢN CHỈNH SỬA MỚI CỦA XUẤT BẢN. SUN MICROSYSTEMS, INC. CÓ THỂ THỰC HIỆN CẢI TIẾN VÀ / HOẶC THAY ĐỔI TRONG (CÁC) SẢN PHẨM VÀ / HOẶC (CÁC) CHƯ Ơ NG TRÌNH ĐƯ ỢC MÔ TẢ TRONG CÔNG BỐ NÀY BẤT CỨ LÚC NÀO.

Tác giả và nhà xuất bản đã cẩn trọng trong việc chuẩn bị cuốn sách này, như ng không đư a ra bất kỳ hình thức bảo hành nào đư ợc nêu rõ hoặc ngụ ý và không chịu trách nhiệm về các sai sót hoặc thiếu sót. Không chịu trách nhiệm pháp lý đối với các thiệt hại ngẫu nhiên hoặc do hậu quả liên quan đến hoặc phát sinh từ việc sử dụng thông tin hoặc chư ơ ng trình có trong tài liệu này.

Nhà xuất bản cung cấp chiết khấu tuyệt vời cho cuốn sách này khi đư ợc đặt hàng với số lư ợng lớn để mua số lư ợng lớn hoặc bán đặc biệt, có thể bao gồm các phiên bản điện tử và / hoặc bìa tùy chỉnh và nội dung cụ thể cho doanh nghiệp của bạn, mục tiêu đào tạo, trọng tâm tiếp thị và sở thích xây dựng thư ơ ng hiệu. Để biết thêm thông tin, vui lòng liên hệ: Bộ phận Bán hàng Doanh nghiệp và Chính phủ Hoa Kỳ (800)

382-3419 Corpsales@pearsontechgroup.com

Để bán hàng bên ngoài Hoa Kỳ, vui lòng liên hệ:

International Sales international@pearsoned.com

Ghé thăm chúng tôi trên Web: Informit.com/aw

Số Kiểm soát của Thư viện Quốc hội: 2008926278 Bản quyền © 2008

Sun Microsystems, Inc.
4150 Network Circle,
Santa Clara, California 95054 Hoa Kỳ

Đã đăng ký Bản quyền. Đư ợc in tại Hoa Kỳ. Ấn phẩm này đư ợc bảo vệ bởi bản quyền và phải đư ợc nhà xuất bản lấy bản quyền trư ớc khi bị cấm sao chép, lư u trữ trong hệ thống truy xuất, hoặc chuyển giao dư ới bất kỳ hình thức nào hoặc bằng bất kỳ phư ơ ng tiện nào, điện tử, cơ khí, photocopy, ghi âm, hoặc tư ơ ng tự. Để biết thông tin về quyền, hãy viết thư tới: Pearson Education, Inc.

Phòng Quyền và Hợp đồng 501 Boylston
Street, Suite 900 Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-35668-0
ISBN-10: 0-321-35668-3

Văn bản in ở Hoa Kỳ trên giấy tái chế tại Courier ở Stoughton, Massachusetts.
Lần in đầu tiên, tháng 5 năm 2008

Machine Translated by Google

Gửi gia đình tôi: Cindy, Tim và Matt

Machine Translated by Google

Trang này cố ý để trống

Machine Translated by Google

Nội dung

Lời tựa xiii

Lời nói đầu. xv

Sự nhìn nhận. xix

1. Giới thiệu 1

2 Tạo và phá hủy đối tượng. 5

Mục 1: Xem xét các phương thức nhà máy tĩnh thay vì các hàm tạo. . . 5

Mục 2: Xem xét một trình xây dựng khi đối mặt với nhiều nhà xây dựng thông số 11

Mục 3: Thực thi thuộc tính singleton với một private hàm tạo hoặc một kiểu enum. 17

Mục 4: Thực thi tính không thay đổi với một phương thức khởi tạo riêng. . . . 19

Mục 5: Tránh tạo các đối tượng không cần thiết. 20

Mục 6: Loại bỏ các tham chiếu đối tượng lỗi thời. 24

Mục 7: Tránh những kẻ cuối cùng 27. . . . . .

3 Phương pháp Chung cho Mọi Đối tượng. 33

Tiết 8: Tuân theo hợp đồng chung khi ghi trên bằng 33

Mục 9: Luôn ghi đè Mã băm khi bạn ghi đè bằng. 45

Mục 10: Luôn ghi đè lên Chuỗi 51

Tiết 11: Ghi đè phân thân một cách thận trọng. 54

Tiết 12: Xem xét thực hiện Có thể so sánh 62

Machine Translated by Google

4 Lớp và Giao diện. 0,67

Mục 13: Giảm thiểu khả năng truy cập của các lớp và thành viên. 67

Tiết 14: Trong các lớp công khai, sử dụng phương thức truy nhập, không sử dụng các trường công khai Tiết 15: Giảm thiểu khả năng 71

thay đổi. 73

Khoản 16: Cấu hướng di sản thừa kế. 81

Mục 17: Thiết kế và tài liệu để thừa kế nếu không thì cấm. . 87

Mục 18: Ưu tiên giao diện cho các lớp trừu tượng. 93

Tiết 19: Chỉ sử dụng giao diện để xác định kiểu. 98

Mục 20: Ưu tiên phân cấp lớp cho các lớp được gắn thẻ. 100

Tiết 21: Sử dụng các đối tượng chức năng để biểu diễn chiến lược. 103

Mục 22: Ủng hộ các lớp thành viên tĩnh hơn là không tĩnh. 106 . . . . . .

5 Generics. 109

Mục 23: Không sử dụng loại thô trong mã mới. 109

Mục 24: Loại bỏ các cảnh báo không được kiểm tra. 116

Tiết 25: Ưu tiên danh sách thành mảng. 119

Tiết 26: Các kiểu chung chung ủng hộ. 124

Tiết 27: Phương pháp chung ủng hộ. 129

Mục 28: Sử dụng các ký tự đại diện có giới hạn để tăng tính linh hoạt của API. 134

Mục 29: Xem xét các thùng chứa không đồng nhất an toàn loại 142 . . . . . .

6 Enums và Chú thích. 147

Mục 30: Sử dụng enum thay vì hằng số int 147

Tiết 31: Sử dụng trường cá thể thay vì thứ tự. 158

Tiết 32: Sử dụng EnumSet thay cho các trường bit. 159

Mục 33: Sử dụng EnumMap thay vì lập chỉ mục theo thứ tự. 161

Tiết 34: Mô phỏng enzim có thể mở rộng bằng giao diện. 165

Tiết 35: Ưu tiên chú thích hơn các mẫu đặt tên. 169

Mục 36: Thường xuyên sử dụng Ghi đè chú thích. 176

Tiết 37: Sử dụng các giao diện đánh dấu để xác định kiểu. 179

7 Phương pháp. 181

Mục 38: Kiểm tra các thông số về tính hợp lệ. 181

Vật phẩm 39: Tạo phó bản phòng thủ khi cần thiết. 184

Tiết 40: Phương pháp thiết kế chữ ký cẩn thận. 189

Tiết 41: Sử dụng quá tải một cách thận trọng. 191

Machine Translated by Google

Tiết 42: Sử dụng kỳ đà một cách thận trọng. 197

Mục 43: Trả về mảng hoặc bộ sư u tập trống, không phải null. 201 . . . . . .

Mục 44: Viết nhận xét doc cho tất cả các phần tử API đư ợc tiếp xúc. 203 . . .

8 Lập trình tổng quát. 209

Tiết 45: Thu nhỏ phạm vi của biến cục bộ. 209

Mục 46: Ư u tiên cho từng vòng lặp thành vòng lặp for truyền thống 212

Tiết 47: Biết và sử dụng thư viện. 215

Tiết 48: Tránh float và double nếu câu trả lời chính xác

đư ợc yêu cầu 218

Tiết 49: Ư u tiên kiểu nguyên sinh hơ n kiểu nguyên hộp. 221

Mục 50: Tránh chuỗi ở những nơ i khác thích hợp hơ n. . 224

Tiết 51: Cẩn thận với hiệu suất của nối chuỗi 227 . . . . . .

Tiết 52: Tham chiếu đến các đối tư ợng theo giao diện của chúng. 228

Mục 53: Thích giao diện để phản chiếu. 230

Tiết 54: Sử dụng các phư ơ ng pháp riêng một cách thận trọng. 233

Tiết 55: Tối ư u hóa một cách sáng suốt. 234

Tiết 56: Tuân thủ các quy ư ớc đặt tên đư ợc chấp nhận chung. 237

9 Ngoại lệ. 241

Mục 57: Chỉ sử dụng các ngoại lệ cho các điều kiện ngoại lệ 241

Mục 58: Sử dụng các ngoại lệ đã kiểm tra cho các điều kiện có thể khôi phục

và ngoại lệ thời gian chạy cho các lỗi lập trình. 244

Mục 59: Tránh sử dụng không cần thiết các ngoại lệ đã kiểm tra. 246

Mục 60: Ủng hộ việc sử dụng các ngoại lệ tiêu chuẩn. 248

Tiết 61: Bỏ ngoại lệ thích hợp với cái trừu tư ợng. 250

Mục 62: Ghi lại tất cả các trư ờng hợp ngoại lệ theo từng phư ơ ng thức. 252

Mục 63: Bao gồm thông tin nắm bắt lỗi trong thông báo chi

tiết. 254

Tiết 64: Phấn đấu về tính nguyên tử không thành công. 256

Tiết 65: Đừng bỏ qua những trư ờng hợp ngoại lệ. 258

10 Đồng thời. 259

Mục 66: Đồng bộ hóa quyền truy cập vào dữ liệu có thể thay đổi đư ợc chia sẻ. 259

Tiết 67: Tránh đồng bộ quá mức. 265

Mục 68: Ư u tiên ngư ời thực thi và nhiệm vụ cho luồng. 271

Mục 69: Ư u tiên các tiện ích đồng thời chờ và thông báo. 273

Machine Translated by Google

Tiết 70: An toàn luồng tài liệu. 278

Tiết 71: Sử dụng chức năng khởi tạo lazy một cách thận trọng. 282

Mục 72: Đừng phụ thuộc vào bộ lập lịch luồng. 286

Tiết 73: Tránh nhóm luồng. 288

11 Tuần tự hóa. 289

Tiết 74: Thực hiện Serializable một cách tư pháp. 289

Mục 75: Cân nhắc sử dụng biểu mẫu được tuần tự hóa tùy chỉnh. 295

Tiết 76: Viết các phương thức readObject một cách phòng thủ. 302

Mục 77: Ví dụ về điều khiển, ưu tiên các kiểu enum hơn là readResolve 308

Mục 78: Xem xét các proxy tuần tự hóa thay vì các phiên bản tuần tự hóa. . . . . .312. . . . . .

Phụ lục: Các vật phẩm tương ứng với phiên bản đầu tiên. 317

Người giới thiệu 321

Mục lục 327

Machine Translated by Google

Lời tựa

NẾU một đồng nghiệp đã nói với bạn, "Vợ / chồng của tôi đêm nay hôm nay chế biến bữa ăn bất thư ờng trong một ngôi nhà. Bạn sẽ tham gia? " ba điều có thể sẽ lư ớt qua tâm trí bạn: thứ ba, rằng bạn đã đư ợc mời đi ăn tối; thứ hai, tiếng Anh đó không phải là ngôn ngữ đầu tiên của liên đoàn bóng đá của bạn; và đầu tiên, rất nhiều sự bối rối.

Nếu bạn đã từng tự học ngôn ngữ thứ hai và sau đó cố gắng sử dụng nó bên ngoài lớp học, bạn biết rằng có ba điều bạn phải nắm vững: ngôn ngữ đư ợc cấu trúc như thế nào (ngữ pháp), cách đặt tên cho những thứ bạn muốn nói (từ vựng.), và những cách thông thư ờng và hiệu quả để nói những điều hàng ngày (cách sử dụng). Thư ờng chỉ có hai câu đầu tiên đư ợc nhắc đến trong lớp học, và bạn thấy ngư ời bản ngữ liên tục kìm nén tiếng cư ời của họ khi bạn cố gắng làm cho mình hiểu.

Nó cũng giống như một ngôn ngữ lập trình. Bạn cần hiểu ngôn ngữ cốt lõi: nó có phải là thuật toán, chức năng, hư ớng đối tư ợng không? Bạn cần biết từ vựng: cấu trúc dữ liệu, hoạt động và cơ sở vật chất nào đư ợc cung cấp bởi các thư viện tiêu chuẩn? Và bạn cần phải làm quen với các cách thông thư ờng và hiệu quả để cấu trúc mã của bạn. Sách về ngôn ngữ lập trình thư ờng chỉ đề cập đến hai phần đầu, hoặc chỉ thảo luận về cách sử dụng một cách sơ sài. Có lẽ đó là vì hai phần đầu dễ viết hơ n về mặt nào đó. Ngữ pháp và từ vựng là mối quan hệ thích hợp của riêng ngôn ngữ, như ng cách sử dụng là đặc trư ng của một cộng đồng sử dụng nó.

Ví dụ, ngôn ngữ lập trình Java là hư ớng đối tư ợng với kế thừa đơ n và hỗ trợ kiểu mã hóa mệnh lệnh (hư ớng câu lệnh) trong mỗi phư ơ ng thức. Các thư viện đề cập đến hỗ trợ hiển thị đồ họa, mạng, máy tính phân tán và bảo mật. Như ng làm thế nào là ngôn ngữ tốt nhất để sử dụng trong thực tế?

Có một điểm khác. Các chư ơ ng trình, không giống như các câu nói và không giống như hầu hết các sách và tạp chí, có thể bị thay đổi theo thời gian. Thông thư ờng, nó không đủ để tạo ra mã hoạt động hiệu quả và đư ợc những ngư ời khác dễ hiểu; ngư ời ta cũng phải tổ chức mã sao cho dễ sửa đổi. Có thể có mư ời cách để viết mã cho một số nhiệm vụ T. Trong mư ời cách đó, bảy cách sẽ khó hiểu, không hiệu quả hoặc khó hiểu. Trong ba mã còn lại, mã nào có nhiều khả năng tư ơ ng tự với mã cần thiết cho nhiệm vụ T ' trong bản phát hành phần mềm năm tới?

Machine Translated by Google

Có rất nhiều sách để bạn có thể học ngữ pháp của Ngôn ngữ lập trình Java, bao gồm Ngôn ngữ lập trình Java ™ của Arnold, Gosling và Holmes [Arnold05] hoặc Đặc tả ngôn ngữ Java ™ của Gos ling, Joy, của bạn thực sự, và Bracha [JLS]. Tư ơ ng tự như vậy, có hàng tá sách về các thư viện và API đư ợc liên kết với ngôn ngữ lập trình Java.

Cuốn sách này giải quyết nhu cầu thứ ba của bạn: sử dụng thông thư ờng và hiệu quả. Joshua Bloch đã dành nhiều năm để mở rộng, triển khai và sử dụng ngôn ngữ lập trình Java tại Sun Microsystems; anh ấy cũng đã đọc rất nhiều mã của ngư ời khác, kể cả của tôi. Tại đây, anh ấy đư a ra những lời khuyên hữu ích, đư ợc sắp xếp có hệ thống, về cách cấu trúc mã của bạn để nó hoạt động tốt, để ngư ời khác có thể hiểu nó, để các sửa đổi và cải tiến trong tư ơ ng lai ít gây đau đầu hơ n— có lẽ, thậm chí, để các chư ơ ng trình của bạn sẽ dễ chịu, thanh lịch và duyên dáng.

Chàng L. Steele Jr.
Burlington, Massachusetts
tháng 4 năm 2001

Machine Translated by Google

Lời nói đầu

Lời nói đầu của Ấn bản thứ hai

MỘT rất nhiều điều đã xảy ra với nền tảng Java kể từ khi tôi viết ấn bản đầu tiên của cuốn sách này vào năm 2001, và đã đến lúc xuất bản lần thứ hai. Tập hợp các thay đổi quan trọng nhất là việc bổ sung generic, kiểu enum, chú thích, autoboxing và vòng lặp for-each trong Java 5. Một giây cuối cùng là việc bổ sung thư viện đồng thời mới, java.util.concurrent, cũng được phát hành trong Java 5. Với Gilad Bracha, tôi đã có may mắn được lãnh đạo nhóm thiết kế các tính năng ngôn ngữ mới. Tôi cũng có may mắn được phục vụ trong nhóm thiết kế và phát triển thư viện đồng thời do Doug Lea đứng đầu.

Sự thay đổi lớn khác trong nền tảng này là việc áp dụng rộng rãi Môi trường phát triển tích hợp (IDE) hiện đại, chẳng hạn như Eclipse, IntelliJ IDEA và NetBeans, và các công cụ phân tích tĩnh, chẳng hạn như FindBugs. Mặc dù tôi không tham gia vào những nỗ lực này, nhưng tôi đã được hưởng lợi rất nhiều từ chúng và học được cách chúng ảnh hưởng đến trải nghiệm phát triển Java.

Năm 2004, tôi chuyển từ Sun sang Google, nhưng tôi vẫn tiếp tục tham gia vào việc phát triển nền tảng Java trong bốn năm qua, đóng góp vào các API đồng thời và thu thập thông qua các văn phòng tốt của Google và Quy trình cộng đồng Java. Tôi cũng rất vui khi sử dụng nền tảng Java để phát triển các thư viện để sử dụng trong Google. Bây giờ tôi biết cảm giác trở thành một người dùng là như thế nào.

Như trường hợp năm 2001 khi tôi viết ấn bản đầu tiên, mục tiêu chính của tôi là chia sẻ kinh nghiệm của tôi với bạn để bạn có thể noi gương những thành công của tôi trong khi tránh những thất bại của tôi. Tài liệu mới tiếp tục sử dụng tự do các kỳ thi trong thế giới thực từ các thư viện nền tảng Java.

Ấn bản đầu tiên đã thành công ngoài mong đợi nhất của tôi và tôi đã cố gắng hết sức để giữ đúng tinh thần của nó trong khi bao gồm tất cả các tài liệu mới cần thiết để cập nhật cuốn sách. Điều không thể tránh khỏi là cuốn sách sẽ phát triển, và nó đã phát triển, từ năm mươi bảy mục lên bảy mươi tám. Tôi không chỉ thêm hai mươi ba mục, mà còn sửa đổi kỹ lưỡng tất cả các tài liệu gốc và gỡ bỏ

Machine Translated by Google

một số mặt hàng có ngày tốt hơ n đã trôi qua. Trong phần Phụ lục, bạn có thể thấy tài liệu trong ấn bản này liên quan như thế nào với tài liệu trong ấn bản đầu tiên.

Trong Lời nói đầu của Ấn bản đầu tiên, tôi đã viết rằng ngôn ngữ lập trình Java và các thư viện của nó vô cùng có lợi cho chất lư ợng và năng suất, đồng thời là một niềm vui khi đư ợc làm việc cùng. Những thay đổi trong bản phát hành 5 và 6 đã mang lại một điều tốt đẹp và khiến nó trở nên tốt hơ n. Nền tảng bây giờ lớn hơ n nhiều so với năm 2001 và phức tạp hơ n, như ng một khi bạn học các mẫu và thành ngữ để sử dụng các tính năng mới, chúng sẽ làm cho các chư ơ ng trình của bạn tốt hơ n và cuộc sống của bạn dễ dàng hơ n. Tôi hy vọng ấn bản này ghi lại sự nhiệt tình liên tục của tôi đối với nền tảng và giúp bạn sử dụng nền tảng và các tính năng mới của nó hiệu quả và thú vị hơ n.

San Jose, California

Tháng 4 năm 2008

Lời nói đầu của Ấn bản đầu tiên

Năm 1996, tôi thu hồi cổ phần và đi về phía tây để làm việc cho JavaSoft, như ngư ời ta đã biết đến nó, bởi vì rõ ràng đó là nơ i hành động. Trong năm năm liên tục, tôi đã từng là kiến trúc sư thư viện nền tảng Java. Tôi đã thiết kế, cố vấn và duy trì nhiều thư viện và làm cố vấn cho nhiều thư viện khác. Quản lý các thư viện này khi nền tảng Java đã trư ởng thành là cơ hội chỉ có một lần trong đời. Không quá lời khi nói rằng tôi có vinh dự đư ợc làm việc với một số kỹ sư phần mềm tuyệt vời của thế hệ chúng tôi. Trong quá trình này, tôi đã học đư ợc rất nhiều về ngôn ngữ lập trình Java — ngôn ngữ nào hoạt động, điều gì không, và cách sử dụng ngôn ngữ cũng như các thư viện của nó để đạt hiệu quả tốt nhất.

Cuốn sách này là nỗ lực của tôi để chia sẻ kinh nghiệm của tôi với bạn để bạn có thể đánh giá những thành công của tôi trong khi tránh những thất bại của tôi. Tôi đã mư ợn định dạng từ C ++ hiệu quả [Meyers98] của Scott Meyers, bao gồm năm mư ơ i mục, mỗi mục chuyển tải một quy tắc cụ thể để cải thiện các chư ơ ng trình và thiết kế của bạn. Tôi thấy định dạng này có hiệu quả đặc biệt và tôi hy vọng bạn cũng vậy.

Trong nhiều trư ờng hợp, tôi đã tự do minh họa các mục bằng các ví dụ trong thế giới thực từ các thư viện nền tảng Java. Khi mô tả điều gì đó đáng lẽ có thể làm tốt hơ n, tôi cố gắng chọn mã do chính tôi viết, như ng thỉnh thoảng tôi lại chọn một thứ do đồng nghiệp viết. Tôi thành thật xin lỗi nếu, mặc dù đã cố gắng hết sức, tôi đã xúc phạm bất kỳ ai. Các ví dụ tiêu cực đư ợc trích dẫn không phải để đổ lỗi

Machine Translated by Google

như ng trên tinh thần hợp tác để tất cả chúng ta cùng có lợi từ kinh nghiệm của những ngư ời đi trư ớc.

Mặc dù cuốn sách này không chỉ nhắm mục tiêu đến các nhà phát triển của các thành phần có thể tái sử dụng, như ng chắc chắn nó sẽ đư ợc tô màu bởi kinh nghiệm viết các thành phần như vậy của tôi trong hai thập kỷ qua. Tôi tự nhiên nghĩ về các API đư ợc xuất (Giao diện lập trình ứng dụng), và tôi khuyến khích bạn làm như vậy. Ngay cả khi bạn không phát triển các thành phần có thể tái sử dụng, suy nghĩ về các thuật ngữ này có xu hư ớng cải thiện chất lư ợng của phần mềm bạn viết. Hơ n nữa, không có gì lạ khi bạn viết một compo nent có thể tái sử dụng mà không biết: Bạn viết một cái gì đó hữu ích, chia sẻ nó với bạn bè của bạn trên khắp hành lang, và chẳng bao lâu nữa bạn có nửa tá ngư ời dùng. Tại thời điểm này, bạn không còn có thể linh hoạt thay đổi API theo ý muốn nữa và cảm ơ n vì tất cả những nỗ lực mà bạn đã bỏ ra để thiết kế API khi lần đầu tiên viết phần mềm.

Việc tôi tập trung vào thiết kế API có vẻ hơ i thiếu tự nhiên đối với những ngư ời yêu thích phư ơ ng pháp luận phát triển phần mềm nhẹ mới, chẳng hạn như Lập trình cực đoan [Beck99]. Những phư ơ ng pháp luận này nhấn mạnh việc viết một chư ơ ng trình đơ n giản nhất có thể hoạt động đư ợc. Nếu bạn đang sử dụng một trong những phư ơ ng pháp này, bạn sẽ thấy rằng việc tập trung vào thiết kế API phục vụ tốt cho bạn trong quá trình tái cấu trúc . Các mục tiêu cơ bản của tái cấu trúc là cải thiện cấu trúc hệ thống và tránh trùng lặp mã. Các mục tiêu này không thể đạt đư ợc nếu không có các API đư ợc thiết kế tốt cho các thành phần của hệ thống.

Không có ngôn ngữ nào là hoàn hảo, như ng một số ngôn ngữ là xuất sắc. Tôi nhận thấy rằng ngôn ngữ lập trình Java và các thư viện của nó vô cùng có lợi cho chất lư ợng và năng suất, và tôi rất vui khi đư ợc làm việc cùng. Tôi hy vọng cuốn sách này nắm bắt đư ợc sự nhiệt tình của tôi và giúp bạn sử dụng ngôn ngữ hiệu quả và thú vị hơ n.

Cupertino, California

Tháng 4 năm 2001

Machine Translated by Google

Trang này cố ý để trống

Machine Translated by Google

Sự nhìn nhận

Lời cảm ơn cho Ấn bản thứ hai

tôi cảm ơn độc giả của ấn bản đầu tiên của cuốn sách này đã dành cho nó sự đón nhận nhiệt tình và ân cần, vì đã ghi nhớ những ý tưởng của nó, và đã cho tôi biết nó có ảnh hưởng tích cực như thế nào đối với họ và công việc của họ. Tôi cảm ơn nhiều giáo sư đã sử dụng cuốn sách trong các khóa học của họ và nhiều nhóm kỹ sư đã sử dụng nó.

Tôi cảm ơn toàn bộ đội tại Addison-Wesley vì sự tử tế, tinh thần nghề nghiệp, sự kiên nhẫn và duyên dáng trước áp lực của họ. Qua tất cả, biên tập viên Greg Doench của tôi vẫn không thể tin được: một biên tập viên giỏi và một quý ông hoàn hảo. Người phụ trách sản xuất của tôi, Julie Nahil, là tất cả những gì mà một giám đốc sản xuất cần phải có: siêng năng, nhanh chóng, có tổ chức và thân thiện. Người biên tập bản sao của tôi, Barbara Wood, rất tỉ mỉ và khéo léo.

Tôi đã một lần nữa được may mắn với đội ngũ đánh giá tốt nhất có thể tưởng tượng được và tôi gửi lời cảm ơn chân thành nhất đến từng người trong số họ. Nhóm cốt lõi, người đã xem lại mọi chương, bao gồm Lexi Baugher, Cindy Bloch, Beth Bottos, Joe Bowbeer, Brian Goetz, Tim Halloran, Brian Kernighan, Rob Konigsberg, Tim Peierls, Bill Pugh, Yoshiki Shibata, Peter Stout, Peter Weinberger, và Frank Yellin. Những người đánh giá khác bao gồm Pablo Bellver, Dan Bloch, Dan Bornstein, Kevin Bourrillion, Martin Buchholz, Joe Darcy, Neal Gafter, Laurence Gonsalves, Aaron Greenhouse, Barry Hayes, Peter Jones, Angelika Langer, Doug Lea, Bob Lee, Jeremy Manson, Tom May, Mike McCloskey, Andriy Tereshchenko và Paul Tyma.

Một lần nữa, những người đánh giá này đã đưa ra nhiều đề xuất dẫn đến những cải thiện đáng kể trong cuốn sách này và giúp tôi thoát khỏi nhiều bối rối. Và một lần nữa, bất kỳ sự bối rối nào còn lại là trách nhiệm của tôi.

Tôi đặc biệt cảm ơn Doug Lea và Tim Peierls, những người đã đóng vai trò là người đóng góp ý kiến cho nhiều ý tưởng trong cuốn sách này. Doug và Tim vô cùng hào hứng với thời gian và kiến thức của họ.

Tôi cảm ơn người quản lý của tôi tại Google, Prabha Krishna, vì đã tiếp tục hỗ trợ và khuyến khích cô ấy.

Machine Translated by Google

Cuối cùng, tôi cảm ơn vợ tôi, Cindy Bloch, đã khuyến khích tôi viết, vì đã đọc từng mục ở dạng thô, vì đã giúp tôi với Framemaker, viết chỉ mục và đã hỗ trợ tôi khi tôi viết.

Lời cảm ơn cho Ấn bản đầu tiên

Tôi cảm ơn Patrick Chan đã gợi ý tôi viết cuốn sách này và trình bày ý tưởng với Lisa Friendly, biên tập viên quản lý bộ truyện; Tim Lindholm, biên tập kỹ thuật loạt phim; và Mike Hendrickson, biên tập viên điều hành của Addison-Wesley. Tôi cảm ơn Lisa, Tim và Mike đã khuyến khích tôi theo đuổi dự án và vì sự kiên nhẫn siêu phàm và niềm tin kiên định của họ rằng một ngày nào đó tôi sẽ viết cuốn sách này.

Tôi cảm ơn James Gosling và nhóm ban đầu của anh ấy đã cho tôi điều gì đó tuyệt vời để viết về nó, và tôi cảm ơn nhiều kỹ sư nền tảng Java đã theo bước James. Đặc biệt, tôi cảm ơn các đồng nghiệp của tôi trong Nhóm thư viện và công cụ nền tảng Java của Sun vì những hiểu biết sâu sắc, sự khuyến khích và cổng hỗ trợ của họ. Đội bao gồm Andrew Bennett, Joe Darcy, Neal Gafter, Iris Garcia, Konstantin Kladko, Ian Little, Mike McCloskey và Mark Reinhold. Các thành viên cũ bao gồm Zhenghua Li, Bill Maddox và Naveen Sanjeeva.

Tôi cảm ơn người quản lý của tôi, Andrew Bennett, và giám đốc của tôi, Larry Abrahams, đã hỗ trợ hết mình và nhiệt tình cho dự án này. Tôi cảm ơn Rich Green, Phó Giám đốc Kỹ thuật của Phần mềm Java, đã cung cấp một môi trường nơi những người sử dụng công nghệ được tự do suy nghĩ sáng tạo và xuất bản công việc của họ.

Tôi đã được ban phước cho đội ngũ đánh giá tốt nhất có thể tưởng tượng được và tôi gửi lời cảm ơn chân thành nhất đến từng người trong số họ: Andrew Bennett, Cindy Bloch, Dan Bloch, Beth Bottos, Joe Bowbeer, Gilad Bracha, Mary Campione, Joe Darcy, David Eckhardt, Joe Fialli, Lisa Friendly, James Gosling, Peter Haggar, David Holmes, Brian Kernighan, Konstantin Kladko, Doug Lea, Zhenghua Li, Tim Lindholm, Mike McCloskey, Tim Peierls, Mark Reinhold, Ken Russell, Bill Shannon, Peter Stout, Phil Wadler , và hai người đánh giá ẩn danh. Họ đã đưa ra nhiều đề xuất dẫn đến những cải tiến tuyệt vời trong cuốn sách này và cứu tôi khỏi nhiều lúng túng. Mọi sự bối rối còn lại là trách nhiệm của tôi.

Nhiều đồng nghiệp, trong và ngoài Sun, đã tham gia vào các cuộc thảo luận kỹ thuật nhằm cải thiện chất lượng của cuốn sách này. Trong số những người khác, Ben Gomes, Steffen Grarup, Peter Kessler, Richard Roda, John Rose và David Stoutamire

Machine Translated by Google

đã đóng góp những hiểu biết hữu ích. Doug Lea, người đã đóng vai trò là người đóng vai trò quan trọng cho nhiều ý tưởng trong cuốn sách này, xin gửi lời cảm ơn đặc biệt đến. Doug vô cùng hào phóng với thời gian và kiến thức của mình.

Tôi cảm ơn Julie Dinicola, Jacqui Doucette, Mike Hendrickson, Heather Olszyk, Tracy Russ và toàn bộ nhóm tại Addison-Wesley vì sự ủng hộ và đánh giá cao chủ nghĩa cá nhân của họ. Ngay cả trong một lịch trình dày đặc đến khó tin, họ vẫn luôn thân thiện và dễ chịu.

Tôi cảm ơn Guy Steele đã viết Lời nói đầu. Tôi rất vinh dự vì anh ấy đã chọn tham gia vào dự án này.

Cuối cùng, tôi cảm ơn vợ tôi, Cindy Bloch, đã khuyến khích và thỉnh thoảng đe dọa tôi viết cuốn sách này, vì đã đọc từng mục ở dạng thô của nó, vì đã giúp tôi với Framemaker, vì đã viết chỉ mục và đã giúp đỡ tôi khi tôi đã viết.

Machine Translated by Google

Trang này cố ý để trống

Machine Translated by Google

CHƯƠNG 1

Giới thiệu

Cuốn sách NÀY được biên soạn để giúp bạn sử dụng Java ™ hiệu quả nhất ngôn ngữ lập trình và các thư viện cơ bản của nó, java.lang, java.util, và ở mức độ thấp hơn, java.util.concurrent và java.io. Cuốn sách bàn về các thư viện khác theo thời gian, nhưng nó không bao gồm giao diện người dùng đồ họa lập trình, API doanh nghiệp hoặc thiết bị di động.

Cuốn sách này bao gồm bảy mươi tám mục, mỗi mục truyền đạt một quy tắc. Các quy tắc nắm bắt thông lệ thường được coi là có lợi cho những người tốt nhất và lập trình viên có kinh nghiệm. Các mục được nhóm lỏng lẻo thành mười chương, mỗi chương liên quan đến một khía cạnh rộng lớn của thiết kế phần mềm. Cuốn sách không nhằm mục đích đọc từ đầu đến bìa: mỗi mục đứng riêng, nhiều hơn hoặc ít hơn. Các mặt hàng là được tham khảo nhiều để bạn có thể dễ dàng lập kế hoạch cho khóa học của mình thông qua cuốn sách.

Nhiều tính năng mới đã được thêm vào nền tảng trong Java 5 (phiên bản 1.5). Hầu hết các mục trong cuốn sách này sử dụng các tính năng này theo một cách nào đó. Bảng sau cho thấy cho bạn nơi để xem phạm vi chính của các tính năng này:

Tính năng	Chương hoặc Mục
Generics	Chương 5
Enums	Các mặt hàng 30-34
Chú thích	Các mục 35–37
Cho mỗi vòng lặp	Mặt hàng 46
Hộp thư tự động	Mục 40, 49
Varargs	Mặt hàng 42
Nhập tĩnh	Mặt hàng 19
java.util.concurrent	Mục 68, 69

Machine Translated by Google

Hầu hết các mục đều được minh họa bằng các ví dụ của chương trình. Một tính năng chính của cuốn sách này là nó chứa các ví dụ mã minh họa nhiều mẫu thiết kế và thành ngữ. Khi thích hợp, chúng được tham chiếu chéo với công việc tham chiếu tiêu chuẩn trong lĩnh vực này [Gamma95].

Nhiều mục chứa một hoặc nhiều ví dụ chương trình minh họa một số thực hành cần tránh. Những ví dụ như vậy, đôi khi được gọi là phản vật chất, được gắn nhãn rõ ràng với chú thích như "// Đừng bao giờ làm điều này!" Trong mỗi trường hợp, mục này giải thích lý do tại sao ví dụ xấu và đề xuất một cách tiếp cận thay thế.

Cuốn sách này không dành cho người mới bắt đầu: nó giả định rằng bạn đã cảm thấy thoải mái với ngôn ngữ lập trình Java. Nếu bạn không, hãy xem xét một trong nhiều văn bản giới thiệu tốt [Arnold05, Sestoft05]. Mặc dù cuốn sách được thiết kế để phù hợp với bất kỳ ai có kiến thức về ngôn ngữ làm việc, nhưng nó sẽ cung cấp nguồn thức ăn để suy nghĩ ngay cả đối với các lập trình viên cao cấp.

Hầu hết các quy tắc trong cuốn sách này bắt nguồn từ một vài nguyên tắc cơ bản. Rõ ràng và đơn giản là điều tối quan trọng. Người dùng của một mô-đun không bao giờ được ngạc nhiên về hành vi của nó. Các mô-đun phải càng nhỏ càng tốt nhưng không được nhỏ hơn. (Như được sử dụng trong cuốn sách này, thuật ngữ mô-đun đề cập đến bất kỳ thành phần phần mềm nào có thể sử dụng lại, từ một phương pháp riêng lẻ đến một hệ thống phức tạp bao gồm nhiều gói.) Mã nên được sử dụng lại thay vì sao chép. Sự phụ thuộc giữa các mô-đun nên được giữ ở mức tối thiểu. Các lỗi phải được phát hiện ngay sau khi chúng được tạo ra, lý tưởng nhất là tại thời điểm biên dịch.

Mặc dù các quy tắc trong cuốn sách này không áp dụng 100% thời gian, nhưng chúng mô tả các phương pháp lập trình tốt nhất trong phần lớn các trường hợp. Bạn không nên tuân theo các quy tắc này một cách phiến diện mà chỉ nên vi phạm chúng thỉnh thoảng và có lý do chính đáng. Học nghệ thuật lập trình, giống như hầu hết các ngành khác, trước tiên bao gồm việc học các quy tắc và sau đó học khi nào nên phá vỡ chúng.

Phần lớn, cuốn sách này không nói về hiệu suất. Đó là về việc viết các gam chuyên nghiệp rõ ràng, chính xác, có thể sử dụng được, mạnh mẽ, linh hoạt và có thể bảo trì. Nếu bạn có thể làm được điều đó, thường là một vấn đề tương đối đơn giản để đạt được hiệu suất bạn cần (Mục 55). Một số mục thảo luận về các mối quan tâm về hiệu suất và một số mục trong số này cung cấp các con số về hiệu suất. Những con số này, được giới thiệu bằng cụm từ "Trên máy của tôi", tốt nhất nên được coi là gần đúng.

Đối với những gì nó đáng giá, máy của tôi là AMD Opteron ™ 170 lõi kép 2,2 GHz xây dựng tại nhà cũ với 2 gigabyte RAM, chạy phiên bản 1.6_05 của Bộ phát triển Java SE (JDK) của Sun trên đầu Microsoft Windows® XP Professional SP2. JDK này có hai máy ảo, Máy khách và Máy chủ Java HotSpot ™
Máy ảo. Số hiệu suất được đo trên Server VM.

Machine Translated by Google

Khi thảo luận về các tính năng của ngôn ngữ lập trình Java và các thư viện của nó, đôi khi cần phải tham khảo các bản phát hành cụ thể. Để ngắn gọn, cuốn sách này sử dụng "Số phiên bản kỹ thuật" tùy thuộc vào tên bản phát hành chính thức. Cái bàn này hiển thị ánh xạ giữa tên bản phát hành và số phiên bản kỹ thuật.

Tên phát hành chính thức	Số phiên bản kỹ thuật
JDK 1.1.x / JRE 1.1.x	1.1
Nền tảng Java 2, Phiên bản tiêu chuẩn, v 1.2	1,2
Nền tảng Java 2, Phiên bản tiêu chuẩn, v 1.3	1,3
Nền tảng Java 2, Phiên bản tiêu chuẩn, v 1.4	1,4
Nền tảng Java 2, Phiên bản tiêu chuẩn, v 5.0	1,5
Nền tảng Java, Phiên bản tiêu chuẩn 6	1,6

Các ví dụ khá đầy đủ, như ng chúng ư u tiên tính dễ đọc hơ n là sự dễ hiểu. Họ tự do sử dụng các lớp từ các gói java.util và java.io. Ở trong để biên dịch các ví dụ, bạn có thể phải thêm một hoặc nhiều lần nhập này các câu lệnh:

```
nhập java.util. *;
nhập java.util.concurrent. *;
nhập java.io. *;
```

Các tấm boilerplate khác cũng bị bỏ qua tư ơ ng tự. Trang web của cuốn sách, http: // java.sun.com/docs/books/effective, chứa phiên bản mở rộng của mỗi ví dụ mà bạn có thể biên dịch và chạy.

Đối với hầu hết các phần, cuốn sách này sử dụng các thuật ngữ kỹ thuật như chúng đư ợc định nghĩa trong Đặc tả ngôn ngữ Java, Ấn bản thứ ba [JLS]. Một số điều khoản đặc biệt xứng đáng đề cập đến. Ngôn ngữ này hỗ trợ bốn loại kiểu: giao diện (bao gồm cả annota tions), lớp (bao gồm cả enum), mảng và nguyên thủy. Ba cái đầu tiên đư ợc biết đến như các loại tham chiếu. Các cá thể và mảng của lớp là các đối tư ợng; giá trị nguyên thủy không. Các thành viên của một lớp bao gồm các trư ờng, phư ơ ng thức, lớp thành viên và thành viên các giao diện. Chữ ký của một phư ơ ng thức bao gồm tên của nó và các loại chính thức của nó thông số; chữ ký không bao gồm kiểu trả về của phư ơ ng thức.

Cuốn sách này sử dụng một số thuật ngữ khác với cuốn Sách cụ thể về ngôn ngữ Java. Không giống như Đặc tả ngôn ngữ Java, cuốn sách này sử dụng kế thừa như một mã bổ trợ syn cho phân lớp. Thay vì sử dụng thuật ngữ kế thừa cho các giao diện, điều này

Machine Translated by Google

book chỉ đơn giản nói rằng một lớp triển khai một giao diện hoặc một giao diện đó mở rộng khác. Để mô tả cấp độ truy cập áp dụng khi không có quy định nào được chỉ định, cuốn sách này sử dụng thuật ngữ mô tả là gói-riêng thay vì thuật ngữ truy cập mặc định về mặt kỹ thuật của thuật ngữ cor trực tràng [JLS, 6.6.1].

Cuốn sách này sử dụng một số thuật ngữ kỹ thuật không được định nghĩa trong Java Lan đặc điểm kỹ thuật guage. Thuật ngữ API đã xuất, hoặc đơn giản là API, đề cập đến các lớp, giao diện, trình xây dựng, thành viên và các biểu mẫu được tuần tự hóa mà một lập trình viên truy cập một lớp, giao diện hoặc gói. (Thuật ngữ API, viết tắt của giao diện lập trình ứng dụng, được sử dụng thay thế cho thuật ngữ thích hợp hơn giao diện để tránh nhầm lẫn với cấu trúc ngôn ngữ của tên đó.) Người chấm điểm chuyên nghiệp viết chương trình sử dụng API được gọi là người dùng API. Một lớp có việc triển khai sử dụng API là một ứng dụng khách của API.

Các lớp, giao diện, trình xây dựng, thành viên và các biểu mẫu được tuần tự hóa được gọi chung là các phần tử API. Một API đã xuất bao gồm các phần tử API có thể truy cập bên ngoài gói định nghĩa API. Đây là các phần mềm API mà bất kỳ khách hàng nào cũng có thể sử dụng và tác giả của API cam kết sẽ hỗ trợ. Không tình cờ, chúng cũng là các phần tử mà tiện ích Javadoc tạo ra tài liệu trong chế độ hoạt động mặc định của nó. Nói một cách dễ hiểu, đã xuất API của một gói bao gồm các thành viên và trình xây dựng công khai và được bảo vệ của mọi lớp công khai hoặc giao diện trong gói.

Machine Translated by Google

CHƯ Ơ NG 2

Tạo và phá hủy các đối tư ợng

CÁI NÀY chư ơ ng liên quan đến việc tạo và phá hủy các đối tư ợng: khi nào và làm thế nào để tạo chúng, khi nào và làm thế nào để tránh tạo chúng, làm thế nào để đảm bảo chúng bị phá hủy kịp thời và cách quản lý bất kỳ hành động dọn dẹp nào trư ớc khi phá hủy chúng.

Mục 1: Xem xét các phư ơ ng pháp nhà máy tĩnh thay vì các hàm tạo

Cách thông thư ờng để một lớp cho phép khách hàng có đư ợc một thể hiện của chính nó là cung cấp một phư ơ ng thức khởi tạo công khai. Có một kỹ thuật khác nên có trong bộ công cụ của mọi lập trình viên. Một lớp có thể cung cấp một phư ơ ng thức nhà máy sản xuất tĩnh công cộng, phư ơ ng thức này chỉ đơ n giản là một phư ơ ng thức tĩnh trả về một thể hiện của lớp. Đây là một bài kiểm tra đơ n giản từ Boolean (lớp nguyên thủy đóng hộp cho kiểu boolean nguyên thủy). Phư ơ ng thức này dịch một giá trị nguyên thủy boolean thành một tham chiếu đối tư ợng Boolean :

```
public static Boolean valueOf (boolean b) {
    trả lại b? Boolean.TRUE: Boolean.FALSE;
}
```

Lư u ý rằng một phư ơ ng pháp nhà máy tĩnh không giống với mẫu Phư ơ ng pháp Nhà máy từ Mẫu thiết kế [Gamma95, p. 107]. Phư ơ ng pháp nhà máy tĩnh đư ợc mô tả trong mục này không có tư ơ ng đư ơ ng trực tiếp trong Mẫu thiết kế.

Một lớp có thể cung cấp cho khách hàng của nó các phư ơ ng thức nhà máy tĩnh thay vì hoặc thêm vào các phư ơ ng thức khởi tạo. Cung cấp một phư ơ ng thức nhà máy tĩnh thay vì một phư ơ ng thức khởi tạo công cộng có cả ư u điểm và như ợc điểm.

Một ư u điểm của các phư ơ ng thức static factory là, không giống như các constructor, chúng có tên. Nếu các tham số của một hàm tạo không mô tả đối tư ợng đư ợc trả về, thì một nhà máy tĩnh có tên đư ợc chọn tốt sẽ dễ sử dụng hơ n và mã máy khách kết quả dễ đọc hơ n. Ví dụ, hàm tạo

Machine Translated by Google

BigInteger (int, int, Random), trả về một BigInteger có thể là
số nguyên tố, lẽ ra sẽ được thể hiện tốt hơn dưới dạng một phương thức nhà máy tĩnh có tên là BigInteger.probablePrime. (Phương pháp này cuối cùng đã được thêm vào trong bản phát hành 1.4.)

Một lớp chỉ có thể có một hàm tạo duy nhất với một chữ ký đã cho. Chương trình mers đã được biết là khắc phục hạn chế này bằng cách cung cấp hai hàm tạo
có danh sách tham số chỉ khác nhau về thứ tự của các loại tham số của chúng. Đây là một
ý tưởng thực sự tồi tệ. Người dùng của một API như vậy sẽ không bao giờ có thể nhớ được
hàm tạo là và cuối cùng sẽ gọi sai một hàm do nhầm lẫn. Mọi người
đọc mã sử dụng các hàm tạo này sẽ không biết mã làm gì nếu không tham khảo tài liệu lớp.

Bởi vì chúng có tên, các phương thức nhà máy tĩnh không chia sẻ hạn chế
đã thảo luận ở đoạn trước. Trong trường hợp một lớp dường như yêu cầu nhiều hàm tạo có cùng một chữ ký, hãy thay thế các hàm tạo bằng các phương thức tĩnh fac tory và các tên được chọn cẩn thận để làm nổi bật sự khác biệt của chúng.

Ưu điểm thứ hai của các phương pháp nhà máy tĩnh là, không giống như các hàm tạo,
chúng không bắt buộc phải tạo một đối tượng mới mỗi khi chúng được gọi. Cái này
cho phép các lớp không thay đổi (Mục 15) sử dụng các thể hiện được tạo sẵn hoặc để lưu vào bộ nhớ cache
các trường hợp khi chúng được xây dựng và phân phối chúng nhiều lần để tránh tạo
các đối tượng trùng lặp không cần thiết. Phương thức Boolean.valueOf (boolean) đánh lừa kỹ thuật này:
nó không bao giờ tạo một đối tượng. Kỹ thuật này tương tự như
Mẫu hạng ruồi [Gamma95 , tr. 195]. Nó có thể cải thiện đáng kể hiệu suất nếu
các đối tượng tương đương được yêu cầu thường xuyên, đặc biệt nếu chúng tốn kém để tạo.

Khả năng của các phương thức nhà máy tĩnh để trả về cùng một đối tượng từ việc lặp lại
lời gọi cho phép các lớp duy trì kiểm soát chặt chẽ đối với những trường hợp nào tồn tại tại
Bất cứ lúc nào. Các lớp thực hiện điều này được cho là được kiểm soát bởi phiên bản. Có một số
lý do để viết các lớp được điều khiển bởi phiên bản. Kiểm soát phiên bản cho phép một lớp
đảm bảo rằng nó là singleton (Mục 3) hoặc không bền (Mục 4). Ngoài ra, nó cho phép
một lớp bất biến (Mục 15) để đảm bảo rằng không có hai trường hợp nào bằng nhau
tồn tại: a.equals (b) nếu và chỉ khi a == b. Nếu một lớp thực hiện đảm bảo này, thì các cli ents của
nó có thể sử dụng toán tử == thay vì phương thức bằng (Đối tượng) , có thể
giúp cải thiện hiệu suất. Các loại Enum (Mục 30) cung cấp sự đảm bảo này.

Ưu điểm thứ ba của các phương pháp nhà máy tĩnh là, không giống như các phương thức xây dựng,
chúng có thể trả về một đối tượng thuộc bất kỳ kiểu con nào trong kiểu trả về của chúng. Điều này mang lại cho bạn
rất linh hoạt trong việc lựa chọn lớp của đối tượng được trả về.

Một ứng dụng của tính linh hoạt này là API có thể trả về các đối tượng mà không
công khai các lớp học của họ. Ẩn các lớp triển khai theo cách này dẫn đến
một API rất nhỏ gọn. Kỹ thuật này tự sử dụng các khuôn khổ dựa trên giao diện
(Mục 18), trong đó các giao diện cung cấp các kiểu trả về tự nhiên cho các phương thức nhà máy tĩnh.

Machine Translated by Google

Các giao diện không thể có các phương thức tĩnh, vì vậy theo quy ước, các phương thức nhà máy tĩnh cho một giao diện có tên là Kiểu được đặt trong một lớp không thể thay đổi (Mục 4) có tên là Kiểu.

Ví dụ: Java Collections Framework có ba mươi hai triển khai thuận tiện cho các giao diện bộ sưu tập của nó, cung cấp các bộ sưu tập không thể sửa đổi, bộ sưu tập được đồng bộ hóa và những thứ tương tự. Gần như tất cả các triển khai này được xuất thông qua các phương thức của nhà máy tĩnh trong một lớp không thể thay đổi (các phần tử java.util.Col). Các lớp của các đối tượng được trả về đều là không công khai.

API khung bộ sưu tập nhỏ hơn nhiều so với những gì nó sẽ có

nó đã xuất ra ba mươi hai lớp công khai riêng biệt, một lớp cho mỗi lớp tham khảo thuận tiện. Nó không chỉ là khối lượng lớn của API bị giảm đi, mà là trọng lượng khái niệm. Người dùng biết rằng đối tượng trả về có chính xác API được chỉ định bởi giao diện của nó, vì vậy không cần phải đọc tài liệu lớp bổ sung cho các lớp triển khai. Hơn nữa, việc sử dụng phương thức nhà máy tĩnh như vậy yêu cầu khách hàng phải tham chiếu đến đối tượng được trả về bởi giao diện của nó chứ không phải là lớp triển khai của nó, đây là cách nói chung (Mục 52).

Lớp của một đối tượng được trả về bởi một phương thức public static factory không chỉ có thể là nonpublic, mà lớp này có thể thay đổi từ lệnh gọi sang lệnh gọi tùy thuộc vào giá trị của các tham số đối với nhà máy tĩnh. Cho phép bất kỳ lớp nào là kiểu con của kiểu trả về đã khai báo. Lớp của đối tượng trả về cũng có thể thay đổi tùy theo từng phiên bản để nâng cao hiệu suất và khả năng bảo trì phần mềm.

Lớp java.util.EnumSet (Mục 32), được giới thiệu trong bản phát hành 1.5, không có hàm tạo công khai, chỉ có các nhà máy tĩnh. Chúng trả về một trong hai cách triển khai, tùy thuộc vào kích thước của kiểu enum bên dưới: nếu nó có sáu mươi tư phần tử trở xuống, như hầu hết các kiểu enum, các nhà máy tĩnh trả về một thể hiện RegularEnumSet , được hỗ trợ bởi một long duy nhất; nếu kiểu enum có sáu mươi lăm phần tử trở lên, các nhà máy sẽ trả về một cá thể JumboEnumSet , được hỗ trợ bởi một mảng dài .

Sự tồn tại của hai lớp thực thi này là vô hình đối với các máy khách. Nếu RegularEnumSet không còn cung cấp lợi thế về hiệu suất cho các loại enum nhỏ, nó có thể bị loại khỏi bản phát hành trong tương lai mà không có tác dụng xấu. Tương tự như vậy, bản phát hành trong tương lai có thể thêm phần triển khai thứ ba hoặc thứ tư của EnumSet nếu nó tỏ ra có lợi cho hiệu suất. Khách hàng không biết hoặc không quan tâm đến loại đối tượng mà họ lấy lại từ nhà máy; họ chỉ quan tâm rằng nó là một số lớp con của EnumSet.

Lớp của đối tượng được trả về bởi một phương thức static factory thậm chí không cần tồn tại tại thời điểm lớp chứa phương thức được viết. Các phương thức nhà máy tĩnh linh hoạt như vậy tạo thành nền tảng của các khuôn khổ nhà cung cấp dịch vụ, chẳng hạn như API kết nối cơ sở dữ liệu Java (JDBC). Khung nhà cung cấp dịch vụ là một hệ thống trong đó nhiều nhà cung cấp dịch vụ triển khai một dịch vụ và hệ thống cung cấp các đề cập tiềm năng cho khách hàng của mình, tách chúng khỏi việc triển khai.

Machine Translated by Google

Có ba thành phần thiết yếu của khung nhà cung cấp dịch vụ: giao diện trợ giúp, mà nhà cung cấp thực hiện; API đăng ký nhà cung cấp, mà hệ thống sử dụng để đăng ký triển khai, cấp cho khách hàng quyền truy cập vào chúng; và một API cấp quyền truy cập ser, mà khách hàng sử dụng để lấy một phiên bản của dịch vụ. API truy cập dịch vụ thư ờng cho phép như ng không yêu cầu khách hàng chỉ định một số tiêu chí để chọn nhà cung cấp. Trong trư ờng hợp không có đặc tả như vậy, API trả về một phiên bản của triển khai mặc định. API truy cập dịch vụ là "nhà máy tĩnh linh hoạt" tạo cơ sở cho khung nhà cung cấp dịch vụ.

Thành phần thứ tư tùy chọn của khung nhà cung cấp dịch vụ là giao diện nhà cung cấp dịch vụ, giao diện mà nhà cung cấp thực hiện để tạo các phiên bản triển khai dịch vụ của họ. Trong trư ờng hợp không có giao diện nhà cung cấp dịch vụ, các triển khai đư ợc đăng ký theo tên lớp và đư ợc khởi tạo phản ánh một cách trực quan (Mục 53). Trong trư ờng hợp của JDBC, Connection đóng vai trò là một phần của giao diện dịch vụ, DriverManager.reg isterDriver là API đăng ký của nhà cung cấp, DriverManager.getConnection là API truy cập dịch vụ và Driver là giao diện của nhà cung cấp dịch vụ.

Có rất nhiều biến thể của mẫu khung nhà cung cấp dịch vụ. Ví dụ: API truy cập dịch vụ có thể trả về giao diện dịch vụ phong phú hơ n so với giao diện yêu cầu của nhà cung cấp, bằng cách sử dụng mẫu Bộ điều hợp [Gamma95, tr. 139]. Đây là cách triển khai đơ n giản với giao diện nhà cung cấp dịch vụ và nhà cung cấp mặc định:

```
// Bản phác thảo khung nhà cung cấp dịch vụ

// Giao diện dịch vụ giao
diện công khai Dịch vụ {... // Các
      phư ơ ng thức dành riêng cho dịch vụ truy cập tại đây
}

// Giao diện nhà cung cấp dịch vụ giao
diện công khai Nhà cung cấp {
      Dịch vụ newService ();
}

// Lớp không bền để đăng ký dịch vụ và truy cập Dịch vụ lớp công khai {

      dịch vụ riêng () {} // Ngăn chặn việc khởi tạo (Mục 4)

      // Ánh xạ tên dịch vụ tới các dịch vụ
      private static final Map <String, Provider> provider = new
            ConcurrentHashMap <String, Provider> (); chuỗi cuối cùng tĩnh công
      khai DEFAULT_PROVIDER_NAME = "<def>";
```

Machine Translated by Google


```
        // API đăng ký nhà cung cấp public
        static void registerDefaultProvider (Nhà cung cấp p) {registerProvider
             (DEFAULT_PROVIDER_NAME, p);

        } public static void registerProvider (String name, Provider p) {provider.put
             (name, p);
        }

        // API truy cập dịch vụ
        public static Service newInstance () {
             trả về newInstance (DEFAULT_PROVIDER_NAME);

        } public static Service newInstance (String name) {Provider
             p = provider.get (name); if (p == null) ném
             IllegalArgumentException mới (

                                                                    "
                       "Không có nhà cung cấp nào được đăng ký với     + tên);
             tên: return p.newService ();
        }
    }
```

Ưu điểm thứ tư của các phương pháp nhà máy tĩnh là chúng giảm bớt tính chất của việc tạo các thể hiện kiểu được tham số hóa. Thật không may, bạn phải chỉ định ify các tham số kiểu khi bạn gọi phương thức khởi tạo của một lớp được tham số hóa ngay cả khi chúng rõ ràng từ ngữ cảnh. Điều này thường yêu cầu bạn cung cấp các tham số kiểu hai lần liên tiếp:

```
    Bản đồ <Chuỗi, Danh sách <Chuỗi>> m
         = new HashMap <Chuỗi, Danh sách <Chuỗi>> ();
```

Đặc điểm kỹ thuật dư thừa này nhanh chóng trở nên khó khăn khi độ dài và độ phức tạp của các tham số kiểu tăng lên. Tuy nhiên, với các nhà máy tĩnh, trình biên dịch có thể tìm ra các tham số kiểu cho bạn. Đây được gọi là kiểu suy luận. Ví dụ: giả sử rằng HashMap đã cung cấp nhà máy tĩnh này:

```
    public static <K, V> HashMap <K, V> newInstance () {return new
         HashMap <K, V> ();
    }
```

Sau đó, bạn có thể thay thế tuyên bố dài dòng ở trên bằng cách thay thế ngắn gọn này:

```
    Bản đồ <Chuỗi, Danh sách <Chuỗi>> m = HashMap.newInstance ();
```

Một ngày nào đó, ngôn ngữ có thể thực hiện kiểu suy luận kiểu này đối với các lệnh gọi phương thức khởi tạo cũng như các lệnh gọi phương thức, nhưng kể từ phiên bản 1.6, thì không.

Machine Translated by Google

Thật không may, các triển khai bộ sưu tập tiêu chuẩn như HashMap không có các phương thức gốc kể từ phiên bản 1.6, nhưng bạn có thể đặt các phương thức này trong lớp tiện ích của riêng mình. Quan trọng hơn, bạn có thể cung cấp các nhà máy tĩnh như vậy trong các lớp được tham số hóa của riêng bạn.

Nhược điểm chính của việc chỉ cung cấp các phương thức nhà máy tĩnh là các lớp không có hàm tạo công khai hoặc được bảo vệ không thể được phân lớp. Điều này cũng đúng đối với các lớp không công khai được trả về bởi các nhà máy tĩnh công cộng. Ví dụ, không thể phân loại bất kỳ lớp nào trong số các lớp triển khai tiện lợi trong Khung bộ sưu tập. Có thể cho rằng đây có thể là một điều may mắn khi ngụy tạo, vì nó khiến các lập trình viên yêu cầu tuổi tác sử dụng thành phần thay vì kế thừa (Mục 16).

Một nhược điểm thứ hai của các phương thức static factory là chúng không dễ phân biệt với các phương thức static khác. Chúng không nổi bật trong tài liệu API theo cách mà các hàm tạo làm, vì vậy có thể khó tìm ra cách khởi tạo một lớp cung cấp các phương thức gốc tĩnh thay vì các mã cấu trúc. Một ngày nào đó, công cụ Javadoc có thể thu hút sự chú ý đến các phương pháp nhà máy tĩnh. Trong thời gian chờ đợi, bạn có thể giảm bớt nhược điểm này bằng cách thu hút sự chú ý đến các câu hỏi thường gặp trong lớp hoặc nhận xét giao diện và bằng cách tuân thủ các quy tắc đặt tên phổ biến. Dưới đây là một số tên phổ biến cho các phương thức nhà máy tĩnh:

- valueOf — Trả về một thể hiện có cùng giá trị với các tham số của nó. Các nhà máy tĩnh như vậy là phương pháp chuyển đổi kiểu hiệu quả.
- of — Một sự thay thế ngắn gọn cho valueOf, được EnumSet phổ biến (Mục 32).
- getInstance — Trả về một thể hiện được mô tả bởi các tham số nhưng không thể được coi là có cùng giá trị. Trong trường hợp của một singleton, getInstance không nhận tham số và trả về cá thể duy nhất.
- newInstance — Giống như getInstance, ngoại trừ việc newInstance đảm bảo rằng mỗi trường hợp được trả về là khác biệt với tất cả các trường hợp khác.
- getType — Giống như getInstance, nhưng được sử dụng khi phương thức gốc nằm trong một lớp ent khác. Loại chỉ ra loại đối tượng được trả về bởi phương thức gốc.
- newType — Giống như newInstance, nhưng được sử dụng khi phương thức gốc nằm trong một lớp ent khác. Loại chỉ ra loại đối tượng được trả về bởi phương thức gốc.

Tóm lại, phương pháp nhà máy tĩnh và phương pháp xây dựng công cộng đều có công dụng của chúng và cần hiểu được giá trị tương đối của chúng. Thường thì các nhà máy tĩnh có thể xây dựng được trước, vì vậy hãy tránh phản xạ cung cấp các nhà xây dựng công cộng mà không xem xét các nhà máy tĩnh trước.

Machine Translated by Google

Mục 2: Hãy xem xét một trình xây dựng khi đối mặt với nhiều tham số của hàm tạo

Các nhà máy và công trình xây dựng tĩnh có chung một hạn chế: chúng không quy mô tốt đến quy mô lớn số lư ợng các tham số tùy chọn. Hãy xem xét trư ờng hợp của một lớp đại diện cho Nhãn Thông tin dinh dư ỡng xuất hiện trên thực phẩm đóng gói. Các nhãn này có một vài các trư ờng bắt buộc — kích thư ớc khẩu phần, khẩu phần trên mỗi thùng chứa và lư ợng calo trên mỗi khẩu phần— và hơ n hai mư ơ i trư ờng tùy chọn — tổng chất béo, chất béo bão hòa, chất béo chuyển hóa, cholesterol, natri, v.v. Hầu hết các sản phẩm có giá trị khác không cho một vài trong số này các trư ờng tùy chọn.

Bạn nên viết loại nhà xây dựng hoặc nhà máy tĩnh nào cho một lớp như vậy? Theo truyền thống, các lập trình viên đã sử dụng mẫu phư ơ ng thức khởi tạo telescoping , trong mà bạn cung cấp một phư ơ ng thức khởi tạo chỉ với các tham số bắt buộc, một phư ơ ng thức khác có một tham số tùy chọn duy nhất, một tham số thứ ba với hai tham số tùy chọn, v.v. Đây là cách nó trông như thế nào thực hành. Để ngắn gọn, chỉ có bốn trư ờng tùy chọn đư ợc hiển thị:

```
// Mẫu phư ơ ng thức khởi tạo Telescoping - không chia tỷ lệ tốt!
lớp công cộng NutritionFacts {
      private cuối cùng int serveSize; // (mL) bắt buộc
      phần ăn int cuối cùng riêng tư ; // (mỗi vùng chứa) bắt buộc
      calo int cuối cùng riêng tư ; // không bắt buộc
      private end int fat; // (g) tùy chọn
      int natri cuối cùng riêng tư ; // (mg) tùy chọn
      riêng cuối cùng int carbohydrate; // (g) tùy chọn

      public NutritionFacts (int serveSize, int khẩu phần) {
            this (serveSize, khẩu phần, 0);
      }

      public NutritionFacts (int serveSize, int khẩu phần,
                  int calo) {
            this (Kích thư ớc khẩu phần, khẩu phần, calo, 0);
      }

      public NutritionFacts (int serveSize, int khẩu phần,
                  int calo, int chất béo) {
            this (Kích thư ớc khẩu phần, khẩu phần, calo, chất béo, 0);
      }

      public NutritionFacts (int serveSize, int khẩu phần,
                  int calo, int chất béo, int natri) {
            this (Khẩu phần, khẩu phần, calo, chất béo, natri, 0);
      }
```

Machine Translated by Google


```
        public NutritionFacts (int serveSize, int khẩu phần,
                int calo, int chất béo, int natri, int carbohydrate) {
            this.servingSize = serveSize;
            this.servings = phần ăn;
            this.calories = calo;
            this.fat = chất béo;
            this.sodium = natri;
            this.carbohydrate = carbohydrate;
        }
    }
```

Khi bạn muốn tạo một thể hiện, bạn sử dụng hàm tạo có giá trị ngắn nhất danh sách tham số chứa tất cả các tham số bạn muốn đặt:

```
    NutritionFacts cocaCola =
        NutritionFacts mới (240, 8, 100, 0, 35, 27);
```

Thông thường, lời gọi hàm tạo này sẽ yêu cầu nhiều tham số mà bạn không muốn đặt, nhưng bạn vẫn buộc phải chuyển một giá trị cho chúng. Trong trường hợp này, chúng tôi đã chuyển giá trị 0 cho chất béo. Với sáu tham số "chỉ", điều này có vẻ không quá tệ, nhưng nó nhanh chóng vượt ra khỏi tầm tay khi số lượng tham số tăng lên.

Nói tóm lại, mẫu phương thức khởi tạo telescoping hoạt động, nhưng rất khó để viết mã máy khách khi có nhiều tham số và khó đọc hơn. Các người đọc còn lại tự hỏi tất cả những giá trị đó có ý nghĩa gì và phải đếm cẩn thận các thông số để tìm hiểu. Chuỗi dài các tham số được nhập giống hệt nhau có thể gây ra lỗi tinh vi. Nếu khách hàng vô tình đảo ngược hai tham số như vậy, trình biên dịch sẽ không phàn nàn, nhưng chương trình sẽ hoạt động sai trong thời gian chạy (Mục 40).

Một giải pháp thay thế thứ hai khi bạn phải đối mặt với nhiều tham số hàm tạo là mẫu JavaBeans , trong đó bạn gọi một hàm tạo không tham số để tạo đối tượng và sau đó gọi các phương thức setter để đặt từng tham số bắt buộc và từng tham số tùy chọn quan tâm:

```
    // JavaBeans Pattern - cho phép sự không nhất quán, yêu cầu khả năng thay đổi
    lớp công cộng NutritionFacts {
        // Các tham số được khởi tạo thành giá trị mặc định (nếu có)
        private int serveSize = -1; // Yêu cầu; không có giá „trị mặc „định    "         "
        phần ăn int riêng = -1; // private int calo
        = 0;
        private int fat = 0;
        riêng int natri = 0;
        int carbohydrate riêng = 0;

        public NutritionFacts () {}
```

Machine Translated by Google


```
    // Bộ thiết
    lập public void setServingSize (int val) {serveSize = val; } public void
    setServings (int val) {serves = val; } public void setCalories (int val) {calories = val; } public void
    setFat (int val) {fat = val; } public void setSodium (int val) {sodium =
    setCarbohydrate (int val) {carbohydrate = val; }

}
```

Mô hình này không có bất kỳ như ợc điểm nào của mô hình xây dựng ống lồng.
Thật dễ dàng, nếu hơ i dài dòng, để tạo các phiên bản và dễ dàng đọc mã kết quả:

```
NutritionFacts cocaCola = new NutritionFacts ();
cocaCola.setServingSize (240); cocaCola.setServings (8);
cocaCola.setCalories (100); cocaCola.setSodium (35);
cocaCola.setCarbohydrat (27);
```

Thật không may, mẫu JavaBeans có những như ợc điểm nghiêm trọng của riêng nó.
Bởi vì quá trình xây dựng đư ợc chia thành nhiều lệnh gọi, một JavaBean có thể ở trạng thái không nhất quán trong suốt quá trình xây dựng của nó. Lớp không có tùy chọn thực thi tính nhất quán chỉ bằng cách kiểm tra tính hợp lệ của các tham số cấu trúc con. Việc cố gắng sử dụng một đối tư ợng khi nó ở trạng thái không nhất quán có thể gây ra lỗi khác xa với mã chứa lỗi, do đó rất khó gỡ lỗi. Một bất lợi liên quan là mẫu JavaBeans đã cho thấy trư ớc khả năng làm cho một lớp trở nên bất biến (Mục 15) và đòi hỏi ngư ời lập trình phải nỗ lực nhiều hơ n để đảm bảo an toàn cho luồng.

Có thể giảm bớt những như ợc điểm này bằng cách "đóng băng" đối tư ợng theo cách thủ công khi quá trình xây dựng của nó hoàn thành và không cho phép sử dụng nó cho đến khi đóng băng, như ng biến thể này khó sử dụng và hiếm khi đư ợc sử dụng trong thực tế. Hơ n nữa, nó có thể gây ra lỗi trong thời gian chạy, vì trình biên dịch không thể đảm bảo rằng lập trình viên gọi phư ơ ng thức đóng băng trên một đối tư ợng trư ớc khi sử dụng nó.

May mắn thay, có một giải pháp thay thế thứ ba kết hợp sự an toàn của mẫu phư ơ ng thức khởi tạo telescoping với khả năng đọc của mẫu JavaBeans. Nó là một dạng của mẫu Builder [Gamma95, tr. 97]. Thay vì tạo đối tư ợng mong muốn trực tiếp, máy khách gọi một phư ơ ng thức khởi tạo (hoặc nhà máy tĩnh) với tất cả các tham số bắt buộc và nhận đư ợc một đối tư ợng xây dựng. Sau đó, khách hàng gọi các phư ơ ng thức giống như setter trên đối tư ợng ngư ời xây dựng để đặt từng tham số tùy chọn quan tâm. Cuối cùng, máy khách gọi một phư ơ ng thức xây dựng vô hạn tham số để tạo đối tư ợng, đối tư ợng này là bất biến. Trình tạo là một lớp thành viên tĩnh (Mục 22) của lớp mà nó xây dựng. Đây là cách nó trông như thế nào trong thực tế:

Machine Translated by Google


```
// Mẫu Builder
lớp công cộng NutritionFacts {
     private cuối cùng int serveSize;
     phần ăn int cuối cùng riêng tư ;
     calo int cuối cùng riêng tư ;
     private end int fat;
     int natri cuối cùng riêng tư ;
     riêng cuối cùng int carbohydrate;

     Trình tạo lớp tĩnh công cộng {
          // Tham số bắt buộc
          private cuối cùng int serveSize;
          phần ăn int cuối cùng riêng tư ;

          // Tham số tùy chọn - được khởi tạo thành giá trị mặc định
          int calo riêng = 0;
          private int fat = 0;
          int carbohydrate riêng = 0;
          riêng int natri = 0;

          public Builder (int serveSize, int serves) {
               this.servingSize = serveSize;
               this.servings = phần ăn;
          }

          Lượng calo công cộng của Builder (int val)
               {calo = val; trả lại cái này; }
          public Builder fat (int val)
               {béo = val; trả lại cái này; }
          carbohydrate Builder công cộng (int val)
               {cacbohydrat = val; trả lại cái này; }
          natri Builder công cộng (int val)
               {natri = val; trả lại cái này; }

          public NutritionFacts build () {
               trả về NutritionFacts mới (this);
          }
     }

     private NutritionFacts (Builder xây dựng) {
          serveSize = builder.servingSize;
          phần ăn = builder.servings;
          calo = builder.calories;
          chất béo = builder.fat;
          natri = người xây dựng.sodium;
          carbohydrate = chất xây dựng.carbohydrate;
     }
}
```

Machine Translated by Google

Lư u ý rằng NutritionFacts là bất biến và tất cả các giá trị mặc định của tham số đều ở một vị trí duy nhất. Các phư ơ ng thức setter của trình xây dựng trả về chính trình xây dựng để các lời gọi có thể đư ợc xâu chuỗi. Đây là giao diện của mã khách hàng:

```
NutritionFacts cocaCola = new NutritionFacts.Builder (240, 8). calo (100) .sodium
   (35) .carbohydrate (27) .build ();
```

Mã ứng dụng này rất dễ viết và quan trọng hơ n là dễ đọc. Builder pat tern mô phỏng các tham số tùy chọn đư ợc đặt tên như đư ợc tìm thấy trong Ada và Python.

Giống như một phư ơ ng thức khởi tạo, một trình xây dựng có thể áp đặt các giá trị bất biến lên các tham số của nó. Phư ơ ng thức xây dựng có thể kiểm tra những bất biến này. Điều quan trọng là chúng phải đư ợc kiểm tra sau khi sao chép các tham số từ trình tạo sang đối tư ợng và chúng đư ợc kiểm tra trên các trư ờng đối tư ợng chứ không phải các trư ờng của trình tạo (Mục 39). Nếu có bất kỳ biến nào đư ợc xác định, phư ơ ng thức xây dựng sẽ ném một IllegalStateException (Mục 60). Phư ơ ng pháp chi tiết của ngoại lệ phải chỉ ra bất biến nào bị vi phạm (Mục 63).

Một cách khác để áp đặt các bất biến liên quan đến nhiều tham số là các phư ơ ng thức setter nhận toàn bộ các nhóm tham số mà một số bất biến phải giữ trên đó. Nếu bất biến không đư ợc thỏa mãn, phư ơ ng thức setter sẽ ném ra một IllegalArgu mentException. Điều này có lợi thế là phát hiện lỗi bất biến ngay sau khi các tham số không hợp lệ đư ợc chuyển qua, thay vì đợi xây dựng đư ợc gọi.

Một lợi thế nhỏ của trình xây dựng so với trình xây dựng là trình xây dựng có thể có nhiều tham số varargs. Các hàm tạo, giống như các phư ơ ng thức, chỉ có thể có một tham số varargs. Bởi vì trình xây dựng sử dụng các phư ơ ng thức riêng biệt để đặt từng tham số, họ có thể có bao nhiêu tham số varargs tùy thích, tối đa một tham số cho mỗi phư ơ ng thức setter.

Mẫu Builder rất linh hoạt. Một trình xây dựng duy nhất có thể đư ợc sử dụng để xây dựng nhiều đối tư ợng. Các tham số của trình xây dựng có thể đư ợc điều chỉnh giữa các lần tạo đối tư ợng để thay đổi các đối tư ợng. Trình tạo có thể tự động điền vào một số trư ờng, chẳng hạn như số sê-ri tự động tăng lên mỗi khi một đối tư ợng đư ợc tạo.

Một ngư ời xây dựng có các thông số đã đư ợc thiết lập sẽ tạo nên một Nhà máy Tóm tắt tốt [Gamma95, tr. 87]. Nói cách khác, một máy khách có thể chuyển một trình tạo như vậy cho một phư ơ ng thức để cho phép phư ơ ng thức tạo một hoặc nhiều đối tư ợng cho máy khách. Để kích hoạt cách sử dụng này, bạn cần một kiểu đại diện cho trình tạo. Nếu bạn đang sử dụng bản phát hành 1.5 hoặc bản phát hành mới hơ n, một loại chung duy nhất (Mục 26) là đủ cho tất cả các trình xây dựng, bất kể họ đang xây dựng loại đối tư ợng nào:

```
// Trình tạo đối tư ợng kiểu T public
interface Builder <T> {public T build ();

}
```

Machine Translated by Google

Lư u ý rằng lớp NutritionFacts.Builder của chúng ta có thể đư ợc khai báo để triển khai Builder <NosystemFacts>.

Các phư ơ ng thức sử dụng phiên bản Builder thư ờng sẽ hạn chế tham số kiểu của trình tạo bằng cách sử dụng kiểu ký tự đại diện đư ợc giới hạn (Mục 28). Ví dụ: đây là một phư ơ ng pháp xây dựng cây bằng cách sử dụng phiên bản Builder do máy khách cung cấp để xây dựng từng nút:

```
Tree buildTree (Builder <? Expand Node> nodeBuilder) {...}
```

Việc triển khai Abstract Factory truyền thống trong Java là đối tư ợng Class , với phư ơ ng thức newInstance đóng vai trò là một phần của phư ơ ng thức xây dựng . Việc sử dụng này có đầy vấn đề. Phư ơ ng thức newInstance luôn cố gắng gọi phư ơ ng thức khởi tạo không tham số của lớp, phư ơ ng thức này thậm chí có thể không tồn tại. Bạn không gặp lỗi thời gian biên dịch nếu lớp không có hàm tạo không tham số có thể truy cập.
Thay vào đó, mã máy khách phải đối phó với InstantiationException hoặc IllegalAc cessException trong thời gian chạy, điều này xấu và bất tiện. Ngoài ra, phư ơ ng thức lập trư ờng newIn truyền bất kỳ ngoại lệ nào do phư ơ ng thức khởi tạo không tham số ném ra, ngay cả khi newInstance thiếu các mệnh đề ném tư ơ ng ứng. Nói cách khác, Class.newInstance phá vỡ kiểm tra ngoại lệ thời gian biên dịch. Giao diện Builder , đư ợc hiển thị ở trên, sửa những thiếu sót này.

Mẫu Builder có những như ợc điểm của riêng nó. Để tạo một đối tư ợng, trư ớc tiên bạn phải tạo trình xây dựng của nó. Mặc dù chi phí tạo trình xây dựng khó có thể đáng chú ý trong thực tế, như ng nó có thể là một vấn đề trong một số tình huống quan trọng về hiệu suất. Ngoài ra, mẫu Builder dài dòng hơ n mẫu phư ơ ng thức tạo telescoping, vì vậy nó chỉ nên đư ợc sử dụng nếu có đủ bốn tham số, chẳng hạn, bốn hoặc nhiều hơ n. Như ng hãy nhớ rằng bạn có thể muốn thêm các tham số trong tư ơ ng lai.

Nếu bạn bắt đầu với các trình xây dựng hoặc nhà máy tĩnh và thêm một trình xây dựng khi lớp phát triển đến mức số lư ợng tham số bắt đầu vư ợt quá tầm kiểm soát, thì các trình xây dựng lỗi thời hoặc nhà máy tĩnh sẽ nhô ra như ngón tay cái. Ở đó

trư ớc hết, tốt hơ n hết là nên bắt đầu với một ngư ời xây dựng ngay từ đầu.

Tóm lại, mẫu Builder là một lựa chọn tốt khi thiết kế các lớp mà các trình tạo hoặc các nhà máy tĩnh sẽ có nhiều hơ n một số

các tham số, đặc biệt nếu hầu hết các tham số đó là tùy chọn. Mã ứng dụng khách dễ đọc và viết hơ n nhiều với các trình xây dựng so với mẫu phư ơ ng thức khởi tạo telescoping truyền thống và các trình xây dựng an toàn hơ n nhiều so với JavaBeans.

Machine Translated by Google

Mục 3: Thực thi thuộc tính singleton với một private hàm tạo hoặc một kiểu enum

Singleton đơn giản là một lớp được khởi tạo chính xác một lần [Gamma95, tr. 127]. Singleton thường đại diện cho một thành phần hệ thống về bản chất là duy nhất, chẳng hạn như trình quản lý cửa sổ hoặc hệ thống tệp. Việc tạo một lớp trở thành một singleton có thể gây khó khăn cho việc kiểm tra các máy khách của nó, vì không thể thay thế một phép cố vấn mô phỏng cho một lớp đơn trừ khi nó triển khai một giao diện phục vụ như kiểu của nó.

Trước khi phát hành 1.5, có hai cách để thực hiện các đĩa đơn. Cả hai đều dựa trên việc giữ phương thức khởi tạo ở chế độ riêng tư và xuất một thành viên tĩnh công khai để cung cấp quyền truy cập vào cá thể duy nhất. Theo một cách tiếp cận, thành viên là một trường cuối cùng:

```
// Singleton với trường cuối cùng công khai
public class Elvis { public static final
     Elvis INSTANCE = new Elvis (); riêng Elvis () {...}


     public void leaveTheBuilding () {...}
}
```

Hàm khởi tạo riêng chỉ được gọi một lần, để khởi tạo trường cuối cùng tĩnh công khai Elvis.INSTANCE. Việc thiếu một phương thức khởi tạo công khai hoặc được bảo vệ đảm bảo một vũ trụ "đơn hình": chính xác một cá thể Elvis sẽ tồn tại sau khi lớp Elvis được khởi tạo — không hơn, không kém. Không có gì mà một ứng dụng khách có thể thay đổi điều này, với một lưu ý: một ứng dụng khách đặc quyền có thể gọi phương thức khởi tạo riêng một cách hoạt động (Mục 53) với sự hỗ trợ của phương thức AccessibleObject.setAccessible . Nếu bạn cần bảo vệ chống lại cuộc tấn công này, hãy sửa đổi hàm tạo để làm cho nó trở thành một ngoại lệ nếu nó được yêu cầu tạo một phiên bản thứ hai.

Trong cách tiếp cận thứ hai để triển khai các singleton, thành viên công cộng là một phương pháp nhà máy tĩnh:

```
// Singleton với static factory public
class Elvis { private static final Elvis
     INSTANCE = new Elvis (); private Elvis () {...} public static
     Elvis getInstance () {return INSTANCE; }


     public void leaveTheBuilding () {...}
}
```

Tất cả các lệnh gọi đến Elvis.getInstance đều trả về cùng một tham chiếu đối tượng và không có cá thể Elvis nào khác sẽ được tạo (với cùng một cảnh báo đã đề cập ở trên).

Machine Translated by Google

Ư u điểm chính của phư ơ ng pháp tiếp cận trư ờng công khai là các khai báo làm rõ rằng lớp là một lớp đơ n: trư ờng tĩnh công khai là trư ờng cuối cùng, vì vậy nó sẽ luôn chứa cùng một tham chiếu đối tư ợng. Không còn bất kỳ lợi thế hiệu suất nào đối với phư ơ ng pháp tiếp cận trư ờng công cộng: việc triển khai máy ảo Java (JVM) hiện đại gần như chắc chắn để nội dòng lệnh gọi đến phư ơ ng thức static factory.

Một ư u điểm của phư ơ ng pháp tiếp cận factory-method là nó cho phép bạn linh hoạt thay đổi ý định về việc liệu lớp có nên là một lớp đơ n mà không cần thay đổi API của nó hay không. Phư ơ ng thức gốc trả về thể hiện duy nhất như ng có thể dễ dàng sửa đổi để trả về, chẳng hạn, một thể hiện duy nhất cho mỗi luồng gọi nó. Ư u điểm thứ hai, liên quan đến các kiểu chung, đư ợc thảo luận trong Mục 27. Thư ờng thì cả hai ư u điểm này đều không liên quan và cách tiếp cận trư ờng cuối cùng đơ n giản hơ n.

Để làm cho một lớp singleton đư ợc triển khai bằng cách sử dụng một trong các cách tiếp cận trư ớc đó có thể tuần tự hóa (Chư ơ ng 11), chỉ cần thêm imple ments Serializable vào khai báo của nó là không đủ. Để duy trì bảo đảm singleton, bạn phải khai báo tạm thời tất cả các trư ờng phiên bản và cung cấp phư ơ ng thức readResolve (Mục 77). Nếu không, mỗi khi một cá thể đư ợc tuần tự hóa đư ợc giải mã hóa, một cá thể mới sẽ đư ợc tạo, dẫn đến việc nhìn thấy Elvis giả, trong trư ờng hợp của chúng tôi, dẫn đến việc nhìn thấy Elvis giả . Để ngăn điều này, hãy thêm phư ơ ng thức readResolve này vào lớp Elvis :

```
// phư ơ ng thức readResolve để bảo toàn thuộc tính singleton private
Object readResolve () {
       // Trả về một Elvis thật và để bộ thu dọn rác // xử lý kẻ mạo danh Elvis. trả
       lại INSTANCE;

}
```

Kể từ bản phát hành 1.5, có một cách tiếp cận thứ ba để triển khai các đĩa đơ n. Đơ n giản tạo một kiểu enum với một phần tử:

```
// Enum singleton - cách tiếp cận ư u tiên public enum
Elvis {
      VÍ DỤ;

      public void leaveTheBuilding () {...}
}
```

Cách tiếp cận này tư ơ ng đư ơ ng về mặt chức năng với cách tiếp cận trư ờng công cộng, ngoại trừ nó ngắn gọn hơ n, cung cấp bộ máy tuần tự hóa miễn phí và cung cấp sự đảm bảo vững chắc chống lại nhiều lần khởi tạo, ngay cả khi đối mặt với các cuộc tấn công tuần tự hóa hoặc phản chiếu tinh vi. Mặc dù cách tiếp cận này vẫn chư a đư ợc áp dụng rộng rãi, như ng kiểu enum một phần tử là cách tốt nhất để triển khai một singleton.

Machine Translated by Google

Mục 4: Thực thi tính không thay đổi với một phư ơ ng thức khởi tạo riêng

Đôi khi bạn sẽ muốn viết một lớp chỉ là một nhóm các phư ơ ng thức tĩnh và trư ờng tĩnh. Những lớp học như vậy đã bị mang tiếng xấu vì một số ngư ời lạm dụng chúng để tránh suy nghĩ về đối tư ợng, như ng chúng có công dụng hợp lệ. Chúng có thể đư ợc sử dụng để nhóm các phư ơ ng thức liên quan trên các giá trị hoặc mảng nguyên thủy, theo cách của java.lang.Math hoặc java.util.Arrays. Chúng cũng có thể đư ợc sử dụng để nhóm các phư ơ ng thức tĩnh, bao gồm các phư ơ ng thức gốc (Mục 1), cho các đối tư ợng triển khai một giao diện nhỏ, theo cách của java.util.Collections. Cuối cùng, chúng có thể đư ợc sử dụng để nhóm các phư ơ ng thức trên một lớp cuối cùng, thay vì mở rộng lớp.

Các lớp tiện ích như vậy không đư ợc thiết kế để đư ợc khởi tạo: một thể hiện sẽ là vô nghĩa. Tuy nhiên, trong trư ờng hợp không có các hàm tạo rõ ràng, trình biên dịch sẽ cung cấp một hàm tạo mặc định công khai, không tham số. Đối với ngư ời dùng, hàm tạo này không thể phân biệt đư ợc với bất kỳ hàm tạo nào khác. Không có gì lạ khi thấy các lớp có thể khởi tạo không chủ ý trong các API đã xuất bản.

Cố gắng thực thi tính không thay đổi bằng cách tạo một lớp trừu tư ợng không hoạt động. Lớp có thể đư ợc phân lớp và lớp con đư ợc khởi tạo. Hơ n nữa, nó đánh lừa ngư ời dùng nghĩ rằng lớp đư ợc thiết kế để kế thừa (Mục 17).
Tuy nhiên, có một thành ngữ đơ n giản để đảm bảo tính không thay đổi. Một tor xây dựng mặc định chỉ đư ợc tạo nếu một lớp không chứa các hàm tạo rõ ràng, do đó, một lớp có thể đư ợc làm cho không thể thay đổi đư ợc bằng cách bao gồm một hàm tạo riêng:

```
// Lớp tiện ích không thể thay đổi lớp công
khai UtilityClass {
      // Loại bỏ phư ơ ng thức khởi tạo mặc định cho không có tính ổn định
      private UtilityClass () {ném new AssertionError ();

      }
      ... // Phần còn lại bị bỏ qua
}
```

Bởi vì hàm tạo rõ ràng là riêng tư , nó không thể truy cập đư ợc bên ngoài lớp. AssertionError không đư ợc yêu cầu nghiêm ngặt, như ng nó cung cấp bảo hiểm trong trư ờng hợp phư ơ ng thức khởi tạo vô tình đư ợc gọi từ bên trong lớp. Nó đảm bảo rằng lớp học sẽ không bao giờ đư ợc khởi tạo trong bất kỳ trư ờng hợp nào. Thành ngữ này hơ i phản trực giác, vì hàm tạo đư ợc cung cấp rõ ràng để nó không thể đư ợc gọi. Do đó, sẽ là khôn ngoan khi bao gồm một bình luận, như đư ợc hiển thị ở trên.

Như một tác dụng phụ, thành ngữ này cũng ngăn lớp đư ợc phân lớp. Tất cả các phư ơ ng thức khởi tạo phải gọi một phư ơ ng thức khởi tạo lớp cha, rõ ràng hoặc ngầm định, và một lớp con sẽ không có phư ơ ng thức khởi tạo lớp cha nào có thể truy cập đư ợc để gọi.

Machine Translated by Google

Mục 5: Tránh tạo các đối tượng không cần thiết

Thường thích hợp sử dụng lại một đối tượng duy nhất thay vì tạo một đối tượng tương đương đồng minh chức năng mới mỗi khi cần. Tái sử dụng có thể nhanh hơn và phong cách hơn. Một đối tượng luôn có thể được sử dụng lại nếu nó là bất biến (Mục 15).

Như một ví dụ điển hình về những điều không nên làm, hãy xem xét tuyên bố này:

```
String s = new String ("stringette"); // ĐỪNG LÀM VIỆC NÀY!
```

Câu lệnh tạo một cá thể Chuỗi mới mỗi khi nó được thực thi và không cần tạo đối tượng nào trong số đó. Đối số của chuỗi tạo tor ("stringette") bản thân nó là một cá thể Chuỗi , giống hệt về mặt chức năng với tất cả các đối tượng được tạo bởi hàm tạo. Nếu việc sử dụng này xảy ra trong một vòng lặp hoặc trong một phương thức được gọi thường xuyên, hàng triệu phiên bản Chuỗi có thể được tạo ra một cách không cần thiết.

Phiên bản cải tiến chỉ đơn giản là sau:

```
String s = "stringette";
```

Phiên bản này sử dụng một cá thể Chuỗi duy nhất , thay vì tạo một cá thể mới mỗi khi nó được thực thi. Hơn nữa, nó được đảm bảo rằng đối tượng sẽ được sử dụng lại bởi bất kỳ mã nào khác đang chạy trong cùng một máy ảo xảy ra với cùng một chuỗi ký tự [JLS, 3.10.5].

Bạn thường có thể tránh tạo các đối tượng không cần thiết bằng cách sử dụng các mã nguồn gốc của nhà máy tĩnh (Mục 1) ưu tiên cho các hàm tạo trên các lớp bất biến cung cấp cả hai. Ví dụ, phương thức nhà máy tĩnh Boolean.valueOf (String) hầu như luôn được ưu tiên hơn phương thức khởi tạo Boolean (String). Hàm tạo tạo một đối tượng mới mỗi khi nó được gọi, trong khi phương thức static factory không bao giờ bắt buộc phải làm như vậy và sẽ không xảy ra trong thực tế.

Ngoài việc tái sử dụng các đối tượng bất biến, bạn cũng có thể sử dụng lại các đối tượng có thể thay đổi nếu bạn biết chúng sẽ không bị sửa đổi. Đây là một ví dụ tinh tế hơn một chút và phổ biến hơn nhiều về những việc không nên làm. Nó liên quan đến các đối tượng Ngày có thể thay đổi không bao giờ được sửa đổi sau khi các giá trị của chúng đã được tính toán. Lớp này lập mô hình một người và có phương thức isBabyBoomer cho biết người đó có phải là “baby boomer” hay không, hay nói cách khác là người đó có sinh từ năm 1946 đến năm 1964 hay không:

```
public class Person
    {private final Ngày sinh Ngày;

    // Các trường, phương thức và hàm tạo khác bị bỏ qua
```

Machine Translated by Google


```
        // ĐỪNG LÀM VIỆC NÀY!
        public boolean isBabyBoomer () { // Phân
             bổ không cần thiết đối tượng đắt tiền Calendar gmtCal =
             Calendar.getInstance (TimeZone.getTimeZone ("GMT"));

             gmtCal.set (1946, Calendar.JANUARY, 1, 0, 0, 0); Ngày
             boomStart = gmtCal.getTime (); gmtCal.set (1965,
             Calendar.JANUARY, 1, 0, 0, 0); Ngày boomEnd = gmtCal.getTime
             (); return BirthDate.compareTo (boomStart)> = 0 &&
             BirthDate.compareTo (boomEnd) <0;

        }
    }
```

Phương thức isBabyBoomer tạo một Lịch, TimeZone và hai phiên bản Ngày mới một cách không cần thiết mỗi khi nó được gọi. Phiên bản sau sẽ tránh được sự kém hiệu quả này với trình khởi tạo tĩnh:

```
    class Person
        {private final Ngày sinh Ngày;
        // Các trường, phương thức và hàm tạo khác bị bỏ qua

        /
         ** * Ngày bắt đầu và kết thúc của thời kỳ bùng nổ trẻ sơ  sinh.
         * / private static Ngày cuối cùng BOOM_START; ngày cuối cùng
        tĩnh riêng BOOM_END;


        static
             {Calendar gmtCal =
                  Calendar.getInstance (TimeZone.getTimeZone ("GMT"));
             gmtCal.set (1946, Calendar.JANUARY, 1, 0, 0, 0); BOOM_START =
             gmtCal.getTime (); gmtCal.set (1965, Calendar.JANUARY, 1, 0,
             0, 0); BOOM_END = gmtCal.getTime ();

        }

        public boolean isBabyBoomer () {return
             BirthDate.compareTo (BOOM_START)> = 0 && BirthDate.compareTo
                      (BOOM_END) <0;
        }
    }
```

Phiên bản cải tiến của lớp Person chỉ tạo các phiên bản Calendar, TimeZone và Date một lần khi nó được khởi tạo, thay vì tạo chúng mỗi khi isBabyBoomer được gọi. Điều này dẫn đến hiệu suất tăng đáng kể nếu

Machine Translated by Google

phương thức được gọi thường xuyên. Trên máy của tôi, phiên bản gốc mất 32.000 mili giây cho 10 triệu lần gọi, trong khi phiên bản cải tiến mất 130 mili giây, nhanh hơn khoảng 250 lần. Không chỉ hiệu suất được cải thiện mà độ rõ nét cũng vậy.
Việc thay đổi boomStart và boomEnd từ các biến cục bộ thành các trường cuối cùng tĩnh làm rõ ràng rằng những ngày này được coi là hằng số, giúp mã dễ hiểu hơn. Vì lợi ích của việc tiết lộ đầy đủ, khoản tiết kiệm từ loại tối ưu hóa này sẽ không phải lúc nào cũng ấn tượng như vậy, vì các phiên bản Lịch đặc biệt đắt để tạo.

Nếu phiên bản cải tiến của lớp Person được khởi chạy nhưng phương thức isBabyBoomer của nó không bao giờ được gọi, các trường BOOM_START và BOOM_END sẽ được khởi chạy một cách không cần thiết. Có thể loại bỏ các khởi tạo không cần thiết bằng cách khởi tạo các trường này một cách lười biếng (Mục 71) lần đầu tiên phương thức isBabyBoomer được gọi, nhưng nó không được khuyến khích. Như thường lệ với khởi tạo lười biếng, nó sẽ làm phức tạp việc triển khai và không có khả năng dẫn đến một thông báo có thể cải thiện hiệu suất vượt quá những gì chúng tôi đã đạt được (Mục 55).

Trong các ví dụ trước trong mục này, rõ ràng là các đối tượng trong question có thể được sử dụng lại vì chúng không bị sửa đổi sau khi khởi tạo. Có những tình huống khác mà nó ít rõ ràng hơn. Hãy xem xét trường hợp của bộ điều hợp [Gamma95, tr. 139], còn được gọi là lượt xem. Bộ điều hợp là một đối tượng ủy quyền cho một đối tượng hỗ trợ, cung cấp một giao diện thay thế cho đối tượng hỗ trợ. Vì một bộ điều hợp không có trạng thái nào ngoài trạng thái của đối tượng hỗ trợ của nó, nên không cần tạo nhiều hơn một phiên bản của bộ điều hợp nhất định cho một đối tượng nhất định.

Ví dụ, phương thức keySet của giao diện Bản đồ trả về dạng xem Tập hợp của đối tượng Bản đồ , bao gồm tất cả các khóa trong bản đồ. Nói một cách ngây thơ , có vẻ như mọi lệnh gọi đến keySet sẽ phải tạo một đối tượng Set mới , nhưng mọi lệnh gọi đến keySet trên một đối tượng Bản đồ nhất định có thể trả về cùng một đối tượng Set . Mặc dù cá thể Set trả về thường có thể thay đổi, tất cả các đối tượng được trả về đều có chức năng identical: khi một trong các đối tượng trả về thay đổi, tất cả các đối tượng khác cũng vậy vì chúng đều được hỗ trợ bởi cùng một cá thể Bản đồ . Mặc dù việc tạo nhiều phiên bản của đối tượng keySet view là vô hại nhưng cũng không cần thiết.

Có một cách mới để tạo các đối tượng không cần thiết trong bản phát hành 1.5. Nó được gọi là autoboxing, và nó cho phép lập trình viên kết hợp các loại nguyên thủy nguyên thủy và đóng hộp, tự động đấm bốc và mở hộp khi cần thiết. Tự động đóng hộp làm mờ nhưng không xóa sự phân biệt giữa các loại nguyên thủy nguyên thủy và đóng hộp. Có sự khác biệt tinh tế về ngữ nghĩa và sự khác biệt về hiệu suất không quá tinh tế (Mục 49).
Hãy xem xét chương trình sau, chương trình này tính tổng của tất cả các số nguyên dương

Machine Translated by Google

các giá trị. Để làm điều này, chương trình phải sử dụng số học dài , bởi vì số int không đủ lớn để chứa tổng của tất cả các giá trị int dương :

```
// Chương trình chậm ghê gớm! Bạn có thể phát hiện ra đối tượng tạo ra không?
public static void main (String [] args) {Long sum = 0L; for (long i = 0; i
     <Integer.MAX_VALUE; i ++) {sum + = i;


     }
     System.out.println (sum);
}
```

Chương trình này đưa ra câu trả lời đúng, nhưng nó chậm hơn nhiều so với bình thường, do lỗi đánh máy một ký tự. Tổng biến được khai báo là Long thay vì dài, có nghĩa là chương trình xây dựng khoảng 231 trường hợp Dài không cần thiết (khoảng một cho mỗi lần dài i được thêm vào tổng dài).
Thay đổi khai báo tổng từ Dài thành dài sẽ giảm thời gian chạy từ 43 giây xuống 6,8 giây trên máy của tôi. Bài học rất rõ ràng: thích nguyên thủy hơn nguyên thủy đóng hộp và đề phòng việc tự động đóng hộp không chủ ý.

Mục này không được hiểu sai để ngụ ý rằng việc tạo đối tượng là hết sức hiệu quả và nên tránh. Ngược lại, việc tạo và cải tạo các đối tượng nhỏ mà các hàm tạo không thực hiện một cách rõ ràng nào là rẻ, đặc biệt là trên các triển khai JVM hiện đại. Tạo các đối tượng bổ sung để nâng cao độ rõ ràng, đơn giản hoặc sức mạnh của một chương trình nói chung là một điều tốt.

Ngược lại, tránh tạo đối tượng bằng cách duy trì nhóm đối tượng của riêng bạn là một ý tưởng tồi trừ khi các đối tượng trong nhóm cực kỳ nặng. Ví dụ cổ điển về một đối tượng biện minh cho một nhóm đối tượng là một kết nối cơ sở dữ liệu. Chi phí thiết lập kết nối đủ cao để có thể sử dụng lại các đối tượng này. Ngoài ra, giấy phép cơ sở dữ liệu của bạn có thể giới hạn bạn ở một số lỗi cố định. Tuy nhiên, nói chung, việc duy trì các nhóm đối tượng của riêng bạn sẽ làm vướng víu mã của bạn, làm tăng dung lượng bộ nhớ và gây hại cho hiệu suất. Các triển khai JVM hiện đại có các bộ thu gom rác được tối ưu hóa cao, dễ dàng thực hiện các nhóm đối tượng như vậy trên các đối tượng nhẹ.

Điểm đối nghịch với vật phẩm này là Vật phẩm 39 về sao chép phòng thủ. Mục 5 nói, "Không tạo một đối tượng mới khi bạn nên sử dụng lại một đối tượng hiện có", trong khi Mục 39 nói, "Không sử dụng lại một đối tượng hiện có khi bạn nên tạo một đối tượng mới." Lưu ý rằng hình phạt cho việc sử dụng lại một đối tượng khi sao chép phòng thủ được yêu cầu lớn hơn nhiều so với hình phạt cho việc tạo ra một đối tượng trùng lặp một cách không cần thiết. Không tạo các bản sao phòng thủ khi cần thiết có thể dẫn đến các lỗi ngấm ngầm và lỗ hổng bảo mật; các đối tượng creating không cần thiết chỉ ảnh hưởng đến phong cách và hiệu suất.

Machine Translated by Google

Mục 6: Loại bỏ các tham chiếu đối tư ợng lỗi thời

Khi bạn chuyển từ một ngôn ngữ có quản lý bộ nhớ thủ công, chẳng hạn như C hoặc C ++, sang một ngôn ngữ đư ợc thu thập rác, công việc của bạn với tư cách là một lập trình viên sẽ dễ dàng hơ n nhiều bởi thực tế là các đối tư ợng của bạn sẽ tự động đư ợc lấy lại khi bạn sử dụng chúng. Nó có vẻ giống như một phép thuật khi bạn lần đầu tiên trải nghiệm nó. Nó có thể dễ dàng dẫn đến ấn tư ợng rằng bạn không cần phải nghĩ đến việc quản lý bộ nhớ, như ng điều này không hoàn toàn đúng.

Hãy xem xét cách triển khai ngăn xếp đơ n giản sau:

```
// Bạn có thể phát hiện ra "bộ nhớ bị rò rỉ"
không? public class Stack {private Object []
      các phần tử; int size = 0; private
      static final int DEFAULT_INITIAL_CAPACITY
      = 16;

      public Stack ()
            {element = new Object [DEFAULT_INITIAL_CAPACITY];
      }

      public void push (Object e)
            {ensureCapacity (); các phần tử
            [size ++] = e;
      }

      public Object pop () {if
            (size == 0) throw new
                  EmptyStackException (); trả về các phần tử
            [- size];
      }

      /
       ** * Đảm bảo không gian cho ít nhất một phần tử nữa, gần như * tăng
       gấp đôi dung lư ợng mỗi khi mảng cần phát triển. * / private void
       ensureCapacity () {if (Elements.length == size) Elements = Arrays.copyOf
      (Elements, 2 * size + 1);


      }
}
```

Rõ ràng là không có gì sai với chư ơ ng trình này (như ng hãy xem Mục 26 để biết phiên bản chung). Bạn có thể kiểm tra nó một cách toàn diện và nó sẽ vư ợt qua mọi bài kiểm tra với màu sắc bay, như ng có một vấn đề đang rình rập. Nói một cách dễ hiểu, chư ơ ng trình bị "rò rỉ bộ nhớ", có thể âm thầm biểu hiện thành hiệu suất giảm do

Machine Translated by Google

tăng hoạt động của bộ thu gom rác hoặc tăng dung lượng bộ nhớ. Trong những trường hợp cực đoan, những rò rỉ bộ nhớ như vậy có thể gây phân trang đĩa và thậm chí là lỗi chương trình với OutOfMemoryError, nhưng những lỗi như vậy tương đối hiếm.

Vậy rò rỉ bộ nhớ là do đâu? Nếu một ngăn xếp lớn lên và sau đó thu hẹp lại, các đối tượng đã bật ra khỏi ngăn xếp sẽ không được thu gom rác, ngay cả khi chương trình sử dụng ngăn xếp không còn tham chiếu đến chúng nữa. Điều này là do ngăn xếp chính có các tham chiếu lỗi thời đến các đối tượng này. Một tham chiếu lỗi thời chỉ đơn giản là một tham chiếu sẽ không bao giờ được tham chiếu nữa. Trong trường hợp này, bất kỳ tham chiếu nào bên ngoài "phần hoạt động" của mảng phần tử đều đã lỗi thời. Phần hoạt động bao gồm các phần tử có chỉ mục nhỏ hơn kích thước.

Rò rỉ bộ nhớ trong các ngôn ngữ được thu thập rác (được gọi đúng hơn là sự cố rút lui đối tượng không chủ định) là ngầm ngầm. Nếu một tham chiếu đối tượng vô tình được giữ lại, không chỉ đối tượng đó bị loại trừ khỏi bộ sưu tập rác mà còn bất kỳ đối tượng nào được tham chiếu bởi đối tượng đó, v.v. Ngay cả khi chỉ có một số đối tượng tham chiếu vô tình được giữ lại, rất nhiều đối tượng có thể bị ngăn cản việc thu gom rác, có khả năng ảnh hưởng lớn đến hiệu suất.

Cách khắc phục cho loại vấn đề này rất đơn giản: loại bỏ các tham chiếu khi chúng trở nên lỗi thời. Trong trường hợp của lớp Stack của chúng ta , tham chiếu đến một mục sẽ trở nên lỗi thời ngay khi nó xuất hiện khỏi ngăn xếp. Phiên bản đã sửa của phương thức pop trông như sau:

```
public Object pop () {if
     (size == 0) throw new
          EmptyStackException ();
     Kết quả đối tượng = các phần tử [- size];
     phần tử [size] = null; // Loại bỏ kết quả trả về tham chiếu lỗi thời;

}
```

Một lợi ích bổ sung của việc loại bỏ các tham chiếu lỗi thời là, nếu sau đó chúng được tham chiếu do nhầm lẫn, chương trình sẽ ngay lập tức bị lỗi với NullPointerException, thay vì lặng lẽ làm điều sai trái. Luôn luôn có lợi khi phát hiện ra các lỗi lập trình càng nhanh càng tốt.

Khi các lập trình viên lần đầu tiên gặp phải vấn đề này, họ có thể bù đắp quá mức bằng cách loại bỏ mọi tham chiếu đối tượng ngay sau khi chương trình kết thúc sử dụng nó. Điều này không cần thiết và cũng không mong muốn, vì nó làm lộn xộn chương trình một cách không cần thiết. Việc hủy bỏ các tham chiếu đối tượng nên là ngoại lệ chứ không phải là chuẩn. Cách tốt nhất để loại bỏ một tham chiếu lỗi thời là để biến chứa tham chiếu nằm ngoài phạm vi. Điều này xảy ra một cách tự nhiên nếu bạn xác định từng biến trong phạm vi hẹp nhất có thể (Mục 45).

Machine Translated by Google

Vì vậy, khi nào bạn nên hủy bỏ một tham chiếu? Khía cạnh nào của lớp Stack làm cho nó dễ bị rò rỉ bộ nhớ? Nói một cách đơn giản, nó quản lý bộ nhớ của chính nó. Nhóm lưu trữ bao gồm các phần tử của mảng phần tử (đối tượng tham chiếu ô ence, không phải chính các đối tượng). Các phần tử trong phần hoạt động của mảng (như đã định nghĩa trước đó) được cấp phát và những mảng trong phần còn lại của mảng là miễn phí. Người thu gom rác không có cách nào biết được điều này; đến người thu gom rác, tất cả các tham chiếu đối tượng trong mảng phần tử đều có giá trị như nhau. Chỉ có chuyên gia grammer mới biết rằng phần không hoạt động của mảng là không quan trọng. Máy ép rác chuyên nghiệp thông báo hiệu quả thực tế này với người thu gom rác bằng cách thủ công vô hiệu hóa các phần tử mảng ngay khi chúng trở thành một phần của phần không hoạt động.

Nói chung, bất cứ khi nào một lớp quản lý bộ nhớ của chính nó, người chấm điểm chuyên nghiệp nên cảnh báo về việc rò rỉ bộ nhớ. Bất cứ khi nào một phần tử được giải phóng, bất kỳ các tham chiếu đối tượng có trong phần tử phải được bỏ trống.

Một nguồn rò rỉ bộ nhớ phổ biến khác là bộ nhớ đệm. Một khi bạn đặt một tham chiếu đối tượng vào bộ nhớ cache, thật dễ dàng quên rằng nó ở đó và để nó trong bộ nhớ cache lâu sau khi nó trở nên không liên quan. Có một số giải pháp cho vấn đề này. Nếu bạn đủ may mắn để triển khai một bộ đệm ẩn mà một mục nhập có liên quan chính xác miễn là có các tham chiếu đến khóa của nó bên ngoài bộ nhớ cache, hãy đại diện cho bộ nhớ cache như một WeakHashMap; các mục nhập sẽ tự động bị xóa sau khi chúng trở thành obso lete. Hãy nhớ rằng WeakHashMap chỉ hữu ích nếu thời gian tồn tại mong muốn của bộ nhớ cache các mục nhập được xác định bởi các tham chiếu bên ngoài tới khóa, không phải giá trị.

Thông thường hơn, thời gian tồn tại hữu ích của mục nhập bộ nhớ cache ít được xác định rõ hơn, với các mục nhập trở nên ít giá trị hơn theo thời gian. Trong những trường hợp này, bộ nhớ cache đôi khi nên được làm sạch các mục đã không sử dụng. Điều này có thể là được thực hiện bởi một luồng nền (có thể là một tor Timer hoặc SchedisedThreadPoolExecu) hoặc như một tác dụng phụ của việc thêm các mục mới vào bộ nhớ cache. Bản đồ liên kết lớp tạo điều kiện thuận lợi cho cách tiếp cận sau với phương thức removeEldestEntry của nó . Vì bộ nhớ đệm phức tạp hơn, bạn có thể cần sử dụng trực tiếp java.lang.ref .

Nguồn rò rỉ bộ nhớ phổ biến thứ ba là người nghe và các cuộc gọi lại khác. Nếu bạn triển khai một API trong đó khách hàng đăng ký các lệnh gọi lại nhưng không hủy đăng ký chúng một cách rõ ràng, chúng sẽ tích lũy trừ khi bạn thực hiện một số hành động. Cách tốt nhất để đảm bảo rằng các cuộc gọi lại được thu thập kịp thời là chỉ lưu trữ các tham chiếu yếu đối với chúng, chẳng hạn, bằng cách chỉ lưu trữ chúng dưới dạng các khóa trong WeakHashMap.

Bởi vì rò rỉ bộ nhớ thường không biểu hiện thành các ures lỗi rõ ràng, chúng có thể vẫn tồn tại trong hệ thống trong nhiều năm. Chúng thường được phát hiện chỉ là kết quả của việc kiểm tra mã cẩn thận hoặc với sự hỗ trợ của công cụ gỡ lỗi được biết đến như một trình biên dịch đống. Vì vậy, chúng tôi rất mong muốn học cách lường trước những lỗi xảy ra như thế này trước khi chúng xảy ra và ngăn ngừa chúng xảy ra.

Machine Translated by Google

Mục 7: Tránh những người hoàn thiện

Finalizers không thể đoán trước, thường nguy hiểm và nói chung là không cần thiết. Việc sử dụng chúng có thể gây ra các hành vi thất thường, hiệu suất kém và các vấn đề về tính di động. Công cụ hoàn thiện có một số cách sử dụng hợp lệ mà chúng tôi sẽ đề cập sau trong mục này, nhưng theo nguyên tắc chung, bạn nên tránh công cụ hoàn thiện.

Các lập trình viên C ++ được cảnh báo là không nên coi các trình hoàn thiện giống như các trình hủy C ++ tương tự của Java. Trong C ++, hàm hủy là cách bình thường để lấy lại các tài nguyên được liên kết với một đối tượng, một đối tác cần thiết cho các hàm tạo. Trong Java, bộ sưu tập gar bage lấy lại dung lượng lưu trữ được liên kết với một đối tượng khi nó không thể truy cập được, không cần nỗ lực đặc biệt nào từ phía lập trình viên. Các trình hủy C ++ cũng được sử dụng để lấy lại các tài nguyên phi bộ nhớ khác. Trong Java, khối try last thường được sử dụng cho mục đích này.

Một thiếu sót của các trình hoàn thiện là không có gì đảm bảo rằng chúng sẽ được thực thi nhanh chóng [JLS, 12.6]. Có thể mất nhiều thời gian tùy ý giữa thời gian mà một đối tượng trở nên không thể truy cập được và thời gian mà trình hoàn thiện của nó được thực thi. Điều này có nghĩa là bạn không bao giờ nên làm bất cứ điều gì quá quan trọng về thời gian trong quá trình hoàn thiện. Ví dụ, sẽ là một lỗi nghiêm trọng khi phụ thuộc vào trình hoàn thiện để đóng tệp, vì bộ mô tả tệp đang mở là một tài nguyên hạn chế. Nếu nhiều tệp bị bỏ ngỏ do JVM bị chậm trong quá trình thực thi các trình hoàn thiện, một chương trình có thể bị lỗi vì nó không thể mở tệp được nữa.

Tính nhanh chóng mà các trình hoàn thiện được thực thi chủ yếu là một chức năng của thuật toán thu gom rác, thuật toán này rất khác nhau từ việc triển khai JVM đến triển khai JVM. Hành vi của một chương trình phụ thuộc vào tốc độ nhanh chóng của việc thực thi trình hoàn thiện cũng có thể khác nhau. Hoàn toàn có khả năng một chương trình như vậy sẽ chạy hoàn hảo trên JVM mà bạn kiểm tra nó và sau đó thất bại thảm hại trên JVM được khách hàng quan trọng nhất của bạn ưa chuộng.

Việc hoàn thành trễ không chỉ là một vấn đề lý thuyết. Việc cung cấp trình hoàn thiện cho một lớp, trong những điều kiện hiếm hoi, có thể tự ý trì hoãn việc khôi phục các thể hiện của nó. Một đồng nghiệp đã gỡ lỗi ứng dụng GUI chạy lâu đang chết một cách bí ẩn bằng OutOfMemoryError. Phân tích tiết lộ rằng vào thời điểm nó chết, ứng dụng có hàng nghìn đối tượng đồ họa trên hàng đợi hoàn thiện của nó chỉ chờ được hoàn thiện và thu hồi. Thật không may, luồng hoàn thiện đang chạy ở mức độ ưu tiên thấp hơn luồng ứng dụng khác, vì vậy các đối tượng không được hoàn thiện ở tốc độ mà chúng đủ điều kiện để hoàn thiện. Đặc tả ngôn ngữ không đảm bảo về việc luồng nào sẽ thực thi các trình hoàn thiện, vì vậy không có cách nào khả thi để ngăn chặn loại vấn đề này ngoài việc hạn chế sử dụng trình hoàn thiện.

Đặc tả ngôn ngữ không chỉ không đảm bảo rằng các trình hoàn thiện sẽ được thực thi ngay lập tức; nó không đảm bảo rằng chúng sẽ được thực thi tại

Machine Translated by Google

tất cả. Hoàn toàn có thể xảy ra, thậm chí có khả năng một chương trình kết thúc mà không thực thi các trình hoàn thiện trên một số đối tượng không thể truy cập được nữa. Do đó, bạn không bao giờ nên phụ thuộc vào một trình hoàn thiện để cập nhật trạng thái liên tục quan trọng. Ví dụ: tùy thuộc vào trình hoàn thiện để phát hành khóa liên tục trên tài nguyên được chia sẻ như cơ sở dữ liệu là một cách tốt để đưa toàn bộ hệ thống phân tán của bạn tạm dừng.

Đừng bị quyến rũ bởi các phương pháp System.gc và System.runFinalization. Họ có thể làm tăng tỷ lệ người hoàn thành được thực hiện, nhưng họ không chấp nhận điều đó. Các phương thức duy nhất yêu cầu đảm bảo quá trình hoàn thiện là System.runFi nalizersOnExit và người anh em sinh đôi xấu xa của nó, Runtime.runFinalizersOnExit. Các phương pháp này có sai sót nghiêm trọng và đã không được dùng nữa [ThreadStop].

Trong trường hợp bạn vẫn chưa tin rằng nên tránh trình hoàn thiện, đây là một mẹo nhỏ khác đáng xem xét: nếu một ngoại lệ chưa được thông báo được đưa ra trong quá trình ization cuối cùng, thì ngoại lệ đó sẽ bị bỏ qua và việc hoàn thiện đối tượng đó sẽ kết thúc [JLS, 12.6]. Các ngoại lệ chưa được lựa chọn có thể khiến các đối tượng ở trạng thái bị hỏng. Nếu một luồng khác cố gắng sử dụng một đối tượng bị hỏng như vậy, hành vi không xác định tùy ý có thể dẫn đến. Thông thường, một ngoại lệ không được ghi chú sẽ kết thúc chuỗi và in dấu vết ngăn xếp, nhưng không phải nếu nó xảy ra trong trình hoàn thiện — nó thậm chí sẽ không in cảnh báo.

Ồ, và một điều nữa: có một hình phạt hiệu suất nghiêm trọng đối với việc sử dụng công cụ hoàn thiện. Trên máy của tôi, thời gian để tạo và hủy một đối tượng đơn giản là khoảng 5,6 ns. Thêm một công cụ hoàn thiện sẽ tăng thời gian lên 2.400 ns. Nói cách khác, việc tạo và phá hủy các đối tượng bằng các trình hoàn thiện chậm hơn khoảng 430 lần.

Vì vậy, bạn nên làm gì thay vì viết một trình hoàn thiện cho một lớp có các đối tượng đóng gói các tài nguyên yêu cầu kết thúc, chẳng hạn như tệp hoặc luồng? Chỉ giới thiệu một phương thức kết thúc rõ ràng và yêu cầu các máy khách của lớp gọi phương thức này trên mỗi phiên bản khi nó không còn cần thiết nữa. Một chi tiết đáng được đề cập trong đó là phiên bản phải theo dõi xem nó đã bị kết thúc hay chưa: phương thức kết thúc rõ ràng phải ghi lại trong trường riêng tư rằng đối tượng không còn hợp lệ và các phương thức khác phải kiểm tra trường này và ném Ngoại lệ trạng thái bất hợp pháp nếu chúng được gọi sau khi đối tượng đã được kết thúc.

Ví dụ điển hình về các phương thức kết thúc rõ ràng là các phương thức đóng trên InputStream, OutputStream và java.sql.Connection. Một ví dụ khác là phương thức hủy trên java.util.Timer, phương thức này thực hiện thay đổi trạng thái cần thiết để khiến luồng liên kết với một cá thể Bộ hẹn giờ tự kết thúc một cách nhẹ nhàng. Ví dụ từ java.awt bao gồm tư thế Graphics.dispose và Window.dis. Những phương pháp này thường bị bỏ qua, với những hậu quả về hiệu suất có thể đoán trước được. Một phương pháp liên quan là Image.flush, phương thức này phân bổ tất cả

Machine Translated by Google

tài nguyên đư ợc liên kết với một cá thể Hình ảnh như ng để nó ở trạng thái vẫn có thể đư ợc sử dụng, phân bổ lại tài nguyên nếu cần.

Các phư ơ ng pháp kết thúc rõ ràng thư ờng đư ợc sử dụng kết hợp với cấu trúc try-last để đảm bảo kết thúc. Việc gọi phư ơ ng thức kết thúc rõ ràng bên trong mệnh đề cuối cùng đảm bảo rằng nó sẽ đư ợc thực thi ngay cả khi một ngoại lệ đư ợc ném ra trong khi đối tư ợng đang đư ợc sử dụng:

```
// khối try-last đảm bảo thực thi phư ơ ng thức kết thúc Foo foo = new Foo (...);
thử {// Làm những gì phải làm với foo


     ...
} cuối cùng {
     foo.termina (); // Phư ơ ng thức kết thúc rõ ràng
}
```

Vì vậy, những gì, nếu có, là những gì tốt cho những gì? Có lẽ có hai cách sử dụng hợp pháp. Một là hoạt động như một "mạng lư ới an toàn" trong trư ờng hợp chủ sở hữu của một đối tư ợng quên gọi phư ơ ng thức chấm dứt rõ ràng của nó. Mặc dù không có gì đảm bảo rằng trình hoàn thiện sẽ đư ợc gọi ngay lập tức, như ng có thể tốt hơ n là giải phóng tài nguyên muộn hơ n là không bao giờ, trong những trư ờng hợp (hy vọng là hiếm) khi khách hàng không gọi đư ợc phư ơ ng thức kết thúc rõ ràng. Như ng trình hoàn thiện sẽ ghi lại một cảnh báo nếu nó phát hiện ra rằng tài nguyên vẫn chư a bị kết thúc, vì điều này cho thấy một lỗi trong mã máy khách, cần đư ợc khắc phục. Nếu bạn đang cân nhắc viết một bản tổng kết mạng lư ới an toàn như vậy, hãy suy nghĩ rất lâu và cẩn thận về việc liệu biện pháp bảo vệ bổ sung có xứng đáng với chi phí bổ sung hay không.

Bốn lớp đư ợc trích dẫn làm ví dụ về mẫu phư ơ ng thức kết thúc rõ ràng (FileInputStream, FileOutputStream, Timer và Connection) có các trình hoàn thiện đóng vai trò là mạng an toàn trong trư ờng hợp các phư ơ ng thức kết thúc của chúng không đư ợc gọi. Thật không may, những trình hoàn thiện này không ghi cảnh báo. Những cảnh báo như vậy thư ờng không thể đư ợc thêm vào sau khi một API đư ợc xuất bản, vì nó có vẻ như sẽ phá vỡ các ứng dụng khách hiện có.

Cách sử dụng hợp pháp thứ hai của trình hoàn thiện liên quan đến các đối tư ợng với các đối tư ợng bản địa. Một ngang hàng gốc là một đối tư ợng gốc mà một đối tư ợng bình thư ờng ủy quyền thông qua các ods meth gốc. Bởi vì một đối tư ợng ngang hàng gốc không phải là một đối tư ợng bình thư ờng, trình thu gom rác không biết về nó và không thể lấy lại nó khi đối tư ợng ngang hàng Java của nó đư ợc thu hồi. Trình hoàn thiện là phư ơ ng tiện thích hợp để thực hiện nhiệm vụ này, giả sử ứng dụng ngang hàng gốc không có tài nguyên quan trọng nào. Nếu mạng ngang hàng gốc nắm giữ các tài nguyên phải đư ợc kết thúc ngay lập tức, thì lớp phải có một phư ơ ng thức kết thúc rõ ràng, như đư ợc mô tả ở trên. Phư ơ ng pháp kết thúc phải làm bất cứ điều gì đư ợc yêu cầu để giải phóng tài nguyên quan trọng. Phư ơ ng thức kết thúc có thể là một phư ơ ng thức gốc hoặc nó có thể gọi một phư ơ ng thức.

Machine Translated by Google

Điều quan trọng cần lư u ý là "chuỗi trình hoàn thiện" không đư ợc thực hiện tự động. Nếu một lớp (không phải Đối tư ợng) có bộ hoàn thiện và một lớp con ghi đè nó, bộ hoàn thiện lớp con phải gọi bộ hoàn thiện lớp cha theo cách thủ công. Bạn nên hoàn thiện lớp con trong một khối try và gọi trình hoàn thiện lớp con trong khối ing cuối cùng tư ơ ng ứng . Điều này đảm bảo rằng trình hoàn thiện lớp cha đư ợc thực thi ngay cả khi việc hoàn thiện lớp con ném ra một ngoại lệ và ngư ợc lại. Đây là cách nó trông.
Lư u ý rằng ví dụ này sử dụng chú thích Ghi đè (@Override), đã đư ợc thêm vào nền tảng trong bản phát hành 1.5. Bạn có thể bỏ qua Ghi đè chú thích ngay bây giờ hoặc xem Mục 36 để tìm hiểu ý nghĩa của chúng:

```
// Chuỗi hoàn thiện thủ công
@Override đư ợc bảo vệ void finalize () ném Có thể ném {
      try
          {... // Hoàn thiện trạng thái lớp con}
      cuối cùng { super.finalize ();

      }
}
```

Nếu một trình triển khai lớp con ghi đè một trình hoàn thiện lớp cha như ng quên gọi nó, trình hoàn thiện lớp cha sẽ không bao giờ đư ợc gọi. Có thể bảo vệ chống lại một lớp con bất cẩn hoặc độc hại như vậy với chi phí tạo ra một đối tư ợng bổ sung cho mọi đối tư ợng sẽ đư ợc hoàn thiện. Thay vì đặt trình hoàn thiện trên lớp yêu cầu hoàn thiện, hãy đặt trình hoàn thiện trên một lớp ẩn danh (Mục 22) có mục đích duy nhất là hoàn thiện phiên bản bao quanh của nó. Một thể hiện duy nhất của lớp ẩn danh, đư ợc gọi là ngư ời giám hộ trình hoàn thiện, đư ợc tạo cho mỗi thể hiện của lớp bao quanh. Bản sao kèm theo lư u trữ tham chiếu duy nhất đến ngư ời giám hộ trình hoàn thiện của nó trong trư ờng phiên bản riêng để ngư ời giám hộ trình hoàn thiện đủ điều kiện để hoàn thiện cùng lúc với bản sao kèm theo. Khi trình giám hộ đư ợc hoàn thiện, nó thực hiện hoạt động hoàn thiện mong muốn cho cá thể bao quanh, giống như thể trình hoàn thiện của nó là một phư ơ ng thức trên lớp bao quanh:

```
// Thành ngữ ngư ời giám hộ Finalizer
public class Foo {
    // Mục đích duy nhất của đối tư ợng này là hoàn thiện đối tư ợng Foo bên ngoài
    private final Object finalizerGuardian = new Object () {
        @Override đư ợc bảo vệ void finalize () ném Throwable {... // Hoàn thiện
            đối tư ợng Foo bên ngoài
        }

    }; ... // Phần còn lại bị bỏ qua
}
```

Machine Translated by Google

Lưu ý rằng lớp công khai, Foo, không có trình hoàn thiện (ngoài lớp tầm thường mà nó kế thừa từ Object), vì vậy không quan trọng liệu trình hoàn thiện lớp con có gọi super.finalize hay không. Kỹ thuật này nên được xem xét cho mọi lớp công cộng không chung cuộc có người hoàn thiện.

Tóm lại, không sử dụng công cụ hoàn thiện ngoại trừ như một mạng lưới an toàn hoặc để chấm dứt các tài nguyên bản địa không phi mã. Trong những trường hợp hiếm hoi mà bạn sử dụng trình hoàn thiện, hãy nhớ gọi super.finalize. Nếu bạn sử dụng trình hoàn thiện như một mạng lưới an toàn, hãy nhớ ghi lại việc sử dụng không hợp lệ từ trình hoàn thiện. Cuối cùng, nếu bạn cần liên kết trình hoàn thiện với một lớp công khai, không chung cuộc, hãy cân nhắc sử dụng người giám hộ trình hoàn thiện, để quá trình hoàn thiện có thể diễn ra ngay cả khi người hoàn thiện lớp con không gọi được super.finalize.

Machine Translated by Google

Trang này cố ý để trống

Machine Translated by Google

CHƯƠNG 3

Các phương pháp chung cho tất cả các đối tượng

MẶC DÙ Đối tượng là một lớp cụ thể, nó được thiết kế chủ yếu để mở rộng. Tất cả các phương thức không chính thức của nó (bằng, mã băm, toString, sao chép và kết thúc) đều có các hợp đồng chung rõ ràng vì chúng được thiết kế để ghi đè. Trách nhiệm của bất kỳ lớp nào ghi đè các phương thức này là tuân theo các đặc điểm chung của chúng; không làm như vậy sẽ ngăn các lớp khác phụ thuộc vào các hợp đồng (chẳng hạn như HashMap và HashSet) hoạt động bình thường cùng với lớp.

Chương này cho bạn biết khi nào và làm thế nào để ghi đè các phương thức đối tượng không phải là đối tượng cuối cùng . Phương thức finalize bị bỏ qua trong chương này vì nó đã được thảo luận trong Mục 7. Mặc dù không phải là một phương thức Object , nhưng phương thức Comp Compare.compareTo được thảo luận trong chương này vì nó có một ký tự tương tự.

Khoản 8: Tuân theo hợp đồng chung khi ghi đè bằng

Ghi đè phương thức bằng có vẻ đơn giản, nhưng có nhiều cách làm sai và hậu quả có thể rất thảm khốc. Cách dễ nhất để tránh các vấn đề là không ghi đè phương thức equals , trong trường hợp này mỗi thể hiện của lớp chỉ bằng với chính nó. Đây là điều đúng đắn cần làm nếu áp dụng bất kỳ điều kiện nào sau đây:

- Mỗi thể hiện của lớp vốn là duy nhất. Điều này đúng với các lớp chẳng hạn như Luồng đại diện cho các thực thể đang hoạt động hơn là các giá trị. Sự cố vấn bằng imple được cung cấp bởi Object có chính xác hành vi phù hợp cho các lớp này.
- Bạn không quan tâm đến việc lớp học có cung cấp một bài kiểm tra "bình đẳng logic" hay không. Ví dụ: java.util.Random có thể đã ghi đè bằng để kiểm tra xem liệu hai phiên bản Ngẫu nhiên có tạo ra cùng một chuỗi số ngẫu nhiên về sau hay không, nhưng các nhà thiết kế không nghĩ rằng khách hàng sẽ cần hoặc muốn chức năng này. Trong những trường hợp này, việc triển khai bằng được kế thừa từ Đối tượng là phù hợp.

Machine Translated by Google

- Một lớp cha đã được ghi đè bằng bằng, và hành vi của lớp cha phù hợp với lớp này. Ví dụ: hầu hết các triển khai Set kế thừa triển khai bằng của chúng từ AbstractSet, triển khai Danh sách từ AbstractList và triển khai Bản đồ từ AbstractMap.

- Lớp là private hoặc package-private và bạn chắc chắn rằng phương thức bằng của nó sẽ không bao giờ được gọi. Có thể cho rằng, phương thức bằng nên được chạy quá mức trong những trường hợp này, trong trường hợp nó vô tình được gọi:

```
@Override public boolean bằng (Đối tượng o) {throw new
     AssertionError (); // Phương thức không bao giờ được gọi
}
```

Vì vậy, khi nào là thích hợp để ghi đè Object.equals? Khi một lớp có khái niệm bình đẳng logic khác với nhận dạng đối tượng đơn thuần và lớp cha chưa ghi đè bằng để thực hiện hành vi mong muốn. Đây là trường hợp chung cho các lớp giá trị. Một lớp giá trị chỉ đơn giản là một lớp đại diện cho một giá trị, chẳng hạn như Số nguyên hoặc Ngày. Một lập trình viên so sánh các tham chiếu với các đối tượng giá trị bằng cách sử dụng phương thức bằng sẽ tìm hiểu xem liệu chúng có tương đương về mặt logic hay không, chứ không phải liệu chúng có tham chiếu đến cùng một đối tượng hay không. Không chỉ ghi đè phương thức bằng cần thiết để đáp ứng kỳ vọng của lập trình viên; nó cho phép các cá thể đóng vai trò là khóa bản đồ hoặc tập hợp các phần tử với hành vi mong muốn, có thể dự đoán được.

Một loại lớp giá trị không yêu cầu phương thức equals được ghi đè den là lớp sử dụng điều khiển phiên bản (Mục 1) để đảm bảo rằng có nhiều nhất một đối tượng tồn tại với mỗi giá trị. Các loại enum (Mục 30) thuộc loại này. Đối với các lớp này, bình đẳng logic giống như nhận dạng đối tượng, do đó, phương thức equals của đối tượng có chức năng như một phương thức bình đẳng logic .

Khi bạn ghi đè phương thức bằng , bạn phải tuân thủ vấn đề chung của nó đường dẫn nước. Đây là hợp đồng, được sao chép từ đặc tả cho Đối tượng [JavaSE6]:

Phương thức bằng thực hiện một quan hệ tương đương. Đó là: • Phản xạ:

Đối với bất kỳ giá trị tham chiếu không rỗng nào x, x.equals (x) phải trả về true. • Đối xứng:

Đối với mọi giá trị tham chiếu không rỗng x và y, x.equals (y) phải trở lại true nếu và chỉ khi y.equals (x) trả về true.

- Transitive: Đối với mọi giá trị tham chiếu không rỗng x, y, z, nếu x.equals (y) trả về true và y.equals (z) trả về true, thì x.equals (z) phải trả về true.

- Nhất quán: Đối với mọi giá trị tham chiếu không rỗng x và y, nhiều lệnh gọi x.equals (y) luôn trả về true hoặc nhất quán trả về false, với điều kiện là không có thông tin nào được sử dụng trong so sánh ngang bằng trên các đối tượng được sửa đổi.

- Đối với bất kỳ giá trị tham chiếu nào không phải null x, x.equals (null) phải trả về false.

Machine Translated by Google

Trừ khi bạn thiên về toán học, nếu không thì điều này có vẻ hơi đáng sợ, nhưng đừng bỏ qua nó! Nếu bạn vi phạm nó, bạn có thể thấy rằng chương trình của bạn hoạt động không tốt hoặc bị treo và có thể rất khó xác định nguồn gốc của lỗi.
Để diễn giải John Donne, không có lớp học nào là một hòn đảo. Các phiên bản của một lớp được chuyển một cách tự do sang lớp khác. Nhiều lớp, bao gồm tất cả các lớp tập hợp, phụ thuộc vào các đối tượng được chuyển cho chúng tuân theo hợp đồng bằng .

Bây giờ bạn đã nhận thức được sự nguy hiểm của việc vi phạm hợp đồng bình đẳng , chúng ta hãy đi tìm hiểu chi tiết về hợp đồng. Tin tốt là, hợp đồng thực sự không phức tạp lắm. Một khi bạn hiểu nó, không khó để tuân thủ nó. Chúng ta hãy lần lượt xem xét năm yêu cầu: Tính phản xạ — Yêu cầu đầu tiên chỉ đơn thuần nói rằng một đối tượng phải bằng chính nó.

Thật khó để tưởng tượng vi phạm yêu cầu này một cách vô ý. Nếu bạn vi phạm nó và sau đó thêm một thể hiện của lớp vào một bộ sưu tập, thì phương thức hàm chứa của bộ sưu tập có thể nói rằng bộ sưu tập không chứa cá thể mà bạn vừa thêm vào.

Tính đối xứng — Yêu cầu thứ hai nói rằng hai đối tượng bất kỳ phải đồng ý về việc chúng có bằng nhau hay không. Không giống như yêu cầu đầu tiên, không khó để tưởng tượng vio vô tình đánh bại yêu cầu này. Ví dụ, hãy xem xét lớp sau, lớp này triển khai một chuỗi không phân biệt chữ hoa chữ thường. Trường hợp của chuỗi được giữ nguyên bởi toString nhưng bị bỏ qua trong các so sánh:

```
// Bị hỏng - vi phạm tính đối xứng!
public final class CaseInsensitiveString {private final
      String s;

      public CaseInsensitiveString (String s) {if (s ==
            null) ném mới NullPointerException (); this.s =
                  s;

      }

      // Bị hỏng - vi phạm tính đối xứng!
      @Override public boolean equals (Object o) {if (o
            instanceof CaseInsensitiveString) return
                  s.equalsIgnoreCase (((CaseInsensitiveString)
                        o) .s);
            if (o instanceof String) // Khả năng tương tác một chiều! return
                  s.equalsIgnoreCase ((Chuỗi) o); trả về sai;

      }
      ... // Phần còn lại bị bỏ qua
}
```

Machine Translated by Google

Phư ơ ng thức bằng có chủ đích tốt trong lớp này cố gắng thực hiện một cách nguyên bản để tăng tỷ lệ tư ơ ng tác với các chuỗi thông thư ờng. Giả sử rằng chúng ta có một chuỗi phân biệt chữ hoa chữ thư ờng và một chuỗi thông thư ờng:

```
CaseInsensitiveString cis = new CaseInsensitiveString ("Tiếng Ba Lan");
String s = "đánh bóng";
```

Như mong đợi, (các) cis.equals trả về true. Vấn đề là trong khi phư ơ ng thức equals trong CaseInsensitiveString biết về các chuỗi thông thư ờng, thì phư ơ ng thức equals trong String lại không biết đến các chuỗi không phân biệt chữ hoa chữ thư ờng. Do đó , s.equals (cis) trả về false, một vi phạm rõ ràng về đối xứng. Giả sử bạn đặt một chuỗi không phân biệt chữ hoa chữ thư ờng vào một bộ sư u tập:

```
List <CaseInsensitiveString> list = new
     ArrayList <CaseInsensitiveString> (); list.add
(cis);
```

Tại thời điểm này (các) list.contains trả về cái gì? Ai biết? Trong quá trình triển khai tiền thuê nhà của Sun, nó sẽ trả về false, như ng đó chỉ là một tác phẩm triển khai. Trong một triển khai khác, nó có thể dễ dàng trả về true hoặc ném một ngoại lệ thời gian chạy. Một khi bạn đã vi phạm hợp đồng bình đẳng , bạn chỉ đơ n giản là không biết các đối tư ợng khác sẽ cư xử như thế nào khi đối đầu với đối tư ợng của bạn.

Để loại bỏ vấn đề, chỉ cần loại bỏ nỗ lực không hợp lý để tư ơ ng tác với String khỏi phư ơ ng thức bằng . Khi bạn thực hiện việc này, bạn có thể cấu trúc lại phư ơ ng thức để cung cấp cho nó một kết quả duy nhất:

```
@Override public boolean bằng (Đối tư ợng o) {
     return o instanceof CaseInsensitiveString &&
          ((CaseInsensitiveString) o) .s.equalsIgnoreCase (s);
}
```

Tính chuyển đổi – Yêu cầu thứ ba của hợp đồng bằng nói rằng nếu một đối tư ợng bằng một vật thứ hai và đối tư ợng thứ hai bằng một phần ba, thì đối tư ợng đầu tiên phải bằng đối tư ợng thứ ba. Một lần nữa, không khó để tư ởng tư ợng việc vô tình vi phạm yêu cầu này. Hãy xem xét trư ờng hợp một lớp con thêm một thành phần giá trị mới vào lớp cha của nó. Nói cách khác, lớp con thêm một phần thông tin

Machine Translated by Google

tion ảnh hưởng đến so sánh ngang bằng . Hãy bắt đầu với một lớp điểm số nguyên hai chiều bất biến đơn giản:

```
public class Point {private
      final int x; private cuối
      cùng int y; public Point
      (int x, int y) {this.x = x; this.y =
            y;

      }

      @Override public boolean bằng (Object o) {if (! (O
            instanceof Point)) return false; Điểm p = (Điểm) o;
                  trả về px == x && py == y;


      }

      ... // Phần còn lại bị bỏ qua
}
```

Giả sử bạn muốn mở rộng lớp này, thêm khái niệm về màu sắc vào một điểm:

```
lớp công khai ColorPoint mở rộng Điểm {màu sắc cuối
      cùng riêng tư ;

      public ColorPoint (int x, int y, Color color) {
            siêu (x, y);
            this.color = màu;
      }

      ... // Phần còn lại bị bỏ qua
}
```

Phương thức bằng nên trông như thế nào? Nếu bạn bỏ qua hoàn toàn, sự gợi ý imple được kế thừa từ Point và thông tin về màu sắc sẽ bị bỏ qua trong các phép so sánh ngang bằng . Mặc dù điều này không vi phạm hợp đồng bình đẳng , nhưng rõ ràng là không thể thực hiện được. Giả sử bạn viết một phương thức bằng chỉ trả về giá trị true nếu điểm đối số của nó là một điểm màu khác có cùng vị trí và màu sắc:

```
// Bị hỏng - vi phạm tính đối xứng!
@Override public boolean bằng (Object o) {if (! (O
      instanceof ColorPoint)) trả về false;

      trả về super.equals (o) && ((ColorPoint) o) .color == color;
}
```

Machine Translated by Google

Vấn đề với phương pháp này là bạn có thể nhận được các kết quả khác nhau khi so sánh một điểm với một điểm màu và ngược lại. So sánh trước bỏ qua màu sắc, trong khi so sánh sau luôn trả về false vì loại đối số không chính xác. Để tạo ra bê tông này, hãy tạo một điểm và một điểm màu:

```
Point p = new Point (1, 2);
ColorPoint cp = ColorPoint mới (1, 2, Color.RED);
```

Khi đó p.equals (cp) trả về true, trong khi cp.equals (p) trả về false. Bạn có thể cố gắng khắc phục sự cố bằng cách để ColorPoint.equals bỏ qua màu khi thực hiện "so sánh hỗn hợp":

```
// Bị hỏng - vi phạm độ nhạy!
@Override public boolean bằng (Object o) {if (! (O
      instanceof Point)) return false;


      // Nếu o là một Điểm bình thường, hãy thực hiện so sánh mù màu if (! (O
      instanceof ColorPoint)) return o.equals (this);


      // o là một ColorPoint; thực hiện so sánh đầy đủ trả về
      super.equals (o) && ((ColorPoint) o) .color == color;
}
```

Cách tiếp cận này thực sự cung cấp sự cân xứng, nhưng với chi phí của sự nhạy cảm:

```
ColorPoint p1 = ColorPoint mới (1, 2, Color.RED); Point p2 =
new Point (1, 2); ColorPoint p3 = ColorPoint mới (1, 2,
Color.BLUE);
```

Bây giờ p1.equals (p2) và p2.equals (p3) trả về true, trong khi p1.equals (p3) trả về false, một sự vi phạm rõ ràng về độ nhạy. Hai phép so sánh đầu tiên là "mù màu", trong khi phép so sánh thứ ba có tính đến màu sắc.

Vậy giải pháp là gì? Nó chỉ ra rằng đây là một vấn đề cơ bản của quan hệ tương đương trong ngôn ngữ hướng đối tượng. Không có cách nào để mở rộng một lớp có thể khởi tạo và thêm một thành phần giá trị trong khi vẫn bảo toàn hợp đồng bằng , trừ khi bạn sẵn sàng từ bỏ những lợi ích của trừu tượng hướng đối tượng.

Machine Translated by Google

Bạn có thể nghe nó nói rằng bạn có thể mở rộng một lớp có thể khởi tạo và thêm một thành phần giá trị trong khi vẫn bảo toàn hợp đồng bằng bằng cách sử dụng thử nghiệm getClass thay cho thử nghiệm instanceof trong phương thức bằng :

```
// Bị hỏng - vi phạm nguyên tắc thay thế Liskov (trang 40)
@Override public boolean bằng (Đối tượng o) {
      if (o == null || o.getClass ()! = getClass ()) return
            false; Điểm p = (Điểm) o; trả về px == x && py == y;


}
```

Điều này có tác dụng cân bằng các đối tượng chỉ khi chúng có cùng một lớp thực thi. Tuy điều này có vẻ không quá tệ nhưng hậu quả là không thể chấp nhận được.

Giả sử chúng ta muốn viết một phương thức để cho biết một điểm nguyên có nằm trên đường tròn đơn vị hay không. Đây là một cách chúng tôi có thể làm điều đó:

```
// Khởi tạo UnitCircle để chứa tất cả các Điểm trên vòng tròn đơn vị private static
final Set <Point> unitCircle; static {unitCircle = new HashSet <Point> ();
unitCircle.add (new Point (1, 0)); unitCircle.add (new Point (0, 1)); unitCircle.add
      (new Point (-1, 0)); unitCircle.add (new Point (0, -1));




}

public static boolean onUnitCircle (Point p) {return
      unitCircle.contains (p);
}
```

Mặc dù đây có thể không phải là cách nhanh nhất để triển khai chức năng, nhưng nó hoạt động tốt. Nhưng giả sử bạn mở rộng Point theo một cách nhỏ nhặt nào đó không thêm giá trị component, chẳng hạn bằng cách để hàm tạo của nó theo dõi số lượng phiên bản đã được tạo:

```
lớp công khai CounterPoint mở rộng Điểm {
      private static cuối cùng bộ đếm AtomicInteger =
            mới AtomicInteger ();

      public CounterPoint (int x, int y) {super (x,
            y); counter.incrementAndGet ();


      } public int numberCreate () {return counter.get (); }
}
```

Machine Translated by Google

Nguyên tắc thay thế Liskov nói rằng bất kỳ thuộc tính quan trọng nào của một kiểu cũng phải được lưu giữ cho các kiểu con của nó, do đó bất kỳ phương thức nào được viết cho kiểu sẽ hoạt động tốt như nhau trên các kiểu con của nó [Liskov87]. Nhưng giả sử chúng ta truyền một cá thể Điểm truy cập vào phương thức onUnitCircle . Nếu lớp Point sử dụng phương thức equals dựa trên getClass , phương thức onUnitCircle sẽ trả về false bất kể giá trị x và y của cá thể CounterPoint . Điều này là như vậy bởi vì các bộ sưu tập, chẳng hạn như HashSet được sử dụng bởi phương thức onUnitCircle , sử dụng phương thức bằng để kiểm tra khả năng chứa và không có cá thể CounterPoint nào bằng với bất kỳ Điểm nào. Nếu, đã bao giờ, bạn sử dụng một phương thức bằng dựa trên instanceof phù hợp trên Point, thì phương thức onUnitCircle tương tự sẽ hoạt động tốt khi được trình bày với CounterPoint.

Mặc dù không có cách nào thỏa đáng để mở rộng một lớp có thể khởi tạo và thêm một thành phần giá trị, nhưng vẫn có một cách giải quyết tốt. Thực hiện theo lời khuyên của Mục 16, "Cơ cấu ủng hộ đối với thừa kế." Thay vì có ColorPoint mở rộng Điểm, hãy cung cấp cho ColorPoint một trường Điểm riêng tư và một phương thức xem công khai (Mục 5) trả về điểm ở cùng vị trí với điểm màu này:

```
// Thêm một thành phần giá trị mà không vi phạm hợp đồng công bằng lớp ColorPoint {

    điểm cuối cùng tư nhân; màu sắc
    cuối cùng riêng tư ;

    public ColorPoint (int x, int y, Color color) {if (color ==
        null)
            ném NullPointerException () mới; point = new
        Point (x, y); this.color = màu;

    }

    /
     ** * Trả về điểm nhìn của điểm màu này. * /

    public Point asPoint () {return
        point;
    }

    @Override public boolean bằng (Object o) {if (! (O
        instanceof ColorPoint)) trả về false; ColorPoint cp =
            (ColorPoint) o; trả về cp.point.equals (điểm) &&
        cp.color.equals (màu);

    }

    ... // Phần còn lại bị bỏ qua
}
```

Machine Translated by Google

Có một số lớp trong thư viện nền tảng Java mở rộng một lớp có thể cài đặt được và thêm một thành phần giá trị. Ví dụ: java.sql.Timestamp mở rộng java.util.Date và thêm trường nano giây . Phương thức triển khai bằng cho Dấu thời gian vi phạm tính đối xứng và có thể gây ra hành vi thất thường nếu các đối tượng Dấu thời gian và Ngày tháng được sử dụng trong cùng một bộ sưu tập hoặc được trộn lẫn với nhau. Lớp Dấu thời gian có tuyên bố từ chối trách nhiệm cảnh báo các lập trình viên chống lại việc trộn ngày tháng và dấu thời gian. Mặc dù bạn sẽ không gặp rắc rối miễn là bạn giữ chúng riêng biệt, nhưng không có gì ngăn cản bạn trộn chúng và các lỗi phát sinh có thể khó gỡ lỗi. Hành vi này của lớp Dấu thời gian là một lỗi và không nên được mô phỏng.

Lưu ý rằng bạn có thể thêm một thành phần giá trị vào lớp con của lớp trừu tượng mà không vi phạm hợp đồng bằng . Điều này quan trọng đối với loại lưu trữ phân cấp lớp mà bạn nhận được bằng cách làm theo lời khuyên trong Mục 20, "Ưu tiên phân cấp lớp hơn các lớp được gắn thẻ." Ví dụ: bạn có thể có một lớp trừu tượng Hình dạng không có thành phần giá trị, một lớp con Vòng tròn thêm trường bán kính và một lớp con Hình chữ nhật thêm các trường chiều dài và chiều rộng . Các vấn đề về sắp xếp được hiển thị ở trên sẽ không xảy ra miễn là không thể tạo trực tiếp một cá thể lớp cha.

Nhất quán — Yêu cầu thứ tư của hợp đồng bình đẳng nói rằng nếu hai đối tượng bằng nhau, chúng phải luôn bằng nhau trừ khi một (hoặc cả hai) trong số chúng được sửa đổi. Nói cách khác, các đối tượng có thể thay đổi có thể bằng với các đối tượng khác nhau tại các thời điểm khác nhau trong khi các đối tượng bất biến thì không. Khi bạn viết một lớp, hãy suy nghĩ kỹ xem nó có nên là bất biến không (Tiết 15). Nếu bạn kết luận rằng điều đó nên xảy ra, hãy đảm bảo rằng phương thức bằng của bạn thực thi hạn chế rằng các đối tượng bằng nhau vẫn bằng nhau và các đối tượng không bằng nhau vẫn không bằng nhau trong mọi thời điểm.

Cho dù một lớp có phải là bất biến hay không, đừng viết một phương thức bằng phụ thuộc vào các tài nguyên không đáng tin cậy. Rất khó để đáp ứng yêu cầu về thời hạn của luật sư nếu bạn vi phạm điều cấm này. Ví dụ: phương thức bằng của java.net.URL dựa trên việc so sánh địa chỉ IP của các máy chủ được liên kết với các URL. Việc dịch tên máy chủ sang địa chỉ IP có thể yêu cầu quyền truy cập mạng và nó không được đảm bảo mang lại kết quả tương tự theo thời gian. Điều này có thể khiến phương thức equals của URL vi phạm hợp đồng bằng và gây ra sự cố trong thực tế. (Rất tiếc, không thể thay đổi hành vi này do các yêu cầu về tính tương thích.) Với rất ít ngoại lệ, các phương thức bằng sẽ thực hiện các phép tính tổng hợp xác định trên các đối tượng thường trú trong bộ nhớ.

"Non-nullity" —Yêu cầu cuối cùng, trong trường hợp không có tên, tôi đã tự do gọi là "non-nullity", nói rằng tất cả các đối tượng phải không bằng nhau để null. Mặc dù thật khó để tưởng tượng việc vô tình trả về true để phản hồi lại lệnh gọi o.equals (null), nhưng không khó để tưởng tượng việc vô tình ném một

Machine Translated by Google

NullPointerException. Hợp đồng chung không cho phép điều này. Nhiều lớp có các phương thức bằng để bảo vệ chống lại điều này bằng một bài kiểm tra rõ ràng cho null:

```
@Override public boolean bằng (Đối tượng o) {if (o ==
     null) return false;

      ...
}
```

Thử nghiệm này là không cần thiết. Để kiểm tra đối số của nó về sự bình đẳng, phương thức equals trước tiên phải truyền đối số của nó đến một kiểu thích hợp để các trình truy cập của nó có thể được gọi hoặc các trường của nó được truy cập. Trước khi thực hiện ép kiểu, phương thức phải sử dụng instanceof operator để kiểm tra xem đối số của nó có đúng kiểu hay không:

```
@Override public boolean bằng (Object o) {if (! (O
     instanceof MyType)) return false; MyType mt =
          (MyType) o;

      ...
}
```

Nếu kiểm tra kiểu này bị thiếu và phương thức equals được thông qua một đối số không đúng loại, phương thức equals sẽ ném ra một ClassCastException, vi phạm hợp đồng bằng . Nhưng toán tử instanceof được chỉ định để trả về false nếu toán hạng đầu tiên của nó là null, bất kể kiểu nào xuất hiện trong toán hạng thứ hai [JLS, 15.20.2]. Do đó, kiểm tra kiểu sẽ trả về false nếu null được chuyển vào, vì vậy bạn không cần kiểm tra null riêng biệt .

Tổng hợp tất cả lại với nhau, đây là công thức cho một phương pháp bằng chất lượng cao :

1. Sử dụng toán tử == để kiểm tra xem đối số có phải là tham chiếu đến đối tượng này hay không. Nếu vậy, hãy trả về true. Đây chỉ là một tối ưu hóa hiệu suất, nhưng một trong những điều đáng làm nếu việc so sánh có khả năng tốn kém.
2. Sử dụng toán tử instanceof để kiểm tra xem đối số có đúng kiểu không. Nếu không, trả về false. Thông thường, loại đúng là lớp mà phương thức xảy ra. Đôi khi, nó là một số giao diện được thực hiện bởi lớp này. Sử dụng interface nếu lớp triển khai giao diện tinh chỉnh hợp đồng bằng để cho phép so sánh giữa các lớp triển khai giao diện. Collection trong các terfaces như Set, List, Map và Map.Entry có thuộc tính này.
3. Truyền đối số đến đúng loại. Bởi vì dàn diễn viên này được đặt trước bởi một thử nghiệm instanceof , nó được đảm bảo sẽ thành công.

Machine Translated by Google

4. Đối với mỗi trường "quan trọng" trong lớp, hãy kiểm tra xem trường đó của đối số khớp với trường tương ứng của đối tượng này. Nếu tất cả các thử nghiệm này thành công, lại biến true; nếu không, trả về false. Nếu loại ở bước 2 là một giao diện, bạn phải truy cập các trường của đối số thông qua các phương thức giao diện; nếu kiểu là một lớp, bạn có thể truy cập trực tiếp vào các trường, tùy thuộc vào khả năng truy cập của chúng.

Đối với các trường nguyên thủy có kiểu không phải là float hoặc double, hãy sử dụng toán tử == cho so sánh; đối với các trường tham chiếu đối tượng, gọi đệ quy phương thức equals ly; đối với các trường float , hãy sử dụng phương thức Float.compare ; và đối với các trường kép , hãy sử dụng Gấp đôi. So sánh. Xử lý đặc biệt của trường float và trường kép được thực hiện cần thiết bởi sự tồn tại của Float.NaN, -0.0f và đôi tương tự các hằng số; xem tài liệu Float.equals để biết chi tiết. Đối với các trường mảng, áp dụng các nguyên tắc này cho từng phần tử. Nếu mọi phần tử trong trường mảng là đáng kể, bạn có thể sử dụng một trong các phương thức Arrays.equals được thêm vào trong bản phát hành 1.5.

Một số trường tham chiếu đối tượng có thể chứa null một cách hợp pháp. Để tránh khả năng hiển thị vị trí của NullPointerException, hãy sử dụng thành ngữ này để so sánh các trường như vậy:

```
(field == null? o.field == null: field.equals (o.field))
```

Sự thay thế này có thể nhanh hơn nếu trường và trường o.field thường giống hệt nhau:

```
(field == o.field || (field! = null && field.equals (o.field)))
```

Đối với một số lớp, chẳng hạn như CaseInsensitiveString ở trên, so sánh trường phức tạp hơn các bài kiểm tra bình đẳng đơn giản. Nếu đúng như vậy, bạn có thể muốn để lưu trữ một dạng chính tắc của trường, vì vậy phương pháp bằng có thể thực hiện chính xác giá rẻ so sánh trên các hình thức chính tắc này thay vì so sánh các hình thức so sánh không chính xác tốn kém hơn. Kỹ thuật này thích hợp nhất cho các lớp bất biến (Mục 15); nếu đối tượng có thể thay đổi, bạn phải giữ cho biểu mẫu chuẩn được cập nhật.

Hiệu suất của phương pháp bằng có thể bị ảnh hưởng bởi thứ tự trong đó các trường được so sánh. Để có hiệu suất tốt nhất, trước tiên bạn nên so sánh các trường có nhiều khả năng khác biệt hơn, ít tốn kém hơn để so sánh, hoặc lý tưởng là cả hai. Bạn phải không so sánh các trường không phải là một phần của trạng thái logic của đối tượng, chẳng hạn như Khóa các trường được sử dụng để đồng bộ hóa các hoạt động. Bạn không cần phải so sánh các trường thừa, có thể được tính toán từ "các trường quan trọng", nhưng làm như vậy có thể cải thiện hiệu suất của phương pháp bằng . Nếu một trường dư thừa tương đương với một mô tả tổng hợp của toàn bộ đối tượng, thì việc so sánh trường này sẽ giúp bạn tiết kiệm chi phí so sánh dữ liệu thực tế nếu so sánh không thành công. Ví dụ, giả sử bạn có một lớp Đa giác và bạn lưu vào bộ nhớ cache của vùng. Nếu hai đa giác có các khu vực không bằng nhau, bạn không cần phải so sánh các cạnh và đỉnh của chúng.

Machine Translated by Google

5. Khi bạn viết xong phương pháp bằng , hãy tự hỏi mình ba câu hỏi: Nó có đối xứng không? Nó có tính bắc cầu không? Nó có nhất quán không? Và đừng chỉ hỏi bản thân; viết các bài kiểm tra đơn vị để kiểm tra xem các thuộc tính này có giữ được không! Nếu không, hãy tìm ra lý do tại sao không và sửa đổi phương thức bằng cho phù hợp. Tất nhiên phương thức bằng của bạn cũng phải thỏa mãn hai thuộc tính khác (tính phản xạ và "tính không"), nhưng hai thuộc tính này thường tự lo liệu.

Đối với một ví dụ cụ thể về một phương pháp bằng được xây dựng theo công thức trên, hãy xem PhoneNumber.equals trong Mục 9. Dưới đây là một số lưu ý cuối cùng:

- Luôn ghi đè Mã băm khi bạn ghi đè bằng (Mục 9).
- Đừng cố tỏ ra quá thông minh. Nếu bạn chỉ đơn giản kiểm tra các trường về sự bình đẳng, thì không khó để tuân thủ hợp đồng bình đẳng . Nếu bạn quá quyết liệt trong việc tìm kiếm sự tương đương, bạn rất dễ gặp rắc rối. Nói chung là một ý tưởng tồi nếu tính đến bất kỳ hình thức nào của bí danh. Ví dụ: lớp Tệp không nên cố gắng đánh đồng các liên kết tượng trưng tham chiếu đến cùng một tệp. Rất may, nó không.
- Không thay thế một kiểu khác cho Đối tượng trong khai báo bằng . Không có gì lạ khi một lập trình viên viết một phương thức bằng giống như thế này, và sau đó dành hàng giờ để tìm hiểu lý do tại sao nó không hoạt động bình thường:

```
công khai boolean bằng (MyClass o) {
      ...
}
```

Vấn đề là phương thức này không ghi đè Object.equals, có hàm ar thuộc loại Đối tượng, nhưng thay vào đó sẽ ghi đè nó (Mục 41). Có thể chấp nhận cung cấp một phương thức bằng "được nhập mạnh" như vậy ngoài phương thức bình thường miễn là hai phương thức trả về cùng một kết quả, nhưng không có lý do thuyết phục nào để làm như vậy. Nó có thể mang lại những lợi ích nhỏ về hiệu suất trong một số trường hợp nhất định, nhưng nó không đáng để tăng thêm độ phức tạp (Mục 55).

Việc sử dụng nhất quán chú thích @Override , như được minh họa trong toàn bộ mục này, sẽ ngăn bạn mắc phải sai lầm này (Mục 36). Phương thức bằng này sẽ không biên dịch và thông báo lỗi sẽ cho bạn biết chính xác điều gì sai:

```
@Override public boolean bằng (MyClass o) {
      ...
}
```

Machine Translated by Google

Mục 9: Luôn ghi đè Mã băm khi bạn ghi đè bằng

Một nguồn lỗi phổ biến là không thể ghi đè phư ơ ng thức hashCode . Bạn phải ghi đè Mã băm trong mọi lớp ghi đè bằng. Không làm đư ợc như vậy sẽ dẫn đến vi phạm hợp đồng chung cho Object.hashCode, điều này sẽ ngăn lớp của bạn hoạt động bình thư ờng cùng với tất cả các mã dựa trên băm bộ sư u tập, bao gồm HashMap, HashSet và Hashtable.

Đây là hợp đồng, đư ợc sao chép từ đặc tả Đối tư ợng [JavaSE6]:

- Bất cứ khi nào nó đư ợc gọi trên cùng một đối tư ợng nhiều hơ n một lần trong quá trình thực thi của một ứng dụng, phư ơ ng thức hashCode phải luôn trả về cùng một số nguyên, miễn là không có thông tin nào đư ợc sử dụng trong các so sánh bằng trên đối tư ợng đư ợc sửa đổi. Số nguyên này không cần phải duy trì nhất quán từ một tệp thực thi của một ứng dụng đến một lần thực thi khác của cùng một ứng dụng.
- Nếu hai đối tư ợng bằng nhau theo phư ơ ng thức bằng (Đối tư ợng) , thì việc gọi ing phư ơ ng thức Mã băm trên mỗi đối tư ợng trong hai đối tư ợng phải tạo ra cùng một kết quả số nguyên.
- Không bắt buộc rằng nếu hai đối tư ợng là không bằng nhau theo phư ơ ng thức bằng (Ob ject) , thì việc gọi phư ơ ng thức hashCode trên mỗi đối tư ợng trong hai đối tư ợng phải tạo ra kết quả số nguyên riêng biệt. Tuy nhiên, lập trình viên nên lư u ý rằng việc tạo ra các kết quả số nguyên riêng biệt cho các đối tư ợng không bằng nhau có thể cải thiện hiệu suất của bảng băm.

Điều khoản quan trọng bị vi phạm khi bạn không ghi đè đư ợc Mã băm là cái thứ hai: các đối tư ợng bằng nhau phải có mã băm bằng nhau. Hai khác biệt các cá thể có thể bằng nhau về mặt logic theo phư ơ ng thức bằng của một lớp , như ng để Phư ơ ng thức hashCode của đối tư ợng , chúng chỉ là hai đối tư ợng không có gì nhiều trong com mon. Do đó , phư ơ ng thức hashCode của Object trả về hai ký tự số dư ờng như ngẫu nhiên thay vì hai số bằng nhau theo yêu cầu của hợp đồng.

Ví dụ: hãy xem xét lớp PhoneNumber đơ n giản sau , có Phư ơ ng thức bằng đư ợc xây dựng theo công thức ở Tiết 8:

```
công khai lớp cuối cùng PhoneNumber {
     mã vùng cuối cùng riêng tư ;
     tiền tố ngắn cuối cùng riêng tư ;
     riêng dòng cuối cùng ngắn gọnNumber;

     public PhoneNumber (int areaCode, int prefix,
                         int lineNumber) {
          rangeCheck (Mã vùng, 999, "mã vùng");
          rangeCheck (tiền tố, 999, "tiền tố");
          rangeCheck (lineNumber, 9999, "số dòng");
```

Machine Translated by Google


```
            this.areaCode = (ngắn) areaCode; tiền tố
            this.prefix = (ngắn); this.lineNumber = (ngắn)
            lineNumber;
        }

        private static void rangeCheck (int arg, int max,
                                                Tên chuỗi) {
            if (arg <0 || arg> max) ném mới
                IllegalArgumentException (name + ":" + arg);
        }

        @Override public boolean bằng (Đối tượng o) {if (o ==
            this) return true; if (! (o instanceof PhoneNumber))
                 trả về false; PhoneNumber pn = (PhoneNumber)
            o; trả về pn.lineNumber == lineNumber



                && pn.prefix == tiền tố &&
                pn.areaCode == areaCode;
        }

        // Bị hỏng - không có phương thức mã băm!

        ... // Phần còn lại bị bỏ qua
    }
```

Giả sử bạn cố gắng sử dụng lớp này với HashMap:

```
        Bản đồ <PhoneNumber, String> m =
            new HashMap <PhoneNumber, String> (); m.put (Số
        điện thoại mới (707, 867, 5309), "Jenny");
```

Tại thời điểm này, bạn có thể mong đợi m.get (new PhoneNumber (707, 867, 5309)) trả về "Jenny", nhưng nó trả về null. Lưu ý rằng hai phiên bản PhoneNumber có liên quan: một phiên bản được sử dụng để chèn vào HashMap và phiên bản thứ hai, bằng nhau, được sử dụng để truy xuất (đã cố gắng). Việc lớp PhoneNumber không ghi đè được Mã băm khiến hai phiên bản bằng nhau có mã băm không bằng nhau, vi phạm hợp đồng Mã băm . Do đó, phương thức get có khả năng tìm kiếm số điện thoại trong một nhóm băm khác với số được lưu trữ bằng phương pháp put . Ngay cả khi hai trường hợp xảy ra để băm cho cùng một nhóm, phương thức get gần như chắc chắn sẽ trả về null, vì HashMap có một tính năng tối ưu hóa lưu mã băm được liên kết với mỗi mục nhập và không bận tâm kiểm tra sự bình đẳng đối tượng nếu mã băm không không khớp.

Machine Translated by Google

Khắc phục sự cố này đơ n giản bằng cách cung cấp phư ơ ng thức mã băm thích hợp cho lớp PhoneNumber . Vậy phư ơ ng thức hashCode sẽ như thế nào? Thật là tầm thư ờng khi viết một cái hợp pháp như ng không tốt. Ví dụ, cái này luôn hợp pháp như ng không bao giờ đư ợc sử dụng:

```
// Hàm băm hợp pháp tệ nhất có thể - không bao giờ sử dụng!
@Override public int hashCode () {return 42; }
```

Nó hợp pháp vì nó đảm bảo rằng các đối tư ợng bằng nhau có cùng mã băm. Nó tàn bạo vì nó đảm bảo rằng mọi đối tư ợng đều có cùng một mã băm. Do đó, mọi đối tư ợng đều đư ợc băm vào cùng một nhóm và bảng băm biến thành danh sách đư ợc liên kết.
Các chư ơ ng trình nên chạy theo thời gian tuyến tính thay vì chạy theo thời gian bậc hai. Đối với các bảng băm lớn, đây là sự khác biệt giữa hoạt động và không hoạt động.

Một hàm băm tốt có xu hư ớng tạo ra các mã băm không bằng nhau cho các đối tư ợng không bằng nhau. Đây chính xác là ý nghĩa của điều khoản thứ ba của con đư ờng mã băm . Lý tư ởng nhất, một hàm băm nên phân phối đồng nhất bất kỳ tập hợp hợp lý nào của các thể hiện không bằng nhau trên tất cả các giá trị băm có thể có. Đạt đư ợc lý tư ởng này có thể khó khăn. May mắn thay, không quá khó để đạt đư ợc giá trị gần đúng. Đây là một công thức đơ n giản:

1. Lư u trữ một số giá trị khác không hằng số, chẳng hạn, 17, trong một biến int đư ợc gọi là kết quả.
2. Đối với mỗi trư ờng quan trọng f trong đối tư ợng của bạn (mỗi trư ờng đư ợc tính đến bởi phư ơ ng thức bằng , nghĩa là), hãy làm như sau:

   Một. Tính toán mã băm int c cho trư ờng:

   tôi. Nếu trư ờng là boolean, compute (f? 1: 0).

   ii. Nếu trư ờng là byte, char, short hoặc int, compute (int) f.

   iii. Nếu trư ờng là một trư ờng dài, compute (int) (f ^ (f >>> 32)).

   iv. Nếu trư ờng là một float, hãy tính Float.floatToIntBits (f).

   v. Nếu trư ờng là một trư ờng kép, hãy tính Double.doubleToLongBits (f), sau đó băm kết quả miễn là trong bư ớc 2.a.iii.

   vi. Nếu trư ờng là một tham chiếu đối tư ợng và phư ơ ng thức equals của lớp này so sánh trư ờng bằng cách gọi đệ quy bằng, gọi đệ quy mã băm trên trư ờng. Nếu cần phải so sánh phức tạp hơ n, hãy tính "biểu diễn chuẩn" cho trư ờng này và gọi Mã băm trên biểu diễn chuẩn. Nếu giá trị của trư ờng là null, trả về 0 (hoặc một số hằng số khác, như ng 0 là truyền thống).

Machine Translated by Google

vii. Nếu trường là một mảng, hãy xử lý nó như thể mỗi phần tử là một trường riêng biệt. Nghĩa là, tính toán mã băm cho mỗi phần tử quan trọng bằng cách áp dụng các quy tắc này một cách đệ quy và kết hợp các giá trị này theo bước 2.b. Nếu mọi phần tử trong trường mảng đều quan trọng, bạn có thể sử dụng một trong các phương thức Arrays.hashCode được thêm vào trong bản phát hành 1.5.

b. Kết hợp mã băm c được tính ở bước 2.a thành kết quả như sau:

```
result = 31 * result + c;
```

3. Trả kết quả.

4. Khi bạn viết xong phương thức Mã băm, hãy tự hỏi bản thân xem các phiên bản bằng nhau có mã băm bằng nhau hay không. Viết các bài kiểm tra đơn vị để xác minh trực giác của bạn! Nếu các trường hợp bằng nhau có mã băm không bằng nhau, hãy tìm hiểu lý do và khắc phục sự cố.

Bạn có thể loại trừ các trường thừa khỏi tính toán mã băm. Nói cách khác, bạn có thể bỏ qua bất kỳ trường nào có giá trị có thể được tính toán từ các trường được bao gồm trong tính toán. Bạn phải loại trừ bất kỳ trường nào không được sử dụng trong equals com parisons, nếu không bạn có nguy cơ vi phạm điều khoản thứ hai của hợp đồng hashCode .

Giá trị ban đầu khác không được sử dụng trong bước 1, vì vậy giá trị băm sẽ bị ảnh hưởng bởi các trường ban đầu có giá trị băm, như được tính ở bước 2.a, bằng 0. Nếu số 0 được sử dụng làm giá trị ban đầu trong bước 1, giá trị băm tổng thể sẽ không bị ảnh hưởng bởi bất kỳ trường ban đầu nào như vậy, điều này có thể làm tăng xung đột. Giá trị 17 là tùy ý.

Phép nhân ở bước 2.b làm cho kết quả phụ thuộc vào thứ tự của các trường, mang lại hàm băm tốt hơn nhiều nếu lớp có nhiều trường giống nhau.
Ví dụ: nếu phép nhân bị bỏ qua khỏi hàm băm chuỗi , tất cả các phép đảo ngữ sẽ có mã băm giống hệt nhau. Giá trị 31 được chọn vì nó là một số nguyên tố lẻ. Nếu nó là số chẵn và phép nhân bị tràn, thông tin sẽ bị mất, vì phép nhân với 2 tương đương với chuyển dịch. Lợi thế của việc sử dụng một số nguyên tố là ít rõ ràng hơn, nhưng nó là truyền thống. Một thuộc tính tốt đẹp của 31 là phép nhân có thể được thay thế bằng một phép chuyển và một phép trừ để có hiệu suất tốt hơn: 31 * i == (i << 5) - i. Các máy ảo hiện đại thực hiện loại tối ưu hóa này một cách tự động.

Hãy áp dụng công thức trên cho lớp PhoneNumber . Có ba trường quan trọng, tất cả đều thuộc loại ngắn:

```
@Override public int hashCode () {int
      result = 17; kết quả = 31 * kết quả
      + mã vùng; result = 31 * kết quả + tiền
      tố; result = 31 * result + lineNumber;
      trả về kết quả;

}
```

Machine Translated by Google

Bởi vì phư ơ ng thức này trả về kết quả của một phép tính xác định đơ n giản mà đầu vào duy nhất là ba trư ờng quan trọng trong một phiên bản PhoneNumber , rõ ràng là các phiên bản PhoneNumber bằng nhau có mã băm bằng nhau. Trên thực tế, phư ơ ng pháp này là một triển khai Mã băm hoàn toàn tốt cho PhoneNumber , ngang bằng với các phư ơ ng pháp trong thư viện nền tảng Java. Nó đơ n giản, nhanh chóng và thực hiện một công việc khả thi là phân tán các số điện thoại không bằng nhau thành các nhóm băm khác nhau.

Nếu một lớp là bất biến và chi phí tính toán mã băm là đáng kể, bạn có thể cân nhắc lư u vào bộ nhớ đệm mã băm trong đối tư ợng thay vì tính toán lại mỗi khi nó đư ợc yêu cầu. Nếu bạn tin rằng hầu hết các đối tư ợng thuộc loại này sẽ đư ợc sử dụng làm khóa băm, thì bạn nên tính toán mã băm khi phiên bản đư ợc tạo.

Nếu không, bạn có thể chọn khởi tạo nó một cách lư ời biếng lần đầu tiên Mã băm đư ợc gọi (Mục 71). Không rõ rằng lớp PhoneNumber của chúng tôi có xứng đáng với việc xử lý này hay không, như ng chỉ để cho bạn thấy nó đư ợc thực hiện như thế nào:

```
// Khởi tạo một cách lư ời biếng, hashCode đư ợc lư u
trong bộ nhớ cache private portable int hashCode; // (Xem Mục 71)

@Override public int hashCode () {int result
      = hashCode; if (result == 0) {result =
      17; kết quả = 31 * kết quả + mã vùng;
            result = 31 * kết quả + tiền tố;
            result = 31 * result + lineNumber; hashCode
            = kết quả;



      } trả về kết quả;
}
```

Mặc dù công thức trong mục này tạo ra các hàm băm hợp lý, như ng nó không mang lại các hàm băm hiện đại, cũng như các thư viện nền tảng Java không cung cấp các hàm băm như vậy kể từ bản phát hành 1.6. Viết các hàm băm như vậy là một chủ đề nghiên cứu, tốt nhất nên dành cho các nhà toán học và nhà khoa học máy tính lý thuyết. Có lẽ bản phát hành sau của nền tảng này sẽ cung cấp các hàm băm hiện đại cho các lớp và các phư ơ ng thức tiện ích của nó để cho phép các lập trình viên trung bình xây dựng các hàm băm như vậy. Trong khi chờ đợi, các kỹ thuật đư ợc mô tả trong mục này phải phù hợp với hầu hết các ứng dụng.

Đừng muốn loại trừ các phần quan trọng của một đối tư ợng khỏi tính toán mã băm để cải thiện hiệu suất. Mặc dù hàm băm kết quả có thể chạy nhanh hơ n, như ng chất lư ợng kém của nó có thể làm giảm hiệu suất của bảng băm đến mức chúng trở nên chậm không thể sử dụng đư ợc. Đặc biệt, hàm băm có thể, trong thực tế

Machine Translated by Google

tice, hãy đối mặt với một bộ sưu tập lớn các trường hợp khác nhau phần lớn ở các khu vực mà bạn đã chọn để bỏ qua. Nếu điều này xảy ra, hàm băm sẽ ánh xạ tất cả các phiên bản thành một số rất ít mã băm và các bộ sưu tập dựa trên băm sẽ hiển thị hiệu suất bậc hai. Đây không chỉ là một vấn đề lý thuyết. Hàm băm chuỗi được triển khai trong tất cả các bản phát hành trước 1.2 được kiểm tra nhiều nhất là mười sáu ký tự charac, cách đều nhau trong suốt chuỗi, bắt đầu bằng ký tự đầu tiên. Đối với bộ sưu tập lớn các tên phân cấp, chẳng hạn như URL, hàm băm này hiển thị chính xác hành vi bệnh lý được lưu ý ở đây.

Nhiều lớp trong thư viện nền tảng Java, chẳng hạn như Chuỗi, Số nguyên và Ngày, bao gồm trong thông số kỹ thuật của chúng giá trị chính xác được trả về bởi phương thức Mã băm của chúng dưới dạng một hàm của giá trị cá thể. Điều này nói chung không phải là một ý kiến hay, vì nó hạn chế nghiêm trọng khả năng cải thiện hàm băm của bạn trong các bản phát hành trong tương lai. Nếu bạn không xác định chi tiết của một hàm băm và một lỗ hổng được tìm thấy hoặc một hàm băm tốt hơn được phát hiện, bạn có thể thay đổi hàm băm trong bản phát hành tiếp theo, tin chắc rằng không có ứng dụng khách nào phụ thuộc vào các giá trị chính xác mà hàm băm trả về.

Machine Translated by Google

Mục 10: Luôn ghi đè lên Chuỗi

Trong khi java.lang.Object cung cấp một triển khai của phương thức toString , thì chuỗi mà nó trả về thường không phải là những gì người dùng trong lớp của bạn muốn thấy. Nó bao gồm tên lớp theo sau là dấu "at" (@) và đại diện hexadecimal không dấu của mã băm, ví dụ: "PhoneNumber @ 163b91". Hợp đồng chung cho toString nói rằng chuỗi trả về phải là "một biểu diễn ngắn gọn nhưng giàu thông tin, dễ đọc cho một người" [JavaSE6]. Mặc dù có thể lập luận rằng "PhoneNumber @ 163b91" là ngắn gọn và dễ đọc, nhưng nó không nhiều thông tin khi so sánh với "(707) 867-5309". Hợp đồng toString tiếp tục có nội dung: "Tất cả các lớp con nên ghi đè phương thức này". Lời khuyên tốt, thực sự!

Mặc dù việc tuân theo các hợp đồng bằng và mã băm (Mục 8, Mục 9) không quan trọng bằng việc cung cấp triển khai toString tốt sẽ làm cho lớp của bạn dễ sử dụng hơn nhiều. Phương thức toString được tự động gọi khi một đối tượng được chuyển tới println, printf, toán tử nối chuỗi, hoặc khẳng định, hoặc được in bởi trình gỡ lỗi. (Phương thức printf đã được thêm vào nền tảng trong bản phát hành 1.5, cũng như các phương thức liên quan bao gồm String.format, gần tương đương với sprintf của C.)

Nếu bạn đã cung cấp một phương pháp toString tốt cho PhoneNumber, thì việc tạo một thông báo chẩn đoán hữu ích sẽ dễ dàng như sau:

```
System.out.println ("Không kết nối được: " + phoneNumber);
```

Các lập trình viên sẽ tạo ra các thông báo chẩn đoán theo cách này cho dù bạn có ghi đè lên Chuỗi hay không, nhưng các thông báo sẽ không hữu ích trừ khi bạn làm như vậy. Lợi ích của việc cung cấp một phương thức toString tốt mở rộng ra ngoài các thể hiện của lớp cho các đối tượng chứa các tham chiếu đến các thể hiện này, đặc biệt là các bộ sưu tập. Bạn muốn xem cái nào khi in bản đồ, "{Jenny = PhoneNumber @ 163b91}" hoặc "{Jenny = (707) 867-5309}"?

Khi thực tế, phương thức toString sẽ trả về tất cả thông tin thú vị có trong đối tượng, như trong ví dụ về số điện thoại vừa được hiển thị. Sẽ là không thực tế nếu đối tượng lớn hoặc nếu nó chứa trạng thái không phù hợp với biểu diễn chuỗi. Trong những trường hợp này, toString sẽ trả về một bản tóm tắt, chẳng hạn như "Các trang trắng Manhattan (1487536 danh sách)" hoặc "Chủ đề [main, 5, main]". Tốt nhất, chuỗi nên tự giải thích. (Ví dụ về Thread làm thất bại bài kiểm tra này.)

Machine Translated by Google

Một quyết định quan trọng bạn sẽ phải thực hiện khi triển khai phương thức toString là có chỉ định định dạng của giá trị trả về trong tài liệu hay không.
Bạn nên làm điều này cho các lớp giá trị, chẳng hạn như số điện thoại hoặc ma trận. Ưu điểm của việc chỉ định định dạng là nó đóng vai trò là một đại diện tiêu chuẩn, rõ ràng, có thể đọc được của con người về đối tượng. Biểu diễn này có thể được sử dụng cho đầu vào và đầu ra và trong các đối tượng dữ liệu liên tục mà con người có thể đọc được, chẳng hạn như tài liệu XML. Nếu bạn chỉ định định dạng, thông thường bạn nên cung cấp một nhà máy hoặc phương thức khởi tạo tĩnh phù hợp để lập trình viên có thể dễ dàng dịch qua lại giữa đối tượng và biểu diễn chuỗi của nó. Cách tiếp cận này được thực hiện bởi nhiều lớp giá trị trong các thư viện nền tảng Java, bao gồm BigInteger, BigDecimal và hầu hết các lớp nguyên thủy đóng hộp.

Nhược điểm của việc chỉ định định dạng của giá trị trả về toString là một khi bạn đã chỉ định nó, bạn sẽ bị mắc kẹt với nó suốt đời, giả sử rằng lớp của bạn được sử dụng rộng rãi. Các lập trình viên sẽ viết mã để phân tích cú pháp biểu diễn, tạo ra nó và nhúng nó vào dữ liệu liên tục. Nếu bạn thay đổi cách trình bày trong bản phát hành trong tương lai, bạn sẽ phá vỡ mã và dữ liệu của chúng và chúng sẽ kêu lên. Bằng cách không chỉ định định dạng, bạn duy trì tính linh hoạt để thêm thông tin hoặc cải thiện định dạng trong bản phát hành tiếp theo.

Cho dù bạn quyết định chỉ định định dạng hay không, bạn nên ghi rõ ý định của mình. Nếu bạn chỉ định định dạng, bạn nên thực hiện chính xác. Ví dụ, đây là một phương thức toString để đi với lớp PhoneNumber trong Mục 9:

```
/
 ** * Trả về biểu diễn chuỗi của số điện thoại này.
 * Chuỗi bao gồm mười bốn ký tự có định dạng * là "(XXX) YYY-ZZZZ", trong
 đó XXX là mã vùng, YYY là * tiền tố và ZZZZ là số dòng. (Mỗi chữ cái * viết
 hoa đại diện cho một chữ số thập phân.)

 *
 * Nếu bất kỳ phần nào trong số ba phần của số điện thoại này quá nhỏ * để lấp
 đầy trường của nó, trường sẽ được đệm bằng các số 0 ở đầu.
 * Ví dụ: nếu giá trị của số dòng là 123, thì * bốn ký tự cuối cùng của biểu diễn
 chuỗi sẽ là "0123".
 *
 * Lưu ý rằng có một khoảng trắng duy nhất ngăn cách giữa dấu đóng ngoặc kép
 * sau mã vùng với chữ số đầu tiên của tiền tố *. * /


@Override public String toString () {
     return String.format ("(% 03d)% 03d-% 04d", areaCode,
                                prefix, lineNumber);
}
```

Machine Translated by Google

Nếu bạn quyết định không chỉ định một định dạng, nhận xét tài liệu sẽ đọc một cái gì đó như sau:

```
/
 ** * Trả về mô tả ngắn gọn về lọ thuốc này. Các chi tiết chính xác * của bản trình
 bày là không xác định và có thể thay đổi, * như ng những điều sau đây có thể đư ợc
 coi là điển hình:
 *
 * "[Potion # 9: type = love, mùi = nhựa thông, look = india ink]" * /

@Override public String toString () {...}
```

Sau khi đọc nhận xét này, các lập trình viên tạo ra mã hoặc dữ liệu liên tục phụ thuộc vào chi tiết của định dạng sẽ không ai phải chịu trách nhiệm khi định dạng bị thay đổi.

Cho dù bạn có chỉ định định dạng hay không, hãy cung cấp quyền truy cập có lập trình vào tất cả thông tin có trong giá trị do toString trả về. Ví dụ, lớp PhoneNumber phải chứa các trình truy cập cho mã vùng, tiền tố và số dòng. Nếu bạn không làm đư ợc điều này, bạn buộc các lập trình viên cần thông tin này phải phân tích cú pháp chuỗi. Bên cạnh việc giảm hiệu suất và thực hiện công việc không cần thiết cho các lập trình viên, quá trình này dễ xảy ra lỗi và dẫn đến các hệ thống dễ hỏng nếu bạn thay đổi định dạng. Bằng cách không cung cấp trình truy cập, bạn biến định dạng chuỗi thành một API trên thực tế, ngay cả khi bạn đã chỉ định rằng nó có thể thay đổi

Machine Translated by Google

Mục 11: Ghi đè phân thân một cách thận trọng

Giao diện có thể nhân bản được dự định là giao diện mixin (Mục 18) cho các đối tượng quảng cáo rằng chúng cho phép sao chép. Thật không may, nó không phục vụ mục đích này. Lỗ hổng chính của nó là nó thiếu phương thức sao chép và phương pháp nhân bản của Object được phát hiện một cách chuyên nghiệp. Bạn không thể, nếu không sử dụng đến sự phản chiếu (Mục 53), gọi phương thức nhân bản trên một đối tượng chỉ vì nó thực hiện Có thể sao chép . Ngay cả một lời gọi phản chiếu có thể không thành công, vì không có gì đảm bảo rằng đối tượng có một phương thức sao chép có thể truy cập được . Bất chấp lỗ hổng này và những lỗ hổng khác, cơ sở này đang được sử dụng rộng rãi nên bạn phải trả tiền để hiểu được nó. Mục này cho bạn biết cách triển khai một phương pháp nhân bản hoạt động tốt , thảo luận về thời điểm thích hợp để làm như vậy và trình bày các lựa chọn thay thế.

Vậy Cloneable làm được gì, vì nó không chứa phương thức nào? Nó xác định hành vi của việc triển khai bản sao được bảo vệ của Đối tượng : nếu một lớp triển khai Bản sao có thể sao chép , phương thức nhân bản của Đối tượng trả về một bản sao theo từng trường của đối tượng; nếu không, nó ném CloneNotSupportedException. Đây là cách sử dụng giao diện không điển hình và không phải là giao diện được mô phỏng. Thông thường, việc triển khai một giao diện nói lên điều gì đó về những gì một lớp có thể làm cho các máy khách của nó. Trong trường hợp Cloneable, nó sửa đổi hành vi của một phương thức được bảo vệ trên một lớp cha.

Nếu việc triển khai giao diện Cloneable có bất kỳ ảnh hưởng nào đến một lớp, thì lớp đó và tất cả các lớp cha của nó phải tuân theo một giao thức khá phức tạp, không thể thực thi và được lập thành văn bản mỏng. Cơ chế kết quả là ngoại giao: nó tạo ra một đối tượng mà không cần gọi một phương thức khởi tạo.

Hợp đồng chung cho phương pháp nhân bản là yếu. Nó đây, được sao chép từ đặc tả cho java.lang.Object [JavaSE6]:

> Tạo và trả về một bản sao của đối tượng này. Ý nghĩa chính xác của “copy” có thể phụ thuộc vào lớp của đối tượng. Mục đích chung là, đối với bất kỳ đối tượng x nào, biểu thức
>
> ```
> x.clone ()! = x
> ```
>
> sẽ đúng, và biểu thức
>
> ```
> x.clone (). getClass () == x.getClass ()
> ```
>
> sẽ đúng, nhưng đây không phải là những yêu cầu tuyệt đối. Trong khi nó thường là trường hợp
>
> ```
> x.clone (). bằng (x)
> ```
>
> sẽ đúng, đây không phải là một yêu cầu tuyệt đối. Việc sao chép một đối tượng sẽ dẫn đến việc tạo một thể hiện mới của lớp nó, nhưng nó cũng có thể yêu cầu sao chép cấu trúc dữ liệu bên trong. Không có hàm tạo nào được gọi.

Machine Translated by Google

Có một số vấn đề với hợp đồng này. Điều khoản "không có hàm tạo nào được gọi" là quá mạnh. Một phương thức nhân bản hoạt động tốt có thể gọi các cấu trúc giả để tạo các đối tượng bên trong bản sao đang được xây dựng. Nếu lớp là cuối cùng, bản sao thậm chí có thể trả về một đối tượng được tạo bởi một phương thức khởi tạo.

Tuy nhiên, điều khoản x.clone (). GetClass () nói chung phải giống với x.getClass (), tuy nhiên, quá yếu. Trong thực tế, các lập trình viên giả định rằng nếu họ mở rộng một lớp và gọi super.clone từ lớp con, thì đối tượng trả về sẽ là một thể hiện của lớp con. Cách duy nhất mà một lớp cha có thể cung cấp chức năng này là trả về một đối tượng có được bằng cách gọi super.clone. Nếu một phương thức clone trả về một đối tượng được tạo bởi một phương thức khởi tạo, nó sẽ có lớp sai.
Do đó, nếu bạn ghi đè phương thức clone trong một lớp không phải là cuối cùng, bạn nên trả về một đối tượng thu được bằng cách gọi super.clone. Nếu tất cả các siêu lớp của một lớp tuân theo quy tắc này, thì việc gọi super.clone cuối cùng sẽ gọi phương thức nhân bản của Object , tạo ra một thể hiện của lớp phù hợp. Cơ chế này gần giống với chuỗi phương thức khởi tạo tự động, ngoại trừ việc nó không được thực thi.

Giao diện Cloneable , kể từ bản phát hành 1.6, không giải thích chi tiết các trách nhiệm mà một lớp đảm nhận khi nó triển khai giao diện này. Trong thực tế, một lớp triển khai Cloneable được mong đợi sẽ cung cấp một phương thức nhân bản công khai hoạt động đúng cách . Nói chung, không thể làm như vậy trừ khi tất cả các lớp cha của lớp cung cấp một triển khai bản sao hoạt động tốt , cho dù là công khai hay được bảo vệ.

Giả sử bạn muốn triển khai Cloneable trong một lớp có các lớp cha cung cấp các phương thức sao chép hoạt động tốt. Đối tượng bạn nhận được từ super.clone () có thể gần hoặc có thể không gần với những gì cuối cùng bạn sẽ trả về, tùy thuộc vào bản chất của lớp. Đối tượng này, từ quan điểm của mỗi lớp cha, sẽ là một bản sao hoàn toàn vui nhộn của đối tượng ban đầu. Các trường được khai báo trong lớp của bạn (nếu có) sẽ có giá trị giống với giá trị của đối tượng được nhân bản. Nếu mọi trường đều chứa giá trị nguyên thủy hoặc tham chiếu đến một đối tượng không thay đổi, đối tượng được trả về có thể là chính xác những gì bạn cần, trong trường hợp đó không cần xử lý thêm. Đây là trường hợp, ví dụ, đối với lớp PhoneNumber trong Mục 9. Trong trường hợp này, tất cả những gì bạn cần làm ngoài việc khai báo rằng bạn triển khai Cloneable là cung cấp quyền truy cập pub lic vào phương thức sao chép được bảo vệ của Object :

```
@Override public PhoneNumber clone ()
    {try {return (PhoneNumber)
        super.clone (); } catch
    (CloneNotSupportedException e) {
        ném AssertionError () mới; // Không thể xảy ra
    }
}
```

Machine Translated by Google

Lư u ý rằng phư ơ ng thức sao chép ở trên trả về PhoneNumber, không phải Đối tư ợng. Kể từ phiên bản 1.5, việc làm này là hợp pháp và mong muốn, bởi vì các kiểu trả về hiệp biến đã đư ợc giới thiệu trong phiên bản 1.5 như một phần của generic. Nói cách khác, bây giờ là hợp pháp khi kiểu trả về của phư ơ ng thức ghi đè là một lớp con của kiểu trả về của phư ơ ng thức bị ghi đè. Điều này cho phép phư ơ ng thức ghi đè cung cấp thêm thông tin về đối tư ợng đư ợc trả về và loại bỏ nhu cầu ép kiểu trong máy khách. Vì Object.clone trả về Object, PhoneNumber.clone phải ép kiểu super.clone () trư ớc khi trả về, như ng điều này tốt hơ n nhiều so với việc yêu cầu mọi ngư ời gọi PhoneNumber.clone truyền kết quả. Nguyên tắc chung ở đây là không bao giờ bắt khách hàng làm bất cứ điều gì mà thư viện có thể làm cho khách hàng.

Nếu một đối tư ợng chứa các trư ờng tham chiếu đến các đối tư ợng có thể thay đổi, việc sử dụng triển khai sao chép đơ n giản đư ợc hiển thị ở trên có thể là một thảm họa. Ví dụ, hãy xem xét lớp Ngăn xếp trong Mục 6:

```
public class Stack
     {private Object [] các phần tử;
     int size = 0; private static
     final int DEFAULT_INITIAL_CAPACITY = 16;

     public Stack ()
          {this.elements = new Object [DEFAULT_INITIAL_CAPACITY];
     }

     public void push (Object e)
          {ensureCapacity (); các phần
          tử [size ++] = e;
     }

     public Object pop () {if
          (size == 0) throw new
               EmptyStackException ();
          Kết quả đối tư ợng = các phần tử [- size];
          phần tử [size] = null; // Loại bỏ kết quả trả về tham chiếu lỗi thời;

     }

     // Đảm bảo không gian cho ít nhất một phần tử nữa. private
     void ensureCapacity () {if (Elements.length == size)
          Elements = Arrays.copyOf (Elements, 2 * size + 1);

     }
}
```

Giả sử bạn muốn làm cho lớp này có thể sao chép đư ợc. Nếu phư ơ ng thức nhân bản của nó chỉ trả về super.clone (), thì cá thể Stack kết quả sẽ có giá trị chính xác trong

Machine Translated by Google

trư ờng kích thư ớc của nó , như ng trư ờng phần tử của nó sẽ tham chiếu đến cùng một mảng với cá thể Stack ban đầu . Sửa đổi bản gốc sẽ phá hủy các bất biến trong bản sao và ngư ợc lại. Bạn sẽ nhanh chóng nhận thấy rằng chư ơ ng trình của bạn tạo ra các kết quả vô nghĩa hoặc ném ra một NullPointerException.

Tình huống này không bao giờ có thể xảy ra do gọi hàm tạo duy nhất trong lớp Stack . Trên thực tế, phư ơ ng thức clone hoạt động như một phư ơ ng thức khởi tạo khác; bạn phải đảm bảo rằng nó không gây hại cho đối tư ợng gốc và nó hỗ trợ thiết lập các bất biến trên bản sao một cách chính xác. Để phư ơ ng thức nhân bản trên Stack hoạt động bình thư ờng, nó phải sao chép phần bên trong của ngăn xếp. Cách dễ nhất để làm điều này là gọi bản sao một cách đệ quy trên mảng các phần tử :

```
@Override public Stack clone () {try
     {Stack result = (Stack) super.clone
          (); result.elements = Elements.clone (); trả về
          kết quả; } catch (CloneNotSupportedException e)
          {ném new AssertionError ();


     }
}
```

Lư u ý rằng chúng ta không phải truyền kết quả của Elements.clone () thành Object []. Kể từ bản phát hành 1.5, việc gọi bản sao trên một mảng trả về một mảng có kiểu thời gian biên dịch giống với kiểu của mảng đang đư ợc sao chép.

Cũng lư u ý rằng giải pháp trên sẽ không hoạt động nếu trư ờng phần tử là cuối cùng, vì bản sao sẽ bị cấm gán giá trị mới cho trư ờng.

Đây là một vấn đề cơ bản: kiến trúc bản sao không tư ơ ng thích với việc sử dụng bình thư ờng các trư ờng cuối cùng đề cập đến các đối tư ợng có thể thay đổi, ngoại trừ trư ờng hợp các đối tư ợng có thể thay đổi có thể đư ợc chia sẻ một cách an toàn giữa một đối tư ợng và bản sao của nó. Để làm cho một lớp có thể sao chép, có thể cần phải xóa các bổ ngữ cuối cùng khỏi một số trư ờng.

Không phải lúc nào cũng đủ để gọi bản sao một cách đệ quy. Ví dụ: giả sử bạn đang viết một phư ơ ng thức sao chép cho một bảng băm có phần bên trong bao gồm một mảng các nhóm, mỗi nhóm tham chiếu đến mục nhập đầu tiên trong danh sách đư ợc liên kết của các cặp khóa-giá trị hoặc là null nếu nhóm trống. Đối với hiệu suất, lớp triển khai danh sách liên kết đơ n lẻ nhẹ của riêng nó thay vì sử dụng java.util.LinkedList trong nội bộ:

```
public class HashTable hiện thực Cloneable {private Entry
     [] buckets = ...;
```

Machine Translated by Google


```
        private static class Entry {khóa đối
            tượng cuối cùng;
            Giá trị đối tượng;
            Mục nhập tiếp theo;

            Mục nhập (Khóa đối tượng, Giá trị đối tượng, Mục nhập tiếp
                theo) {this.key = key; this.value = giá trị; this.next
                = tiếp theo;

            }
        }

        ... // Phần còn lại bị bỏ qua
    }
```

Giả sử bạn chỉ sao chép đệ quy mảng thùng, như chúng tôi đã làm đối với Stack:

```
    // Bị hỏng - kết quả ở trạng thái nội bộ được chia sẻ!
    @Override public HashTable clone () {try
        {HashTable result = (HashTable) super.clone
            (); result.buckets = buckets.clone (); trả về kết quả; }
            catch (CloneNotSupportedException e) {ném new AssertionError
            ();


        }
    }
```

Mặc dù bản sao có mảng thùng riêng của nó, mảng này tham chiếu đến các danh sách được liên kết giống như bản gốc, điều này có thể dễ dàng gây ra hành vi không xác định trong cả bản sao và bản gốc. Để khắc phục sự cố này, bạn sẽ phải sao chép danh sách được liên kết bao gồm từng nhóm riêng lẻ. Đây là một cách tiếp cận phổ biến:

```
    public class HashTable hiện thực Cloneable {private Entry []
        buckets = ...;

        private static class Entry {khóa đối
            tượng cuối cùng;
            Giá trị đối tượng;
            Mục nhập tiếp theo;

            Mục nhập (Khóa đối tượng, Giá trị đối tượng, Mục nhập tiếp
                theo) {this.key = key; this.value = giá trị; this.next
                = tiếp theo;

            }
```

Machine Translated by Google


```
            // Sao chép đệ quy danh sách liên kết đứng đầu bởi Mục nhập này deepCopy ()
            {return new Entry (key, value, next == null? Null: next.deepCopy ());


            }
      }

      @Override public HashTable clone () {try
            {HashTable result = (HashTable) super.clone
                  (); result.buckets = new Entry [buckets.length]; for (int i
                  = 0; i <buckets.length; i ++) if (buckets [i]! = null)


                              result.buckets [i] = buckets [i] .deepCopy ();
                  trả về kết quả; }
            catch (CloneNotSupportedException e) {ném new
                  AssertionError ();
            }
      }
      ... // Phần còn lại bị bỏ qua
}
```

Lớp riêng HashTable.Entry đã đư ợc tăng cư ờng để hỗ trợ phư ơ ng pháp “bản sao sâu”. Phư ơ ng thức nhân bản trên HashTable phân bổ một mảng nhóm mới có kích thư ớc phù hợp và lặp lại trên mảng nhóm ban đầu , sao chép sâu từng nhóm không có gì. Phư ơ ng thức sao chép sâu trên Entry gọi chính nó một cách đệ quy để sao chép toàn bộ danh sách đư ợc liên kết đứng đầu bởi mục nhập. Mặc dù kỹ thuật này rất dễ thư ơ ng và hoạt động tốt nếu các nhóm không quá dài, như ng nó không phải là một cách tốt để sao chép một danh sách đư ợc liên kết vì nó tiêu tốn một khung ngăn xếp cho mỗi phần tử trong danh sách. Nếu danh sách dài, điều này có thể dễ dàng gây ra tràn ngăn xếp. Để ngăn điều này xảy ra, bạn có thể thay thế đệ quy trong deepCopy bằng phép lặp:

```
// Sao chép lặp đi lặp lại danh sách đư ợc liên kết đứng đầu bởi Mục nhập này
Entry deepCopy () {
    Kết quả nhập = Mục nhập mới (khóa, giá trị, tiếp theo);

    for (Mục p = result; p.next! = null; p = p.next)
        p.next = new Entry (p.next.key, p.next.value, p.next.next);

    trả về kết quả;
}
```

Cách tiếp cận cuối cùng để nhân bản các đối tư ợng phức tạp là gọi super.clone, đặt tất cả các trư ờng trong đối tư ợng kết quả về trạng thái nguyên thủy của chúng, sau đó gọi các phư ơ ng thức cấp cao hơ n để tạo lại trạng thái của đối tư ợng. Trong trư ờng hợp bài kiểm tra HashTable của chúng tôi-

Machine Translated by Google

ple, trường bucket sẽ được khởi tạo thành một mảng bucket mới và phương thức put (khóa, giá trị) (không được hiển thị) sẽ được gọi cho mỗi ping ánh xạ khóa-giá trị trong bảng băm đang được sao chép. Cách tiếp cận này thường mang lại một phương thức sao chép đơn giản, rất thanh lịch nhưng thường không chạy nhanh như một phương pháp thao tác trực tiếp các phần bên trong của đối tượng và bản sao của nó.

Giống như một phương thức khởi tạo, một phương thức sao chép không được gọi bất kỳ phương thức nào không phải phương thức chính trên bản sao đang được xây dựng (Mục 17). Nếu bản sao gọi một phương thức được ghi đè, phương thức này sẽ thực thi trước khi lớp con mà nó được định nghĩa có cơ hội để sửa chữa trạng thái của nó trong bản sao, hoàn toàn có thể dẫn đến hỏng bản sao và bản gốc. Do đó , phương thức đặt (khóa, giá trị) được thảo luận trong đoạn trước phải là phương thức cuối cùng hoặc riêng tư . (Nếu nó là riêng tư , nó có lẽ là "phương pháp trợ giúp" cho một phương thức công khai không chính thức.)

Phương thức nhân bản của đối tượng được khai báo để ném CloneNotSupportedException, nhưng ghi đè các phương thức nhân bản có thể bỏ qua khai báo này. Các phương thức nhân bản công khai nên bỏ qua nó vì các phương thức không ném các ngoại lệ đã kiểm tra sẽ dễ sử dụng hơn (Mục 59). Nếu một lớp được thiết kế để kế thừa (Mục 17) ghi đè lên bản sao, thì phương thức ghi đè phải bắt chước hành vi của Object.clone: nó phải được khai báo là bảo vệ, nó phải được khai báo để ném CloneNotSupportedExcep tion và lớp không được triển khai Cloneable. Điều này cho phép các lớp con tự do triển khai Có thể sao chép hoặc không, giống như khi chúng mở rộng Đối tượng trực tiếp.

Một chi tiết đáng chú ý. Nếu bạn quyết định tạo cho một lớp an toàn luồng có thể sao chép , hãy nhớ rằng phương thức nhân bản của nó phải được đồng bộ đúng cách giống như bất kỳ phương thức nào khác (Mục 66). Phương thức sao chép của đối tượng không được đồng bộ hóa, vì vậy ngay cả khi nó thỏa mãn, bạn có thể phải viết một phương thức sao chép đồng bộ hóa gọi super.clone ().

Tóm lại, tất cả các lớp triển khai Cloneable nên ghi đè bản sao bằng một phương thức công khai có kiểu trả về là chính lớp đó. Phương thức này trước tiên nên gọi super.clone và sau đó sửa bất kỳ trường nào cần sửa. Thông thường, điều này có nghĩa là sao chép bất kỳ đối tượng có thể thay đổi nào bao gồm "cấu trúc sâu" bên trong của đối tượng được nhân bản và thay thế các tham chiếu của bản sao đến các đối tượng này bằng tham chiếu đến các bản sao. Mặc dù các bản sao nội bộ này thường có thể được thực hiện bằng cách gọi đệ quy bản sao , nhưng đây không phải lúc nào cũng là cách tiếp cận tốt nhất. Nếu lớp chỉ chứa các trường nguyên thủy hoặc các tham chiếu đến các đối tượng không thể thay đổi, thì có thể là trường hợp không cần sửa trường nào. Có những ngoại lệ cho quy tắc này. Ví dụ: trường đại diện cho số sê-ri hoặc ID duy nhất khác hoặc trường đại diện cho thời gian tạo của đối tượng sẽ cần được cố định, ngay cả khi nó là nguyên thủy hoặc bất biến.

Tất cả sự phức tạp này có thực sự cần thiết? Hiếm. Nếu bạn mở rộng một lớp triển khai Cloneable, bạn có rất ít lựa chọn ngoài việc triển khai một

Machine Translated by Google

phư ơ ng pháp nhân bản . Nếu không, tốt hơ n hết bạn nên cung cấp một phư ơ ng tiện sao chép đối tư ợng thay thế, hoặc chỉ đơ n giản là không cung cấp khả năng. Ví dụ, sẽ không hợp lý khi các lớp không thay đổi hỗ trợ sao chép đối tư ợng, vì các bản sao hầu như không thể phân biệt đư ợc với bản gốc.

Một cách tiếp cận tốt để sao chép đối tư ợng là cung cấp một phư ơ ng thức tạo bản sao hoặc nhà máy sao chép. Một phư ơ ng thức khởi tạo sao chép chỉ đơ n giản là một phư ơ ng thức khởi tạo nhận một đối số duy nhất có kiểu là lớp chứa phư ơ ng thức khởi tạo, ví dụ:

```
Yum công cộng (Yum yum);
```

Nhà máy sao chép là tư ơ ng tự nhà máy tĩnh của một phư ơ ng thức khởi tạo sao chép:

```
public static Yum newInstance (Yum yum);
```

Phư ơ ng pháp tiếp cận phư ơ ng thức tạo bản sao và biến thể static factory của nó có nhiều lợi thế hơ n so với Cloneable / clone: chúng không dựa vào cơ chế tạo đối tư ợng ngoại lai dễ gặp rủi ro; họ không yêu cầu sự tuân thủ không thể thực thi đối với các quy ư ớc đư ợc ghi chép mỏng; chúng không xung đột với việc sử dụng thích hợp các trư ờng cuối cùng; họ không ném các ngoại lệ đã kiểm tra không cần thiết; và họ không yêu cầu phôi.
Mặc dù không thể đặt một hàm tạo bản sao hoặc nhà máy sản xuất trong một giao diện, như ng Cloneable không thể hoạt động như một giao diện vì nó thiếu phư ơ ng thức sao chép công khai . Do đó, bạn không từ bỏ chức năng giao diện bằng cách sử dụng phư ơ ng thức tạo bản sao hoặc nhà máy thay thế cho phư ơ ng pháp sao chép .

Hơ n nữa, một phư ơ ng thức khởi tạo hoặc nhà máy sao chép có thể nhận một đối số có kiểu là một giao diện đư ợc thực thi bởi lớp. Ví dụ, theo quy ư ớc, tất cả các triển khai bộ sư u tập có mục đích chung đều cung cấp một phư ơ ng thức khởi tạo có đối số thuộc kiểu Bộ sư u tập hoặc Bản đồ. Nhà máy và trình tạo bản sao dựa trên giao diện, đư ợc gọi đúng hơ n là trình tạo chuyển đổi và nhà máy chuyển đổi, cho phép khách hàng chọn kiểu triển khai của bản sao thay vì buộc khách hàng chấp nhận kiểu triển khai của bản gốc. Giả sử bạn có một HashSet và bạn muốn sao chép nó dư ới dạng TreeSet. Phư ơ ng thức sao chép không thể cung cấp tính đồng nhất của hàm này, như ng nó dễ dàng với một phư ơ ng thức khởi tạo chuyển đổi: (các) TreeSet mới.

Do tất cả các vấn đề liên quan đến Cloneable, có thể nói rằng các giao diện khác không nên mở rộng nó và các lớp đư ợc thiết kế để kế thừa (Mục 17) không nên triển khai nó. Bởi vì nhiều thiếu sót của nó, một số lập trình viên chuyên nghiệp chỉ đơ n giản là chọn không bao giờ ghi đè phư ơ ng thức sao chép và không bao giờ gọi nó ngoại trừ, có lẽ, để sao chép các mảng. Nếu bạn thiết kế một lớp để kế thừa, hãy lư u ý rằng nếu bạn chọn không cung cấp một phư ơ ng thức nhân bản đư ợc bảo vệ hoạt động tốt , thì các lớp con sẽ không thể thực hiện Cloneable.

Machine Translated by Google

Mục 12: Xem xét triển khai Có thể so sánh

Không giống như các phư ơ ng pháp khác đư ợc thảo luận trong chư ơ ng này, phư ơ ng thức CompareTo không đư ợc khai báo trong Object. Đúng hơ n, nó là phư ơ ng thức duy nhất trong giao diện So sánh . Nó là có đặc điểm tư ơ ng tự như phư ơ ng thức equals của Object , ngoại trừ việc nó cho phép đặt hàng com paris ngoài các so sánh bình đẳng đơ n giản và nó là chung chung. Bằng cách nhắc đến So sánh, một lớp chỉ ra rằng các thể hiện của nó có thứ tự tự nhiên.
Việc sắp xếp một mảng các đối tư ợng triển khai So sánh đơ n giản như sau:

```
Arrays.sort (a);
```

Tư ơ ng tự, việc tìm kiếm, tính toán các giá trị cực trị và duy trì các bộ sư u tập các đối tư ợng có thể so sánh đư ợc sắp xếp một cách tự động cũng rất dễ dàng . Ví dụ: pro gram sau đây, dựa trên thực tế là Chuỗi thực hiện có thể so sánh, in ra một
danh sách đư ợc sắp xếp theo thứ tự bảng chữ cái các đối số dòng lệnh của nó với các đối số trùng lặp đã bị loại bỏ:

```
lớp công khai WordList {
     public static void main (String [] args) {
          Đặt <Chuỗi> s = new TreeSet <Chuỗi> ();
          Collections.addAll (s, args);
          System.out.println (s);
     }
}
```

Bằng cách triển khai Có thể so sánh, bạn cho phép lớp của mình tư ơ ng tác với tất cả nhiều thuật toán chung và triển khai bộ sư u tập phụ thuộc vào điều này
giao diện. Bạn có đư ợc một sức mạnh to lớn chỉ với một nỗ lực nhỏ.
Hầu như tất cả các lớp giá trị trong thư viện nền tảng Java đều triển khai Compa rable. Nếu bạn đang viết một lớp giá trị với một thứ tự tự nhiên rõ ràng, chẳng hạn như
thứ tự bảng chữ cái, thứ tự số hoặc thứ tự thời gian, bạn nên
xem xét triển khai giao diện:

```
giao diện công cộng Có thể so sánh đư ợc <T> {
     int so sánhTo (T t);
}
```

Hợp đồng chung của phư ơ ng thức CompareTo tư ơ ng tự như hợp đồng của phư ơ ng thức bằng:

So sánh đối tư ợng này với đối tư ợng đư ợc chỉ định để đặt hàng. Trả về số âm dư ới dạng teger, không hoặc số nguyên dư ơ ng vì đối tư ợng này nhỏ hơ n, bằng hoặc lớn hơ n hơ n đối tư ợng đư ợc chỉ định. Ném ClassCastException nếu kiểu của ob ject đư ợc chỉ định ngăn nó đư ợc so sánh với đối tư ợng này.

Machine Translated by Google

Trong mô tả sau, ký hiệu sgn (biểu thức) chỉ định hàm ký hiệu ematical trong toán học , đư ợc xác định để trả về -1, 0 hoặc 1, tùy theo giá trị của biểu thức là âm, 0 hay dư ơ ng.

- Ngư ời triển khai phải đảm bảo sgn (x.compareTo (y)) == -sgn (y.compare To (x)) cho mọi x và y. (Điều này ngụ ý rằng x.compareTo (y) phải ném một ex ception nếu và chỉ khi y.compareTo (x) ném ra một ngoại lệ.)
- Ngư ời triển khai cũng phải đảm bảo rằng quan hệ là bắc cầu: (x.com pareTo (y)> 0 && y.compareTo (z)> 0) ngụ ý x.compareTo (z)> 0.
- Cuối cùng, ngư ời triển khai phải đảm bảo rằng x.compareTo (y) == 0 ngụ ý rằng sgn (x.compareTo (z)) == sgn (y.compareTo (z)), với mọi z.
- Chúng tôi thực sự khuyến khích, như ng không bắt buộc nghiêm ngặt, rằng (x.compareTo (y) == 0) == (x.equals (y)). Nói chung, bất kỳ lớp nào triển khai giao diện Có thể so sánh và vi phạm điều kiện này phải chỉ rõ điều này. Ngôn ngữ đư ợc đề xuất là "Lư u ý: Lớp này có thứ tự tự nhiên không nhất quán với bằng."

Đừng bỏ qua bản chất toán học của hợp đồng này. Giống như hợp đồng bình đẳng (Khoản 8), hợp đồng này không phức tạp như vẻ ngoài của nó. Trong một lớp học, bất kỳ thứ tự hợp lý nào sẽ đáp ứng nó. Trên các lớp, CompareTo, không giống như bằng, không phải hoạt động: nó đư ợc phép ném ClassCastException nếu hai tham chiếu đối tư ợng đư ợc so sánh tham chiếu đến các đối tư ợng thuộc các lớp khác nhau. Thông thư ờng, đó chính xác là những gì so sánh phải làm và những gì nó sẽ làm nếu lớp đư ợc tham số hóa đúng cách. Mặc dù hợp đồng không loại trừ việc so sánh giữa các lớp, như ng kể từ phiên bản 1.6, không có lớp nào trong thư viện nền tảng Java hỗ trợ chúng.

Cũng giống như một lớp vi phạm hợp đồng Mã băm có thể phá vỡ các lớp khác phụ thuộc vào phép băm, một lớp vi phạm hợp đồng So sánh có thể phá vỡ các lớp khác phụ thuộc vào phép so sánh. Các lớp phụ thuộc vào việc so sánh bao gồm các bộ sư u tập đã đư ợc sắp xếp TreeSet và TreeMap, và các lớp tiện ích Bộ sư u tập và Mảng, chứa các thuật toán tìm kiếm và sắp xếp.

Chúng ta hãy xem xét các điều khoản của hợp đồng so sánh. Điều khoản đầu tiên nói rằng nếu bạn đảo ngư ợc hư ớng so sánh giữa hai đối tư ợng tham chiếu, điều mong đợi sẽ xảy ra: nếu đối tư ợng đầu tiên nhỏ hơ n đối tư ợng thứ hai, thì đối tư ợng thứ hai phải lớn hơ n đối tư ợng đầu tiên; nếu vật thứ nhất bằng vật thứ hai thì vật thứ hai phải bằng vật thứ nhất; và nếu vật thứ nhất lớn hơ n vật thứ hai, thì vật thứ hai phải nhỏ hơ n vật thứ nhất. Điều khoản thứ hai nói rằng nếu một đối tư ợng lớn hơ n một giây và đối tư ợng thứ hai lớn hơ n một phần ba, thì đối tư ợng đầu tiên phải lớn hơ n đối tư ợng thứ ba. Điều khoản cuối cùng nói rằng tất cả các đối tư ợng đư ợc so sánh ngang bằng phải mang lại kết quả như nhau khi so sánh với bất kỳ đối tư ợng nào khác.

Machine Translated by Google

Một hệ quả của ba điều khoản này là phép thử bình đẳng được áp dụng bởi phương pháp so sánh phải tuân theo các hạn chế tương tự do phương pháp bình đẳng áp đặt : tính phản xạ, tính đối xứng và độ nhạy. Do đó, điều tương tự cũng được áp dụng: không có cách nào để mở rộng một lớp có thể khởi tạo với một thành phần giá trị mới trong khi bảo toàn hợp đồng CompareTo , trừ khi bạn sẵn sàng từ bỏ các lợi ích của trừu tượng hướng đối tượng (Mục 8). Cách giải quyết tương tự cũng được áp dụng. Nếu bạn muốn thêm một thành phần giá trị vào một lớp có triển khai So sánh, đừng mở rộng nó; viết một lớp không liên quan chứa một thể hiện của lớp đầu tiên. Sau đó, cung cấp một phương thức "view" trả về trường hợp này. Điều này giải phóng bạn triển khai bất kỳ phương thức CompareTo nào bạn thích trên lớp thứ hai, đồng thời cho phép cli ent của nó xem một thể hiện của lớp thứ hai như một thể hiện của lớp đầu tiên khi

cần thiết.

Đoạn cuối cùng của hợp đồng CompareTo , là một gợi ý mạnh mẽ chứ không phải là một điều khoản thực sự, chỉ đơn giản nói rằng kiểm tra bình đẳng được áp đặt bởi phương pháp CompareTo nói chung sẽ trả về kết quả giống như phương pháp bằng . Nếu điều khoản này được tuân theo, thứ tự áp đặt bởi phương pháp CompareTo được cho là nhất quán với bằng. Nếu nó bị vi phạm, thứ tự được cho là không phù hợp với bằng. Một lớp có phương thức CompareTo áp đặt một thứ tự không phù hợp với bằng sẽ vẫn hoạt động, nhưng các tập hợp đã sắp xếp chứa các phần tử của lớp có thể không tuân theo hợp đồng chung của các giao diện trường hợp thích hợp (Bộ sưu tập, Tập hợp hoặc Bản đồ). Điều này là do các hợp đồng chung cho các giao diện này được xác định theo phương pháp bằng , nhưng các đầu nối được sắp xếp sử dụng phép thử bình đẳng được áp đặt bởi CompareTo thay cho bằng. Nó không phải là một thảm họa nếu điều này xảy ra, nhưng đó là điều cần phải lưu ý.

Ví dụ: hãy xem xét lớp BigDecimal , có phương thức CompareTo không phù hợp với bằng. Nếu bạn tạo một cá thể HashSet và thêm BigDecimal mới ("1.0") và BigDecimal mới ("1.00"), tập hợp sẽ chứa hai phần tử vì hai cá thể BigDecimal được thêm vào tập hợp là không bằng nhau khi so sánh bằng phương pháp bằng . Tuy nhiên, nếu bạn thực hiện quy trình tương tự bằng cách sử dụng TreeSet thay vì HashSet, thì tập hợp sẽ chỉ chứa một phần tử vì hai cá thể BigDecimal bằng nhau khi được so sánh bằng phương thức CompareTo . (Xem tài liệu BigDecimal để biết chi tiết.)

Viết phương thức CompareTo tương tự như viết phương thức bằng , nhưng có một vài điểm khác biệt chính. Bởi vì giao diện So sánh được ized tham số, phương thức CompareTo được nhập tĩnh, vì vậy bạn không cần nhập kiểm tra hoặc ép kiểu đối số của nó. Nếu đối số không đúng loại, lời gọi thậm chí sẽ không được biên dịch. Nếu đối số là null, lời gọi sẽ ném một ception NullPointerEx và nó sẽ xuất hiện ngay sau khi phương thức cố gắng truy cập các thành viên của nó.

Machine Translated by Google

Các so sánh trư ờng trong phư ơ ng thức CompareTo là so sánh theo thứ tự hơ n là so sánh bình đẳng. So sánh các trư ờng tham chiếu đối tư ợng bằng cách gọi đệ quy phư ơ ng thức CompareTo . Nếu một trư ờng không triển khai So sánh hoặc bạn cần sử dụng một thứ tự không chuẩn, bạn có thể sử dụng một Trình so sánh rõ ràng để thay thế.
Viết của riêng bạn hoặc sử dụng một cái có sẵn như trong phư ơ ng thức CompareTo này cho lớp CaseInsensitiveString trong Mục 8.

```
public cuối cùng lớp CaseInsensitiveString thực hiện
            có thể so sánh đư ợc <CaseInsensitiveString> {public int so
      sánhTo (CaseInsensitiveString cis) {
            return String.CASE_INSENSITIVE_ORDER.compare (s, cis.s);
      }
      ... // Phần còn lại bị bỏ qua
}
```

Lư u ý rằng lớp CaseInsensitiveString triển khai Compara ble <CaseInsensitiveString>. Điều này có nghĩa là tham chiếu CaseInsensitiveString chỉ có thể đư ợc so sánh với các tham chiếu CaseInsensitiveString khác . Đây là mô hình bình thư ờng cần tuân theo khi khai báo một lớp để triển khai So sánh.
Cũng lư u ý rằng tham số của phư ơ ng thức CompareTo là một Chuỗi phân biệt chữ hoa chữ thư ờng, không phải là một Đối tư ợng. Điều này đư ợc yêu cầu bởi khai báo lớp nói trên.

So sánh các trư ờng nguyên thủy tích phân bằng cách sử dụng các toán tử quan hệ < và >. Đối với các trư ờng dấu phẩy động, hãy sử dụng Double.compare hoặc Float.compare thay cho các toán tử quan hệ, các toán tử này không tuân theo hợp đồng chung cho CompareTo khi áp dụng cho các giá trị dấu chấm động. Đối với các trư ờng mảng, hãy áp dụng các nguyên tắc này cho từng phần tử.

Nếu một lớp có nhiều trư ờng quan trọng, thứ tự mà bạn so sánh chúng là rất quan trọng. Bạn phải bắt đầu với lĩnh vực quan trọng nhất và làm việc theo cách của bạn.
Nếu kết quả so sánh là bất kỳ điều gì khác với 0 (đại diện cho sự bình đẳng), bạn đã hoàn thành; chỉ cần trả lại kết quả. Nếu các trư ờng quan trọng nhất bằng nhau, hãy tiếp tục so sánh các trư ờng quan trọng nhất tiếp theo, v.v. Nếu tất cả các trư ờng đều bằng nhau thì các đối tư ợng bằng nhau; trả về số không. Kỹ thuật này đư ợc thể hiện bằng phư ơ ng thức CompareTo cho lớp PhoneNumber trong Mục 9:

```
public int CompareTo (PhoneNumber pn) {// So sánh
      các mã vùng if (areaCode <pn.areaCode) return
      -1; if (areaCode> pn.areaCode) return 1;
```

Machine Translated by Google


```
        // Các mã vùng bằng nhau, hãy so sánh các tiền tố if
        (prefix <pn.prefix) return -1; if (prefix> pn.prefix)
             return 1;



        // Mã vùng và tiền tố bằng nhau, hãy so sánh số dòng if (lineNumber
        <pn.lineNumber) return -1;

        if (lineNumber> pn.lineNumber) return 1;


        trả về 0; // Tất cả các trư ờng đều bằng nhau
    }
```

Trong khi phư ơ ng pháp này hoạt động, nó có thể đư ợc cải thiện. Hãy nhớ lại rằng hợp đồng cho com pareTo không chỉ định độ lớn của giá trị trả về, chỉ có dấu hiệu. Bạn có thể tận dụng điều này để đơ n giản hóa mã và có thể làm cho nó chạy nhanh hơ n một chút:

```
    public int CompareTo (PhoneNumber pn) {// So
        sánh các mã vùng int areaCodeDiff = areaCode
        - pn.areaCode; if (areaCodeDiff! = 0)

             trả về areaCodeDiff;

        // Các mã vùng bằng nhau, hãy so sánh các tiền tố
        int prefixDiff = prefix - pn.prefix; if (prefixDiff!
        = 0) trả về prefixDiff;


        // Mã vùng và tiền tố bằng nhau, so sánh số dòng trả về lineNumber -
        pn.lineNumber;
    }
```

Thủ thuật này hoạt động tốt ở đây như ng nên đư ợc sử dụng hết sức thận trọng. Không sử dụng trừ khi bạn chắc chắn các trư ờng đư ợc đề cập là không âm hoặc nói chung là sự khác biệt giữa giá trị trư ờng thấp nhất và cao nhất có thể nhỏ hơ n hoặc bằng Integer.MAX_VALUE (231-1). Lý do mà thủ thuật này không phải lúc nào cũng hoạt động là số nguyên 32 bit có dấu không đủ lớn để chứa sự khác biệt giữa hai số nguyên 32 bit có dấu tùy ý. Nếu i là int dư ơ ng lớn và j là int âm lớn , (i - j) sẽ tràn và trả về giá trị âm. Phư ơ ng thức CompareTo kết quả sẽ trả về kết quả không chính xác đối với một số đối số và vi phạm các điều khoản thứ nhất và thứ hai của hợp đồng CompareTo . Đây không phải là một vấn đề lý thuyết thuần túy: nó đã gây ra lỗi trong các hệ thống thực. Những lỗi này có thể khó gỡ lỗi, vì phư ơ ng thức CompareTo bị hỏng hoạt động bình thư ờng đối với hầu hết các giá trị đầu vào.

Machine Translated by Google

CHƯ Ơ NG 4

Lớp học và Giao diện

LỚP và giao diện nằm ở trung tâm của ngôn ngữ lập trình Java. Chúng là các đơ n vị trừu tư ợng cơ bản của nó. Ngôn ngữ này cung cấp nhiều tính năng mạnh mẽ mà bạn có thể sử dụng để thiết kế các lớp và giao diện. Chư ơ ng này bao gồm hư ớng dẫn để giúp bạn sử dụng tốt nhất các yếu tố này để các lớp học của bạn và giao diện có thể sử dụng đư ợc, mạnh mẽ và linh hoạt.

Mục 13: Giảm thiểu khả năng truy cập của các lớp và thành viên

Yếu tố quan trọng nhất giúp phân biệt mô-đun đư ợc thiết kế tốt với đư ợc thiết kế kém là mức độ mà mô-đun ẩn dữ liệu bên trong của nó và chi tiết triển khai khác từ các mô-đun khác. Một mô-đun đư ợc thiết kế tốt ẩn tất cả các chi tiết triển khai của nó, tách biệt rõ ràng API của nó khỏi triển khai của nó. Sau đó, các mô-đun chỉ giao tiếp thông qua các API của chúng và không để ý đến hoạt động bên trong của nhau. Khái niệm này, đư ợc gọi là ẩn thông tin hoặc tổng hợp thông tin , là một trong những nguyên lý cơ bản của thiết kế phần mềm [Parnas72].

Việc che giấu thông tin quan trọng vì nhiều lý do, hầu hết đều xuất phát từ thực tế là nó tách các mô-đun bao gồm một hệ thống, cho phép chúng đư ợc phát triển, thử nghiệm, tối ư u hóa, sử dụng, hiểu và sửa đổi một cách riêng biệt. Cái này tăng tốc độ phát triển hệ thống vì các mô-đun có thể đư ợc phát triển song song. Nó giảm bớt gánh nặng bảo trì bởi vì các mô-đun có thể đư ợc hiểu nhiều hơ n nhanh chóng và đư ợc gỡ lỗi mà ít sợ làm hại các mô-đun khác. Mặc dù ẩn thông tin không gây ra hiệu suất tốt, như ng nó cho phép hiệu quả điều chỉnh hiệu suất: sau khi hệ thống hoàn tất và cấu hình đã đư ợc xác định mô-đun nào đang gây ra sự cố hiệu suất (Mục 55), những mô-đun đó có thể đư ợc tối ư u hóa mà không ảnh hư ởng đến tính đúng đắn của các mô-đun khác. Việc ẩn thông tin làm tăng khả năng sử dụng lại phần mềm vì các mô-đun thư ờng không đư ợc kết hợp chặt chẽ với nhau tỏ ra hữu ích trong các bối cảnh khác ngoài bối cảnh mà chúng đã đư ợc phát triển.

Machine Translated by Google

Cuối cùng, việc che giấu thông tin làm giảm rủi ro trong việc xây dựng các hệ thống lớn, bởi vì các mô-đun riêng lẻ có thể chứng minh là thành công ngay cả khi hệ thống không thành công.

Java có nhiều phương tiện hỗ trợ cho việc che giấu thông tin. Cơ chế kiểm soát truy cập nism [JLS, 6.6] chỉ định khả năng truy cập của các lớp, giao diện và thành viên. Khả năng truy cập của một thực thể được xác định bởi vị trí khai báo của nó và theo đó, nếu có, các bổ trợ truy cập (riêng tư, bảo vệ và công khai) có trên khai báo. Sử dụng hợp lý các bổ ngữ này là điều cần thiết để che giấu thông tin.

Quy tắc ngón tay cái rất đơn giản: làm cho mỗi lớp hoặc thành viên không thể truy cập được càng tốt. Nói cách khác, hãy sử dụng mức truy cập thấp nhất có thể phù hợp với hoạt động bình thường của phần mềm mà bạn đang viết.

Đối với các lớp và giao diện cấp cao nhất (không lồng nhau), chỉ có hai cấp độ truy cập có thể có: gói-riêng tư và công khai. Nếu bạn khai báo một lớp cấp cao nhất hoặc mặt liên với công cụ sửa đổi công khai, nó sẽ là công khai; nếu không, nó sẽ là package-pri vate. Nếu một lớp hoặc giao diện cấp cao nhất có thể được đặt ở dạng gói riêng tư, thì nó nên như vậy. Bằng cách đặt nó ở chế độ riêng tư, bạn biến nó thành một phần của việc triển khai thay vì API đã xuất và bạn có thể sửa đổi nó, thay thế nó hoặc loại bỏ nó trong một bản phát hành tiếp theo mà không sợ gây hại cho các ứng dụng khách hiện có. Nếu bạn đặt nó ở chế độ công khai, bạn có nghĩa vụ phải hỗ trợ nó mãi mãi để duy trì khả năng tương thích.

Nếu một lớp (hoặc giao diện) cấp cao nhất của gói-private chỉ được sử dụng bởi một lớp, hãy xem xét việc đặt lớp cấp cao nhất trở thành lớp lồng nhau riêng của lớp duy nhất sử dụng nó (Mục 22). Điều này làm giảm khả năng truy cập của nó từ tất cả các lớp trong gói của nó xuống một lớp sử dụng nó. Nhưng điều quan trọng hơn là giảm khả năng truy cập của lớp công khai vô cớ so với lớp cấp cao nhất của gói-private: lớp công khai là một phần của API của gói, trong khi lớp cấp cao nhất của gói-riêng đã là một phần của nó thực hiện.

Đối với các thành viên (trường, phương thức, lớp lồng nhau và giao diện lồng nhau), có bốn cấp độ truy cập có thể có, được liệt kê ở đây để tăng khả năng truy cập:

- riêng tư — Thành viên chỉ có thể truy cập được từ lớp cấp cao nhất ở đó khai báo.
- package-private — Thành viên có thể truy cập từ bất kỳ lớp nào trong gói mà nó được khai báo. Về mặt kỹ thuật được gọi là quyền truy cập mặc định, đây là quyền truy cập mà bạn nhận được nếu không có công cụ sửa đổi quyền truy cập nào được chỉ định.
- protected — Thành viên có thể truy cập từ các lớp con của lớp nơi nó được khai báo (tuân theo một số hạn chế [JLS, 6.6.2]) và từ bất kỳ lớp nào trong gói nơi nó được khai báo.
- public — Thành viên có thể truy cập từ mọi nơi.

Machine Translated by Google

Sau khi thiết kế cẩn thận API công khai của lớp, phản xạ của bạn là đặt tất cả các thành viên khác ở chế độ riêng tư . Chỉ khi một lớp khác trong cùng một gói thực sự cần truy cập một thành viên thì bạn mới nên xóa phần bổ trợ riêng tư , đặt gói thành viên ở chế độ riêng tư . Nếu bạn thấy mình làm điều này thư ờng xuyên, bạn nên kiểm tra lại thiết kế của hệ thống của mình để xem liệu sự phân rã khác có thể mang lại các lớp đư ợc tách rời khỏi nhau tốt hơ n hay không. Điều đó nói rằng, cả thành viên private và package-private đều là một phần trong quá trình triển khai của lớp và thư ờng không ảnh hư ởng đến API đã xuất của nó. Tuy nhiên, những trư ờng này có thể "rò rỉ" vào API đã xuất nếu lớp triển khai Serializable (Mục 74, Mục 75).

Đối với các thành viên của các lớp công khai, khả năng truy cập sẽ tăng lên đáng kể khi cấp độ truy cập chuyển từ gói riêng tư sang đư ợc bảo vệ. Thành viên đư ợc bảo vệ là một phần của API đã xuất của lớp và phải đư ợc hỗ trợ mãi mãi. Ngoài ra, một thành viên đư ợc bảo vệ của một lớp đã xuất đại diện cho một cam kết công khai đối với một chi tiết triển khai (Mục 17). Nhu cầu về các thành viên đư ợc bảo vệ nên tư ơ ng đối hiếm.

Có một quy tắc hạn chế khả năng của bạn là giảm khả năng truy cập của meth ods. Nếu một phư ơ ng thức ghi đè một phư ơ ng thức của lớp cha, thì nó không đư ợc phép có mức truy cập thấp hơ n trong lớp con so với mức của nó trong lớp cha [JLS, 8.4.8.3]. Điều này là cần thiết để đảm bảo rằng một thể hiện của lớp con có thể sử dụng đư ợc ở bất kỳ đâu mà một thể hiện của lớp cha có thể sử dụng đư ợc. Nếu bạn vi phạm quy tắc này, trình biên dịch sẽ tạo ra thông báo lỗi khi bạn cố gắng biên dịch lớp con. Một trư ờng hợp đặc biệt của quy tắc này là nếu một lớp triển khai một giao diện, thì tất cả các phư ơ ng thức của lớp cũng có trong giao diện này phải đư ợc khai báo công khai. Điều này là như vậy bởi vì tất cả các thành viên của một giao diện là hoàn toàn công khai [JLS, 9.1.5].

Để tạo điều kiện thuận lợi cho quá trình thử nghiệm, bạn có thể muốn làm cho một lớp, giao diện hoặc mem ber dễ truy cập hơ n. Điều này là tốt cho đến một thời điểm. Có thể chấp nhận việc đặt một thành viên private của gói public class-private để kiểm tra nó, như ng không thể chấp nhận đư ợc việc nâng khả năng truy cập cao hơ n mức đó. Nói cách khác, không thể chấp nhận việc biến một lớp, giao diện hoặc thành viên trở thành một phần của API đã xuất của gói để tạo điều kiện cho việc thử nghiệm. May mắn thay, nó cũng không cần thiết, vì các bài kiểm tra có thể đư ợc thực hiện để chạy như một phần của gói đang đư ợc kiểm tra, do đó có quyền truy cập vào các phần tử gói-riêng của nó.

Các trư ờng cá thể không bao giờ đư ợc công khai (Mục 14). Nếu một trư ờng thể hiện không phải là trư ờng cuối cùng hoặc là tham chiếu cuối cùng đến một đối tư ợng có thể thay đổi, thì bằng cách đặt trư ờng ở chế độ công khai, bạn sẽ loại bỏ khả năng giới hạn các giá trị có thể đư ợc lư u trữ trong trư ờng. Điều này có nghĩa là bạn cũng từ bỏ khả năng thực thi các bất biến liên quan đến trư ờng. Ngoài ra, bạn từ bỏ khả năng thực hiện bất kỳ hành động nào khi trư ờng đư ợc sửa đổi, vì vậy các lớp có trư ờng có thể thay đổi công khai không an toàn cho chuỗi. Ngay cả khi một trư ờng là cuối cùng và tham chiếu đến một đối tư ợng bất biến, bằng cách đặt trư ờng ở chế độ công khai, bạn sẽ từ bỏ tính linh hoạt để chuyển sang một biểu diễn dữ liệu nội bộ mới mà trư ờng đó không tồn tại.

Machine Translated by Google

Lời khuyên tương tự cũng áp dụng cho các trường tĩnh, với một ngoại lệ. Bạn có thể hiển thị các hằng số thông qua các trường cuối cùng tĩnh công khai, giả sử các hằng số tạo thành một phần tổng thể của phần trừu tượng được cung cấp bởi lớp. Theo quy ước, các trường như vậy có tên bao gồm các chữ cái viết hoa, với các từ được phân tách bằng dấu gạch dưới (Mục 56). Điều quan trọng là các trường này phải chứa các giá trị nguyên thủy hoặc các tham chiếu đến các đối tượng bất biến (Mục 15). Trường cuối cùng chứa tham chiếu đến một đối tượng có thể thay đổi có tất cả các những đặc điểm của trường không phải là trường cuối cùng. Mặc dù không thể sửa đổi tham chiếu, nhưng đối tượng được tham chiếu có thể được sửa đổi - với kết quả thảm hại.

Lưu ý rằng một mảng có độ dài khác không luôn có thể thay đổi, vì vậy sẽ sai khi một lớp có trường mảng cuối cùng tĩnh công khai hoặc một trình truy cập trả về trường như vậy. Nếu một lớp có trường hoặc bộ truy cập như vậy, các máy khách sẽ có thể sửa đổi lều che của mảng. Đây là một nguồn thường xuyên của các lỗ hổng bảo mật:

```
// Lỗ hổng bảo mật tiềm ẩn! public
static final Thing [] VALUES = {...};
```

Hãy cẩn thận với thực tế là nhiều IDE tạo ra các trình truy cập trả về các tham chiếu đến các trường mảng private, dẫn đến chính xác vấn đề này. Có hai cách để khắc phục sự cố. Bạn có thể đặt mảng công khai ở chế độ riêng tư và thêm một danh sách bất biến công khai:

```
private static final Thing [] PRIVATE_VALUES = {...}; public static
final List <Thing> VALUES =
    Collections.unmodifiableList (Arrays.asList (PRIVATE_VALUES));
```

Ngoài ra, bạn có thể đặt mảng ở chế độ riêng tư và thêm phương thức công khai trả về bản sao của mảng riêng tư :

```
private static final Thing [] PRIVATE_VALUES = {...}; public static
final Thing [] giá trị () {return PRIVATE_VALUES.clone ();

}
```

Để chọn giữa các lựa chọn thay thế này, hãy nghĩ về những gì khách hàng có thể làm với kết quả. Loại trả lại nào sẽ thuận tiện hơn? Cái nào sẽ mang lại hiệu suất đặt cược?

Tóm lại, bạn nên luôn giảm khả năng truy cập càng nhiều càng tốt.
Sau khi thiết kế cẩn thận một API công khai tối thiểu, bạn nên ngăn mọi lớp, giao diện hoặc thành viên bị lạc trở thành một phần của API. Với ngoại trừ các trường cuối cùng tĩnh công khai, các lớp công khai sẽ không có trường công cộng nào.

Đảm bảo rằng các đối tượng được tham chiếu bởi các trường cuối cùng tĩnh công khai là không thay đổi.

Machine Translated by Google

Mục 14: Trong các lớp công khai, sử dụng các phương thức của trình truy cập, không sử dụng các trường công khai

Đôi khi, bạn có thể bị cám dỗ để viết các lớp thoái hóa không phục vụ tư thế nào khác ngoài nhóm các trường cá thể:

```
// Degenerate các lớp như thế này không nên công khai! class Point
{public double x; công đôi y;


}
```

Bởi vì các trường dữ liệu của các lớp như vậy được truy cập trực tiếp, các lớp này không cung cấp các lợi ích của việc đóng gói (Mục 13). Bạn không thể thay đổi đại diện mà không thay đổi API, bạn không thể thực thi các biến bất biến và bạn không thể thực hiện hành động bổ trợ khi một trường được truy cập. Các lập trình viên hướng đối tượng dòng cứng cảm thấy rằng các lớp như vậy là anathema và phải luôn được thay thế bằng các lớp có trường riêng và phương thức truy cập công khai (getters) và đối với các lớp có thể thay đổi, trình đột biến (setters):

```
// Đóng gói dữ liệu bằng các phương thức của bộ truy cập và lớp bộ đột
biến Point {private double x; tư nhân đôi y;



     public Point (double x, double y) {this.x
          = x; this.y = y;

     }

     public double getX () {return x; } public
     double getY () {return y; }

     public void setX (double x) {this.x = x; } public
     void setY (double y) {this.y = y; }
}
```

Chắc chắn, hard-liner là đúng khi nói đến các lớp công khai: nếu một lớp có thể truy cập được bên ngoài gói của nó, hãy cung cấp các phương thức của trình truy cập, để duy trì tính linh hoạt trong việc thay đổi biểu diễn bên trong của lớp. Nếu một lớp công khai để lộ các trường dữ liệu của nó, tất cả hy vọng thay đổi cách biểu diễn của nó sẽ bị mất, vì mã máy khách có thể bị đánh giá xa và rộng.

Tuy nhiên, nếu một lớp là gói-riêng hoặc là một lớp lồng nhau riêng tư , thì không có gì sai khi để lộ các trường dữ liệu của nó — giả sử chúng thực hiện một

Machine Translated by Google

công việc thích hợp của việc mô tả phần trừu tượng được cung cấp bởi lớp. Cách tiếp cận này tạo ra ít lộn xộn trực quan hơn so với cách tiếp cận phương pháp truy cập, cả trong định nghĩa lớp và mã máy khách sử dụng nó. Trong khi mã khách hàng được liên kết với đại diện bên trong của lớp, mã này được giới hạn trong gói chứa lớp. Nếu mong muốn thay đổi cách trình bày, bạn có thể thực hiện thay đổi mà không cần chạm vào bất kỳ mã nào bên ngoài gói. Trong trường hợp của một lớp lồng nhau riêng tư , phạm vi thay đổi bị hạn chế hơn nữa đối với lớp bao quanh.

Một số lớp trong thư viện nền tảng Java vi phạm lời khuyên rằng các lớp công khai không nên hiển thị trực tiếp các trường. Các ví dụ nổi bật bao gồm các lớp Điểm và Thứ nguyên trong gói java.awt . Thay vì những ví dụ được nêu ra, những lớp này nên được coi là những câu chuyện cảnh giác. Như được mô tả trong Mục 55, quyết định để lộ phần bên trong của lớp Thứ nguyên đã dẫn đến một vấn đề về hiệu suất vẫn còn tồn tại với chúng ta ngày nay.

Mặc dù không bao giờ là một ý tưởng hay để một lớp công khai hiển thị trực tiếp các trường, nhưng sẽ ít có hại hơn nếu các trường là bất biến. Bạn không thể thay đổi biểu diễn của một lớp như vậy mà không thay đổi API của nó và bạn không thể thực hiện các hành động bổ trợ khi một trường được đọc, nhưng bạn có thể thực thi các biến bất biến. Ví dụ, lớp này đảm bảo rằng mỗi cá thể đại diện cho một thời gian hợp lệ:

```
// Lớp công khai với các trường không thay đổi được tiếp xúc - lớp cuối cùng
công khai có vấn đề Thời gian {private static final int HOURS_PER_DAY = 24;
      private static cuối cùng int MINUTES_PER_HOUR = 60;


      công khai int giờ cuối cùng;
      công khai cuối cùng int phút;

      Giờ công khai (int giờ, int phút) {
            nếu (giờ <0 || giờ> = HOURS_PER_DAY)
                ném IllegalArgumentException mới ("Giờ:" if (phút <0 ||     + giờ);
            phút> = MINUTES_PER_HOUR) ném IllegalArgumentException mới
                ("Tối thiểu:" + phút); this.hour = giờ; this.minute = phút;


      }
      ... // Phần còn lại bị bỏ qua
}
```

Tóm lại, các lớp công khai không bao giờ được để lộ các trường có thể thay đổi. Mặc dù vẫn còn nhiều nghi vấn, việc các lớp công khai để lộ các trường bất biến sẽ ít có hại hơn. Tuy nhiên, đôi khi điều mong muốn đối với các lớp lồng nhau gói-riêng hoặc riêng tư để hiển thị các trường, cho dù có thể thay đổi hoặc bất biến.

Machine Translated by Google

Mục 15: Giảm thiểu khả năng đột biến

Một lớp bất biến chỉ đơ n giản là một lớp mà các thể hiện của nó không thể sửa đổi đư ợc. Tất cả thông tin chứa trong mỗi phiên bản đư ợc cung cấp khi nó đư ợc tạo và cố định trong suốt thời gian tồn tại của đối tư ợng. Các thư viện nền tảng Java chứa nhiều các lớp bất biến, bao gồm String, các lớp nguyên thủy đóng hộp, BigInte ger và BigDecimal. Có nhiều lý do chính đáng cho điều này:
dễ thiết kế, triển khai và sử dụng hơ n các lớp có thể thay đổi. Họ ít bị để tránh lỗi và an toàn hơ n.

Để làm cho một lớp trở nên bất biến, hãy làm theo năm quy tắc sau:

1. Không cung cấp bất kỳ phư ơ ng thức nào sửa đổi trạng thái của đối tư ợng (đư ợc gọi là muta thân).

2. Đảm bảo rằng lớp học không thể đư ợc mở rộng. Điều này ngăn chặn sự bất cẩn hoặc độc hại các lớp con không làm ảnh hư ởng đến hành vi bất biến của lớp bằng cách cư xử như thể trạng thái của đối tư ợng đã thay đổi. Việc ngăn chặn phân lớp con thư ờng đư ợc thực hiện bằng cách đặt lớp cuối cùng, như ng có một giải pháp thay thế mà chúng tôi sẽ thảo luận sau.

3. Làm cho tất cả các trư ờng cuối cùng. Điều này thể hiện rõ ràng ý định của bạn theo cách mà hệ thống bắt buộc. Ngoài ra, cần phải đảm bảo hành vi chính xác nếu một tham chiếu đến một phiên bản mới đư ợc tạo đư ợc chuyển từ một chuỗi này sang một chuỗi khác mà không đồng bộ hóa, như đư ợc mô tả trong mô hình bộ nhớ [JLS, 17.5; Goetz06 16].

4. Đặt tất cả các trư ờng ở chế độ riêng tư . Điều này ngăn không cho các máy khách truy cập vào các đối tư ợng muta ble đư ợc các trư ờng tham chiếu đến và sửa đổi các đối tư ợng này một cách trực tiếp. Trong khi đó đư ợc phép về mặt kỹ thuật đối với các lớp không thay đổi có các trư ờng cuối cùng công khai kết nối các giá trị nguyên thủy hoặc các tham chiếu đến các đối tư ợng bất biến, nó không đư ợc đề xuất sửa chữa vì nó ngăn cản việc thay đổi biểu diễn bên trong sau này phát hành (Tiết 13).

5. Đảm bảo quyền truy cập độc quyền vào bất kỳ thành phần có thể thay đổi nào. Nếu lớp của bạn có bất kỳ các trư ờng tham chiếu đến các đối tư ợng có thể thay đổi, đảm bảo rằng các máy khách của lớp không thể lấy đư ợc tham chiếu đến các đối tư ợng này. Không bao giờ khởi tạo trư ờng như vậy thành tham chiếu ob ject do khách hàng cung cấp hoặc trả về tham chiếu đối tư ợng từ trình truy cập. Phòng thủ các bản sao (Mục 39) trong các hàm tạo, trình truy cập và các phư ơ ng thức readObject (Mục 76).

Machine Translated by Google

Nhiều lớp mẫu trong các mục trước là bất biến. Một trong những lớp như vậy là PhoneNumber trong Mục 9, có các trình truy cập cho mỗi thuộc tính nhưng không có các trình đột biến tương ứng. Đây là một ví dụ phức tạp hơn một chút:

```
public final class Complex { private
     final double re; tư nhân cuối
     cùng im lặng;

     public Complex (double re, double im) {this.re =
          re; this.im = im;

     }

     // Các trình truy cập không có trình đột biến tương ứng
     public double realPart () {return re; } public double imaginaryPart ()
     {return im; }

     public Complex add (Complex c) {
          return new Complex (re + c.re, im + c.im);
     }

     public Complex trừ (Complex c) {
          return new Complex (re - c.re, im - c.im);
     }

     public Complex nhân (Complex c) {
          return new Complex (re * c.re - im * c.im, re * c.im
                                  + im * c.re);
     }

     public Phép chia phức hợp (Complex c) {double
          tmp = c.re * c.re + c.im * c.im; return new Complex
          ((re * c.re + im * c.im) / tmp, (im * c.re - re * c.im) / tmp);

     }

     @Override public boolean bằng (Đối tượng o) {if (o ==
         this) return true; if (! (o instanceof Complex)) trả
              về false;


         Phức hợp c = (Phức tạp) o;

         // Xem trang 43 để tìm hiểu lý do tại sao chúng ta sử dụng so sánh thay vì
         == return Double.compare (re, c.re) == 0 && Double.compare (im, c.im) == 0;

     }
```

Machine Translated by Google


```
    @Override public int hashCode () {
        int result = 17 + hashDouble (re);
        result = 31 * result + hashDouble (im);
        trả về kết quả;
    }

    private int hashDouble (double val) {
        long longBits = Double.doubleToLongBits (lại);
        return (int) (longBits ^ (longBits >>> 32));
    }

    @Override public String toString () {
        return "(" + re + + im + "i)";
    }
}
```

Lớp này đại diện cho một số phức (một số có cả phần thực và phần ảo). Ngoài các phư ơ ng thức Đối tư ợng tiêu chuẩn , nó cung cấp các trình truy cập cho phần thực và phần ảo và cung cấp bốn phép toán số học cơ bản: cộng, trừ, nhân và chia. Chú ý cách số học các hoạt động tạo và trả về một phiên bản Complex mới thay vì sửa đổi nó ví dụ. Mẫu này đư ợc sử dụng trong hầu hết các lớp bất biến quan trọng. Nó đư ợc biết đến như phư ơ ng pháp tiếp cận chức năng vì các phư ơ ng thức trả về kết quả của việc áp dụng một chức năng vào toán hạng của họ mà không sửa đổi nó. Đối lập điều này với phư ơ ng pháp tiếp cận song phư ơ ng hoặc mệnh lệnh phổ biến hơ n, trong đó các phư ơ ng pháp áp dụng một quy trình cho hoạt động của chúng và khiến trạng thái của nó thay đổi.

Cách tiếp cận chức năng có thể có vẻ không tự nhiên nếu bạn không quen với nó, như ng nó cho phép tính bất biến, có nhiều ư u điểm. Các đối tư ợng bất biến là giản dị. Một đối tư ợng bất biến có thể ở chính xác một trạng thái, trạng thái mà nó đã từng là tạo. Nếu bạn đảm bảo rằng tất cả các hàm tạo đều thiết lập các bất biến của lớp, thì đảm bảo rằng những bất biến này sẽ vẫn đúng cho mọi thời đại mà không cần nỗ lực thêm về phía bạn hoặc về phía lập trình viên sử dụng lớp học. Đối tư ợng có thể thay đổi, mặt khác, có thể có không gian trạng thái phức tạp tùy ý. Nếu tài liệu không cung cấp mô tả chính xác về các chuyển đổi trạng thái đư ợc thực hiện bởi các phư ơ ng thức muta tor, có thể khó hoặc không thể sử dụng một lớp có thể thay đổi một cách đáng tin cậy.

Các đối tư ợng bất biến vốn dĩ an toàn theo luồng; chúng không yêu cầu phân vùng đồng bộ hóa. Chúng không thể bị hỏng bởi nhiều luồng truy cập đồng thời với nhau. Đây là cách tiếp cận dễ dàng nhất để đạt đư ợc sự an toàn của luồng. Trên thực tế, không một luồng nào có thể quan sát bất kỳ ảnh hư ởng nào của một luồng khác lên một đối tư ợng không thể thay đổi. Do đó, các đối tư ợng bất biến có thể đư ợc chia sẻ một cách tự do. Các lớp bất biến nên tận dụng lợi thế này bằng cách khuyến khích khách hàng sử dụng lại các phiên bản hiện có ở bất kỳ đâu

Machine Translated by Google

có thể được. Một cách dễ dàng để làm điều này là cung cấp các hằng số cuối cùng tĩnh công khai cho các giá trị được sử dụng thường xuyên. Ví dụ: lớp Complex có thể cung cấp những stants:

```
public static final Complex ZERO = new Complex (0, 0);
public static final Complex ONE = new Complex (1, 0);
public static final Complex I = new Complex (0, 1);
```

Cách tiếp cận này có thể được thực hiện thêm một bước nữa. Một lớp không thay đổi có thể cung cấp các nhà máy tĩnh (Mục 1) để lưu vào bộ nhớ cache các phiên bản được yêu cầu thường xuyên để tránh tạo ra các phiên bản mới khi những phiên bản hiện có sẽ làm. Tất cả các lớp nguyên thủy đóng hộp và BigInteger đều thực hiện điều này. Việc sử dụng các nhà máy tĩnh như vậy khiến các máy khách chia sẻ các phiên bản thay vì tạo các phiên bản mới, giảm dung lượng bộ nhớ và chi phí thu gom rác. Việc chọn các nhà máy tĩnh thay cho các công cụ xây dựng công cộng khi thiết kế một lớp mới cho phép bạn thêm vào bộ nhớ đệm sau này mà không cần sửa đổi các ứng dụng khách.

Một hệ quả của thực tế là các đối tượng bất biến có thể được chia sẻ tự do là bạn không bao giờ phải tạo các bản sao phòng thủ (Mục 39). Trên thực tế, bạn không bao giờ phải tạo bất kỳ bản sao nào bởi vì các bản sao sẽ mãi mãi tương đương với niên hạn gốc. Do đó, bạn không cần và không nên cung cấp phương thức sao chép hoặc phương thức sao chép (Mục 11) trên một lớp không thay đổi. Điều này chưa được hiểu rõ trong những ngày đầu của nền tảng Java, vì vậy lớp String có một hàm tạo bản sao, nhưng nó hiếm khi được sử dụng (Mục 5).

Bạn không chỉ có thể chia sẻ các đối tượng bất biến mà còn có thể chia sẻ giữa các đối tượng của chúng. Ví dụ, lớp BigInteger sử dụng biểu diễn độ lớn dấu hiệu bên trong. Dấu hiệu được biểu thị bằng một int và độ lớn được biểu thị bằng một mảng int . Phương pháp phủ định tạo ra một BigInteger mới có độ lớn và dấu hiệu ngược lại. Nó không cần sao chép mảng; mầm BigInte mới được tạo trỏ đến mảng bên trong giống như mảng ban đầu.

Các đối tượng bất biến tạo nên các khối xây dựng tuyệt vời cho các đối tượng khác, cho dù có thể thay đổi hay bất biến. Sẽ dễ dàng hơn nhiều để duy trì các bất biến của một đối tượng phức tạp nếu bạn biết rằng các đối tượng thành phần của nó sẽ không thay đổi bên dưới nó. Một trường hợp đặc biệt của nguyên tắc này là các đối tượng bất biến tạo ra các khóa bản đồ và các phần tử thiết lập tuyệt vời: bạn không phải lo lắng về việc giá trị của chúng thay đổi khi chúng ở trong bản đồ hoặc tập hợp, điều này sẽ phá hủy các bất biến của bản đồ hoặc tập hợp.

Nhược điểm thực sự duy nhất của các lớp bất biến là chúng yêu cầu một đối tượng riêng biệt cho mỗi giá trị riêng biệt. Việc tạo các đối tượng này có thể tốn kém, đặc biệt nếu chúng lớn. Ví dụ: giả sử bạn có Số nguyên lớn triệu bit và bạn muốn thay đổi bit bậc thấp của nó:

Machine Translated by Google


```
BigInteger moby = ...;
moby = moby.flipBit (0);
```

Phư ơ ng thức flipBit tạo ra một phiên bản BigInteger mới , cũng dài một triệu bit, khác với bản gốc chỉ một bit. Hoạt động yêu cầu thời gian và không gian tỷ lệ với kích thư ớc của BigInteger. Ngư ợc lại điều này với java.util.BitSet. Giống như BigInteger, BitSet đại diện cho một chuỗi bit dài tùy ý, như ng không giống như BigInteger, BitSet có thể thay đổi đư ợc. Lớp BitSet cung cấp một phư ơ ng thức cho phép bạn thay đổi trạng thái của một bit đơ n lẻ của một triệu bit trong thời gian không đổi.

Vấn đề hiệu suất sẽ đư ợc phóng đại nếu bạn thực hiện một thao tác nhiều bư ớc tạo ra một đối tư ợng mới ở mỗi bư ớc, cuối cùng loại bỏ tất cả các đối tư ợng ngoại trừ kết quả cuối cùng. Có hai cách tiếp cận để đối phó với vấn đề này. Đầu tiên là đoán những hoạt động nhiều bư ớc nào sẽ thư ờng đư ợc yêu cầu và cung cấp chúng dư ới dạng nguyên thủy. Nếu một hoạt động nhiều bư ớc đư ợc cung cấp dư ới dạng nguyên thủy, thì lớp bất biến không phải tạo một đối tư ợng riêng biệt ở mỗi bư ớc. Về mặt nội bộ, lớp immuta ble có thể đư ợc tùy ý thông minh. Ví dụ: BigInteger có một "lớp đồng hành" có thể thay đổi gói-riêng mà nó sử dụng để tăng tốc các hoạt động nhiều bư ớc như lũy thừa mô-đun. Việc sử dụng lớp đồng hành có thể thay đổi sẽ khó hơ n nhiều so với sử dụng BigInteger vì tất cả các lý do đã nêu trư ớc đó, như ng may mắn là bạn không phải làm vậy: những ngư ời triển khai BigInteger đã làm công việc khó khăn cho bạn.

Phư ơ ng pháp tiếp cận lớp đồng hành có thể thay đổi theo gói-riêng hoạt động tốt nếu bạn có thể dự đoán chính xác các hoạt động đa tầng phức tạp mà khách hàng sẽ muốn thực hiện trên lớp bất biến của bạn. Nếu không, thì cách tốt nhất của bạn là cung cấp một lớp đồng hành có thể thay đổi công khai . Ví dụ chính của cách tiếp cận này trong các thư viện nền tảng Java là lớp String , có lớp đồng hành có thể thay đổi là StringBuilder (và StringBuffer phần lớn đã lỗi thời). Có thể cho rằng, BitSet đóng vai trò là ngư ời bạn đồng hành có thể thay đổi đư ợc với BigInteger trong một số trư ờng hợp nhất định.

Bây giờ bạn đã biết cách tạo một lớp bất biến và bạn hiểu những ư u và như ợc điểm của tính bất biến, chúng ta hãy thảo luận về một vài lựa chọn thay thế thiết kế. Nhớ lại rằng để đảm bảo tính bất biến, một lớp không đư ợc cho phép chính nó đư ợc phân lớp.
Thông thư ờng, điều này đư ợc thực hiện bằng cách đư a ra kết quả cuối cùng của lớp học, như ng có một cách khác linh hoạt hơ n để thực hiện. Giải pháp thay thế để tạo một lớp cuối cùng không thay đổi là đặt tất cả các hàm tạo của nó ở chế độ riêng tư hoặc gói riêng tư và thêm các nhà máy tĩnh công cộng thay cho các hàm tạo công cộng (Mục 1).

Machine Translated by Google

Để tạo ra bê tông này, đây là cách Complex sẽ trông như thế nào nếu bạn thực hiện theo cách tiếp cận này:

```
// Lớp bất biến với các nhà máy tĩnh thay vì các hàm tạo public class Complex
{private final double re; tư nhân cuối cùng im lặng;



     private Complex (double re, double im) {this.re =
          re; this.im = im;

     }

     public static Giá trị phức hợpOf (double re, double im) {
          return new Complex (re, im);
     }

     ... // Phần còn lại không thay đổi
}
```

Mặc dù cách tiếp cận này không đư ợc sử dụng phổ biến, như ng nó thư ờng là giải pháp thay thế tốt nhất. Nó linh hoạt nhất vì nó cho phép sử dụng nhiều lớp tation implemen gói-riêng. Đối với các máy khách của nó cư trú bên ngoài gói của nó, lớp không thay đổi có hiệu quả là cuối cùng vì không thể mở rộng một lớp đến từ một gói khác và lớp đó thiếu hàm tạo công khai hoặc đư ợc bảo vệ. Bên cạnh việc cho phép tính linh hoạt của nhiều lớp triển khai, cách tiếp cận này giúp bạn có thể điều chỉnh hiệu suất của lớp trong các bản phát hành tiếp theo bằng cách cải thiện khả năng lư u vào bộ nhớ đệm đối tư ợng của các nhà máy tĩnh.

Các nhà máy tĩnh có nhiều lợi thế khác so với các nhà máy xây dựng, như đã thảo luận trong Mục 1. Ví dụ, giả sử rằng bạn muốn cung cấp một phư ơ ng tiện để tạo một số com plex dựa trên các tọa độ cực của nó. Điều này sẽ rất lộn xộn khi sử dụng các cấu trúc con vì hàm tạo tự nhiên sẽ có cùng một chữ ký mà chúng ta đã sử dụng: Complex (double, double). Với các nhà máy tĩnh, điều đó thật dễ dàng. Chỉ cần thêm một nhà máy tĩnh thứ hai với tên xác định rõ ràng chức năng của nó:

```
public static Complex valueOfPolar (double r, double theta) {return new
     Complex (r * Math.cos (theta), r * Math.sin (theta));

}
```

Ngư ời ta không hiểu rộng rãi rằng các lớp bất biến phải là lớp cuối cùng một cách hiệu quả khi BigInteger và BigDecimal đư ợc viết, vì vậy tất cả các phư ơ ng thức của chúng có thể là

Machine Translated by Google

ghi đè. Thật không may, điều này không thể đư ợc sửa chữa sau khi thực tế trong khi vẫn duy trì khả năng tư ơ ng thích ngư ợc. Nếu bạn viết một lớp có tính bảo mật phụ thuộc vào tính bất biến của đối số BigInteger hoặc BigDecimal từ một máy khách không đáng tin cậy, bạn phải kiểm tra xem đối số đó là BigInteger hoặc BigDecimal "thực" chứ không phải là một phiên bản của lớp con không đáng tin cậy. Nếu nó là cái thứ hai, bạn phải xác định sao chép nó một cách cẩn thận với giả định rằng nó có thể thay đổi đư ợc (Mục 39):

```
public static BigInteger safeInstance (BigInteger val) {if (val.getClass
     ()! = BigInteger.class)
          trả về BigInteger mới (val.toByteArray ());
     trả lại val;
}
```

Danh sách các quy tắc cho các lớp bất biến ở đầu mục này nói rằng không có phư ơ ng thức nào có thể sửa đổi đối tư ợng và tất cả các trư ờng của nó phải là trư ờng cuối cùng. Trên thực tế, các quy tắc này mạnh hơ n một chút so với mức cần thiết và có thể đư ợc nới lỏng để cải thiện hiệu suất. Trên thực tế, không có phư ơ ng pháp nào có thể tạo ra sự thay đổi có thể nhìn thấy bên ngoài trong trạng thái của đối tư ợng. Tuy nhiên, một số lớp không thay đổi có một hoặc nhiều trư ờng không phải là trư ờng cuối cùng, trong đó chúng lư u vào bộ nhớ cache các kết quả của các phép tính tốn kém trong lần đầu tiên cần thiết. Nếu yêu cầu lại cùng một giá trị, giá trị đã lư u trong bộ nhớ cache sẽ đư ợc trả về, giúp tiết kiệm chi phí tính toán lại. Thủ thuật này hoạt động chính xác vì đối tư ợng là bất biến, điều này đảm bảo rằng phép tính sẽ mang lại kết quả tư ơ ng tự nếu nó đư ợc lặp lại.

Ví dụ: phư ơ ng thức hashCode của PhoneNumber (Mục 9, trang 49) tính toán mã băm lần đầu tiên nó đư ợc gọi và lư u vào bộ nhớ cache trong trư ờng hợp nó đư ợc gọi lại.
Kỹ thuật này, một ví dụ về khởi tạo lư ời biếng (Mục 71), cũng đư ợc sử dụng bởi String.

Một lư u ý cần đư ợc thêm vào liên quan đến khả năng tuần tự hóa. Nếu bạn chọn để lớp không thay đổi của mình triển khai Serializable và nó chứa một hoặc nhiều trư ờng tham chiếu đến các đối tư ợng có thể thay đổi, bạn phải cung cấp phư ơ ng thức readObject hoặc readResolve rõ ràng hoặc sử dụng phư ơ ng thức ObjectOutputStream.writeUnshared và ObjectInputStream.readUnshared , ngay cả khi các phư ơ ng thức đư ợc tuần tự hóa mặc định hình thức có thể chấp nhận đư ợc. Nếu không, kẻ tấn công có thể tạo một phiên bản có thể thay đổi của lớp không hoàn toàn bất biến của bạn. Chủ đề này đư ợc đề cập chi tiết trong Mục 76.

Tóm lại, hãy chống lại sự thôi thúc viết một phư ơ ng thức thiết lập cho mọi phư ơ ng thức get . Các lớp phải là bất biến trừ khi có lý do chính đáng để khiến chúng có thể thay đổi. Các lớp bất biến cung cấp nhiều lợi thế, và điểm trừ duy nhất của chúng là khả năng gây ra các vấn đề về hiệu suất trong một số trư ờng hợp nhất định. Bạn nên luôn làm cho các đối tư ợng có giá trị nhỏ, chẳng hạn như PhoneNumber và Complex, không thể thay đổi. (Có một số lớp trong thư viện nền tảng Java, chẳng hạn như

Machine Translated by Google

java.util.Date và java.awt.Point, lẽ ra phải là bất biến nhưng không phải vậy.) Bạn nên nghiêm túc xem xét việc tạo các đối tượng có giá trị lớn hơn, chẳng hạn như String và BigInteger, cũng không thể thay đổi. Bạn chỉ nên cung cấp một lớp đồng hành có thể thay đổi công khai cho lớp bất biến của mình khi bạn đã xác nhận rằng nó cần thiết để đạt được hiệu suất thỏa đáng (Mục 55).

Có một số lớp mà tính bất biến là không thực tế. Nếu một lớp không thể được thực hiện là bất biến, hãy hạn chế khả năng thay đổi của nó càng nhiều càng tốt. Việc giảm số lượng trạng thái mà một đối tượng có thể tồn tại sẽ giúp dễ dàng lập luận về đối tượng hơn và giảm khả năng xảy ra lỗi. Do đó, hãy đưa mọi lĩnh vực vào chung kết trừ khi có lý do thuyết phục để biến nó thành không chung kết.

Các trình xây dựng nên tạo các đối tượng được khởi tạo đầy đủ với tất cả các bất biến của chúng được thiết lập. Không cung cấp phương thức khởi tạo công khai tách biệt với trình cấu trúc con hoặc nhà máy tĩnh trừ khi có lý do thuyết phục để làm như vậy. Tương tự, không cung cấp phương thức "reinitialize" cho phép sử dụng lại một đối tượng như thể nó đã được xây dựng với một trạng thái ban đầu khác. Các phương pháp như vậy thường mang lại rất ít lợi ích về hiệu suất với chi phí là sự phức tạp tăng lên.

Lớp TimerTask minh họa cho những nguyên tắc này. Nó có thể thay đổi, nhưng không gian trạng thái của nó được giữ cố ý nhỏ. Bạn tạo một thể hiện, lên lịch cho nó để thực thi và hủy nó một cách tùy ý. Khi nhiệm vụ hẹn giờ đã hoàn thành hoặc đã có thể hoàn thành, bạn không thể lên lịch lại.

Một lưu ý cuối cùng cần được thêm vào liên quan đến lớp Complex trong mục này. Ví dụ này chỉ nhằm minh họa tính bất biến. Nó không phải là một triển khai số phức có sức mạnh công nghiệp. Nó sử dụng các công thức tiêu chuẩn cho phép nhân và phép chia phức tạp, không được làm tròn chính xác và cung cấp ngữ nghĩa kém cho các NaN và số nguyên phức tạp [Kahan91, Smith62, Thomas94].

Machine Translated by Google

Khoản 16: Cơ cấu ủng hộ đối với thừa kế

Kế thừa là một cách hiệu quả để tái sử dụng mã, như ng nó không phải lúc nào cũng là tốt nhất công cụ cho công việc. Sử dụng không phù hợp dẫn đến phần mềm dễ hỏng. Nó là an toàn để sử dụng kế thừa trong một gói, trong đó lớp con và lớp cha triển khai dư ới sự kiểm soát của cùng một lập trình viên. Cũng an toàn khi sử dụng kế thừa ance khi mở rộng các lớp đư ợc thiết kế và lập tài liệu đặc biệt cho phần mở rộng (Tiết 17). Kế thừa từ các lớp bê tông thông thư ờng qua các ranh giới gói, tuy nhiên, là nguy hiểm. Xin nhắc lại, cuốn sách này sử dụng từ "kế thừa" để nghĩa là kế thừa thực thi (khi một lớp này mở rộng lớp khác). Vấn đề đư ợc thảo luận trong mục này không áp dụng cho việc kế thừa giao diện (khi một lớp ghép vào một giao diện hoặc khi một giao diện này mở rộng giao diện khác).

Không giống như lệnh gọi phư ơ ng thức, kế thừa vi phạm tính đóng gói [Snyder86]. Nói cách khác, một lớp con phụ thuộc vào chi tiết triển khai của lớp cha của nó cho chức năng thích hợp của nó. Việc triển khai lớp cha có thể thay đổi từ khi phát hành để phát hành và nếu có, lớp con có thể bị hỏng, mặc dù mã của nó không đã đư ợc xúc động. Do đó, một lớp con phải phát triển song song với lớp siêu của nó, trừ khi các tác giả của lớp cha đó đã thiết kế và ghi lại nó một cách cụ thể cho mục đích đư ợc mở rộng.

Để làm cho điều này cụ thể, giả sử chúng ta có một chư ơ ng trình sử dụng HashSet. Để điều chỉnh hiệu suất của chư ơ ng trình của chúng tôi, chúng tôi cần truy vấn HashSet về cách nhiều phần tử đã đư ợc thêm vào kể từ khi nó đư ợc tạo (đừng nhầm với kích thư ớc hiện tại, sẽ giảm xuống khi một phần tử bị xóa). Để cung cấp tính chất vui nhộn này, chúng tôi viết một biến thể HashSet giúp đếm số lần thử chèn phần tử và xuất một trình truy cập cho số lư ợng này. Lớp HashSet có hai phư ơ ng thức có khả năng thêm phần tử, add và addAll, vì vậy chúng tôi ghi đè cả hai phư ơ ng pháp này:

```
// Bị hỏng - Sử dụng quyền thừa kế không phù hợp!
public class InstrumentedHashSet <E> mở rộng HashSet <E> {
     // Số lần chèn phần tử đã cố gắng
     private int addCount = 0;

     public InstrumentedHashSet () {
     }

     public InstrumentedHashSet (int initCap, float loadFactor) {
          siêu (initCap, loadFactor);
     }
```

Machine Translated by Google


```
        @Override public boolean add (E e) {addCount
            ++; trả về super.add (e);

        }
        @Override public boolean addAll (Collection <? Expand E> c) {addCount + =
            c.size (); trả về super.addAll (c);


        } public int getAddCount ()
            {return addCount;
        }
    }
```

Lớp này trông hợp lý, như ng nó không hoạt động. Giả sử chúng ta tạo một thể hiện và thêm ba phần tử bằng phư ơ ng thức addAll :

```
    InstrumentedHashSet <String> s =
        new InstrumentedHashSet <String> ();
    s.addAll (Arrays.asList ("Snap", "Crackle", "Pop"));
```

Chúng tôi mong đợi phư ơ ng thức getAddCount trả về ba tại thời điểm này, như ng nó trả về sáu. Có chuyện gì? Bên trong, phư ơ ng thức addAll của HashSet đư ợc đề cập bên trên phư ơ ng thức add của nó , mặc dù khá hợp lý, HashSet không ghi lại chi tiết triển khai này. Phư ơ ng thức addAll trong InstrumentedHash Set đã thêm ba vào addCount và sau đó gọi triển khai addAll của HashSet bằng cách sử dụng super.addAll. Điều này lần lư ợt gọi phư ơ ng thức thêm , như đư ợc ghi đè trong InstrumentedHashSet, một lần cho mỗi phần tử. Mỗi trong số ba lời gọi này đã thêm một lệnh nữa vào addCount, với tổng số tăng thêm là sáu: mỗi phần tử đư ợc thêm bằng phư ơ ng thức addAll đư ợc tính hai lần.

Chúng tôi có thể "sửa chữa" lớp con bằng cách loại bỏ việc ghi đè phư ơ ng thức addAll của nó . Mặc dù lớp kết quả sẽ hoạt động, như ng nó sẽ phụ thuộc vào chức năng thích hợp của nó trên thực tế là phư ơ ng thức addAll của HashSet đư ợc triển khai trên phư ơ ng thức add của nó . "Tự sử dụng" này là một chi tiết triển khai, không đư ợc đảm bảo giữ trong tất cả các đề cập liên quan đến nền tảng Java và có thể thay đổi từ bản phát hành này sang lần phát hành khác. Do đó, lớp InstrumentedHashSet kết quả sẽ rất mỏng manh.

Sẽ tốt hơ n một chút nếu ghi đè phư ơ ng thức addAll để lặp lại bộ sư u tập đư ợc chỉ định, gọi phư ơ ng thức thêm một lần cho mỗi phần tử. Điều này sẽ đảm bảo kết quả chính xác cho dù phư ơ ng thức addAll của HashSet có đư ợc triển khai trên phư ơ ng thức add của nó hay không , bởi vì việc triển khai addAll của HashSet sẽ không còn đư ợc gọi nữa. Tuy nhiên, kỹ thuật này không giải quyết đư ợc tất cả các vấn đề của chúng ta. Nó tư ơ ng đư ơ ng với việc thực hiện lại các phư ơ ng thức siêu lớp có thể có hoặc không

Machine Translated by Google

dẫn đến việc tự sử dụng khó khăn, mất thời gian và dễ xảy ra sai sót.
Ngoài ra, không phải lúc nào cũng có thể thực hiện được, vì một số phương pháp không thể được triển khai nếu không có quyền truy cập vào các trường riêng tư không thể truy cập vào lớp con.

Một nguyên nhân liên quan của sự dễ vỡ trong các lớp con là lớp cha của chúng có thể có được các phương thức mới trong các bản phát hành tiếp theo. Giả sử một chương trình phụ thuộc vào tính bảo mật của nó trên thực tế là tất cả các phần tử được chèn vào một số tập hợp thỏa mãn một số vị từ. Điều này có thể được đảm bảo bằng cách phân lớp tập hợp và ghi đè mỗi phương thức có khả năng thêm một phần tử để đảm bảo rằng vị từ được thỏa mãn trước khi thêm phần tử. Điều này hoạt động tốt cho đến khi một phương thức mới có khả năng chèn một phần tử được thêm vào lớp cha trong bản phát hành tiếp theo. Khi điều này xảy ra, có thể thêm một phần tử "bất hợp pháp" chỉ bằng cách gọi phương thức mới, phương thức này không bị ghi đè trong lớp con. Đây không phải là một vấn đề lý thuyết thuần túy. Một số lỗ hổng bảo mật về bản chất này đã phải được sửa khi Hashtable và Vector được trang bị lại để tham gia vào Khung bộ sưu tập.

Cả hai vấn đề trên đều bắt nguồn từ các phương pháp overriding. Bạn có thể nghĩ rằng việc mở rộng một lớp là an toàn nếu bạn chỉ thêm các phương thức mới và không ghi đè các phương thức hiện có. Mặc dù loại tiện ích mở rộng này an toàn hơn nhiều, nhưng không phải là không có rủi ro. Nếu lớp cha có được một phương thức mới trong bản phát hành tiếp theo và bạn không may mắn đã cung cấp cho lớp con một phương thức có cùng chữ ký và kiểu trả về khác, lớp con của bạn sẽ không biên dịch nữa [JLS, 8.4.8.3]. Nếu bạn đã cấp cho lớp con một phương thức có cùng chữ ký và kiểu trả về như phương thức lớp cha mới, thì bây giờ bạn đang ghi đè nó, vì vậy bạn phải đối mặt với hai vấn đề được mô tả ở trên. Hơn nữa, có thể nghi ngờ rằng phương thức của bạn sẽ hoàn thành hợp đồng của phương thức lớp cha mới, vì hợp đồng đó chưa được ghi mười khi bạn viết phương thức lớp con.

May mắn thay, có một cách để tránh tất cả các vấn đề được mô tả trước đó. Thay vì mở rộng một lớp hiện có, hãy cung cấp cho lớp mới của bạn một trường riêng tư tham chiếu đến một thể hiện của lớp hiện có. Thiết kế này được gọi là thành phần vì lớp hiện có trở thành một thành phần của lớp mới. Mỗi phương thức thể hiện trong lớp mới gọi phương thức tương ứng trên thể hiện chứa của lớp hiện có và trả về kết quả. Điều này được gọi là chuyển tiếp và các meth ods trong lớp mới được gọi là các phương thức chuyển tiếp. Lớp kết quả sẽ vững chắc, không phụ thuộc vào chi tiết triển khai của lớp hiện có. Ngay cả việc thêm các phương thức mới vào lớp hiện có cũng sẽ không ảnh hưởng đến lớp mới. Để làm cho điều này cụ thể, đây là một sự thay thế cho InstrumentedHashSet sử dụng phương pháp thành phần và chuyển tiếp. Lưu ý rằng việc triển khai được chia thành hai phần, chính lớp đó và một lớp chuyển tiếp có thể sử dụng lại, chứa tất cả các phương thức chuyển tiếp và không có gì khác:

Machine Translated by Google


```
// Lớp Wrapper - sử dụng thành phần thay cho kế thừa
public class InstrumentedSet <E> mở rộng ForwardingSet <E> {
     private int addCount = 0;

     public InstrumentedSet (Set <E> s) {
          siêu (các);
     }

     @Override public boolean add (E e) {
          addCount ++;
          trả về super.add (e);
     }
     @Override public boolean addAll (Collection <? Expand E> c) {
          addCount + = c.size ();
          trả về super.addAll (c);
     }
     public int getAddCount () {
          trả về addCount;
     }
}

// Lớp chuyển tiếp có thể tái sử dụng
public class ForwardingSet <E> thực hiện Set <E> {
     private final Set <E> s;
     public ForwardingSet (Set <E> s) {this.s = s; }

     public void clear ()}                        {s.clear ();
     public boolean chứa (Object o) {return s.contains (o); }
     public boolean isEmpty () {return s.isEmpty (); }
     public int size () {return s.size (); }
     public Iterator <E> iterator () {return s.iterator (); }
     public boolean add (E e) {return s.add (e); }
     public boolean remove (Object o) {return s.remove (o); }
     boolean công cộng chứaAll (Bộ sưu tập <?> c)
                                          {return s.containsAll (c); }
     public boolean addAll (Collection <? expand E> c)
                                          {return s.addAll (c);              }
     public boolean removeAll (Bộ sưu tập <?> c)
                                          {return s.removeAll (c); }
     public boolean keepAll (Bộ sưu tập <?> c)
                                          {return s.retainAll (c); }
     public Object [] toArray () {return s.toArray (); }
     public <T> T [] toArray (T [] a) {return s.toArray (a); }
     @Override public boolean bằng (Đối tượng o)
                                               {return s.equals (o); }
     @Override public int hashCode () {return s.hashCode (); }
     @Override public String toString () {return s.toString (); }
}
```

Machine Translated by Google

Thiết kế của lớp InstrumentedSet được kích hoạt bởi sự tồn tại của giao diện Set , giao diện này nắm bắt chức năng của lớp HashSet . Bên cạnh sự mạnh mẽ, thiết kế này còn vô cùng linh hoạt. Lớp InstrumentedSet chèn giao diện Set và có một hàm tạo duy nhất có đối số cũng thuộc kiểu Set. Về bản chất, lớp học chuyển đổi một Set thành một Set khác, thêm chức năng cố vấn nhạc cụ. Không giống như cách tiếp cận dựa trên kế thừa, chỉ hoạt động cho một lớp cụ thể và yêu cầu một phương thức khởi tạo riêng cho mỗi phương thức khởi tạo được hỗ trợ trong lớp cha, lớp wrapper có thể được sử dụng để thiết bị bất kỳ triển khai Set nào và sẽ hoạt động cùng với bất kỳ phương thức khởi tạo nào đã có từ trước:

```
Đặt <Ngày> s = new InstrumentedSet <Ngày> (TreeSet mới <Ngày> (cmp));
Đặt <E> s2 = new InstrumentedSet <E> (new HashSet <E> (dung lượng));
```

Lớp InstrumentedSet thậm chí có thể được sử dụng để tạo thiết bị tạm thời cho một tập hợp phiên bản đã được sử dụng mà không có thiết bị đo đạc:

```
static void walk (Đặt <Dog> dog)
     {InstrumentedSet <Dog> iDogs = new InstrumentedSet <Dog>
     (dog); ... // Trong phương thức này, sử dụng iDogs thay vì dog
}
```

Lớp InstrumentedSet được gọi là lớp bao bọc bởi vì mỗi cá thể InstrumentedSet chứa (“bao bọc”) một cá thể Set khác . Đây còn được gọi là mẫu Decorator [Gamma95, p. 175], bởi vì lớp Instrumented Set “trang trí” một bộ bằng cách thêm thiết bị đo đạc. Đôi khi sự kết hợp của thành phần và chuyển tiếp được gọi một cách lỏng lẻo là ủy quyền.

Về mặt kỹ thuật, nó không phải là ủy quyền trừ khi đối tượng wrapper tự chuyển sang đối tượng được bao bọc [Lieberman86; Gamma95, tr. 20].

Nhược điểm của các lớp wrapper là rất ít. Một lưu ý là các lớp trình bao bọc không thích hợp để sử dụng trong các khuôn khổ gọi lại, trong đó các đối tượng chuyển tự tham chiếu đến các đối tượng khác cho các lần gọi tiếp theo (“gọi lại”). Bởi vì một đối tượng được bọc không biết về trình bao bọc của nó, nên nó chuyển một tham chiếu đến chính nó (cái này) và gọi lại loại bỏ trình bao bọc. Đây được gọi là vấn đề TỰ TIN [Lieberman86]. Một số người lo lắng về tác động hiệu suất của lời gọi phương thức chuyển tiếp hoặc tác động đến dấu chân bộ nhớ của các đối tượng trình bao bọc. Cả hai đều không có tác động nhiều trong thực tế. Thật là tẻ nhạt khi viết các phương thức chuyển tiếp, nhưng bạn phải viết lớp chuyển tiếp cho mỗi giao diện chỉ một lần và đối với các lớp chuyển tiếp có thể được cung cấp cho bạn bởi gói chứa giao diện.

Kế thừa chỉ thích hợp trong các trường hợp mà lớp con thực sự là một kiểu con của lớp cha. Nói cách khác, một lớp B chỉ nên mở rộng một lớp A nếu

Machine Translated by Google

giữa hai lớp tồn tại một mối quan hệ "là một". Nếu bạn bị cám dỗ để có một lớp B kéo dài một lớp A, hãy tự hỏi bản thân mình câu hỏi: Mọi B có thực sự là A không? Nếu bạn không thể thành thật trả lời có cho câu hỏi này, B không nên mở rộng A. Nếu câu trả lời là không, thường xảy ra trường hợp B phải chứa một phiên bản riêng của A và hiển thị một API nhỏ hơn và đơn giản hơn: A không phải là một phần thiết yếu của B, chỉ đơn thuần là một chi tiết của việc thực hiện nó.

Có một số vi phạm rõ ràng nguyên tắc này trong các thư viện nền tảng Java. Ví dụ: ngăn xếp không phải là vectơ , vì vậy Ngăn xếp không được mở rộng Vectơ . Tương tự, một danh sách thuộc tính không phải là một bảng băm, vì vậy Thuộc tính không nên mở rộng Hashtable. Trong cả hai trường hợp, bố cục sẽ được ưu tiên hơn.

Nếu bạn sử dụng kế thừa khi bố cục phù hợp, bạn không cần thiết phải tiết lộ chi tiết triển khai. API kết quả ràng buộc bạn với sự cố vấn ban đầu, giới hạn mãi mãi hiệu suất của lớp của bạn. Nghiêm trọng hơn, bằng cách tiết lộ nội dung bên trong, bạn cho phép khách hàng truy cập trực tiếp vào chúng. Ít nhất, điều này có thể dẫn đến ngữ nghĩa khó hiểu. Ví dụ: nếu p tham chiếu đến một cá thể Thuộc tính , thì p.getProperty (khóa) có thể mang lại kết quả khác với p.get (khóa): phương thức trước có tính đến giá trị mặc định, trong khi phương thức sau, được kế thừa từ Hashtable , không phải. Nghiêm trọng nhất, máy khách có thể làm hỏng các bất biến của lớp con bằng cách sửa đổi lớp cha trực tiếp. Trong trường hợp Thuộc tính, các nhà thiết kế dự định rằng chỉ các chuỗi được phép làm khóa và giá trị, nhưng quyền truy cập trực tiếp vào Hashtable cơ bản cho phép bất biến này bị vi phạm. Khi phần bất biến này bị vi phạm, bạn không thể sử dụng các phần khác của API thuộc tính (tải và lưu trữ) được nữa. Vào thời điểm lỗi này được phát hiện, đã quá muộn để sửa nó vì các máy khách phụ thuộc vào việc sử dụng các khóa và giá trị nonstring.

Có một bộ câu hỏi cuối cùng mà bạn nên tự hỏi trước khi quyết định sử dụng tài sản thừa kế thay cho sáng tác. Lớp mà bạn dự định mở rộng có bất kỳ sai sót nào trong API của nó không? Nếu vậy, bạn có cảm thấy thoải mái khi tuyên truyền những sai sót đó vào API của lớp mình không? Tính kế thừa truyền bá bất kỳ sai sót nào trong API của lớp cha, trong khi thành phần cho phép bạn thiết kế một API mới để che giấu những sai sót này.

Tóm lại, kế thừa rất mạnh, nhưng nó có vấn đề vì nó vi phạm tính đóng gói. Nó chỉ thích hợp khi tồn tại một mối quan hệ kiểu con thực sự giữa lớp con và lớp cha. Ngay cả khi đó, kế thừa có thể dẫn đến sự dễ vỡ nếu lớp con nằm trong một gói khác với lớp cha và lớp cha không được thiết kế để kế thừa. Để tránh sự mong manh này, hãy sử dụng thành phần và chuyển tiếp thay vì kế thừa, đặc biệt nếu tồn tại một giao diện thích hợp để triển khai một lớp trình bao bọc. Các lớp wrapper không chỉ mạnh hơn các lớp con mà còn mạnh hơn.

Machine Translated by Google

Mục 17: Thiết kế và tài liệu để thừa kế nếu không thì cấm nó

Mục 16 đã cảnh báo bạn về những nguy cơ khi phân lớp một lớp "ngoại lai" không đư ợc thiết kế và ghi chép lại để kế thừa. Vậy một lớp đư ợc thiết kế và lập thành tài liệu để kế thừa có nghĩa là gì?

Đầu tiên, lớp phải ghi lại chính xác ảnh hư ởng của việc ghi đè bất kỳ phư ơ ng thức nào. Nói cách khác, lớp phải ghi lại việc tự sử dụng các phư ơ ng thức có thể ghi đè của nó. Đối với mỗi phư ơ ng thức hoặc hàm tạo công khai hoặc đư ợc bảo vệ, tài liệu phải chỉ ra các phư ơ ng thức có thể ghi đè mà phư ơ ng thức hoặc hàm tạo gọi ra, theo trình tự nào và kết quả của mỗi lệnh gọi ảnh hư ởng như thế nào đến quá trình xử lý tiếp theo. (Bằng cách ghi đè, chúng tôi có nghĩa là không chính thức và công khai hoặc đư ợc bảo vệ.) Nói chung, một lớp phải ghi lại bất kỳ trư ờng hợp nào mà nó có thể gọi một phư ơ ng thức có thể ghi đè. Ví dụ, lời gọi có thể đến từ các luồng nền hoặc bộ khởi tạo tĩnh.

Theo quy ư ớc, một phư ơ ng thức gọi các phư ơ ng thức có thể ghi đè chứa một phần mô tả về các lệnh gọi này ở cuối chú thích tài liệu của nó. Mô tả bắt đầu bằng cụm từ "Việc triển khai này". Cụm từ này không nên đư ợc sử dụng để chỉ ra rằng hành vi có thể thay đổi từ phát hành sang phát hành. Nó ngụ ý rằng mô tả liên quan đến hoạt động bên trong của phư ơ ng pháp. Đây là một đề thi, đư ợc sao chép từ đặc tả cho java.util.

public boolean remove (Object o)

> Xóa một phiên bản của phần tử đã chỉ định khỏi bộ sư u tập này, nếu nó có mặt (thao tác tùy chọn). Chính thức hơ n, loại bỏ một phần tử e sao cho (o == null? E == null: o.equals (e)), nếu tập hợp chứa một hoặc nhiều phần tử như vậy. Trả về true nếu tập hợp chứa phần tử đư ợc chỉ định (hoặc tư ơ ng đư ơ ng, nếu tập hợp thay đổi do lệnh gọi).
>
> Việc triển khai này lặp lại bộ sư u tập để tìm kiếm phần tử đư ợc chỉ định. Nếu nó tìm thấy phần tử, nó sẽ xóa phần tử khỏi bộ sư u tập bằng cách sử dụng phư ơ ng thức remove của trình lặp . Lư u ý rằng việc triển khai này sẽ ném ra một ngoại lệ UnsupportedOperationException nếu trình vòng lặp đư ợc trả về bởi phư ơ ng thức trình vòng lặp của bộ sư u tập này không triển khai phư ơ ng thức loại bỏ .

Tài liệu này chắc chắn rằng việc ghi đè phư ơ ng thức trình lặp sẽ ảnh hư ởng đến hoạt động của phư ơ ng thức remove . Hơ n nữa, nó mô tả chính xác hành vi của Iterator đư ợc trả về bởi phư ơ ng thức iterator sẽ ảnh hư ởng như thế nào đến hành vi của phư ơ ng thức remove . Đối chiếu điều này với tình huống trong Mục 16, trong đó

Machine Translated by Google

lập trình viên phân lớp HashSet đơn giản là không thể nói liệu có ghi đè phương thức add sẽ ảnh hưởng đến hoạt động của phương thức addAll .

Nhưng điều này không vi phạm quy định rằng tài liệu API tốt phải mô tả những gì một phương thức nhất định làm và không nó hoạt động như thế nào? Có, nó có! Đây là hậu quả đáng tiếc của thực tế là thừa kế vi phạm tính đóng gói. Đến ghi lại một lớp để nó có thể được phân lớp một cách an toàn, bạn phải mô tả các chi tiết liên quan đến thiết lập mà nếu không thì không được chỉ định.

Thiết kế để kế thừa bao gồm nhiều thứ hơn là chỉ ghi lại các kiểu sử dụng của bản thân. Để cho phép các lập trình viên viết các lớp con hiệu quả mà không gây đau đớn quá mức , lớp có thể phải cung cấp các hook vào hoạt động bên trong của nó dưới dạng các phương thức được bảo vệ được lựa chọn cẩn thận hoặc trong một số trường hợp hiếm hoi là các trường được bảo vệ. Vì ví dụ, hãy xem xét phương thức removeRange từ java.util.AbstractList:

được bảo vệ void removeRange (int fromIndex, int toIndex)

Xóa khỏi danh sách này tất cả các phần tử có chỉ mục nằm giữa fromIndex, bao gồm và toIndex, độc quyền. Thay đổi bất kỳ thành công các phần tử ở bên trái (giảm chỉ số của chúng). Lệnh gọi này rút ngắn ArrayList bởi các phần tử (toIndex - fromIndex) . (Nếu toIndex == fromIndex, điều này hoạt động không có hiệu lực.)

Phương thức này được gọi bằng thao tác rõ ràng trên danh sách này và các danh sách con của nó. Ghi đè phương pháp này để tận dụng lợi thế bên trong của việc đề cập đến danh sách có thể cải thiện đáng kể hiệu suất của hoạt động rõ ràng trên danh sách này và các danh sách phụ của nó.

Việc triển khai này nhận được một trình lặp danh sách được định vị trước fromIndex và thường xuyên gọi lại ListIterator.next theo sau bởi ListIterator.remove, cho đến khi toàn bộ phạm vi đã bị loại bỏ. Lưu ý: Nếu ListIterator.remove yêu cầu thời gian tuyến tính, việc thực hiện này yêu cầu thời gian bậc hai.

Thông số:

fromIndex	chỉ mục của phần tử đầu tiên bị xóa.
Chỉ mục	chỉ mục sau phần tử cuối cùng được xóa.

Phương pháp này không quan tâm đến người dùng cuối của việc triển khai Danh sách . Nó là được cung cấp chỉ để giúp các lớp con dễ dàng cung cấp một phương pháp rõ ràng nhanh chóng trên danh sách phụ. Trong trường hợp không có phương thức removeRange , các lớp con sẽ phải làm với hiệu suất bậc hai khi phương thức rõ ràng được gọi vào danh sách con hoặc viết lại toàn bộ cơ chế Danh sách con từ đầu — không phải là một nhiệm vụ dễ dàng!

Machine Translated by Google

Vì vậy, làm thế nào để bạn quyết định những gì các thành viên được bảo vệ sẽ hiển thị khi bạn thiết kế một lớp để kế thừa? Thật không may, không có viên đạn thần kỳ. Điều tốt nhất bạn có thể làm là suy nghĩ kỹ, đưa ra dự đoán tốt nhất của bạn và sau đó kiểm tra nó bằng cách viết các lớp con. Bạn nên tiết lộ càng ít thành viên được bảo vệ càng tốt, bởi vì mỗi thành viên đại diện cho một cam kết về một chi tiết triển khai. Mặt khác, bạn không được để lộ quá ít, vì thiếu thành viên được bảo vệ có thể làm cho một lớp thực tế không thể sử dụng để kế thừa.

Cách duy nhất để kiểm tra một lớp được thiết kế để kế thừa là viết các lớp con. Nếu bạn bỏ qua một thành viên quan trọng được bảo vệ, việc cố gắng viết một lớp con sẽ khiến việc bỏ sót trở nên rõ ràng một cách đau đớn. Ngược lại, nếu một số lớp con được viết và không có lớp nào sử dụng thành viên được bảo vệ, bạn có thể nên đặt nó ở chế độ riêng tư . Kinh nghiệm cho thấy rằng ba lớp con thường là đủ để kiểm tra một lớp có thể mở rộng. Một hoặc nhiều lớp con này phải được viết bởi một người nào đó không phải là tác giả của lớp con.

Khi bạn thiết kế để kế thừa một lớp có khả năng đạt được mục đích sử dụng rộng rãi, hãy nhận ra rằng bạn đang cam kết mãi mãi với các mẫu tự sử dụng mà bạn ghi lại và các quyết định triển khai tiềm ẩn trong các phương thức và trường được bảo vệ của nó. Những cam kết này có thể gây khó khăn hoặc không thể cải thiện hiệu quả hoặc chức năng của lớp trong bản phát hành tiếp theo. Do đó, bạn phải kiểm tra lớp của mình bằng cách viết các lớp con trước khi phát hành.

Ngoài ra, hãy lưu ý rằng tài liệu đặc biệt cần thiết để thừa kế lộn xộn với tài liệu thông thường, được thiết kế cho các lập trình viên tạo các thể hiện của lớp bạn và gọi các phương thức trên chúng. Kể từ khi viết bài này, có rất ít công cụ hoặc quy ước nhận xét để tách tài liệu API thông thường khỏi thông tin chỉ quan tâm đến các lập trình viên thực hiện các lớp con.

Có một vài hạn chế nữa mà một lớp phải tuân theo để cho phép kế thừa. Trình tạo không được gọi các phương thức có thể ghi đè, trực tiếp hoặc gián tiếp. Nếu bạn vi phạm quy tắc này, chương trình sẽ bị lỗi. Phương thức khởi tạo lớp cha chạy trước phương thức khởi tạo lớp con, vì vậy phương thức ghi đè trong lớp con sẽ được gọi trước khi phương thức khởi tạo lớp con chạy. Nếu phương thức ghi đè phụ thuộc vào bất kỳ quá trình khởi tạo nào được thực hiện bởi phương thức khởi tạo lớp con, phương thức sẽ không hoạt động như mong đợi. Để làm cho điều này cụ thể, đây là một lớp vi phạm quy tắc này:

```
lớp công cộng Super {
     // Broken - constructor gọi một phương thức có thể ghi đè public Super
     () {overrideMe ();


     } public void overrideMe () {}

}
```

Machine Translated by Google

Đây là một lớp con ghi đè phương thức overrideMe, một cách sai lầm được gọi bởi hàm tạo duy nhất của Super :

```
public final class Sub mở rộng Super {private
     final Date date; // Cuối cùng trống, được đặt bởi hàm tạo

     Sub ()
          {date = new Date ();
     }

     // Phương thức ghi đè được gọi bởi hàm tạo lớp siêu cấp
     @Override public void overrideMe () {
          System.out.println (ngày);
     }

     public static void main (String [] args) {Sub sub
          = new Sub (); sub.overrideMe ();

     }
}
```

Bạn có thể mong đợi chương trình này in ra ngày hai lần, nhưng nó sẽ in ra null lần đầu tiên, bởi vì phương thức overrideMe được gọi bởi Super construc tor trước khi Hàm dựng phụ có cơ hội khởi tạo trường ngày . Lưu ý rằng chương trình này quan sát một trường cuối cùng ở hai trạng thái khác nhau! Cũng lưu ý rằng nếu over rideMe đã gọi bất kỳ phương thức nào vào ngày, thì lời gọi đó sẽ ném ra một NullPointerException khi Super constructor gọi overrideMe. Lý do duy nhất mà chương trình này không ném ra một NullPointerException như nó viết tắt là phương thức println có các điều khoản đặc biệt để xử lý đối số null.

Các giao diện có thể sao chép và nối tiếp gây ra những khó khăn đặc biệt khi thiết kế để kế thừa. Nói chung không phải là ý kiến hay cho một lớp được thiết kế để thừa kế thực thi một trong hai giao diện này, vì chúng tạo ra gánh nặng đáng kể cho các lập trình viên mở rộng lớp. Tuy nhiên, có những hành động đặc biệt mà bạn có thể thực hiện để cho phép các lớp con triển khai các giao diện này mà không cần hẹn hò với người đàn ông mà chúng làm như vậy. Những hành động này được mô tả trong Mục 11 và Mục 74.

Nếu bạn quyết định triển khai Cloneable hoặc Serializable trong một lớp được thiết kế để kế thừa, bạn nên lưu ý rằng vì các phương thức clone và readObject hoạt động rất giống với các hàm tạo, một hạn chế tương tự cũng áp dụng: không clone và readObject có thể gọi một phương thức ghi đè, trực tiếp hoặc gián tiếp. Trong trường hợp của phương thức readObject , phương thức ghi đè sẽ chạy trước khi trạng thái của lớp con được giải mã hóa. Trong trường hợp của phương thức nhân bản , phương thức ghi đè sẽ chạy trước khi phương thức nhân bản của lớp con có cơ hội

Machine Translated by Google

sửa chữa trạng thái của bản sao. Trong cả hai trường hợp, có khả năng xảy ra lỗi chương trình. Trong trường hợp nhân bản, sự thất bại có thể làm hỏng đối tượng gốc cũng như bản sao. Điều này có thể xảy ra, chẳng hạn, nếu phương thức ghi đè giả định rằng nó đang sửa đổi bản sao của cấu trúc sâu của đối tượng, nhưng bản sao vẫn chưa được tạo ra.

Cuối cùng, nếu bạn quyết định triển khai Serializable trong một lớp được thiết kế để kế thừa và lớp có phương thức readResolve hoặc writeReplace , bạn phải đặt phương thức readResolve hoặc writeReplace được bảo vệ thay vì riêng tư . Nếu các phương thức này là private, chúng sẽ bị bỏ qua bởi các lớp con. Đây là một trường hợp nữa trong đó chi tiết triển khai trở thành một phần của API của lớp để cho phép kế thừa.

Bây giờ, rõ ràng là việc thiết kế một lớp để kế thừa đặt ra những hạn chế đáng kể cho lớp đó. Đây không phải là một quyết định được thực hiện một cách nhẹ nhàng. Có một số tình huống mà rõ ràng đó là điều đúng đắn cần làm, chẳng hạn như các lớp trừu tượng, bao gồm cả việc triển khai bộ xương của các giao diện (Mục 18). Có những tình huống khác mà rõ ràng đó là điều sai trái phải làm, chẳng hạn như các lớp bất biến (Mục 15).

Nhưng đối với các lớp bê tông thông thường thì sao? Theo truyền thống, chúng không phải là cuối cùng cũng như được thiết kế và lập thành văn bản để phân loại con, nhưng tình trạng này rất nguy hiểm. Mỗi khi một thay đổi được thực hiện trong một lớp như vậy, sẽ có khả năng các lớp khách hàng mở rộng lớp đó sẽ bị phá vỡ. Đây không chỉ là một vấn đề lý thuyết. Không có gì lạ khi nhận được các báo cáo lỗi liên quan đến lớp con sau khi sửa đổi phần bên trong của một lớp cụ thể không phải là nguyên bản không được thiết kế và ghi lại để kế thừa.

Giải pháp tốt nhất cho vấn đề này là cấm phân lớp trong các lớp không được thiết kế và ghi chép để được phân lớp một cách an toàn. Có hai cách để cấm phân lớp con. Việc dễ dàng hơn trong cả hai là tuyên bố cuối cùng của lớp. Giải pháp thay thế là đặt tất cả các nhà xây dựng ở chế độ riêng tư hoặc riêng tư trọn gói và thêm các nhà máy tĩnh công cộng thay cho các nhà xây dựng. Phương pháp thay thế này, cung cấp khả năng linh hoạt khi sử dụng các lớp con trong nội bộ, được thảo luận trong Mục 15. Cả hai cách tiếp cận đều được chấp nhận.

Lời khuyên này có thể gây tranh cãi đôi chút, vì nhiều lập trình viên đã quen với việc phân lớp các lớp bê tông thông thường để thêm các tiện ích như thiết bị đo đạc, thông báo và đồng bộ hóa hoặc để giới hạn chức năng. Nếu một lớp triển khai một số giao diện nắm bắt được bản chất của nó, chẳng hạn như Tập hợp, Danh sách hoặc Bản đồ, thì bạn sẽ không cảm thấy phiền phức về việc cấm phân lớp. Mẫu lớp trình bao bọc , được mô tả trong Mục 16, cung cấp một giải pháp thay thế vượt trội cho tính kế thừa để tăng cường chức năng.

Machine Translated by Google

Nếu một lớp cụ thể không triển khai giao diện chuẩn, thì bạn có thể gây bất tiện cho một số lập trình viên bằng cách cấm kế thừa. Nếu bạn cảm thấy rằng bạn phải cho phép kế thừa từ một lớp như vậy, một cách tiếp cận hợp lý là đảm bảo rằng lớp không bao giờ gọi bất kỳ phư ơ ng thức nào có thể ghi đè của nó và để ghi lại điều này thực tế. Nói cách khác, loại bỏ hoàn toàn việc lớp tự sử dụng các phư ơ ng thức có thể ghi đè. Khi làm như vậy, bạn sẽ tạo ra một lớp an toàn hợp lý cho lớp con. Ghi đè một phư ơ ng thức sẽ không bao giờ ảnh hư ởng đến hành vi của bất kỳ phư ơ ng thức nào khác.

Bạn có thể loại bỏ việc tự sử dụng các phư ơ ng thức có thể ghi đè của một lớp một cách máy móc, mà không thay đổi hành vi của nó. Di chuyển phần thân của mỗi phư ơ ng thức có thể ghi đè sang một "phư ơ ng thức trợ giúp" riêng và để mỗi phư ơ ng thức có thể ghi đè gọi trình trợ giúp riêng của nó phư ơ ng pháp. Sau đó, thay thế mỗi lần tự sử dụng một phư ơ ng thức có thể ghi đè bằng một mã hóa đơ n trực tiếp của phư ơ ng thức trợ giúp riêng của phư ơ ng thức có thể ghi đè.

Machine Translated by Google

Mục 18: Ư u tiên giao diện cho các lớp trừu tư ợng

Ngôn ngữ lập trình Java cung cấp hai cơ chế để xác định một kiểu cho phép nhiều triển khai: giao diện và lớp trừu tư ợng. Sự khác biệt rõ ràng nhất giữa hai cơ chế là các lớp trừu tư ợng đư ợc sử dụng để chứa các triển khai cho một số phư ơ ng thức trong khi các giao diện thì không. Một sự khác biệt quan trọng hơ n là để triển khai kiểu đư ợc định nghĩa bởi một lớp trừu tư ợng, một lớp phải là một lớp con của lớp trừu tư ợng. Bất kỳ lớp nào xác định tất cả các phư ơ ng thức đư ợc yêu cầu và tuân theo hợp đồng chung đều đư ợc phép triển khai một giao diện, bất kể lớp đó cư trú ở đâu trong hệ thống phân cấp lớp. Bởi vì Java chỉ cho phép kế thừa một lần, hạn chế này đối với các lớp trừu tư ợng đã hạn chế nghiêm trọng việc sử dụng chúng làm định nghĩa kiểu.

Các lớp hiện có có thể dễ dàng đư ợc trang bị thêm để triển khai một giao diện mới. Tất cả những gì bạn phải làm là thêm các phư ơ ng thức bắt buộc nếu chúng chư a tồn tại và thêm một mệnh đề triển khai vào khai báo lớp. Ví dụ: nhiều lớp hiện có đã đư ợc trang bị thêm để triển khai giao diện Có thể so sánh khi nó đư ợc đư a vào nền tảng. Nói chung, các lớp hiện có không thể đư ợc trang bị thêm để mở rộng một lớp trừu tư ợng mới. Nếu bạn muốn có hai lớp mở rộng cùng một lớp trừu tư ợng, bạn phải đặt lớp trừu tư ợng lên cao trong hệ thống phân cấp kiểu nơ i nó phân lớp tổ tiên của cả hai lớp. Thật không may, điều này gây ra thiệt hại lớn đối với hệ thống phân cấp kiểu, buộc tất cả con cháu của tổ tiên chung phải mở rộng lớp trừu tư ợng mới cho dù có phù hợp với họ hay không.

Các giao diện là lý tư ởng để xác định các mixin. Nói một cách dễ hiểu, mixin là một kiểu mà một lớp có thể triển khai ngoài “kiểu chính” của nó để tuyên bố rằng nó cung cấp một số hành vi tùy chọn. Ví dụ, có thể so sánh là một giao diện mixin cho phép một lớp tuyên bố rằng các thể hiện của nó đư ợc sắp xếp theo thứ tự đối với các đối tư ợng có thể so sánh của đồng minh mutu khác. Giao diện như vậy đư ợc gọi là mixin vì nó cho phép chức năng tùy chọn đư ợc “trộn lẫn” vào chức năng chính của loại.
Các lớp trừu tư ợng không thể đư ợc sử dụng để định nghĩa các mixin vì cùng một lý do là chúng không thể đư ợc trang bị thêm vào các lớp hiện có: một lớp không thể có nhiều hơ n một lớp cha và không có vị trí hợp lý trong hệ thống phân cấp lớp để chèn một mixin.

Các giao diện cho phép xây dựng các khuôn khổ kiểu phi phân cấp.
Cấu trúc phân cấp kiểu rất tốt để tổ chức một số thứ, như ng những thứ khác không nằm gọn trong một hệ thống phân cấp cứng nhắc. Ví dụ: giả sử chúng ta có một giao diện đại diện cho ca sĩ và một giao diện khác đại diện cho một nhạc sĩ:

```
giao diện công cộng Ca sĩ {
    AudioClip hát (Bài hát);
}
```

Machine Translated by Google


```
giao diện công cộng Người viết nhạc {
     Sáng tác bài hát (boolean hit);
}
```

Ngoài đời, một số ca sĩ cũng là nhạc sĩ. Bởi vì chúng tôi đã sử dụng các giao diện chứ không phải các lớp trừu tượng để định nghĩa các kiểu này, nên hoàn toàn có thể cho phép một lớp duy nhất triển khai cả Ca sĩ và Nhạc sĩ. Trên thực tế, chúng ta có thể xác định giao diện thứ ba mở rộng cả Ca sĩ và Nhạc sĩ và thêm các phương pháp mới phù hợp với sự kết hợp:

```
giao diện công cộng SingerSongwriter mở rộng Ca sĩ, Nhạc sĩ {
     AudioClip strum ();
     void actSensitive ();
}
```

Không phải lúc nào bạn cũng cần mức độ linh hoạt này, nhưng khi bạn làm vậy, các giao diện là một cứu cánh. Giải pháp thay thế là một hệ thống phân cấp lớp mở rộng có chứa một lớp riêng biệt cho mọi tổ hợp thuộc tính được hỗ trợ. Nếu có n thuộc tính trong hệ thống loại, thì có thể có 2n sự kết hợp mà bạn có thể phải hỗ trợ. Đây là những gì được gọi là một vụ nổ tổ hợp. Cấu trúc phân cấp lớp bị phình ra có thể dẫn đến các lớp bị phồng chứa nhiều phương thức chỉ khác nhau về kiểu của các phương thức lập luận của chúng, vì không có kiểu nào trong hệ thống phân cấp lớp để nắm bắt các hành vi phổ biến.

Các giao diện cho phép các cải tiến chức năng an toàn và mạnh mẽ thông qua thành ngữ quấn mỗi lớp , được mô tả trong Mục 16. Nếu bạn sử dụng các lớp trừu tượng để xác định kiểu, bạn để lập trình viên muốn thêm chức năng mà không có sự thay thế nào khác ngoài sử dụng tính kế thừa. Các lớp kết quả ít mạnh hơn và dễ vỡ hơn các lớp trình bao bọc.

Mặc dù các giao diện không được phép chứa các triển khai phương pháp, nhưng việc sử dụng các giao diện để xác định kiểu không ngăn cản bạn cung cấp hỗ trợ triển khai cho các lập trình viên. Bạn có thể kết hợp các ưu điểm của giao diện và các lớp trừu tượng bằng cách cung cấp một lớp triển khai bộ xương trừu tượng để đi với mỗi giao diện tầm thường mà bạn xuất. Giao diện vẫn xác định kiểu, nhưng việc triển khai bộ xương sẽ mất tất cả công việc khi thực hiện nó.

Theo quy ước, các triển khai bộ xương được gọi là AbstractInterface, trong đó Giao diện là tên của giao diện mà chúng thực hiện. Ví dụ, Collec tions Framework cung cấp một triển khai khung để đi cùng với mỗi giao diện bộ sưu tập chính: AbstractCollection, AbstractSet, AbstractList và AbstractMap. Có thể cho rằng sẽ có lý khi gọi chúng là SkeletalCollec tion, SkeletalSet, SkeletalList và SkeletalMap, nhưng khái niệm trừu tượng conven hiện đã được thiết lập vững chắc.

Machine Translated by Google

Khi được thiết kế phù hợp, các triển khai khung có thể giúp các lập trình viên rất dễ dàng cung cấp các triển khai của riêng họ cho các giao diện của bạn. Đối với bài kiểm tra, đây là phương thức static factory chứa triển khai Danh sách đầy đủ, đầy đủ chức năng :

```
// Triển khai cụ thể được xây dựng trên đỉnh bộ xương thực thi static List
<Integer> intArrayAsList (final int [] a) {if (a == null) ném mới
     NullPointerException ();


     return new AbstractList <Integer> () {public
          Integer get (int i) {return a [i]; //
               Hộp tự động (Mục 5)
          }

          @Override public Integer set (int i, Integer val) {int oldVal =
               a [i]; a [i] = val;
                                    // Tự động mở hộp
               trả về oldVal; // Hộp thư  tự động
          }

          public int size ()
               {return a.length;
          }
     };
}
```

Khi bạn xem xét tất cả những gì mà một triển khai Danh sách mang lại cho bạn, ví dụ này là một minh chứng ấn tượng về sức mạnh của việc triển khai bộ xương. Inciden kiểm đếm, ví dụ là một Bộ điều hợp [Gamma95, p. 139] cho phép một mảng int được xem như một danh sách các thể hiện Số nguyên . Bởi vì tất cả các bản dịch qua lại giữa các giá trị int và các cá thể Integer (quyền anh và unboxing), tính hiệu quả của nó không tốt lắm. Lưu ý rằng một nhà máy tĩnh được cung cấp và lớp đó là một lớp ẩn danh không thể truy cập được (Mục 22) ẩn bên trong nhà máy tĩnh.

Vẻ đẹp của các triển khai khung là chúng cung cấp sự hỗ trợ triển khai của các lớp trừu tượng mà không áp đặt các ràng buộc nghiêm trọng mà các lớp trừu tượng áp đặt khi chúng dùng làm định nghĩa kiểu. Đối với hầu hết các phần thân của giao diện, việc mở rộng triển khai khung là lựa chọn hiển nhiên, nhưng nó hoàn toàn là tùy chọn. Nếu không thể tạo một lớp có sẵn để mở rộng triển khai skele tal, thì lớp đó luôn có thể triển khai giao diện theo cách thủ công. Furrmore, việc triển khai bộ xương vẫn có thể hỗ trợ nhiệm vụ của người thực hiện. Lớp thực thi giao diện có thể chuyển tiếp các lệnh gọi của các phương thức giao diện tới một thể hiện chứa của một lớp bên trong riêng tư mở rộng triển khai khung

Machine Translated by Google

sự. Kỹ thuật này, đư ợc gọi là đa kế thừa mô phỏng, có liên quan chặt chẽ đến thành ngữ lớp wrapper đư ợc thảo luận trong Mục 16. Nó cung cấp hầu hết các lợi ích của đa kế thừa, đồng thời tránh đư ợc các cạm bẫy.

Viết một bản triển khai bộ xư ơ ng là một quá trình tư ơ ng đối đơ n giản, nếu hơ i tẻ nhạt. Đầu tiên, bạn phải nghiên cứu giao diện và quyết định phư ơ ng thức nào là nguyên thủy về mặt mà các phư ơ ng thức khác có thể đư ợc thực hiện. Những nguyên thủy này sẽ là những phư ơ ng thức trừu tư ợng trong quá trình thực thi bộ xư ơ ng của bạn. Sau đó, bạn phải cung cấp các triển khai cụ thể của tất cả các phư ơ ng pháp khác trong giao diện. Ví dụ: đây là một triển khai cơ bản của giao diện Map.Entry :

```
// Thực hiện Skeletal lớp trừu
tư ợng công khai AbstractMapEntry <K, V> thực hiện
          Map.Entry <K, V> {// Các phép toán nguyên thủy
     public abstract K getKey (); công khai trừu tư ợng V
     getValue ();


     // Các mục trong bản đồ có thể sửa đổi phải ghi đè phư ơ ng thức này
     public V setValue (V value) {throw new UnsupportedOperationException ();

     }

     // Thực hiện hợp đồng chung của Map.Entry.equals @Override public boolean
     equals (Object o) {if (o == this) return true; if (! (o instanceof
          Map.Entry)) trả về false; Map.Entry <?,?> Arg = (Map.Entry) o; trả
               về bằng (getKey (), arg.getKey ()) && bằng (getValue (),
          arg.getValue ());





     } private static boolean bằng (Đối tư ợng o1, Đối tư ợng o2) {return o1
          == null? o2 == null: o1.equals (o2);
     }

     // Thực hiện hợp đồng chung của Map.Entry.hashCode @Override public int
     hashCode () {return hashCode (getKey ()) ^ hashCode (getValue ());


     } private static int hashCode (Object obj) {
          trả về obj == null? 0: obj.hashCode ();
     }
}
```

Machine Translated by Google

Bởi vì các triển khai khung được thiết kế để kế thừa, bạn nên tuân theo tất cả các hướng dẫn về thiết kế và tài liệu trong Mục 17. Để ngắn gọn, các nhận xét tài liệu đã được bỏ qua trong ví dụ trước, nhưng tài liệu tốt là hoàn toàn cần thiết cho việc triển khai khung.

Một biến thể nhỏ trong việc triển khai bộ xương là cách triển khai đơn giản, được minh họa bởi AbstractMap.SimpleEntry. Một triển khai đơn giản giống như một triển khai bộ xương trong đó nó thực hiện một giao diện và được thiết kế để kế thừa, nhưng nó khác ở chỗ nó không trừu tượng: nó là triển khai làm việc đơn giản nhất có thể. Bạn có thể sử dụng nó dưới dạng đứng hoặc phân loại nó tùy trường hợp.

Sử dụng các lớp trừu tượng để xác định các kiểu cho phép nhiều triển khai có một lợi thế lớn so với việc sử dụng các giao diện: Việc phát triển một lớp trừu tượng dễ dàng hơn nhiều so với một giao diện. Nếu, trong bản phát hành tiếp theo, bạn muốn thêm một phương thức mới vào một lớp trừu tượng, bạn luôn có thể thêm một phương thức cụ thể có chứa một triển khai mặc định hợp lý. Sau đó, tất cả các triển khai hiện có của lớp trừu tượng sẽ cung cấp phương thức mới. Điều này không hoạt động đối với các giao diện.

Nói chung, không thể thêm một phương thức vào giao diện công cộng mà không phá vỡ tất cả các lớp hiện có triển khai giao diện. Các lớp đã triển khai giao diện trước đó sẽ thiếu phương thức mới và sẽ không biên dịch nữa. Bạn có thể hạn chế phần nào thiệt hại bằng cách thêm phương pháp mới vào việc triển khai bộ xương cùng lúc khi bạn thêm nó vào mặt liên, nhưng điều này thực sự sẽ không giải quyết được vấn đề. Bất kỳ triển khai nào không kế thừa từ triển khai bộ xương sẽ vẫn bị hỏng.

Do đó, các giao diện công cộng phải được thiết kế cẩn thận. Một khi giao diện được phát hành và triển khai rộng rãi thì hầu như không thể thay đổi được. Bạn thực sự phải làm đúng ngay lần đầu tiên. Nếu một giao diện có một lỗ hổng nhỏ, nó sẽ khiến bạn và người dùng của nó khó chịu mãi mãi. Nếu một giao diện bị thiếu nghiêm trọng, nó có thể hủy hoại một API. Điều tốt nhất cần làm khi phát hành giao diện mới là có càng nhiều chương trình mers triển khai giao diện theo nhiều cách càng tốt trước khi giao diện bị đóng băng. Điều này sẽ cho phép bạn phát hiện ra những sai sót trong khi bạn vẫn có thể chỉnh sửa chúng.

Tóm lại, giao diện thường là cách tốt nhất để xác định một kiểu cho phép nhiều triển khai. Một ngoại lệ đối với quy tắc này là trường hợp sự dễ dàng tiến hóa được coi là quan trọng hơn tính linh hoạt và sức mạnh. Trong những trường hợp này, bạn nên sử dụng một lớp trừu tượng để định nghĩa kiểu, nhưng chỉ khi bạn hiểu và có thể chấp nhận các hạn chế. Nếu bạn xuất một giao diện tầm thường, bạn nên xem xét cung cấp một triển khai khung để đi kèm với nó.

Cuối cùng, bạn nên thiết kế tất cả các giao diện công cộng của mình một cách cẩn thận nhất và kiểm tra chúng kỹ lưỡng bằng cách viết nhiều triển khai.

Machine Translated by Google

Mục 19: Chỉ sử dụng giao diện để xác định loại

Khi một lớp triển khai một giao diện, giao diện đó đóng vai trò là một kiểu có thể được sử dụng để tham chiếu đến các thể hiện của lớp. Do đó, một lớp triển khai một giao diện nên nói lên điều gì đó về những gì một máy khách có thể làm với các thể hiện của lớp. Không thích hợp để xác định một giao diện cho bất kỳ mục đích nào khác.

Một loại giao diện không thành công trong thử nghiệm này là cái gọi là giao diện không đổi. Một giao diện như vậy không chứa phương thức nào; nó chỉ bao gồm các trường cuối cùng tĩnh, mỗi trường xuất một hằng số. Các lớp sử dụng các hằng số này triển khai giao diện để tránh nhu cầu đặt tên hằng số đủ điều kiện với một tên lớp. Đây là một ví dụ:

```
// Constant interface antipattern - không sử dụng! giao diện
chung PhysicalConstants {// Số Avogadro (1 / mol) tĩnh kép cuối
     cùng AVOGADROS_NUMBER = 6.02214199e23;


     // Hằng số Boltzmann (J / K) tĩnh
     kép cuối cùng BOLTZMANN_CONSTANT = 1.3806503e-23;

     // Khối lượng của electron (kg) tĩnh
     điện tử kép cuối cùng ELECTRON_MASS                = 9.10938188e-31;
}
```

Mẫu giao diện không đổi là cách sử dụng giao diện kém. Việc một lớp sử dụng một số hằng số bên trong là một chi tiết triển khai. Việc triển khai một giao diện không đổi khiến chi tiết triển khai này bị rò rỉ vào API đã xuất của lớp. Không có hậu quả gì đối với người dùng của một lớp mà lớp đó thực hiện một giao diện không đổi. Trên thực tế, nó thậm chí có thể khiến họ nhầm lẫn. Tệ hơn nữa, nó đại diện cho một cam kết: nếu trong một bản phát hành trong tương lai, lớp được sửa đổi để nó không còn cần sử dụng các con stants nữa, nó vẫn phải triển khai giao diện để đảm bảo khả năng tương thích nhị phân. Nếu một lớp nonfinal triển khai một giao diện không đổi, tất cả các lớp con của nó sẽ có không gian tên của chúng bị ô nhiễm bởi các hằng số trong giao diện.

Có một số giao diện không đổi trong các thư viện nền tảng Java, chẳng hạn như java.io.ObjectStreamConstants. Những giao diện này nên được coi là dị thường và không nên được mô phỏng.

Nếu bạn muốn xuất các hằng số, có một số lựa chọn hợp lý. Nếu các hằng được liên kết chặt chẽ với một lớp hoặc giao diện hiện có, bạn nên thêm chúng vào lớp hoặc giao diện. Ví dụ: tất cả các lớp nguyên thủy số được đóng hộp, chẳng hạn như Số nguyên và Đôi, xuất các hằng số MIN_VALUE và MAX_VALUE . Nếu các hằng được xem tốt nhất là thành viên của kiểu liệt kê, bạn nên xuất

Machine Translated by Google

chúng với một kiểu enum (Mục 30). Nếu không, bạn nên xuất các hằng với một lớp tiện ích không ổn định (Mục 4). Đây là phiên bản lớp tiện ích của ví dụ PhysicalConstants ở trên:

```
// Gói lớp tiện ích không đổi
com.effectivejava.science;

public class PhysicalConstants {private
   PhysicalConstants () {} // Ngăn chặn việc khởi tạo

   công khai tĩnh cuối cùng kép AVOGADROS_NUMBER = 6.02214199e23; công khai tĩnh cuối
   cùng kép BOLTZMANN_CONSTANT = 1.3806503e-23; công cộng tĩnh cuối cùng ELECTRON_MASS
   = 9.10938188e-31;
}
```

Thông thư ờng một lớp tiện ích yêu cầu máy khách đủ điều kiện cho các tên không đổi với tên lớp, ví dụ: PhysicalConstants.AVOGADROS_NUMBER. Nếu bạn sử dụng nhiều các hằng số đư ợc xuất bởi một lớp tiện ích, bạn có thể tránh nhu cầu nhập các hằng số có tên lớp đủ điều kiện bằng cách sử dụng phư ơ ng tiện nhập tĩnh , đư ợc giới thiệu trong bản phát hành 1.5:

```
// Sử dụng nhập tĩnh để tránh các hằng số đủ điều kiện nhập tĩnh
com.effectivejava.science.PhysicalConstants. *;

public class Kiểm tra
      {nguyên tử kép (double mols) {return
            AVOGADROS_NUMBER * mols;
      }
      ...
      // Nhiều công dụng khác của PhysicalConstants biện minh cho việc nhập tĩnh
}
```

Tóm lại, các giao diện chỉ nên đư ợc sử dụng để xác định các kiểu. Họ không nên đư ợc sử dụng để xuất các hằng số.

Machine Translated by Google

Mục 20: Ưu tiên phân cấp lớp cho các lớp được gắn thẻ

Đôi khi, bạn có thể gặp phải một lớp mà các phiên bản của nó có từ hai phiên bản trở lên và chứa một trường thẻ biểu thị hướng vị của phiên bản đó. Ví dụ: hãy xem xét lớp này, lớp này có khả năng đại diện cho hình tròn hoặc hình chữ nhật:

```
// Lớp được gắn thẻ - kém hơn rất nhiều so với hệ thống phân cấp lớp!
class Hình {enum Shape {RECTANGLE, CIRCLE};


     // Trường thẻ - hình dạng của hình này Hình dạng
     cuối cùng;

     // Các trường này chỉ được sử dụng nếu hình dạng có chiều dài gấp
     đôi RECTANGLE; chiều rộng gấp đôi;


     // Trường này chỉ được sử dụng nếu hình dạng là bán kính
     kép CIRCLE;

     // Hàm tạo cho vòng tròn
     Hình (bán kính gấp đôi) {
          shape = Shape.CIRCLE;
          this.radius = bán kính;
     }

     // Hàm tạo cho hình chữ nhật Hình
     (chiều dài gấp đôi, chiều rộng gấp đôi) {shape
          = Shape.RECTANGLE; this.length = chiều
          dài; this.width = width;

     }

     double area ()
          {switch (shape) {
            case RECTANGLE:
              trả về chiều dài * chiều
            rộng; case CIRCLE: return
              Math.PI * (radius * bán kính); default: ném
            AssertionError () mới;

          }
     }
}
```

Machine Translated by Google

Các lớp được gắn thẻ như vậy có nhiều thiếu sót. Chúng lộn xộn với bảng soạn sẵn, bao gồm khai báo enum, trường thẻ và câu lệnh switch. Khả năng đọc bị tổn hại nhiều hơn vì nhiều triển khai bị lộn xộn với nhau trong một lớp duy nhất. Dấu chân bộ nhớ được tăng lên bởi vì các cá thể bị gánh nặng với các trường không liên quan thuộc các hương vị khác. Các trường không thể được tạo cuối cùng trừ khi các hàm khởi tạo khởi tạo các trường không liên quan, dẫn đến nhiều bản soạn sẵn hơn. Các trình xây dựng phải đặt trường thẻ và khởi tạo các trường dữ liệu phù hợp mà không cần sự trợ giúp từ com piler: nếu bạn khởi tạo các trường sai, chương trình sẽ không thành công trong thời gian chạy. Bạn không thể thêm hương vị vào một lớp được gắn thẻ trừ khi bạn có thể sửa đổi tệp nguồn của nó. Nếu bạn thêm một hương vị, bạn phải nhớ thêm một trường hợp vào mỗi câu lệnh switch, nếu không lớp sẽ bị lỗi trong thời gian chạy. Cuối cùng, kiểu dữ liệu của một cá thể không đưa ra manh mối nào về fla vor của nó. Nói tóm lại, các lớp được gắn thẻ dài dòng, dễ xảy ra lỗi và kém hiệu quả.

May mắn thay, các ngôn ngữ hướng đối tượng như Java cung cấp một giải pháp thay thế tốt hơn nhiều để xác định một kiểu dữ liệu duy nhất có khả năng đại diện cho các đối tượng có nhiều hương vị: kiểu con. Một lớp được gắn thẻ chỉ là một sự bắt chước tồi tệ của một hệ thống phân cấp lớp.

Để chuyển đổi một lớp được gắn thẻ thành một hệ thống phân cấp lớp, trước tiên hãy xác định một lớp trừu tượng chứa một phương thức trừu tượng cho mỗi phương thức trong lớp được gắn thẻ có hành vi phụ thuộc vào giá trị thẻ. Trong lớp Hình , chỉ có một phương thức như vậy, đó là diện tích. Lớp trừu tượng này là gốc của hệ thống phân cấp lớp. Nếu có bất kỳ phương thức nào có hành vi không phụ thuộc vào giá trị của thẻ, hãy đặt chúng vào lớp này. Tương tự, nếu có bất kỳ trường dữ liệu nào được sử dụng bởi tất cả các phiên bản, hãy đặt chúng vào lớp này. Không có phương thức hoặc trường độc lập hương vị nào như vậy trong lớp Hình .

Tiếp theo, xác định một lớp con cụ thể của lớp gốc cho từng hương vị của lớp được gắn thẻ ban đầu. Trong ví dụ của chúng tôi, có hai: hình tròn và hình chữ nhật. Bao gồm trong mỗi lớp con các trường dữ liệu đặc biệt cho hương vị của nó. Trong ví dụ của chúng tôi, bán kính dành riêng cho hình tròn và chiều dài và chiều rộng dành riêng cho hình chữ nhật. Cũng bao gồm trong mỗi lớp con sự triển khai thích hợp của mỗi phương thức trừu tượng trong lớp gốc.

Đây là phân cấp lớp tương ứng với lớp Hình ban đầu :

```
// Thay thế phân cấp lớp cho một lớp trừu tượng lớp được gắn
thẻ Hình {
     vùng kép trừu tượng ();
}

class Circle mở rộng Hình {bán kính
     kép cuối cùng;

     Circle (bán kính kép) {this.radius = radius; }

     double area () {return Math.PI * (radius * bán kính); }
}
```

Machine Translated by Google


```
lớp Hình chữ nhật kéo dài Hình {độ dài
     gấp đôi cuối cùng; chiều rộng kép
     cuối cùng;

     Rectangle (chiều dài gấp đôi, chiều rộng gấp đôi)
          {this.length = length; this.width = width;


     } double area () {return length * width; }
}
```

Hệ thống phân cấp lớp này sửa chữa mọi thiếu sót của các lớp đư ợc gắn thẻ đư ợc ghi chú trư ớc đó. Mã này đơ n giản và rõ ràng, không chứa bảng soạn sẵn nào đư ợc tìm thấy trong bản gốc. Việc triển khai mỗi hư ơ ng vị đư ợc phân bổ cho lớp riêng của nó và không lớp nào trong số này bị cản trở bởi các trư ờng dữ liệu không liên quan. Tất cả các lĩnh vực là cuối cùng. Trình biên dịch đảm bảo rằng phư ơ ng thức khởi tạo của mỗi lớp khởi tạo các trư ờng dữ liệu của nó và mỗi lớp có một triển khai cho mọi phư ơ ng thức trừu tư ợng đư ợc khai báo trong lớp gốc. Điều này giúp loại bỏ khả năng lỗi thời gian chạy do trư ờng hợp công tắc bị thiếu. Nhiều lập trình viên có thể mở rộng phân cấp một cách độc lập và xóa tư ơ ng tác mà không cần truy cập vào nguồn cho lớp gốc. Có một kiểu dữ liệu riêng biệt đư ợc liên kết với mỗi hư ơ ng vị, cho phép các lập trình viên chỉ ra hư ơ ng vị của một biến thể có thể và hạn chế các biến và tham số đầu vào đối với một hư ơ ng vị cụ thể.

Một ư u điểm khác của phân cấp lớp là chúng có thể đư ợc tạo ra để phản ánh các mối quan hệ phân cấp tự nhiên giữa các loại, cho phép tăng tính linh hoạt và kiểm tra loại thời gian biên dịch tốt hơ n. Giả sử lớp đư ợc gắn thẻ trong đề thi ban đầu cũng đư ợc phép cho các ô vuông. Phân cấp lớp có thể đư ợc tạo ra để phản ánh thực tế rằng hình vuông là một loại hình chữ nhật đặc biệt (giả sử cả hai đều bất biến):

```
class Hình vuông kéo dài Hình chữ nhật {
     Square (hai mặt) {super
          (side, side);
     }
}
```

Lư u ý rằng các trư ờng trong cấu trúc phân cấp trên đư ợc truy cập trực tiếp chứ không phải bằng các phư ơ ng thức của trình truy cập. Điều này đư ợc thực hiện cho ngắn gọn và sẽ không thể chấp nhận đư ợc nếu hệ thống phân cấp đư ợc công khai (Khoản 14).

Tóm lại, các lớp đư ợc gắn thẻ hiếm khi thích hợp. Nếu bạn muốn viết một lớp có trư ờng thẻ rõ ràng, hãy nghĩ xem liệu thẻ có thể bị loại bỏ và lớp đư ợc thay thế bằng hệ thống phân cấp hay không. Khi bạn gặp một lớp hiện có với trư ờng thẻ, hãy xem xét việc cấu trúc lại nó thành một hệ thống phân cấp.

Machine Translated by Google

Mục 21: Sử dụng các đối tượng chức năng để biểu diễn các chiến lược

Một số ngôn ngữ hỗ trợ con trỏ hàm, đại biểu, biểu thức lambda hoặc cơ sở mô phỏng cho phép các chương trình lưu trữ và truyền khả năng gọi một hàm particular. Các phương tiện như vậy thường được sử dụng để cho phép người gọi một hàm chuyên biệt hóa hành vi của nó bằng cách truyền vào một hàm thứ hai. Ví dụ, hàm qsort trong thư viện chuẩn của C nhận một con trỏ đến một hàm so sánh , hàm này sử dụng qsort để so sánh các phần tử sẽ được sắp xếp. Hàm so sánh nhận hai tham số, mỗi tham số là một con trỏ đến một phần tử. Nó trả về một số nguyên âm nếu phần tử được chỉ ra bởi tham số đầu tiên nhỏ hơn phần tử được chỉ ra bởi tham số thứ hai, bằng 0 nếu hai phần tử bằng nhau và một số nguyên dương nếu phần tử được chỉ ra bởi tham số đầu tiên lớn hơn phần được chỉ ra bởi thứ hai.
Các thứ tự sắp xếp khác nhau có thể nhận được bằng cách chuyển các hàm so sánh khác nhau. Đây là một ví dụ về mô hình Chiến lược [Gamma95, tr. 315]; hàm so sánh đại diện cho một chiến lược để sắp xếp các phần tử.

Java không cung cấp con trỏ hàm, nhưng các tham chiếu đối tượng có thể được sử dụng để đạt được hiệu quả tương tự. Việc gọi một phương thức trên một đối tượng thường thực hiện một số thao tác trên đối tượng đó. Tuy nhiên, có thể xác định một đối tượng mà meth ods thực hiện các hoạt động trên các đối tượng khác, được truyền rõ ràng cho các phương thức. Một thể hiện của lớp xuất chính xác một phương thức như vậy sẽ là một con trỏ đến phương thức đó. Các cá thể như vậy được gọi là các đối tượng chức năng. Ví dụ, hãy xem xét lớp sau:

```
class StringLengthComparator
    {public int so sánh (String s1, String s2)
        {return s1.length () - s2.length ();
    }
}
```

Lớp này xuất một phương thức duy nhất nhận hai chuỗi và trả về một số nguyên âm nếu chuỗi đầu tiên ngắn hơn chuỗi thứ hai, bằng 0 nếu hai chuỗi có độ dài bằng nhau và một số nguyên dương nếu chuỗi đầu tiên dài hơn. Phương pháp này là một bộ so sánh sắp xếp thứ tự các chuỗi dựa trên độ dài của chúng thay vì thứ tự từ điển điển hình hơn. Tham chiếu đến một đối tượng StringLengthComparator đóng vai trò như một “con trỏ hàm” tới bộ so sánh này, cho phép nó được gọi trên các cặp chuỗi tùy ý. Nói cách khác, một cá thể StringLengthComparator là một chiến lược khó hiểu để so sánh chuỗi.

Như là điển hình cho các lớp chiến lược cụ thể, lớp StringLengthComparator là không trạng thái: nó không có trường, do đó tất cả các phiên bản của lớp đều có chức năng

Machine Translated by Google

tư ơ ng đư ơ ng. Vì vậy, nó nên là một đơ n để tiết kiệm chi phí tạo đối tư ợng không cần thiết (Mục 3, Mục 5):

```
class StringLengthComparator {private
     StringLengthComparator () {} public static
     final StringLengthComparator
          INSTANCE = new StringLengthComparator ();
     public int so sánh (String s1, String s2) {return
          s1.length () - s2.length ();
     }
}
```

Để truyền một cá thể StringLengthComparator cho một phư ơ ng thức, chúng ta cần một kiểu thích hợp cho tham số. Sẽ không tốt nếu sử dụng StringLengthComparator vì các máy khách sẽ không thể vư ợt qua bất kỳ chiến lư ợc so sánh nào khác. Thay vào đó, chúng ta cần xác định giao diện Comparator và sửa đổi StringLengthComparator để triển khai giao diện này. Nói cách khác, chúng ta cần xác định một giao diện chiến lư ợc để đi cùng với lớp chiến lư ợc cụ thể. Nó đây:

```
// Giao diện chiến lư ợc
giao diện công khai Bộ so sánh <T> {
     public int so sánh (T t1, T t2);
}
```

Định nghĩa về giao diện Bộ so sánh này tình cờ đến từ gói java.util , như ng không có gì kỳ diệu về nó: bạn cũng có thể tự viết nó. Giao diện Bộ so sánh là chung (Mục 26) để nó có thể áp dụng cho các bộ so sánh cho các đối tư ợng không phải là chuỗi. Phư ơ ng thức so sánh của nó nhận hai tham số kiểu T (tham số kiểu chính thức của nó) chứ không phải là Chuỗi. Lớp StringLengthComparator đư ợc hiển thị ở trên có thể đư ợc thực hiện để triển khai Compar ator <String> chỉ bằng cách khai báo nó để làm như vậy:

```
class StringLengthComparator triển khai Comparator <String> {... // phần thân
     của lớp giống với phần đư ợc hiển thị ở trên
}
```

Các lớp chiến lư ợc cụ thể thư ờng đư ợc khai báo bằng cách sử dụng các lớp ẩn danh (Mục 22). Câu lệnh sau đây sắp xếp một mảng chuỗi theo độ dài:

```
Arrays.sort (stringArray, new Comparator <Chuỗi> () {public int
     so sánh (Chuỗi s1, Chuỗi s2) {return s1.length () -
          s2.length ();
     }
});
```

Machine Translated by Google

Như ng lư u ý rằng việc sử dụng một lớp ẩn danh theo cách này sẽ tạo ra một thể hiện mới mỗi khi lệnh gọi đư ợc thực thi. Nếu nó phải đư ợc thực thi lặp đi lặp lại, hãy xem xét việc lư u trữ đối tư ợng hàm trong trư ờng cuối cùng tĩnh riêng tư và sử dụng lại nó. Một ư u điểm khác của việc này là bạn có thể đặt tên mô tả cho trư ờng đối với đối tư ợng hàm.

Vì giao diện chiến lư ợc đóng vai trò là một loại cho tất cả các trư ờng hợp chiến lư ợc cụ thể của nó, nên một lớp chiến lư ợc cụ thể không cần phải đư ợc công khai để xuất một chiến lư ợc cụ thể. Thay vào đó, một "lớp máy chủ" có thể xuất một trư ờng tĩnh công khai (hoặc phư ơ ng thức nhà máy tĩnh) có kiểu là giao diện chiến lư ợc và lớp chiến lư ợc cụ thể có thể là một lớp lồng nhau riêng của máy chủ. Trong ví dụ sau, một lớp thành viên tĩnh đư ợc sử dụng ư u tiên cho một lớp ẩn danh để cho phép lớp chiến lư ợc cụ thể triển khai giao diện thứ hai, Serializable:

```
// Xuất một lớp chiến lư ợc cụ thể Máy chủ
{private static class StrLenCmp triển khai
     Comparator <String>, Serializable {

          public int so sánh (String s1, String s2) {return
               s1.length () - s2.length ();
          }
     }

     // Bộ so sánh đư ợc trả về là bộ so sánh công khai
     tĩnh cuối cùng có thể tuần tự hóa <Chuỗi>
          STRING_LENGTH_COMPARATOR = new StrLenCmp ();

     ... // Hàng loạt lớp bị bỏ qua
}
```

Lớp String sử dụng mẫu này để xuất tham số com chuỗi không phân biệt chữ hoa chữ thư ờng qua trư ờng CASE_INSENSITIVE_ORDER của nó .

Tóm lại, cách sử dụng chính của con trỏ hàm là thực hiện mẫu Chiến lư ợc. Để triển khai mẫu này trong Java, hãy khai báo một giao diện để đại diện cho chiến lư ợc và một lớp triển khai giao diện này cho từng chiến lư ợc cụ thể.

Khi một chiến lư ợc cụ thể chỉ đư ợc sử dụng một lần, nó thư ờng đư ợc khai báo và tức thời đư ợc coi là một lớp ẩn danh. Khi một chiến lư ợc cụ thể đư ợc thiết kế để sử dụng nhiều lần, nó thư ờng đư ợc triển khai dư ới dạng một lớp thành viên tĩnh riêng và đư ợc xuất trong trư ờng cuối cùng tĩnh công khai có kiểu là giao diện chiến lư ợc.

Machine Translated by Google

Mục 22: Ủng hộ các lớp thành viên tĩnh hơn là không tĩnh

Một lớp lồng nhau là một lớp được định nghĩa bên trong một lớp khác. Một lớp lồng nhau nên tồn tại chỉ để phục vụ lớp bao quanh của nó. Nếu một lớp lồng nhau sẽ hữu ích trong một số ngữ cảnh khác, thì nó phải là một lớp cấp cao nhất. Có bốn loại lớp lồng nhau: lớp thành viên tĩnh, lớp thành viên không tĩnh, lớp ẩn danh và lớp cục bộ. Tất cả trừ loại đầu tiên được gọi là các lớp bên trong. Mục này cho bạn biết khi nào sử dụng loại lớp lồng nhau nào và tại sao.

Lớp thành viên tĩnh là loại lớp lồng nhau đơn giản nhất. Tốt nhất nó nên được coi như một lớp bình thường được khai báo bên trong một lớp khác và có quyền truy cập vào tất cả các thành viên của lớp bao quanh, ngay cả những thành viên được khai báo là private. Lớp thành viên tĩnh là một thành viên tĩnh của lớp bao quanh nó và tuân theo các quy tắc trợ năng giống như các thành viên tĩnh khác. Nếu nó được khai báo là private, nó chỉ có thể truy cập được trong lớp bao quanh, v.v.

Một cách sử dụng phổ biến của lớp thành viên tĩnh là như một lớp trợ giúp công khai, chỉ hữu ích khi kết hợp với lớp bên ngoài của nó. Ví dụ, hãy xem xét một enum mô tả các phép toán được hỗ trợ bởi máy tính (Mục 30). Enum hoạt động phải là một lớp thành viên tĩnh công khai của lớp Máy tính . Khách hàng của Máy tính sau đó có thể tham chiếu đến các hoạt động sử dụng các tên như Calculator.Operation.PLUS và Calculator.Operation.MINUS.

Về mặt cú pháp, sự khác biệt duy nhất giữa các lớp thành viên tĩnh và không có tĩnh là các lớp thành viên tĩnh có tĩnh sửa đổi trong khai báo của chúng.

Mặc dù có sự giống nhau về cú pháp, hai loại lớp lồng nhau này rất khác nhau. Mỗi thể hiện của một lớp thành viên không tĩnh được liên kết ngầm với một thể hiện bao quanh của lớp chứa nó. Trong các phương thức cá thể của một lớp thành viên không tĩnh, bạn có thể gọi các phương thức trên cá thể bao quanh hoặc lấy tham chiếu đến cá thể bao quanh bằng cách sử dụng cấu trúc này đủ điều kiện [JLS, 15.8.4]. Nếu một thể hiện của lớp lồng nhau có thể tồn tại tách biệt với một thể hiện của lớp bao quanh nó, thì lớp lồng nhau phải là lớp thành viên tĩnh: không thể tạo một thể hiện của lớp thành viên không tĩnh mà không có thể hiện bao quanh.

Sự liên kết giữa thể hiện lớp thành viên không tĩnh và thể hiện bao quanh của nó được thiết lập khi cá thể cũ được tạo; nó không thể được sửa đổi sau đó. Thông thường, liên kết được thiết lập tự động bằng cách gọi một phương thức khởi tạo lớp thành viên không tĩnh từ bên trong một phương thức thể hiện của

lớp bao quanh. Mặc dù hiếm khi có thể thiết lập liên kết theo cách thủ công bằng cách sử dụng biểu thức enclosingInstance.new MemberClass (args). Như bạn mong đợi, liên kết chiếm không gian trong cá thể lớp thành viên không tĩnh và thêm thời gian để xây dựng nó.

Machine Translated by Google

Một cách sử dụng phổ biến của lớp thành viên không tĩnh là định nghĩa Bộ điều hợp [Gamma95, p. 139] cho phép một thể hiện của lớp ngoài được xem như là một thể hiện của một số lớp không liên quan. Ví dụ, các triển khai của mặt liên bản đồ thường sử dụng các lớp thành viên không tĩnh để triển khai các chế độ xem tập hợp của chúng, được trả về bởi các phương thức keySet, entrySet và giá trị của Map . Tương tự, việc triển khai các giao diện bộ sưu tập, chẳng hạn như Tập hợp và Danh sách, thường sử dụng các lớp thành viên không tĩnh để triển khai các trình vòng lặp của chúng:

```
// Việc sử dụng điển hình của lớp thành viên không có
lớp công khai MySet <E> kéo dài AbstractSet <E> {... // Hàng
      loạt lớp bị bỏ qua

      public Iterator <E> iterator ()
           {return new MyIterator ();
      }

      lớp riêng MyIterator triển khai Iterator <E> {
           ...
      }
}
```

Nếu bạn khai báo một lớp thành viên không yêu cầu quyền truy cập vào một cá thể bao quanh, hãy luôn đặt công cụ sửa đổi tĩnh trong phần khai báo của nó, làm cho nó trở thành một lớp tĩnh thay vì một lớp thành viên không tĩnh. Nếu bạn bỏ qua công cụ sửa đổi này, mỗi phiên bản sẽ có một tham chiếu không liên quan đến phiên bản kèm theo của nó. Việc lưu trữ tham chiếu này tốn thời gian và không gian, đồng thời có thể dẫn đến việc giữ lại phiên bản kèm theo khi nó đủ điều kiện để thu gom rác (Mục 6). Và nếu bạn cần phải cấp phát một cá thể mà không có một cá thể kèm theo, bạn sẽ không thể làm như vậy, vì các cá thể lớp thành viên không tĩnh được yêu cầu phải có một cá thể kèm theo.

Cách sử dụng phổ biến của các lớp thành viên tĩnh riêng là đại diện cho các thành phần của đối tượng được đại diện bởi lớp bao quanh của chúng. Ví dụ: hãy xem xét một cá thể Bản đồ , nó liên kết các khóa với các giá trị. Nhiều triển khai Bản đồ có đối tượng Mục nhập nội bộ cho mỗi cặp khóa-giá trị trong bản đồ. Trong khi mỗi mục nhập được gán với một bản đồ, các phương thức trên một mục nhập (getKey, getValue và setValue) không cần quyền truy cập vào bản đồ. Do đó, sẽ rất lãng phí nếu sử dụng một lớp thành viên không tĩnh để biểu diễn các mục nhập: tốt nhất là một lớp thành viên tĩnh riêng tư . Nếu bạn vô tình bỏ qua công cụ sửa đổi tĩnh trong khai báo mục nhập, bản đồ sẽ vẫn hoạt động, nhưng mỗi mục nhập sẽ chứa một tham chiếu thừa đến bản đồ, điều này gây lãng phí không gian và thời gian.

Điều quan trọng gấp đôi là phải chọn đúng giữa một lớp thành viên tĩnh và không tĩnh nếu lớp được đề cập là thành viên công khai hoặc được bảo vệ của một

Machine Translated by Google

lớp đã xuất. Trong trường hợp này, lớp thành viên là một phần tử API được xuất và không thể thay đổi từ lớp không tĩnh thành lớp thành viên tĩnh trong bản phát hành tiếp theo mà không vi phạm tính tương thích nhị phân.

Các lớp ẩn danh không giống bất cứ thứ gì khác trong lan guage lập trình Java. Như bạn mong đợi, một lớp ẩn danh không có tên. Nó không phải là thành viên của lớp bao quanh nó. Thay vì được khai báo cùng với các thành viên khác, nó được khai báo và khởi tạo đồng thời tại điểm sử dụng. Các lớp ẩn danh được cho phép tại bất kỳ điểm nào trong mã mà biểu thức là hợp pháp. Các lớp ẩn danh có các trường hợp bao quanh nếu và chỉ khi chúng xảy ra trong ngữ cảnh không ổn định.

Nhưng ngay cả khi chúng xảy ra trong bối cảnh tĩnh, chúng không thể có bất kỳ thành viên tĩnh nào.

Có nhiều hạn chế về khả năng áp dụng của các lớp ẩn danh. Bạn không thể khởi tạo chúng ngoại trừ thời điểm chúng được khai báo. Bạn không thể thực hiện kiểm tra instanceof hoặc làm bất kỳ điều gì khác yêu cầu bạn đặt tên cho lớp. Bạn không thể khai báo một lớp ẩn danh để triển khai nhiều giao diện hoặc để mở rộng một lớp và triển khai một giao diện cùng một lúc. Khách hàng của một lớp ẩn danh không thể gọi bất kỳ thành viên nào ngoại trừ những thành viên mà nó kế thừa từ supertype của nó. Bởi vì các lớp ẩn danh xuất hiện ở giữa các biểu thức, chúng phải được giữ ngắn - khoảng mười dòng trở xuống - nếu không khả năng đọc sẽ bị ảnh hưởng.

Một cách sử dụng phổ biến của các lớp ẩn danh là tạo các đối tượng hàm (Mục 21) một cách nhanh chóng. Ví dụ: lệnh gọi phương thức sắp xếp trên trang 104 sắp xếp một mảng các chuỗi theo độ dài của chúng bằng cách sử dụng đối tượng Bộ so sánh ẩn danh .
Một cách sử dụng phổ biến khác của các lớp ẩn danh là tạo các đối tượng quy trình, chẳng hạn như các thể hiện Runnable, Thread hoặc TimerTask . Cách sử dụng phổ biến thứ ba là trong các phương thức static factory (xem phương thức intArrayAsList trong Mục 18).

Các lớp cục bộ là lớp ít được sử dụng nhất trong bốn loại lớp lồng nhau. Một lớp cục bộ có thể được khai báo ở bất cứ đâu mà một biến cục bộ có thể được khai báo và tuân theo các quy tắc xác định phạm vi giống nhau. Các lớp cục bộ có các thuộc tính chung với từng loại lớp lồng nhau khác. Giống như các lớp thành viên, chúng có tên và có thể được sử dụng nhiều lần. Giống như các lớp ẩn danh, chúng chỉ có các cá thể bao quanh nếu chúng được xác định trong ngữ cảnh không tĩnh và chúng không thể chứa các thành viên tĩnh. Và giống như các lớp ẩn danh, chúng nên được giữ ngắn gọn để không làm tổn hại đến khả năng đọc.

Tóm lại, có bốn loại lớp lồng nhau khác nhau và mỗi loại đều có vị trí của nó. Nếu một lớp lồng nhau cần hiển thị bên ngoài một phương thức đơn lẻ hoặc quá dài để phù hợp thoải mái bên trong một phương thức, hãy sử dụng một lớp thành viên. Nếu mỗi thể hiện của lớp thành viên cần một tham chiếu đến thể hiện bao quanh của nó, hãy đặt nó là không tĩnh; không ngoan khác, làm cho nó tĩnh. Giả sử lớp thuộc về bên trong một phương thức, nếu bạn chỉ cần tạo các thể hiện từ một vị trí và có một kiểu đã tồn tại trước đó mà charac sẽ biến lớp đó thành một lớp ẩn danh; nếu không, hãy biến nó thành một lớp cục bộ.

Machine Translated by Google

CHƯ Ơ NG 5

Generics

IN phiên bản 1.5, generic đã đư ợc thêm vào Java. Trư ớc khi có generic, bạn phải truyền mọi đối tư ợng bạn đọc từ một bộ sư u tập. Nếu ai đó vô tình chèn một đối tư ợng không đúng loại, quá trình truyền có thể bị lỗi trong thời gian chạy. Với generic, bạn cho trình biên dịch biết loại đối tư ợng nào đư ợc phép trong mỗi bộ sư u tập. Trình biên dịch sẽ tự động chèn các phôi cho bạn và cho bạn biết tại thời điểm biên dịch nếu bạn cố chèn một đối tư ợng không đúng loại. Điều này dẫn đến các chư ơ ng trình an toàn hơ n và rõ ràng hơ n, như ng những lợi ích này đi kèm với các biến chứng. Chư ơ ng này cho bạn biết cách tối đa hóa lợi ích và giảm thiểu các biến chứng. Để có cách xử lý chi tiết hơ n về vật liệu này, hãy xem hư ớng dẫn của Langer [Langer08] hoặc sách của Naftalin và Wadler [Naftalin07].

Mục 23: Không sử dụng loại thô trong mã mới

Đầu tiên, một vài điều khoản. Một lớp hoặc giao diện có khai báo có một hoặc nhiều tham số kiểu là một lớp hoặc giao diện chung [JLS, 8.1.2, 9.1.2]. Ví dụ: kể từ bản phát hành 1.5, giao diện Danh sách có một tham số kiểu duy nhất, E, đại diện cho kiểu nâng cao của danh sách. Về mặt kỹ thuật, tên của giao diện bây giờ là Danh sách <E> (đọc là “danh sách của E”), như ng mọi ngư ời thư ờng gọi tắt là Danh sách . Các lớp và giao diện chung đư ợc gọi chung là các kiểu chung.

Mỗi kiểu chung xác định một tập hợp các kiểu đư ợc tham số hóa, bao gồm tên lớp hoặc tên giao diện, theo sau là một danh sách có dấu ngoặc nhọn của các tham số kiểu thực tư ơ ng ứng với các tham số kiểu chính thức của kiểu chung [JLS, 4.4, 4.5]. Ví dụ, List <String> (đọc là "danh sách chuỗi") là một kiểu tham số hóa gửi một danh sách có các phần tử thuộc kiểu String. (Chuỗi là tham số kiểu thực tư ơ ng ứng với tham số kiểu chính thức E.)

Cuối cùng, mỗi kiểu chung xác định một kiểu thô, là tên của kiểu chung đư ợc sử dụng mà không có bất kỳ tham số kiểu thực tế nào đi kèm [JLS, 4.8]. Cho kỳ thi-

Machine Translated by Google

ple, kiểu thô tương ứng với List <E> là List. Kiểu thô hoạt động như thể tất cả thông tin kiểu chung đã bị xóa khỏi khai báo kiểu. Đối với tất cả các mục đích thực tế, Danh sách kiểu thô hoạt động giống như kiểu Giao diện Danh sách đã làm trước khi các thuốc chung được thêm vào nền tảng.

Trước khi phát hành 1.5, đây sẽ là một tuyên bố thu thập mẫu mực:

```
// Bây giờ là kiểu tập hợp thô - đừng làm điều này!

/
 ** * Bộ sưu tập tem của tôi. Chỉ chứa các phiên bản Tem.
 * /
tem Bộ sưu tập cuối cùng riêng tư =        ...
```

Nếu bạn vô tình đặt một đồng xu vào bộ sưu tập tem của mình, phần chèn sai sẽ biên dịch và chạy mà không có lỗi:

```
// Nhét nhầm đồng xu vào bộ sưu tập tem Tem.add (new Coin (...));
```

Bạn không gặp lỗi cho đến khi lấy đồng xu từ bộ sưu tập tem:

```
// Bây giờ là một kiểu trình vòng lặp thô - đừng làm điều
này! for (Iterator i = Tem.iterator (); i.hasNext ();) {
     Stamp s = (Dấu) i.next (); // Ném ClassCastException ... // Làm gì đó
     với tem
}
```

Như đã đề cập trong suốt cuốn sách này, việc phát hiện ra các lỗi ngay khi pos sible được thực hiện, lý tưởng nhất là vào thời điểm biên dịch. Trong trường hợp này, bạn không phát hiện ra lỗi cho đến thời gian chạy, rất lâu sau khi nó xảy ra và trong mã khác xa với mã chứa lỗi. Khi bạn nhìn thấy ClassCastException, bạn phải tìm kiếm thông qua cơ sở mã để tìm lời gọi phương thức đưa đồng xu vào bộ sưu tập tem. Trình biên dịch không thể giúp bạn, bởi vì nó không thể hiểu nhận xét có nội dung "Chỉ chứa các phiên bản Tem."

Với generics, bạn thay thế nhận xét bằng một khai báo kiểu được cải thiện cho bộ sưu tập để cho trình biên dịch biết thông tin trước đó đã được ẩn đến trong nhận xét:

```
// Kiểu bộ sưu tập được tham số hóa - stylesafe private
final Bộ sưu tập <Stamp> Tem = ...;
```

Machine Translated by Google

Từ khai báo này, trình biên dịch biết rằng các tem chỉ nên chứa các phiên bản Tem và đảm bảo điều này đúng với trư ờng hợp này, giả sử toàn bộ cơ sở mã của bạn đư ợc biên dịch bằng trình biên dịch từ phiên bản 1.5 trở lên và tất cả mã biên dịch mà không phát ra (hoặc triệt tiêu; xem Mục 24) bất kỳ cảnh báo nào. Khi tem đư ợc khai báo với kiểu tham số hóa, việc chèn sai sẽ tạo ra thông báo lỗi thời gian biên dịch cho bạn biết chính xác điều gì sai:

```
Test.java:9: không thể áp dụng thêm (Tem) trong Bộ sư u tập <Stamp> cho (Đồng xu)
Tem.add (Đồng xu mới ());
                          ^
```

Như một lợi ích bổ sung, bạn không còn phải truyền theo cách thủ công khi xóa các phần tử trong bộ sư u tập. Trình biên dịch chèn các phôi vô hình cho bạn và tin rằng chúng sẽ không bị lỗi (giả sử, một lần nữa, tất cả mã của bạn đư ợc biên dịch bằng trình biên dịch nhận biết chung và không tạo ra hoặc ngăn chặn bất kỳ cảnh báo nào). Điều này đúng cho dù bạn sử dụng vòng lặp for-each (Mục 46):

```
// vòng lặp for-each qua bộ sư u tập đư ợc tham số hóa - stylesafe for (Stamp s:
Tem) { // Không ép kiểu
     ... // Làm gì đó với con tem
}
```

hoặc vòng lặp for truyền thống :

```
// vòng lặp for với khai báo trình lặp đư ợc tham số hóa - stylesafe for (Iterator
<Stamp> i = Tem.iterator (); i.hasNext ();) {Stamp s = i.next (); // Không cần ép
     kiểu ... // Làm gì đó với tem

}
```

Mặc dù viễn cảnh vô tình nhét một đồng xu vào bộ sư u tập tem có vẻ xa vời, như ng vấn đề là có thật. Ví dụ, có thể dễ dàng tư ởng tư ợng ai đó đặt một cá thể java.util.Date vào một bộ sư u tập đư ợc cho là chỉ chứa các cá thể java.sql.Date .

Như đã lư u ý ở trên, vẫn hợp pháp để sử dụng các kiểu tập hợp và các kiểu chung khác mà không cung cấp các tham số kiểu, như ng bạn không nên làm điều đó. Nếu bạn sử dụng các loại thô, bạn sẽ mất tất cả các lợi ích an toàn và biểu cảm của thuốc gốc. Cho rằng bạn không nên sử dụng các kiểu thô, tại sao các nhà thiết kế ngôn ngữ lại cho phép chúng? Để tư ơ ng thích với vide chuyên nghiệp. Nền tảng Java sắp bư ớc vào thập kỷ thứ hai khi các generic đư ợc giới thiệu và có một lư ợng lớn mã Java trong

Machine Translated by Google

sự tồn tại không sử dụng generic. Điều quan trọng được coi là tất cả mã này vẫn hợp pháp và có thể tương tác với mã mới sử dụng generic. Nó phải hợp pháp để chuyển các phiên bản của các kiểu tham số hóa sang các phương thức được thiết kế để sử dụng với các kiểu thông thường và ngược lại. Yêu cầu này, được gọi là khả năng tương thích di chuyển, đã dẫn đến quyết định hỗ trợ các loại thô.

Mặc dù bạn không nên sử dụng các kiểu thô như Danh sách trong mã mới, nhưng bạn có thể sử dụng các kiểu được tham số hóa để cho phép chèn các đối tượng tùy ý, chẳng hạn như Danh sách <Đối tượng>. Chỉ là sự khác biệt giữa Danh sách kiểu thô và kiểu được tham số hóa Danh sách <Đối tượng> là gì? Nói một cách dễ hiểu, cái trước đã chọn không tham gia kiểm tra kiểu chung, trong khi cái sau đã nói rõ ràng với trình biên dịch rằng nó có khả năng giữ các đối tượng thuộc bất kỳ kiểu nào. Mặc dù bạn có thể truyền một Danh sách <Chuỗi> cho một tham số kiểu Danh sách, nhưng bạn không thể truyền nó cho một tham số kiểu Danh sách <Đối tượng>.

Có các quy tắc định kiểu con cho các tổng thể và Danh sách <Chuỗi> là một kiểu con của Danh sách kiểu thô , nhưng không thuộc kiểu được tham số hóa Danh sách <Đối tượng> (Mục 25). Vì vậy, bạn sẽ mất an toàn kiểu nếu bạn sử dụng kiểu thô như Danh sách, nhưng không phải nếu bạn sử dụng kiểu được tham số hóa như Danh sách <Đối tượng>.

Để làm cho điều này cụ thể, hãy xem xét chương trình sau:

```
// Sử dụng kiểu thô (Danh sách) - không thành công trong
thời gian chạy! public static void main (String [] args)
      {Danh sách <Chuỗi> chuỗi = new ArrayList <Chuỗi> (); không an
      toànAdd (chuỗi, số nguyên mới (42)); String s = string.get
      (0); // Diễn viên do trình biên dịch tạo
}

private static void secureAdd (List list, Object o) {list.add (o);

}
```

Chương trình này biên dịch, nhưng vì nó sử dụng Danh sách loại thô , bạn nhận được cảnh báo:

```
Test.java:10: warning: lệnh gọi bỏ chọn để thêm (E) vào kiểu thô List list.add (o); ^
```

Và thực sự, nếu bạn chạy chương trình, bạn sẽ nhận được ClassCastException khi chương trình cố gắng truyền kết quả của các lệnh gọi string.get (0) thành một Chuỗi.

Đây là một quá trình ép kiểu do trình biên dịch tạo ra, vì vậy nó thường được đảm bảo thành công, nhưng trong trường hợp này, chúng tôi đã bỏ qua cảnh báo của trình biên dịch và phải trả giá.

Machine Translated by Google

Nếu bạn thay thế Danh sách kiểu thô bằng Danh sách kiểu đư ợc tham số hóa là Danh sách <Đối tư ợng> trong khai báo secureAdd và cố gắng biên dịch lại chư ơ ng trình, bạn sẽ thấy rằng nó không còn biên dịch nữa. Đây là thông báo lỗi:

```
Test.java:5: không an toànAdd (Danh sách <Đối tư ợng>, Đối tư ợng) không thể đư ợc áp
dụng cho (Danh sách <Chuỗi>, Số nguyên) không an toànAdd (chuỗi, Số nguyên mới (42));

      ^
```

Bạn có thể bị cám dỗ để sử dụng kiểu thô cho một tập hợp có kiểu phần tử không xác định và không quan trọng. Ví dụ: giả sử bạn muốn viết một phư ơ ng thức nhận hai tập hợp và trả về số phần tử mà chúng có điểm chung.
Đây là cách bạn có thể viết một phư ơ ng pháp như vậy nếu bạn chư a quen với thuốc generic:

```
// Sử dụng kiểu thô cho kiểu phần tử không xác định - đừng làm điều này! static
int numElementsInCommon (Đặt s1, Đặt s2) {int result = 0; for (Đối tư ợng o1:
      s1) if (s2.contains (o1)) result ++; trả về kết quả;




}
```

Phư ơ ng pháp này hiệu quả như ng nó sử dụng các loại thô, rất nguy hiểm. Kể từ bản phát hành 1.5, Java đã cung cấp một giải pháp thay thế an toàn đư ợc gọi là các loại ký tự đại diện không bị ràng buộc. Nếu bạn muốn sử dụng một kiểu chung như ng bạn không biết hoặc không quan tâm đến tham số kiểu thực tế là gì, bạn có thể sử dụng dấu chấm hỏi để thay thế. Ví dụ: kiểu ký tự đại diện không bị ràng buộc cho kiểu chung Đặt <E> là Đặt <?> (Đọc là "tập hợp của một số loại"). Đây là loại Tập hợp đư ợc tham số hóa chung nhất , có khả năng giữ bất kỳ tập hợp nào. Đây là cách phư ơ ng thức numElementsInCommon trông như thế nào với các loại ký tự đại diện không bị ràng buộc:

```
// Kiểu ký tự đại diện không giới hạn - typeafe và static static
int numElementsInCommon (Set <?> S1, Set <?> S2) {
      int kết quả = 0;
      for (Đối tư ợng o1: s1) if
            (s2.contains (o1)) result
                  ++; trả về kết quả;

}
```

Sự khác biệt giữa kiểu ký tự đại diện không liên kết Set <?> Và kiểu thô Set là gì? Những dấu chấm hỏi có thực sự mua đư ợc cho bạn điều gì không? Không phải để tin vào quan điểm, như ng kiểu ký tự đại diện là an toàn còn kiểu thô thì không. Bạn có thể đặt bất kỳ phần tử nào

Machine Translated by Google

vào một tập hợp với kiểu thô, dễ dàng làm hỏng kiểu bất biến của tập hợp (như được minh họa bằng phương thức Không an toàn ở trang 112); bạn không thể đặt bất kỳ ele ment (ngoài null) vào Bộ sưu tập <?>. Cố gắng làm như vậy gener sẽ nhận được thông báo lỗi thời gian biên dịch như sau:

```
WildCard.java:13: không tìm thấy biểu tượng ký
hiệu: method add (String) location: interface
Bộ sưu tập <capture # 825 of?> C.add ("verboten");
                ^
```

Phải thừa nhận rằng thông báo lỗi này để lại điều gì đó mong muốn, nhưng com piler đã thực hiện công việc của mình, ngăn bạn làm hỏng kiến thức invari loại của bộ sưu tập. Bạn không chỉ không thể đặt bất kỳ phần tử nào (ngoài null) vào Bộ sưu tập <?>, Mà còn không thể giả định bất kỳ điều gì về loại đối tượng mà bạn lấy ra. Nếu những hạn chế này là không thể chấp nhận được, bạn có thể sử dụng các phương pháp chung (Mục 27) hoặc các loại ký tự đại diện có giới hạn (Mục 28).

Có hai ngoại lệ nhỏ đối với quy tắc rằng bạn không nên sử dụng kiểu thô trong mã mới, cả hai đều xuất phát từ thực tế là thông tin kiểu chung bị xóa trong thời gian chạy (Mục 25). Bạn phải sử dụng các kiểu thô trong các ký tự của lớp. Sự hư cấu cụ thể không cho phép sử dụng các kiểu được tham số hóa (mặc dù nó cho phép các kiểu mảng và kiểu nguyên thủy) [JLS, 15.8.2]. Nói cách khác, lớp List.class, String []. Và int.class đều hợp pháp, nhưng lớp List <String> .class và List <?>. Thì không.

Ngoại lệ thứ hai đối với quy tắc liên quan đến toán tử instanceof . Bởi vì thông tin kiểu chung bị xóa trong thời gian chạy, việc sử dụng toán tử instanceof trên các kiểu được tham số hóa khác với kiểu ký tự đại diện không bị ràng buộc là bất hợp pháp. Việc sử dụng các kiểu ký tự đại diện không bị ràng buộc thay cho kiểu thô không ảnh hưởng đến hoạt động của toán tử instanceof theo bất kỳ cách nào. Trong trường hợp này, dấu ngoặc nhọn và dấu hỏi chỉ là nhiễu. Đây là cách ưa thích để sử dụng toán tử instanceof với các kiểu chung:

```
// Sử dụng hợp pháp kiểu thô - toán tử instanceof if (o instanceof
Set) {                                  // Loại thô
     Đặt <?> M = (Đặt <?>) O;           // Loại ký tự đại diện
      ...
}
```

Lưu ý rằng khi bạn đã xác định rằng o là một Tập hợp, bạn phải truyền nó sang kiểu ký tự đại diện là Set <?>, Không phải kiểu thô là Set. Đây là kiểu ép kiểu đã được kiểm tra, vì vậy nó sẽ không gây ra cảnh báo trình biên dịch.

Machine Translated by Google

Tóm lại, việc sử dụng các loại thô có thể dẫn đến ngoại lệ trong thời gian chạy, vì vậy không sử dụng chúng trong mã mới. Chúng chỉ đư ợc cung cấp cho khả năng tư ơ ng thích và tư ơ ng tác với mã kế thừa trư ớc khi giới thiệu thuốc gốc. Như một đánh giá nhanh, Tập hợp <Đối tư ợng> là một kiểu tham số đại diện cho một tập hợp có thể chứa các đối tư ợng của bất kỳ loại nào, Set <?> là kiểu ký tự đại diện đại diện cho một tập hợp chỉ có thể chứa các đối tư ợng thuộc một số loại không xác định và Set là một loại thô, chọn không tham gia hệ thống loại chung. Hai cái đầu tiên là an toàn và cái cuối cùng thì không.

Để tham khảo nhanh, các điều khoản đư ợc giới thiệu trong mục này (và một số điều khoản đư ợc giới thiệu ở những nơ i khác trong chư ơ ng này) đư ợc tóm tắt trong bảng sau:

Thuật ngữ	Ví dụ	Mục
Loại đư ợc tham số hóa	Danh sách <Chuỗi>	Mặt hàng 23
Tham số loại thực tế	Sợi dây	Mặt hàng 23
Loại chung	Danh sách <E>	Mục 23, 26
Tham số kiểu chính thức	E	Mặt hàng 23
Danh sách loại ký tự đại diện không bị ràng buộc <?>		Mặt hàng 23
Loại thô	Danh sách	Mặt hàng 23
Tham số loại có giới hạn	<E mở rộng số>	Mặt hàng 26
Loại đệ quy bị ràng buộc	<T mở rộng Có thể so sánh đư ợc <T>>	Mặt hàng 27
Loại ký tự đại diện đư ợc giới hạn	Danh sách <? mở rộng Số>	Mặt hàng 28
Phư ơ ng pháp chung	static <E> List <E> asList (E [] a)	Mặt hàng 27
Nhập mã thông báo	String.class	Mặt hàng 29

Machine Translated by Google

Mục 24: Loại bỏ các cảnh báo không được kiểm tra

Khi bạn lập trình với generic, bạn sẽ thấy nhiều cảnh báo trình biên dịch: cảnh báo truyền bỏ chọn, cảnh báo gọi phương thức không được chọn, cảnh báo tạo mảng chung không được chọn và cảnh báo chuyển đổi không được chọn. Bạn càng có nhiều kinh nghiệm với generic, bạn sẽ nhận được ít cảnh báo hơn, nhưng đừng mong đợi mã mới được viết sử dụng generic để biên dịch rõ ràng.

Nhiều cảnh báo không được kiểm tra rất dễ bị loại bỏ. Ví dụ, giả sử bạn vô tình viết khai báo này:

```
Đặt <Lark> exaltation = new HashSet ();
```

Trình biên dịch sẽ nhẹ nhàng nhắc nhở bạn những gì bạn đã làm sai:

```
Venery.java:4: warning: Đã tìm thấy chuyển đổi không được kiểm tra [chưa
được kiểm tra]: HashSet, bắt buộc: Đặt <Lark>
     Đặt <Lark> exaltation = new HashSet ();
                             ^
```

Sau đó, bạn có thể thực hiện chỉnh sửa được chỉ định, làm cho cảnh báo biến mất:

```
Đặt <Lark> exaltation = new HashSet <Lark> ();
```

Một số cảnh báo sẽ khó loại bỏ hơn nhiều . Chương này chứa đầy các ví dụ về những cảnh báo như vậy. Khi bạn nhận được cảnh báo cần phải suy nghĩ, hãy kiên trì! Loại bỏ mọi cảnh báo không được kiểm tra mà bạn có thể. Nếu bạn loại bỏ tất cả các cảnh báo, bạn yên tâm rằng mã của bạn là loại an toàn, đó là một điều rất tốt. Điều đó có nghĩa là bạn sẽ không nhận được ClassCastException trong thời gian chạy và nó làm tăng sự tin tưởng của bạn rằng chương trình của bạn đang hoạt động như bạn dự định.

Nếu bạn không thể loại bỏ cảnh báo và bạn có thể chứng minh rằng mã kích hoạt cảnh báo là an toàn kiểu chữ, thì (và chỉ sau đó) loại bỏ cảnh báo bằng chú thích @SuppressWarnings ("bỏ chọn") . Nếu bạn ngăn chặn việc nhập cảnh báo mà không chứng minh trước rằng mã đó là an toàn kiểu chữ, bạn chỉ đang tạo cho mình một cảm giác an toàn sai lầm. Mã có thể biên dịch mà không đưa ra bất kỳ cảnh báo nào, nhưng nó vẫn có thể ném ra một ClassCastException trong thời gian chạy. Tuy nhiên, nếu bạn bỏ qua các cảnh báo chưa được kiểm tra mà bạn biết là an toàn (thay vì loại bỏ chúng), bạn sẽ không nhận thấy khi một cảnh báo mới xuất hiện đại diện cho một vấn đề thực sự. Cảnh báo mới sẽ bị mất giữa tất cả các cảnh báo giả mà bạn không tắt tiếng.

Machine Translated by Google

Chú thích SuppressWarnings có thể đư ợc sử dụng ở bất kỳ mức độ chi tiết nào, từ khai báo biến cục bộ riêng lẻ đến toàn bộ lớp. Luôn sử dụng chú thích Cảnh báo đàn áp trên phạm vi nhỏ nhất có thể. Thông thư ờng, đây sẽ là một khai báo biến hoặc một phư ơ ng thức hoặc hàm tạo rất ngắn. Không bao giờ sử dụng Cảnh báo đàn áp trên toàn bộ lớp. Làm như vậy có thể che giấu các cảnh báo quan trọng.

Nếu bạn thấy mình đang sử dụng chú thích SuppressWarnings trên một phư ơ ng thức hoặc hàm tạo dài hơ n một dòng, bạn có thể chuyển nó vào một khai báo biến cục bộ. Bạn có thể phải khai báo một biến cục bộ mới, như ng nó đáng giá. Ví dụ: hãy xem xét phư ơ ng thức toArray này, đến từ ArrayList:

```
public <T> T [] toArray (T [] a) {if
     (a.length <size) return (T [])
          Arrays.copyOf (các phần tử, kích thư ớc, a.getClass ()); System.arraycopy
     (phần tử, 0, a, 0, kích thư ớc); if (a.length> size) a [size] = null; trả lại
     a;


}
```

Nếu bạn biên dịch ArrayList, phư ơ ng thức này sẽ tạo ra cảnh báo này:

```
ArrayList.java:305: cảnh báo: [không kiểm tra] đúc không kiểm tra đư ợc
tìm thấy: Đối tư ợng [], bắt buộc: T [] return (T []) Arrays.copyOf (các
     phần tử, kích thư ớc, a.getClass ());
                                          ^
```

Việc đặt chú thích SuppressWarnings trên câu lệnh return là bất hợp pháp vì nó không phải là một khai báo [JLS, 9.7]. Bạn có thể muốn đặt chú thích trên toàn bộ phư ơ ng pháp, như ng đừng. Thay vào đó, hãy khai báo một biến cục bộ để giữ giá trị trả về và chú thích phần khai báo của nó, như sau:

```
// Thêm biến cục bộ để giảm phạm vi của @SuppressWarnings public <T> T [] toArray
(T [] a) {if (a.length <size) {

          // Việc ép kiểu này là đúng vì mảng mà chúng ta đang tạo // có cùng kiểu
          với mảng đư ợc truyền vào, là T [].
          @SuppressWarnings ("bỏ chọn") T [] result =
               (T []) Arrays.copyOf (các phần tử, kích thư ớc, a.getClass ());
          trả về kết quả;
     }
     System.arraycopy (phần tử, 0, a, 0, kích thư ớc); if
     (a.length> size) a [size] = null; trả lại a;


}
```

Machine Translated by Google

Phương pháp này biên dịch một cách rõ ràng và giảm thiểu phạm vi trong đó các lệnh cảnh báo không được kiểm tra sẽ bị loại bỏ.

Mỗi khi bạn sử dụng chú thích @SuppressWarnings ("bỏ chọn") , thêm một bình luận nói lý do tại sao nó an toàn để làm như vậy. Điều này sẽ giúp những người khác hiểu mã, và quan trọng hơn, nó sẽ giảm khả năng ai đó sẽ sửa đổi mã ify để làm cho việc tính toán không an toàn. Nếu bạn cảm thấy khó viết như vậy một nhận xét, tiếp tục suy nghĩ. Cuối cùng, bạn có thể nhận ra rằng cuối cùng thì vở opera không được kiểm tra không an toàn.

Tóm lại, các cảnh báo không được kiểm tra là quan trọng. Đừng bỏ qua chúng. Mỗi cảnh báo không được đánh dấu thể hiện khả năng xảy ra ClassCastException tại thời điểm chạy. Cố gắng hết sức để loại bỏ những cảnh báo này. Nếu bạn không thể loại bỏ một cảnh báo bỏ chọn và bạn có thể chứng minh rằng mã kích động nó là an toàn kiểu chữ, ngăn cảnh báo bằng chú thích @SuppressWarnings ("bỏ chọn") trong phạm vi hẹp nhất có thể. Ghi lại cơ sở lý luận cho quyết định của bạn để ngăn chặn cảnh báo trong một bình luận.

Machine Translated by Google

Mục 25: Ư u tiên danh sách thành mảng

Mảng khác với các kiểu chung theo hai cách quan trọng. Đầu tiên, các mảng là hiệp phư ơ ng sai. Từ nghe có vẻ đáng sợ này có nghĩa đơ n giản là nếu Sub là một kiểu con của Super, thì kiểu mảng Sub [] là một kiểu con của Super []. Ngư ợc lại, generic là bất biến: đối với bất kỳ hai kiểu riêng biệt nào là Type1 và Type2, List <Type1> không phải là kiểu con cũng không phải là siêu kiểu của List <Type2> [JLS, 4.10; Naftalin07, 2,5]. Bạn có thể nghĩ rằng điều này có nghĩa là các generic bị thiếu, như ng có thể cho rằng đó là các mảng bị thiếu hụt.

Đoạn mã này là hợp pháp:

```
// Không thực hiện đư ợc trong thời gian chạy!
Object [] objectArray = new Long [1];
objectArray [0] = "Tôi không phù hợp"; // Ném ArrayStoreException
```

như ng cái này không phải là:

```
// Sẽ không biên dịch!
Danh sách <Đối tư ợng> ol = new ArrayList <Dài> (); // Các kiểu không tư ơ ng thích
ol.add ("Tôi không phù hợp");
```

Dù bằng cách nào, bạn không thể đặt một Chuỗi vào một vùng chứa Dài , như ng với một mảng, bạn phát hiện ra rằng bạn đã mắc lỗi trong thời gian chạy; với một danh sách, bạn sẽ tìm ra tại thời điểm biên dịch. Tất nhiên bạn muốn tìm hiểu tại thời điểm biên dịch.

Sự khác biệt lớn thứ hai giữa mảng và generic là mảng đư ợc sửa đổi [JLS , 4.7]. Điều này có nghĩa là các mảng biết và thực thi các kiểu phần tử của chúng trong thời gian chạy. Như đã lư u ý ở trên, nếu bạn cố gắng lư u trữ một Chuỗi vào một mảng Long, bạn sẽ nhận đư ợc ArrayStoreException. Ngư ợc lại, generic đư ợc thực hiện bằng cách tẩy xóa [JLS, 4.6]. Điều này có nghĩa là chúng thực thi các ràng buộc kiểu của chúng chỉ tại thời điểm biên dịch và loại bỏ (hoặc xóa) thông tin loại phần tử của chúng trong thời gian chạy. Erasure là thứ cho phép các loại generic tư ơ ng tác tự do với mã kế thừa không sử dụng generic (Mục 23).

Do những khác biệt cơ bản này, mảng và generic không kết hợp tốt với nhau. Ví dụ: việc tạo một mảng có kiểu chung, kiểu đư ợc tham số hóa hoặc tham số kiểu là bất hợp pháp. Không có biểu thức tạo mảng nào trong số này là hợp pháp: new List <E> [], new List <String> [], new E []. Tất cả sẽ dẫn đến lỗi chung của mảng sáng tạo tại thời điểm biên dịch.

Tại sao việc tạo một mảng chung là bất hợp pháp? Bởi vì nó không an toàn. Nếu nó là hợp pháp, các phôi đư ợc tạo bởi trình biên dịch trong một chư ơ ng trình đúng cách khác có thể bị lỗi trong thời gian chạy với ClassCastException. Điều này sẽ vi phạm tee guaran cơ bản đư ợc cung cấp bởi hệ thống loại chung.

Machine Translated by Google

Để làm cho điều này cụ thể hơn, hãy xem xét đoạn mã sau:

```
// Tại sao tạo mảng chung chung là bất hợp pháp - sẽ không biên dịch!
Danh sách <Chuỗi> [] stringLists = new Danh sách <Chuỗi> [1]; // (1)
Danh sách <Integer> intList = Arrays.asList (42); // (2)
Đối tượng [] đối tượng = stringLists; // (3)
đối tượng [0] = intList; // (4)
String s = stringLists [0] .get (0); // (5)
```

Hãy giả sử rằng dòng 1, tạo ra một mảng chung, là hợp pháp. Dòng 2 tạo và khởi tạo một Danh sách <Integer> chứa một phần tử duy nhất. Dòng 3 lưu trữ Liệt kê mảng <Chuỗi> thành một biến mảng Đối tượng , biến này hợp pháp vì mảng đồng biến. Dòng 4 lưu trữ Danh sách <Integer> vào phần tử duy nhất của Mảng đối tượng , thành công vì các số liệu chung được triển khai bằng cách tẩy: kiểu thời gian chạy của một cá thể List <Integer> chỉ đơn giản là Danh sách và kiểu thời gian chạy của một cá thể List <String> [] là List [], vì vậy việc gán này không tạo ra một ArrayStoreException. Bây giờ chúng tôi đang gặp rắc rối. Chúng tôi đã lưu trữ một Danh sách <Integer> thể hiện thành một mảng được khai báo chỉ chứa các thể hiện của Danh sách <Chuỗi> . Ở trong dòng 5, chúng tôi lấy phần tử duy nhất từ danh sách duy nhất trong mảng này. Trình biên dịch tự động chuyển phần tử được truy xuất thành Chuỗi, nhưng đó là Số nguyên, vì vậy chúng tôi nhận được một ClassCastException trong thời gian chạy. Để ngăn điều này xảy ra, dòng 1 (tạo ra một mảng chung) tạo ra lỗi thời gian biên dịch.

Các kiểu như E, Danh sách <E> và Danh sách <Chuỗi> về mặt kỹ thuật được gọi là các kiểu không thể thay thế [JLS, 4.7]. Nói một cách trực quan, loại không thể sử dụng lại là loại có biểu diễn thời gian chạy chứa ít thông tin hơn biểu diễn thời gian biên dịch của nó. Các loại tham số duy nhất có thể kết nối lại là các loại ký tự đại diện không bị ràng buộc chẳng hạn như Danh sách <?> và Bản đồ <?,?> (Mục 23). Nó là hợp pháp, mặc dù không thường xuyên hữu ích, để tạo mảng các kiểu ký tự đại diện không bị ràng buộc.

Việc cấm tạo mảng chung có thể gây khó chịu. Điều đó có nghĩa là, đối với đề thi, nói chung không thể cho một kiểu chung trả về một mảng thuộc kiểu ele ment của nó (nhưng hãy xem Mục 29 để biết giải pháp từng phần). Nó cũng có nghĩa là bạn có thể nhận được cảnh báo khó hiểu khi sử dụng các phương thức varargs (Mục 42) kết hợp với các loại chung chung. Điều này là do mỗi khi bạn gọi một phương thức varargs, một mảng sẽ được tạo để giữ các tham số varargs. Nếu loại phần tử của mảng này không thể tìm lại được, bạn sẽ nhận được một cảnh báo. Bạn có thể làm ít điều gì về những cảnh báo này ngoài việc để ngăn chặn chúng (Mục 24) và tránh trộn lẫn các loại generic và varargs trong các API của bạn.

Khi bạn gặp lỗi tạo mảng chung, giải pháp tốt nhất thường là sử dụng kiểu tập hợp Danh sách <E> ưu tiên cho kiểu mảng E []. Bạn có thể hy sinh một số hiệu suất hoặc tính ngắn gọn, nhưng đổi lại, bạn có được sự an toàn về loại tốt hơn và khả năng tương tác.

Machine Translated by Google

Ví dụ: giả sử bạn có một danh sách được đồng bộ hóa (thuộc loại được trả về bởi Collections.synchronizedList) và một hàm nhận hai giá trị của loại được giữ bởi danh sách và trả về một phần ba. Bây giờ, giả sử bạn muốn viết một phương thức để "giảm" danh sách bằng cách áp dụng hàm trên nó. Nếu danh sách chứa các số nguyên và hàm thêm hai giá trị nguyên, thì phương thức Reduce trả về tổng của tất cả các giá trị trong danh sách. Nếu hàm nhân hai giá trị nguyên, phương thức trả về tích của các giá trị trong danh sách. Nếu danh sách chứa các chuỗi và hàm func nối hai chuỗi, phương thức trả về một chuỗi bao gồm tất cả các chuỗi trong danh sách theo thứ tự. Ngoài một danh sách và một hàm, phương thức rút gọn nhận một giá trị ban đầu cho việc giảm bớt, giá trị này được trả về nếu danh sách trống. (Giá trị ban đầu thường là phần tử nhận dạng cho hàm, giá trị này là 0 đối với phép cộng, 1 đối với phép nhân và "" đối với nối chuỗi.) Đây là cách mã có thể trông như thế nào nếu không có số liệu chung:

```
// Giảm không có generic và có lỗ hổng đồng thời! static Object Reduce
(List list, Function f, Object initVal) {
      đồng bộ hóa (danh sách)
            {Kết quả đối tượng = initVal;
            for (Đối tượng o: list) result
                  = f.apply (result, o); trả về kết
            quả;
      }
}

giao diện Chức năng {
      Áp dụng đối tượng (Đối tượng arg1, Đối tượng arg2);
}
```

Bây giờ, giả sử bạn đã đọc Mục 67, mục này yêu cầu bạn không gọi “các phương pháp ngoại lai” từ một vùng được đồng bộ hóa. Vì vậy, bạn sửa đổi phương thức giảm để sao chép các lều con của danh sách trong khi giữ khóa, điều này cho phép bạn thực hiện thu nhỏ trên bản sao. Trước khi phát hành 1.5, cách tự nhiên để thực hiện việc này là sử dụng phương thức toArray của List (khóa danh sách bên trong):

```
// Giảm thiểu không có chung chung hoặc lỗ hổng đồng thời Đối
tượng tĩnh Giảm (Danh sách liệt kê, Hàm f, Đối tượng initVal) {
      Object [] snapshot = list.toArray (); // Khóa danh sách bên trong Object result
      = initVal; for (Đối tượng o: list) result = f.apply (result, o); trả về kết
      quả;


}
```

Machine Translated by Google

Nếu bạn cố gắng làm điều này với thuốc chung, bạn sẽ gặp rắc rối về loại mà chúng tôi Thảo luận ở trên. Đây là phiên bản chung của giao diện Hàm :

```
giao diện Chức năng <T> {
     T áp dụng (T arg1, T arg2);
}
```

Và đây là một nỗ lực ngây thơ để áp dụng generic vào phiên bản sửa đổi của phư ơ ng pháp rút gọn . Đây là một phư ơ ng pháp chung (Tiết 27). Đừng lo lắng nếu bạn không hiểu khai báo. Đối với mục đích của mục này, bạn nên tập trung vào phần thân của phư ơ ng thức:

```
// Phiên bản giảm thiểu chung chung ngây thơ - sẽ không biên dịch!
static <E> E Reduce (List <E> list, Function <E> f, E initVal) {E [] snapshot =
     list.toArray (); // Khóa danh sách E result = initVal; for (E e: snapshot)
     result = f.apply (result, e); trả về kết quả;



}
```

Nếu bạn cố gắng biên dịch phư ơ ng thức này, bạn sẽ gặp lỗi sau:

```
Reduce.java:12: tìm thấy các loại không
tư ơ ng thích: Đối tư ợng [], bắt buộc: E []
          E [] snapshot = list.toArray (); // Danh sách khóa
                                           ^
```

Bạn nói không có vấn đề gì lớn, tôi sẽ truyền mảng Đối tư ợng thành mảng E :

```
E [] snapshot = (E []) list.toArray ();
```

Điều đó sẽ loại bỏ lỗi, như ng bây giờ bạn nhận đư ợc cảnh báo:

```
Reduce.java:12: warning: [chư a chọn] đã tìm thấy diễn viên bỏ
chọn: Object [], bắt buộc: E []
          E [] snapshot = (E []) list.toArray (); // Danh sách khóa
                                                  ^
```

Trình biên dịch cho bạn biết rằng nó không thể kiểm tra độ an toàn của quá trình truyền trong thời gian chạy bởi vì nó không biết E là gì trong thời gian chạy — hãy nhớ rằng, thông tin loại phần tử sẽ bị xóa khỏi generic trong thời gian chạy. Chư ơ ng trình này sẽ hoạt động? Có, nó chỉ ra là nó sẽ, như ng nó không an toàn. Với những sửa đổi nhỏ, bạn có thể làm cho nó

Machine Translated by Google

ClassCastException trên một dòng không chứa một kiểu truyền rõ ràng. Kiểu thời gian biên dịch của ảnh chụp nhanh là E [] có thể là Chuỗi [], Số nguyên [] hoặc bất kỳ loại mảng nào khác. Loại thời gian chạy là Đối tượng [], và điều đó thật nguy hiểm. Truyền tới các mảng thuộc loại không thể sử dụng lại chỉ được sử dụng trong các trường hợp đặc biệt (Mục 26).

Vậy bạn nên làm gì? Sử dụng một danh sách thay vì một mảng. Đây là một phiên bản của Giảm phương thức biên dịch mà không có lỗi hoặc cảnh báo:

```
// Giảm tổng quát dựa trên danh sách
static <E> E Reduce (Danh sách <E> danh sách, Hàm <E> f, E initVal) {
      Liệt kê ảnh chụp nhanh
      <E>; đồng bộ hóa (danh
            sách) {snapshot = new ArrayList <E> (danh sách);
      }
      Kết quả E = initVal; for
      (E e: snapshot) result =
            f.apply (result, e); trả về kết quả;

}
```

Phiên bản này dài dòng hơn một chút so với phiên bản mảng, nhưng nó đáng để bạn yên tâm khi biết rằng bạn sẽ không nhận được ClassCastException khi chạy.

Tóm lại, mảng và generic có các quy tắc kiểu rất khác nhau. Mảng có tính đồng biến và thay đổi; generics là bất biến và bị xóa. Do đó, các mảng cung cấp an toàn kiểu thời gian chạy nhưng không an toàn kiểu thời gian biên dịch và ngược lại cho các kiểu chung. Nói chung, mảng và generic không kết hợp tốt với nhau. Nếu bạn thấy mình trộn chúng và nhận được lỗi hoặc cảnh báo thời gian biên dịch, điều đầu tiên của bạn nên là thay thế các mảng bằng danh sách.

Machine Translated by Google

Mục 26: Các loại chung chung ủng hộ

Nói chung không quá khó để tham số hóa các khai báo tập hợp của bạn và sử dụng các kiểu và phư ơ ng pháp chung đư ợc cung cấp bởi JDK. Viết các kiểu chung chung của riêng bạn hơ i khó hơ n một chút, như ng bạn cũng cần cố gắng tìm hiểu cách thực hiện.

Hãy xem xét việc triển khai ngăn xếp đơ n giản từ Mục 6:

```
// Tập hợp dựa trên đối tư ợng - một ứng cử viên chính cho các phần tử chung
lớp public class Stack {private Object [] ; int size = 0; private static
     final int DEFAULT_INITIAL_CAPACITY = 16;



     public Stack ()
          {element = new Object [DEFAULT_INITIAL_CAPACITY];
     }

     public void push (Object e)
          {ensureCapacity (); các phần
          tử [size ++] = e;
     }

     public Object pop () {if
          (size == 0) throw new
               EmptyStackException ();
          Kết quả đối tư ợng = các phần tử [- size];
          phần tử [size] = null; // Loại bỏ kết quả trả về tham chiếu lỗi thời;

     }

     public boolean isEmpty () {return
          size == 0;
     }

     private void ensureCapacity () {if
          (Elements.length == size) Elements
               = Arrays.copyOf (Elements, 2 * size + 1);
     }
}
```

Nói cách khác, lớp này là một ứng cử viên hàng đầu cho việc tổng quát hóa, vì đã đư ợc tăng cư ờng một cách dễ dàng để tận dụng lợi thế của các kiểu chung. Như hiện tại, bạn phải ép các đối tư ợng đư ợc bật ra khỏi ngăn xếp và những lần ép đó có thể không thành công trong thời gian chạy. Bư ớc đầu tiên trong việc tạo ra một lớp là thêm một hoặc nhiều tham số kiểu vào decla-

Machine Translated by Google

khẩu phần ăn. Trong trư ờng hợp này có một tham số kiểu, đại diện cho kiểu phần tử của ngăn xếp và tên quy ư ớc cho tham số này là E (Mục 44).

Bư ớc tiếp theo là thay thế tất cả việc sử dụng kiểu Đối tư ợng bằng tham số kiểu thích hợp, sau đó thử biên dịch chư ơ ng trình kết quả:

```
// Nỗ lực ban đầu để tổng hợp hóa Stack = sẽ không biên dịch! lớp công
khai Stack <E> {
      các phần tử E [] private ;
      int size = 0; private
      static final int DEFAULT_INITIAL_CAPACITY = 16;

      public Stack ()
            {element = new E [DEFAULT_INITIAL_CAPACITY];
      }

      public void push (E e)
            {ensureCapacity (); các
            phần tử [size ++] = e;
      }

      public E pop () {if
            (size == 0) ném
                  mới EmptyStackException ();
            E result = các phần tử [- size];
            phần tử [size] = null; // Loại bỏ kết quả trả về tham chiếu lỗi thời;


      } ... // không có thay đổi nào trong isEmpty hoặc ensureCapacity
}
```

Nói chung, bạn sẽ gặp ít nhất một lỗi hoặc cảnh báo, và lớp này không phải là ngoại lệ. May mắn thay, lớp này chỉ tạo ra một lỗi:

```
Stack.java:8: phần tử tạo mảng chung = new E
            [DEFAULT_INITIAL_CAPACITY];
                              ^
```

Như đã giải thích trong Mục 25, bạn không thể tạo một mảng thuộc kiểu không thể kết hợp lại, chẳng hạn như E. Vấn đề này phát sinh mỗi khi bạn viết một kiểu chung đư ợc hỗ trợ bởi một mảng. Có hai cách để giải quyết nó. Giải pháp đầu tiên trực tiếp phá vỡ lệnh cấm tạo mảng chung: tạo một mảng Đối tư ợng và truyền nó tới

Machine Translated by Google

kiểu mảng chung. Bây giờ thay vì lỗi, trình biên dịch sẽ phát ra cảnh báo.

Việc sử dụng này là hợp pháp, nhưng nó không phải là loại an toàn (nói chung):

```
Stack.java:8: warning: [unhecked] cast unhecked found: Object [],
bắt buộc: E [] element = (E []) new Object [DEFAULT_INITIAL_CAPACITY];
                                        ^
```

Trình biên dịch có thể không chứng minh được rằng chương trình của bạn là an toàn về kiểu chữ, nhưng bạn có thể. Bạn phải thuyết phục bản thân rằng dàn diễn viên không được chọn sẽ không ảnh hưởng đến sự an toàn về loại hình của chương trình. Mảng được đề cập (các phần tử) được lưu trữ trong một trường riêng tư và không bao giờ được trả lại cho máy khách hoặc được chuyển cho bất kỳ phương thức nào khác. Các phần mười m duy nhất được lưu trữ trong mảng là những phần được chuyển đến phương thức push , thuộc loại E, vì vậy việc ép kiểu không được kiểm tra có thể không gây hại gì.

Sau khi bạn đã chứng minh rằng việc truyền không được chọn là an toàn, hãy chặn cảnh báo trong phạm vi càng hẹp càng tốt (Mục 24). Trong trường hợp này, hàm tạo chỉ chứa phần tạo mảng chưa được kiểm tra, do đó, rất thích hợp để loại bỏ cảnh báo trong toàn bộ hàm tạo. Với việc bổ sung chú thích để thực hiện việc này, Stack biên dịch rõ ràng và bạn có thể sử dụng nó mà không cần phân loại rõ ràng hoặc sợ ClassCastException:

```
// Mảng phần tử sẽ chỉ chứa E thể hiện từ push (E).
// Điều này là đủ để đảm bảo an toàn cho kiểu, nhưng // kiểu thời gian chạy của
mảng sẽ không phải là E []; nó sẽ luôn là Đối tượng []!
@SuppressWarnings ("bỏ chọn") public
Stack () {element = (E []) new Object
     [DEFAULT_INITIAL_CAPACITY];
}
```

Cách thứ hai để loại bỏ lỗi tạo mảng chung trong Stack là thay đổi kiểu của các phần tử trường từ E [] thành Object []. Nếu bạn làm điều này, bạn sẽ gặp một lỗi khác:

```
Stack.java:19: tìm thấy các kiểu không
tương thích: Đối tượng, bắt buộc: E E
           result = element [- size];
                                    ^
```

Bạn có thể thay đổi lỗi này thành một cảnh báo bằng cách truyền phần tử được truy xuất từ mảng từ Đối tượng sang E:

```
Stack.java:19: warning: [unshecked] unhecked cast found found: Đối
tượng, bắt buộc: Kết quả EE = (E) phần tử [- size]; ^
```

Machine Translated by Google

Bởi vì E là kiểu không thể sử dụng lại, không có cách nào trình biên dịch có thể kiểm tra quá trình truyền trong thời gian chạy. Một lần nữa, bạn có thể dễ dàng chứng minh với bản thân rằng dàn diễn viên chư a đư ợc kiểm tra là an toàn, vì vậy, việc loại bỏ cảnh báo là phù hợp. Theo lời khuyên của Mục 24, chúng tôi chỉ loại bỏ cảnh báo trên bài tập có chứa diễn viên chư a đư ợc kiểm tra, không phải trên toàn bộ phư ơ ng thức bật :

```
// Loại bỏ thích hợp cảnh báo không đư ợc kiểm tra public E pop
() {if (size == 0) throw new EmptyStackException ();



      // push yêu cầu các phần tử phải thuộc loại E, vì vậy ép kiểu là chính xác
      @SuppressWarnings ("bỏ chọn") E result =
            (E) các phần tử [- size];

      phần tử [size] = null; // Loại bỏ kết quả trả về tham chiếu lỗi thời;

}
```

Kỹ thuật nào trong hai kỹ thuật bạn chọn để xử lý lỗi tạo mảng chung phần lớn là do sở thích. Tất cả những thứ khác đều bình đẳng, sẽ rủi ro hơ n nếu loại bỏ một kiểu ép kiểu không đư ợc kiểm tra đối với kiểu mảng hơ n là kiểu vô hư ớng, điều này sẽ gợi ý giải pháp thứ hai. Như ng trong một lớp chung chung thực tế hơ n Stack, bạn có thể sẽ đọc từ mảng ở nhiều điểm trong mã, vì vậy việc chọn giải pháp thứ hai sẽ yêu cầu nhiều lần truyền tới E hơ n là một lần truyền tới E [], đó là lý do tại sao giải pháp đầu tiên đư ợc sử dụng phổ biến hơ n [Naftalin07, 6.7].

Chư ơ ng trình sau đây trình bày việc sử dụng lớp Stack chung của chúng ta . Chư ơ ng trình in các đối số dòng lệnh của nó theo thứ tự ngư ợc lại và đư ợc chuyển đổi thành chữ hoa. Không cần ép kiểu rõ ràng để gọi phư ơ ng thức toUpperCase của String trên các phần tử xuất hiện từ ngăn xếp và quá trình ép kiểu đư ợc tạo tự động đư ợc đảm bảo sẽ thành công:

```
// Chư ơ ng trình nhỏ để thực hiện Stack chung của chúng ta
public static void main (String [] args) {
      Stack <Chuỗi> ngăn xếp = new Stack <Chuỗi> (); for
      (String arg: args) stack.push (arg); while (!
            stack.isEmpty ())

            System.out.println (stack.pop (). ToUpperCase ());
}
```

Machine Translated by Google

Ví dụ trên có thể mâu thuẫn với Mục 25, trong đó khuyến khích việc sử dụng danh sách ưu tiên cho mảng. Không phải lúc nào cũng có thể hoặc mong muốn sử dụng danh sách bên trong các loại chung của bạn. Java không hỗ trợ danh sách nguyên bản, vì vậy một số kiểu chung, chẳng hạn như ArrayList, phải được triển khai trên các mảng. Các kiểu chung khác, chẳng hạn như HashMap, được triển khai trên các mảng để đạt hiệu suất.

Phần lớn các kiểu chung chung giống như ví dụ về Stack của chúng tôi ở chỗ các tham số kiểu của chúng không có hạn chế: bạn có thể tạo Stack <Object>, Stack <int []>, Stack <List <String>> hoặc một Stack của bất kỳ loại nào khác kiểu tham chiếu đối tượng. Lưu ý rằng bạn không thể tạo một Ngăn xếp kiểu nguyên thủy: việc cố gắng tạo một Ngăn xếp <int> hoặc Ngăn xếp <đôi> sẽ dẫn đến lỗi thời gian biên dịch. Đây là một hạn chế thú vị của hệ thống kiểu chung của Java. Bạn có thể khắc phục hạn chế này bằng cách sử dụng các kiểu nguyên thủy đóng hộp (Mục 49).

Có một số kiểu chung hạn chế các giá trị cho phép của các tham số kiểu của chúng. Ví dụ: hãy xem xét java.util.concurrent.DelayQueue, có khai báo giống như sau:

lớp DelayQueue <E kéo dài Delayed> triển khai BlockingQueue <E>;

Danh sách tham số kiểu (<E expand Delayed>) yêu cầu tham số kiểu thực tế E phải là một kiểu con của java.util.concurrent.Delayed. Điều này cho phép triển khai DelayQueue và các máy khách của nó tận dụng các phương thức Delayed trên các phần tử của DelayQueue, mà không cần truyền rõ ràng hoặc rủi ro về ClassCastException. Tham số kiểu E được gọi là tham số kiểu bị giới hạn. Lưu ý rằng quan hệ kiểu con được định nghĩa sao cho mọi kiểu đều là kiểu con của chính nó [JLS, 4.10], vì vậy việc tạo DelayQueue <Đã hoãn> là hợp pháp.

Tóm lại, các kiểu chung an toàn hơn và dễ sử dụng hơn các kiểu yêu cầu phôi trong mã máy khách. Khi bạn thiết kế các loại mới, hãy đảm bảo rằng chúng có thể được sử dụng mà không cần phôi như vậy. Điều này thường có nghĩa là làm cho các loại chung chung. Tạo các loại hiện có của bạn khi thời gian cho phép. Điều này sẽ giúp cuộc sống của những người dùng mới thuộc loại này dễ dàng hơn mà không làm hỏng các khách hàng hiện tại (Mục 23).

Machine Translated by Google

Tiết 27: Phương pháp chung ủng hộ

Giống như các lớp có thể được hưởng lợi từ việc tổng quát hóa, các phương thức cũng vậy. Các meth ods tiện ích tĩnh là những ứng cử viên đặc biệt tốt cho quá trình tổng quát hóa. Tất cả các phương thức "thuật toán" trong Bộ sưu tập (chẳng hạn như Tìm kiếm nhị phân và sắp xếp) đã được tổng hợp hóa.

Cách viết generic tương tự như viết kiểu generic. Xem xét điều này phương thức, trả về kết hợp của hai tập hợp:

```
// Sử dụng kiểu thô - không thể chấp nhận được! (Mục 23)
public static Set union (Set s1, Set s2) {Set result = new
     HashSet (s1); result.addAll (s2); trả về kết quả;


}
```

Phương thức này biên dịch, nhưng với hai cảnh báo:

```
Union.java:5: warning: lệnh gọi [chưa được chọn] tới HashSet
(Collection <? Expand E>) như một thành viên của kiểu thô HashSet Set result = new
          HashSet (s1);
                          ^

Union.java:6: warning: lệnh gọi [chưa được chọn] đến addAll (Bộ sưu
tập <? Kéo dài E>) làm thành viên của kiểu thô Set
          result.addAll (s2);
                          ^
```

Để khắc phục những cảnh báo này và làm cho phương thức an toàn, hãy sửa đổi phương thức dec laration để khai báo một tham số kiểu đại diện cho kiểu phần tử cho ba tập hợp (hai đối số và giá trị trả về) và sử dụng tham số kiểu trong phương thức.
Danh sách tham số kiểu, khai báo tham số kiểu, nằm giữa các sửa đổi của phương thức và kiểu trả về của nó. Trong ví dụ này, danh sách tham số kiểu là <E> và kiểu trả về là Đặt <E>. Các quy ước đặt tên cho các tham số kiểu giống nhau đối với các phương thức chung như đối với các kiểu chung (Mục 26, 44):

```
// Phương thức chung
public static <E> Set <E> union (Set <E> s1, Set <E> s2) { Set <E>
     result = new HashSet <E> (s1); result.addAll (s2); trả về kết quả;


}
```

Ít nhất là đối với các phương pháp chung đơn giản, đó là tất cả những gì cần có. Giờ đây, phương thức này sẽ biên dịch mà không tạo ra bất kỳ cảnh báo nào và cung cấp sự an toàn cho kiểu cũng như dễ dàng

Machine Translated by Google

sử dụng. Đây là một chương trình đơn giản để thực hiện phương pháp của chúng tôi. Chương trình không chứa phôi và biên dịch không có lỗi hoặc cảnh báo:

```
// Chương trình đơn giản để thực hiện phương thức chung
public static void main (String [] args) {
      Đặt <String> guys = new HashSet <String> (
            Arrays.asList ("Tom", "Dick", "Harry")); Đặt
      <String> stoges = new HashSet <String> (
            Arrays.asList ("Larry", "Moe", "Curly"));
      Đặt <String> aflCio = union (guys, stoges);
      System.out.println (aflCio);
}
```

Khi bạn chạy chương trình, nó sẽ in [Moe, Harry, Tom, Curly, Larry, Dick].
Thứ tự của các phần tử phụ thuộc vào việc triển khai.

Một hạn chế của phương thức union là kiểu của cả ba bộ (cả tham số đầu vào và giá trị trả về) phải giống nhau. Bạn có thể làm cho phương pháp linh hoạt hơn bằng cách sử dụng các loại ký tự đại diện có giới hạn (Mục 28).

Một tính năng đáng chú ý của các phương thức chung là bạn không cần chỉ định giá trị của tham số kiểu một cách rõ ràng như bạn phải khi gọi các cấu trúc con chung. Trình biên dịch tính ra giá trị của các tham số kiểu bằng cách kiểm tra các kiểu của các đối số của phương thức. Trong trường hợp của chương trình trên, trình biên dịch thấy rằng cả hai đối số của union đều thuộc kiểu Set <String>, vì vậy nó biết rằng tham số kiểu E phải là String. Quá trình này được gọi là kiểu suy luận.

Như đã thảo luận trong Mục 1, bạn có thể khai thác kiểu suy luận được cung cấp bởi lệnh gọi phương thức chung để dễ dàng quá trình tạo các thể hiện kiểu được tham số hóa. Để làm mới bộ nhớ của bạn, nhu cầu truyền các giá trị của tham số kiểu một cách rõ ràng khi gọi các hàm tạo chung có thể gây khó chịu. Các tham số kiểu xuất hiện dư thừa ở phía bên trái và bên phải của khai báo biến:

```
// Tạo cá thể kiểu được tham số hóa với hàm tạo
Bản đồ <Chuỗi, Danh sách <Chuỗi>> đảo ngữ =
      new HashMap <Chuỗi, Danh sách <Chuỗi>> ();
```

Để loại bỏ sự dư thừa này, hãy viết một phương thức nhà máy tĩnh chung tương ứng với mỗi phương thức khởi tạo mà bạn muốn sử dụng. Ví dụ: đây là một phương thức nhà máy tĩnh chung tương ứng với phương thức khởi tạo HashMap không tham số :

```
// Phương thức nhà máy tĩnh chung public
static <K, V> HashMap <K, V> newHashMap () {return new HashMap
      <K, V> ();
}
```

Machine Translated by Google

Với phương pháp nhà máy tĩnh chung này, bạn có thể thay thế khai báo lặp đi lặp lại ở trên bằng một khai báo ngắn gọn sau:

```
// Tạo phiên bản kiểu được tham số hóa với nhà máy tĩnh
Bản đồ <Chuỗi, Danh sách <Chuỗi>> anagrams = newHashMap ();
```

Sẽ thật tuyệt nếu ngôn ngữ thực hiện cùng một kiểu suy luận khi gọi các hàm tạo trên các kiểu chung như khi gọi các phương thức chung.

Một ngày nào đó nó có thể, nhưng kể từ phiên bản 1.6, nó không.

Một mô hình liên quan là nhà máy singleton chung. Đôi khi, bạn sẽ cần tạo một đối tượng bất biến nhưng có thể áp dụng cho nhiều kiểu khác nhau.

Bởi vì generic được thực hiện bằng cách tẩy (Mục 25), bạn có thể sử dụng một đối tượng duy nhất cho tất cả các tham số kiểu bắt buộc, nhưng bạn cần viết một phương thức static factory để loại bỏ nhiều lần đối tượng cho từng tham số kiểu được yêu cầu. Mẫu này được sử dụng thường xuyên nhất cho các đối tượng hàm (Mục 21) như Collections.reverseOrder, nhưng nó cũng được sử dụng cho các bộ sưu tập như Collec tions.emptySet.

Giả sử bạn có một giao diện mô tả một hàm chấp nhận và trả về một giá trị thuộc kiểu T:

```
giao diện công cộng UnaryFunction <T> {
     T áp dụng (T arg);
}
```

Bây giờ, giả sử rằng bạn muốn cung cấp một chức năng nhận dạng. Sẽ rất lãng phí nếu tạo một cái mới mỗi khi nó được yêu cầu, vì nó không có trạng thái. Nếu các generic được sử dụng lại, bạn sẽ cần một hàm nhận dạng cho mỗi loại, nhưng vì chúng bị xóa nên bạn chỉ cần một hàm đơn chung chung. Đây là cách nó trông:

```
// Mẫu nhà máy sản xuất singleton chung private
static UnaryFunction <Object> IDENTITY_FUNCTION =
     new UnaryFunction <Object> () {
          public Đối tượng áp dụng (Object arg) {return arg; }
     };

// IDENTITY_FUNCTION là không trạng thái và tham số kiểu của nó là // không bị
ràng buộc nên có thể an toàn khi chia sẻ một phiên bản trên tất cả các loại.
@SuppressWarnings ("chưa được kiểm
tra") public static <T> UnaryFunction <T> IDFunction () {
     return (UnaryFunction <T>) IDENTITY_FUNCTION;
}
```

Machine Translated by Google

Việc ép kiểu IDENTITY_FUNCTION tới (UnaryFunction <T>) tạo ra một cảnh báo kết hợp không được kiểm tra, vì UnaryFunction <Object> không phải là một UnaryFunction <T> cho mọi T. Nhưng hàm nhận dạng là đặc biệt: nó trả về đối số không được sửa đổi, vì vậy chúng tôi biết rằng Sử dụng nó như một UnaryFunction <T> cho dù giá trị là T. Sau khi chúng tôi thực hiện xong việc này, mã sẽ biên dịch mà không có lỗi hoặc cảnh báo.

Đây là một chương trình mẫu sử dụng singleton chung của chúng ta như một hàm UnaryFunc <Chuỗi> và một UnaryFunction <Số>. Như thường lệ, nó không chứa phôi và biên dịch không có lỗi hoặc cảnh báo:

```
// Chương trình mẫu bài tập chung singleton public static
void main (String [] args) {String [] string = {"jute",
     "hemp", "nylon"}; UnaryFunction <String> sameString =
     IDFunction (); cho (Chuỗi s: chuỗi)

          System.out.println (sameString.apply (s));

     Number [] số = {1, 2.0, 3L}; UnaryFunction
     <Number> sameNumber = IDFunction (); cho (Số n: số)

          System.out.println (sameNumber.apply (n));
}
```

Được phép, mặc dùtương đối hiếm, khi một tham số kiểu được giới hạn bởi một số biểu thức liên quan đến chính tham số kiểu đó. Đây là những gì được gọi là một kiểu đệ quy bị ràng buộc. Việc sử dụng phổ biến nhất của giới hạn kiểu đệ quy là kết hợp với giao diện So sánh , xác định thứ tự tự nhiên của một kiểu:

```
giao diện công cộng Có thể so sánh được
     <T> {int so sánhTo (T o);
}
```

Tham số kiểu T xác định kiểu mà các phần tử của kiểu thực hiện Comp Compare <T> có thể được so sánh. Trong thực tế, gần như tất cả các loại chỉ có thể được so sánh với các phần tử của loại riêng của chúng. Vì vậy, ví dụ, chuỗi thực hiện Compa rable <Chuỗi>, Số nguyên thực hiện <Integer> có thể so sánh, v.v.

Có nhiều phương pháp lấy danh sách các phần tử triển khai Compara ble, để sắp xếp danh sách, tìm kiếm bên trong nó, tính toán tối thiểu hoặc tối đa của nó, và những thứ tương tự. Để thực hiện bất kỳ điều nào trong số này, yêu cầu mọi phần tử trong danh sách

Machine Translated by Google

có thể so sánh với mọi phần tử khác trong danh sách, nói cách khác, các phần tử của danh sách có thể so sánh lẫn nhau. Đây là cách thể hiện ràng buộc đó:

```
// Sử dụng kiểu đệ quy được ràng buộc để thể hiện khả năng so sánh lẫn nhau public
static <T kéo dài có thể so sánh được <T>> T max (List <T> list) {...}
```

Kiểu ràng buộc <T kéo dài Có thể so sánh được <T>> có thể được đọc là "đối với mọi kiểu T có thể được so sánh với chính nó," tương ứng ít nhiều chính xác với khái niệm so sánh lẫn nhau.

Đây là một phương pháp để đi với khai báo ở trên. Nó tính toán giá trị lớn nhất của một danh sách theo thứ tự tự nhiên của các phần tử của nó và nó biên dịch mà không có lỗi hoặc cảnh báo:

```
// Trả về giá trị lớn nhất trong danh sách - sử dụng kiểu đệ quy ràng buộc public
static <T kéo dài có thể so sánh được <T>> T max (Danh sách <T> danh sách) {
      Lặp lại <T> i = list.iterator (); Kết quả
      T = i.next (); while (i.hasNext ()) {T t =
      i.next (); if (t.compareTo (result)> 0)
            result = t;



      } trả về kết quả;
}
```

Các giới hạn kiểu đệ quy có thể phức tạp hơn nhiều so với điều này, nhưng may mắn là nó không xảy ra quá thường xuyên. Nếu bạn hiểu thành ngữ này và biến thể ký tự đại diện của nó (Mục 28), bạn sẽ có thể đối phó với nhiều giới hạn kiểu đệ quy mà bạn thấy trong thực tế.

Tóm lại, các phương thức chung, giống như các kiểu chung, an toàn hơn và dễ sử dụng hơn các phương thức yêu cầu máy khách của họ ép các tham số đầu vào và trả về giá trị.
Giống như các loại, bạn nên đảm bảo rằng các phương pháp mới của mình có thể được sử dụng mà không có phôi, điều này thường có nghĩa là làm cho chúng chung chung. Và giống như các loại, bạn nên tổng quát hóa các phương pháp hiện có của mình để giúp người dùng mới dễ dàng sử dụng hơn mà không làm hỏng các khách hàng hiện có (Mục 23).

Machine Translated by Google

Mục 28: Sử dụng các ký tự đại diện có giới hạn để tăng tính linh hoạt của API

Như đã lưu ý trong Mục 25, các kiểu tham số là bất biến. Nói cách khác, đối với bất kỳ hai kiểu riêng biệt nào là Type1 và Type2, List <Type1> không phải là kiểu con cũng không phải là siêu kiểu của List <Type2>. Mặc dù việc List <String> không phải là một kiểu con của List <Object> là phản trực giác , nhưng nó thực sự có ý nghĩa. Bạn có thể đặt bất kỳ đối tượng nào vào Danh sách <Đối tượng>, nhưng bạn chỉ có thể đặt các chuỗi vào Danh sách <Chuỗi>.

Đôi khi bạn cần sự linh hoạt hơn so với cách nhập bất biến có thể cung cấp. Consider ngăn xếp từ Mục 26. Để làm mới bộ nhớ của bạn, đây là API công khai của nó:

```
public class Stack <E> {public
      Stack (); công khai khoảng
      trống đẩy (E e); public E pop
      (); public boolean isEmpty ();

}
```

Giả sử chúng ta muốn thêm một phương thức có một chuỗi các phần tử và đẩy tất cả chúng lên ngăn xếp. Đây là lần thử đầu tiên:

```
// phương thức pushAll không có kiểu ký tự đại diện - thiếu! public
void pushAll (Iterable <E> src) {for (E e: src) push (e);


}
```

Phương pháp này biên dịch rõ ràng, nhưng nó không hoàn toàn thỏa đáng. Nếu loại phần tử của src Iterable khớp chính xác với loại phần tử của ngăn xếp, thì nó sẽ hoạt động tốt. Nhưng giả sử bạn có Stack <Number> và bạn gọi push (intVal), trong đó intVal thuộc kiểu Integer. Điều này hoạt động, vì Số nguyên là một kiểu con của Số. Về mặt logic, có vẻ như điều này cũng nên hoạt động:

```
Stack <Number> numberStack = new Stack <Number> (); Lặp lại
<Integer> Integers = numberStack.pushAll (số nguyên);
```

Tuy nhiên, nếu bạn thử nó, bạn sẽ nhận được thông báo lỗi này bởi vì, như đã lưu ý ở trên, các kiểu được tham số hóa là bất biến:

```
StackTest.java:7: pushAll (Lặp lại <Số>) trong Ngăn xếp <Số> không thể được áp
dụng cho (Lặp lại <Số>) numberStack.pushAll (số nguyên);
                                ^
```

Machine Translated by Google

May mắn thay, có một lối thoát. Ngôn ngữ này cung cấp một loại tham số đặc biệt gọi kiểu ký tự đại diện có giới hạn để đối phó với các tình huống như thế này. Loại tham số đầu vào cho pushAll không đư ợc là " Có thể lặp lại của E" mà là " Có thể lặp lại của một số kiểu con của E" và có một kiểu ký tự đại diện có nghĩa chính xác rằng: Iter could <? mở rộng E>. (Việc sử dụng từ khóa kéo dài hơ i gây hiểu lầm: hãy nhớ lại từ Mục 26 rằng kiểu con đư ợc định nghĩa để mọi kiểu là một kiểu con của chính nó, mặc dù nó không tự mở rộng.) Hãy sửa đổi pushAll để sử dụng kiểu này:

```
// Kiểu ký tự đại diện cho tham số phục vụ như một trình sản xuất E public
void pushAll (Iterable <? Expand E> src) {for (E e: src) push (e);


}
```

Với thay đổi này, không chỉ Stack biên dịch rõ ràng mà còn cả mã máy khách không biên dịch với khai báo pushAll ban đầu . Bởi vì Stack và ứng dụng khách của nó biên dịch sạch sẽ, bạn biết rằng mọi thứ đều an toàn.

Bây giờ, giả sử bạn muốn viết một phư ơ ng thức popAll để đi cùng với pushAll. Phư ơ ng thức popAll bật từng phần tử ra khỏi ngăn xếp và thêm các phần tử vào bộ sư u tập đã cho. Đây là cách nỗ lực đầu tiên để viết phư ơ ng thức popAll có thể trông như thế nào:

```
// phư ơ ng thức popAll không có kiểu ký tự đại diện - thiếu! public
void popAll (Collection <E> dst) {
     while (! isEmpty ())
          dst.add (pop ());
}
```

Một lần nữa, điều này biên dịch rõ ràng và hoạt động tốt nếu loại phần tử của tập hợp đích khớp chính xác với loại phần tử của ngăn xếp. Như ng một lần nữa, nó có vẻ không hoàn toàn thỏa đáng. Giả sử bạn có một Stack <Số> và biến kiểu Đối tư ợng. Nếu bạn bật một phần tử từ ngăn xếp và lư u trữ nó trong biến, nó sẽ biên dịch và chạy mà không có lỗi. Vì vậy, bạn cũng không thể làm điều này?

```
Stack <Number> numberStack = new Stack <Number> (); Collection
<Object> objects = numberStack.popAll (đối tư ợng);
```

Nếu bạn cố gắng biên dịch mã khách hàng này với phiên bản của popAll ở trên, bạn sẽ gặp lỗi tư ơ ng tự như lỗi mà chúng tôi gặp phải với phiên bản pushAll đầu tiên của chúng tôi: Bộ sư u tập <Đối tư ợng> không phải là một loại con của Bộ sư u tập <Số>. Một lần nữa, các loại ký tự đại diện cung cấp một lối thoát. Loại tham số đầu vào cho popAll

Machine Translated by Google

không nên là "tập hợp của E" mà là "tập hợp của một số siêu kiểu của E" (trong đó siêu kiểu được định nghĩa sao cho E là siêu kiểu của chính nó [JLS, 4.10]). Một lần nữa, có một loại ký tự đại diện có nghĩa chính xác rằng: Bộ sưu tập <? siêu E>. Hãy sửa đổi popAll để sử dụng nó:

```
// Kiểu ký tự đại diện cho tham số đóng vai trò là người tiêu dùng E public
void popAll (Collection <? Super E> dst) {while (! IsEmpty ()) dst.add (pop ());

}
```

Với thay đổi này, cả Stack và mã máy khách đều biên dịch rõ ràng.

Bài học rõ ràng. Để có tính linh hoạt tối đa, hãy sử dụng các loại ký tự đại diện trên các tham số đầu vào đại diện cho nhà sản xuất hoặc người tiêu dùng. Nếu một tham số đầu vào vừa là nhà sản xuất vừa là người tiêu dùng, thì các loại ký tự đại diện sẽ không tốt cho bạn: bạn cần một đối sánh loại chính xác, đó là những gì bạn nhận được mà không có bất kỳ ký tự đại diện nào.

Đây là cách ghi nhớ để giúp bạn nhớ loại ký tự đại diện nào sẽ sử dụng:

PECS là viết tắt của từ sản xuất mở rộng, siêu tiêu dùng.

Nói cách khác, nếu một kiểu tham số đại diện cho một trình sản xuất T , hãy sử dụng <? kéo dài T>; nếu nó đại diện cho một người tiêu dùng T , hãy sử dụng <? siêu T>. Trong ví dụ về Stack của chúng ta, tham số src của pushAll tạo ra các cá thể E để Stack sử dụng, vì vậy kiểu thích hợp cho src là Iterable <? mở rộng E>; Tham số dst của popAll sử dụng E thể hiện từ Ngăn xếp, vì vậy kiểu thích hợp cho dst là Bộ sưu tập <? siêu E>. PECS ghi nhớ nắm bắt nguyên tắc cơ bản hướng dẫn việc sử dụng các loại thẻ đại diện. Naftalin và Wadler gọi nó là Nguyên tắc Nhận và Đặt [Naftalin07, 2.4].

Với cách ghi nhớ này, chúng ta hãy xem một số khai báo phương thức từ các mục trước đó. Phương thức rút gọn trong Mục 25 có khai báo sau:

```
static <E> E giảm (Danh sách <E> danh sách, Hàm <E> f, E initVal)
```

Mặc dù danh sách có thể vừa tiêu thụ vừa tạo ra giá trị, nhưng phương thức Reduce chỉ sử dụng tham số danh sách của nó như một trình tạo E , do đó, khai báo của nó nên sử dụng kiểu ký tự đại diện mở rộng E. Tham số f đại diện cho một hàm vừa tiêu thụ vừa tạo ra các cá thể E , vì vậy a loại ký tự đại diện sẽ không thích hợp cho nó. Đây là khai báo phương thức kết quả:

```
// Kiểu ký tự đại diện cho tham số phục vụ như một trình tạo E tĩnh <E> E giảm
(Danh sách <? Expand E> danh sách, Hàm <E> f, E initVal)
```

Machine Translated by Google

Và liệu sự thay đổi này có tạo ra sự khác biệt nào trong thực tế không? Khi nó quay ra, nó sẽ. Giả sử bạn có một Danh sách <Con số> và bạn muốn giảm nó bằng một Hàm <Con số>. Điều này sẽ không biên dịch với khai báo ban đầu, như ng nó sẽ thực hiện khi bạn thêm loại ký tự đại diện đư ợc giới hạn.

Bây giờ chúng ta hãy xem xét phư ơ ng thức union từ Mục 27. Đây là khai báo:

```
public static <E> Set <E> union (Đặt <E> s1, Đặt <E> s2)
```

Cả hai tham số, s1 và s2, đều là các nhà sản xuất E , vì vậy phư ơ ng pháp ghi nhớ PECS cho chúng ta biết rằng khai báo phải là:

```
public static <E> Set <E> union (Đặt <? kéo dài E> s1, Đặt <? mở
                                                rộng E> s2)
```

Lư u ý rằng kiểu trả về vẫn là Đặt <E>. Không sử dụng các loại ký tự đại diện làm loại trả về. Thay vì cung cấp thêm tính linh hoạt cho ngư ời dùng của bạn, nó sẽ buộc họ sử dụng các loại ký tự đại diện trong mã khách hàng.

Đư ợc sử dụng đúng cách, các loại ký tự đại diện gần như vô hình đối với ngư ời dùng của một lớp. Chúng tạo ra các phư ơ ng thức chấp nhận các tham số mà chúng nên chấp nhận và từ chối những tham số mà chúng nên loại bỏ. Nếu ngư ời dùng của một lớp phải suy nghĩ về các loại ký tự đại diện, có thể có gì đó sai với API của lớp.

Thật không may, các quy tắc suy luận kiểu khá phức tạp. Chúng chiếm sáu trang dành cho thanh thiếu niên trong đặc tả ngôn ngữ [JLS, 15.12.2.7–8] và không phải lúc nào chúng cũng làm những gì bạn muốn. Nhìn vào tuyên bố sửa đổi cho union, bạn có thể nghĩ rằng bạn có thể làm điều này:

```
Đặt <Integer> integer = ...; Đặt
<Double> double = ...; Đặt <Số> số =
union
(số nguyên, nhân đôi);
```

Nếu bạn thử nó, bạn sẽ nhận đư ợc thông báo lỗi sau:

```
Union.java:14: Đã tìm thấy loại không tư ơ ng
thích: Đặt <Số & Có thể so sánh <? mở rộng Số & Có thể so sánh <?
                                                              >>>
bắt buộc: Đặt <Số>
          Đặt <Số> số = union (số nguyên, nhân đôi);
                                              ^
```

May mắn thay, có một cách để đối phó với loại lỗi này. Nếu trình biên dịch không suy ra loại mà bạn muốn nó có, bạn có thể cho nó biết loại nào để sử dụng với một

Machine Translated by Google

tham số kiểu. Đây không phải là điều mà bạn phải làm thường xuyên, đó là một điều tốt, vì các tham số kiểu rõ ràng không đẹp lắm. Với việc bổ sung này tham số kiểu rõ ràng, chương trình biên dịch rõ ràng:

```
Đặt <Number> number = Union. <Number> union (số nguyên, nhân đôi);
```

Tiếp theo, hãy chuyển sự chú ý của chúng ta đến phương thức max từ Mục 27. Đây là khai báo ban đầu:

```
public static <T kéo dài Có thể so sánh được <T>> T max (Danh sách <T> danh sách)
```

Đây là một khai báo đã sửa đổi sử dụng các loại ký tự đại diện:

```
public static <T kéo dài Có thể so sánh được <? siêu T >> T cực đại (
                Danh sách <? mở rộng T> danh sách)
```

Để có được bản kê khai đã sửa đổi từ bản gốc, chúng tôi áp dụng hình thức chuyển đổi PECS hai lần. Ứng dụng đơn giản là vào danh sách tham số. Nó ủng hộ các trường hợp T , vì vậy chúng tôi thay đổi kiểu từ Danh sách <T> thành Danh sách <? kéo dài T>.
Ứng dụng khó khăn là tham số kiểu T. Đây là lần đầu tiên chúng tôi thấy một ký tự đại diện được áp dụng cho một tham số kiểu. T ban đầu được chỉ định để mở rộng Compa rable <T>, nhưng một so sánh của T tiêu thụ các cá thể T (và tạo ra các số nguyên biểu thị quan hệ thứ tự). Do đó, kiểu tham số có thể so sánh được <T> là được thay thế bởi kiểu ký tự đại diện có giới hạn Có thể so sánh được <? siêu T>. Có thể so sánh được luôn luôn là người tiêu dùng, vì vậy bạn nên luôn sử dụng Có thể so sánh <? super T> in prefer ence <T> có thể so sánh được. Điều này cũng đúng với các trình so sánh, vì vậy bạn nên luôn sử dụng Bộ so sánh <? super T> ưu tiên cho Bộ so sánh <T>.

Khai báo max đã sửa đổi có lẽ là khai báo phương thức phức tạp nhất trong toàn bộ cuốn sách. Sự phức tạp thêm vào có thực sự mua được cho bạn điều gì không? Có nó làm. Đây là một ví dụ đơn giản về danh sách sẽ bị loại trừ bởi danh sách ban đầu khai báo phương thức được cung cấp bởi một trong những bản đồi.

```
Danh sách <SchedisedFuture <?>> SchedisedFutures = . . .
```

Lý do mà bạn không thể áp dụng khai báo phương pháp ban đầu cho danh sách này là rằng java.util.concurrent.SchedonedFuture không triển khai Compara ble <SchedonedFuture>. Thay vào đó, nó là một giao diện con của Delayed, mở rộng Có thể so sánh <Đã trả sau>. Nói cách khác, một phiên bản SchedisedFuture không chỉ đơn thuần là có thể so sánh với các phiên bản SchedisedFuture khác ; nó có thể so sánh với bất kỳ Delayed nào và điều đó đủ để khiến khai báo ban đầu từ chối nó.

Machine Translated by Google

Có một vấn đề nhỏ với khai báo đã sửa đổi cho max: nó ngăn phư ơ ng pháp biên dịch. Đây là phư ơ ng thức với khai báo đã sửa đổi:

```
// Sẽ không biên dịch - các ký tự đại diện có thể yêu cầu thay đổi trong thân phư ơ ng
thức! public static <T kéo dài Có thể so sánh đư ợc <? super T >> T max (List <?
          expand T> list) {Iterator <T> i = list.iterator (); Kết quả T = i.next
     (); while (i.hasNext ()) {T t = i.next (); if (t.compareTo (result)> 0) result
     = t;





     } trả về kết quả;
}
```

Đây là những gì sẽ xảy ra khi bạn cố gắng biên dịch nó:

```
Max.java:7: Đã tìm thấy các loại không
tư ơ ng thích: Iterator <capture # 591 of? mở rộng T> yêu cầu:
Iterator <T> Iterator <T> i = list.iterator ();

                                                  ^
```

Thông báo lỗi này có nghĩa là gì và làm cách nào để khắc phục sự cố? Nó có nghĩa là danh sách đó không phải là một List <T>, vì vậy phư ơ ng thức trình lặp của nó không trả về Iterator <T>. Nó trả về một trình vòng lặp của một số kiểu con của T, vì vậy chúng tôi thay thế khai báo trình vòng lặp bằng kiểu khai báo này, sử dụng kiểu ký tự đại diện có giới hạn:

```
Trình lặp <? mở rộng T> i = list.iterator ();
```

Đó là thay đổi duy nhất mà chúng tôi phải thực hiện đối với phần thân của phư ơ ng pháp. Ele ments đư ợc trả về bởi phư ơ ng thức tiếp theo của trình vòng lặp thuộc một số kiểu con của T, vì vậy chúng có thể đư ợc lư u trữ một cách an toàn trong một biến kiểu T.

Có một chủ đề khác liên quan đến ký tự đại diện mà các chú gấu đang thảo luận. Có sự đối ngẫu giữa các tham số kiểu và các ký tự đại diện, và nhiều phư ơ ng thức có thể đư ợc khai báo bằng cách sử dụng phư ơ ng thức này hoặc phư ơ ng thức kia. Ví dụ, đây là hai khai báo có thể có cho một phư ơ ng thức tĩnh để hoán đổi hai mục đư ợc lập chỉ mục trong một danh sách. Đầu tiên sử dụng tham số loại không bị ràng buộc (Mục 27) và thứ hai là ký tự đại diện không bị ràng buộc:

```
// Hai khai báo có thể có cho phư ơ ng thức hoán đổi public
static <E> void swap (List <E> list, int i, int j); public static void swap
(List <?> list, int i, int j);
```

Machine Translated by Google

Tuyên bố nào trong hai khai báo này là thích hợp hơn, và tại sao? Trong một API công khai, thứ hai tốt hơn vì nó đơn giản hơn. Bạn chuyển vào một danh sách — bất kỳ danh sách nào — và phương thức hoán đổi các phần tử được lập chỉ mục. Không có tham số kiểu nào để lo lắng. Theo quy tắc, nếu một tham số kiểu chỉ xuất hiện một lần trong khai báo phương thức, hãy thay thế nó bằng một ký tự đại diện. Nếu đó là một tham số kiểu không bị ràng buộc, hãy thay thế nó bằng một ký tự đại diện không bị ràng buộc; nếu đó là một tham số kiểu giới hạn, hãy thay thế nó bằng một ký tự đại diện bị giới hạn.

Có một vấn đề với khai báo thứ hai cho swap, sử dụng thẻ đại diện thay thế cho tham số kiểu: triển khai đơn giản sẽ không biên dịch:

```
public static void swap (List <?> list, int i, int j)
    {list.set (i, list.set (j, list.get (i)));
}
```

Cố gắng biên dịch nó sẽ tạo ra thông báo lỗi ít hữu ích này:

```
Swap.java:5: set (int, capture # 282 of?) Trong List <capture # 282 of?> Không thể
đương áp dụng cho (int, Object) list.set (i, list.set (j, list.get ( tôi)));
                                                                    ^
```

Có vẻ như không ổn khi chúng tôi không thể đưa một phần tử trở lại danh sách mà chúng tôi vừa lấy nó ra. Vấn đề là loại danh sách là Danh sách <?>, Và bạn không thể đặt bất kỳ giá trị nào ngoại trừ null vào Danh sách <?>. May mắn thay, có một cách để thực hiện phương pháp này mà không cần sử dụng đến loại nguyên liệu hoặc nguyên liệu không an toàn. Ý tưởng là viết một phương thức trợ giúp riêng để nắm bắt kiểu ký tự đại diện. Phương thức trợ giúp phải là một phương thức chung để nắm bắt kiểu. Đây là cách nó trông:

```
public static void swap (List <?> list, int i, int j)
    {swapHelper (list, i, j);
}

// Phương thức private helper để chụp ký tự đại diện
private static <E> void swapHelper (List <E> list, int i, int j) {list.set (i,
    list.set (j, list.get (i)));
}
```

Phương thức swapHelper biết rằng danh sách là một List <E>. Do đó, nó biết rằng bất kỳ giá trị nào mà nó đưa ra khỏi danh sách này đều thuộc loại E và sẽ an toàn khi đưa bất kỳ giá trị nào của loại E vào danh sách. Việc triển khai các biên dịch hoán đổi hơi phức tạp này một cách rõ ràng. Nó cho phép chúng tôi xuất khai báo hoán đổi dựa trên ký tự đại diện, đồng thời tận dụng lợi thế của phương pháp chung phức tạp hơn trong nội bộ. Khách hàng của

Machine Translated by Google

phư ơ ng thức hoán đổi không cần phải đối mặt với khai báo swapHelper phức tạp hơ n , như ng họ đư ợc lợi từ nó

Tóm lại, việc sử dụng các loại ký tự đại diện trong các API của bạn, mặc dù phức tạp, như ng khiến các API linh hoạt hơ n nhiều. Nếu bạn viết một thư viện sẽ đư ợc sử dụng rộng rãi, thì việc sử dụng hợp lý các loại ký tự đại diện nên đư ợc coi là bắt buộc. Hãy nhớ quy tắc cơ bản: pro ducer-expand, tiêu dùng-super (PECS). Và hãy nhớ rằng tất cả các sản phẩm có thể so sánh đư ợc và ngư ời so sánh là ngư ời tiêu dùng.

Machine Translated by Google

Mục 29: Xem xét các thùng chứa không đồng nhất an toàn cho các loại

Việc sử dụng phổ biến nhất của generic là cho các tập hợp, chẳng hạn như Tập hợp và Bản đồ, và các vùng chứa phần tử sin, chẳng hạn như ThreadLocal và AtomicReference. Trong tất cả các cách sử dụng này, nó là vùng chứa được tham số hóa. Điều này giới hạn bạn ở một số tham số loại cố định cho mỗi vùng chứa. Bình thường đó là chính xác những gì bạn muốn. Một Tập hợp có một tham số kiểu duy nhất, đại diện cho kiểu phần tử của nó; một Bản đồ có hai, đại diện gửi lại các loại khóa và giá trị của nó; và kể từ đó trở đi.

Tuy nhiên, đôi khi bạn cần linh hoạt hơn. Ví dụ, một hàng cơ sở dữ liệu có thể có nhiều cột tùy ý, và sẽ rất tuyệt nếu có thể truy cập tất cả chúng theo cách an toàn về kiểu chữ. May mắn thay, có một cách dễ dàng để đạt được hiệu ứng này. Ý tưởng là tham số hóa khóa thay vì vùng chứa. Sau đó, trình bày khóa được xác định tham số cho vùng chứa để chèn hoặc truy xuất một giá trị. Hệ thống loại chung được sử dụng để đảm bảo rằng loại giá trị đồng ý với khóa của nó.

Như một ví dụ đơn giản của cách tiếp cận này, hãy xem xét một lớp Yêu thích cho phép khách hàng của nó lưu trữ và truy xuất một thể hiện "yêu thích" của nhiều lớp khác tùy ý. Đối tượng Class sẽ đóng vai trò của khóa được tham số hóa. Lý do điều này hoạt động là vì lớp Class đã được tổng quát hóa trong bản phát hành 1.5. Kiểu của một lớp học thấp sáng eral không còn đơn giản là Lớp nữa, mà là Lớp <T>. Ví dụ, String.class thuộc loại Class <String> và Integer.class thuộc loại Class <Integer>. Khi một ký tự lớp được chuyển giữa các phương thức để truyền đạt cả thông tin về kiểu thời gian biên dịch và thời gian chạy, nó được gọi là mã thông báo kiểu [Bracha04].

API cho lớp Yêu thích rất đơn giản. Nó trông giống như một bản đồ đơn giản, ngoại trừ khóa được tham số hóa thay vì bản đồ. Máy khách trình bày một đối tượng Class khi thiết lập và nhận mục yêu thích. Đây là API:

```
// Mẫu vùng chứa không đồng nhất Typeafe - API public class
Favorites {public <T> void putFavorite (Class <T> type, T
     instance); public <T> T getFavorite (Loại <T> Class);

}
```

Đây là một chương trình mẫu thực hiện lớp Yêu thích , lưu trữ, truy xuất ing và in một cá thể Chuỗi, Số nguyên và Lớp yêu thích :

```
// Mẫu vùng chứa không đồng nhất Typeafe - client public static
void main (String [] args) {Favorites f = new Favorites ();
     f.putFavorite (String.class, "Java"); f.putFavorite
     (Integer.class, 0xcafebabe); f.putFavorite (Class.class,
     Favorites.class);
```

Machine Translated by Google


```
        String favouriteString = f.getFavorite (String.class); int
        favouriteInteger = f.getFavorite (Integer.class); Lớp <?>
        FavouriteClass = f.getFavorite (Lớp.class); System.out.printf ("%
        s% x% s% n", favouriteString, favouriteInteger,
            favouriteClass.getName ());
    }
```

Như bạn có thể mong đợi, chư ơ ng trình này in các mục Yêu thích Java cafebabe.

Một cá thể Ư a thích là an toàn kiểu chữ: nó sẽ không bao giờ trả về một Số nguyên khi bạn yêu cầu nó cho một Chuỗi. Nó cũng không đồng nhất: không giống như một bản đồ thông thư ờng, tất cả các chìa khóa đều thuộc các loại khác nhau. Do đó, chúng tôi gọi Mục ư a thích là một vùng chứa không đồng nhất an toàn kiểu loại.

Việc thực hiện Mục yêu thích là rất nhỏ một cách đáng ngạc nhiên. Toàn bộ đây rồi:

```
    // Mẫu vùng chứa không đồng nhất Typeafe - thực hiện lớp công khai Yêu thích
    {private Map <Class <?>, Object> favorites = new HashMap <Class <?>, Object>
        ();


        public <T> void putFavorite (Class <T> type, T instance) {
            if (type == null) ném
                NullPointerException mới ("Kiểu là null"); favorites.put (loại,
            trư ờng hợp);
        }

        public <T> T getFavorite (Class <T> type) {return
            type.cast (favorites.get (type));
        }
    }
```

Có một vài điều nhỏ nhặt đang diễn ra ở đây. Mỗi cá thể Mục ư a thích đư ợc hỗ trợ bởi một Bản đồ riêng tư <Lớp <?>, Đối tư ợng> đư ợc gọi là mục ư a thích. Bạn có thể nghĩ rằng bạn không thể đư a bất cứ thứ gì vào Bản đồ này vì loại ký tự đại diện không bị ràng buộc, như ng sự thật thì hoàn toàn ngư ợc lại. Điều cần chú ý là kiểu ký tự đại diện đư ợc lồng vào nhau: không phải loại Bản đồ là kiểu ký tự đại diện mà là kiểu khóa của nó. Điều này có nghĩa là mọi khóa đều có thể có một kiểu tham số hóa khác nhau: một có thể là Lớp <Chuỗi>, Lớp tiếp theo <Bộ ghi nhớ>, v.v. Đó là nơ i bắt nguồn của tính dị bản.

Điều tiếp theo cần chú ý là loại giá trị của Bản đồ yêu thích chỉ đơ n giản là Đối tư ợng. Nói cách khác, Bản đồ không đảm bảo mối quan hệ kiểu giữa khóa và giá trị, tức là mọi giá trị đều thuộc kiểu đư ợc đại diện bởi khóa của nó. Trên thực tế, hệ thống kiểu của Java không đủ mạnh để thể hiện điều này. Như ng chúng tôi biết đó là sự thật và chúng tôi tận dụng lợi thế của nó khi có thời gian để lấy một mục yêu thích.

Machine Translated by Google

Việc triển khai putFavorite là không đáng kể: nó chỉ đơn giản là đưa vào các mục ưa thích một ánh xạ từ đối tượng Class đã cho đến đối tượng ưa thích nhất định. Như đã lưu ý, điều này loại bỏ "liên kết kiểu" giữa khóa và giá trị; nó làm mất đi kiến thức rằng giá trị là một thể hiện của khóa. Nhưng điều đó không sao, vì phương thức getFavorites có thể và thực sự thiết lập lại liên kết này.

Việc triển khai phương thức getFavorite phức tạp hơn so với phương thức put Favourite. Đầu tiên, nó nhận được từ bản đồ yêu thích giá trị tương ứng với đối tượng Class đã cho . Đây là tham chiếu đối tượng chính xác để trả về, nhưng nó có kiểu thời gian biên dịch sai. Kiểu của nó chỉ đơn giản là Đối tượng (kiểu giá trị của bản đồ ưu tiên) và chúng ta cần trả về T. Vì vậy, việc triển khai getFavorite tự động chuyển đổi tượng tham chiếu đến kiểu được đại diện bởi đối tượng Lớp , sử dụng phương thức ép kiểu của Lớp .

Phương thức ép kiểu là tương tự động của toán tử ép kiểu của Java. Nó chỉ đơn giản là kiểm tra xem đối số của nó có phải là một thể hiện của kiểu được đại diện bởi đối tượng Class hay không . Nếu vậy, nó trả về đối số; nếu không, nó ném một ClassCastException. Chúng tôi biết rằng lệnh gọi ép kiểu trong getFavorite sẽ không bao giờ ném ClassCastExcep tion, giả sử mã máy khách được biên dịch sạch sẽ. Có nghĩa là, chúng ta biết rằng các giá trị trong bản đồ yêu thích luôn khớp với các loại khóa.

Vậy phương thức ép kiểu có tác dụng gì đối với chúng ta, khi nó chỉ đơn giản trả về phương thức lập luận của nó? Chữ ký của phương thức ép kiểu tận dụng tối đa thực tế là lớp Class đã được tổng quát hóa. Kiểu trả về của nó là tham số kiểu của đối tượng Lớp :

```
lớp công khai Lớp <T> {
    T cast (Đối tượng đối tượng);
}
```

Đây chính là thứ cần thiết của phương thức getFavorite . Đó là thứ cho phép chúng tôi tạo An toàn cho các loại Mục ưa thích mà không cần sử dụng đến việc chuyển thành T không được chọn .

Có hai hạn chế đối với lớp Yêu thích đáng lưu ý. Đầu tiên, một ứng dụng khách độc hại có thể dễ dàng làm hỏng sự an toàn kiểu của một cá thể Yêu thích , sim ply bằng cách sử dụng một đối tượng Lớp ở dạng thô của nó. Nhưng mã khách hàng kết quả sẽ tạo ra một cảnh báo không được kiểm tra khi nó được biên dịch. Điều này không khác gì các triển khai bộ sưu tập bình thường như HashSet và HashMap. Bạn có thể dễ dàng đặt một chuỗi vào một HashSet <Integer> bằng cách sử dụng kiểu thô HashSet (Mục 23). Điều đó nói rằng, bạn có thể có an toàn kiểu thời gian chạy nếu bạn sẵn sàng trả tiền cho nó. Cách để đảm bảo rằng Mục yêu thích không bao giờ vi phạm loại bất biến của nó là có

Machine Translated by Google

Phư ơ ng thức putFavorite hãy kiểm tra xem phiên bản đó có thực sự là một phiên bản của kiểu đại diện đư ợc gửi theo kiểu hay không. Và chúng tôi đã biết cách làm điều này. Chỉ cần sử dụng một dàn diễn viên động:

```
// Đạt đư ợc sự an toàn của kiểu thời gian chạy với kiểu truyền động
public <T> void putFavorite (Class <T> type, T instance) {favorites.put
     (type, type.cast (instance));
}
```

Có những trình bao bọc bộ sư u tập trong java.util.Collections chơ i cùng một mẹo. Chúng đư ợc gọi là checksSet, checkList, checkMap, v.v. Các nhà máy tĩnh của họ có một (hoặc hai) đối tư ợng Lớp ngoài một bộ sư u tập (hoặc bản đồ).
Các nhà máy tĩnh là các phư ơ ng thức chung, đảm bảo rằng các kiểu thời gian biên dịch của đối tư ợng Lớp và bộ sư u tập khớp với nhau. Các trình bao bọc thêm sự cải tiến cho các đư ờng dẫn col mà chúng bọc. Ví dụ: trình bao bọc ném một ClassCastException trong thời gian chạy nếu ai đó cố gắng đư a Coin vào Bộ sư u tập <Stamp> của bạn. Những ngư ời bọc này hữu ích để theo dõi xem ai thêm phần tử đư ợc nhập không chính xác vào mục col trong một ứng dụng kết hợp mã chung và mã kế thừa.

Hạn chế thứ hai của lớp Yêu thích là nó không thể đư ợc sử dụng cho loại không thể sử dụng lại (Mục 25). Nói cách khác, bạn có thể lư u trữ Chuỗi hoặc Chuỗi [] yêu thích của mình, như ng không lư u trữ Danh sách <Chuỗi> yêu thích của bạn. Nếu bạn cố gắng lư u trữ Danh sách <Chuỗi> yêu thích của mình, chư ơ ng trình của bạn sẽ không biên dịch. Lý do là bạn không thể lấy đối tư ợng Class cho List <String>: List <String> .class là một lỗi cú pháp và đó cũng là một điều tốt. List <String> và List <Integer> chia sẻ một đối tư ợng Class duy nhất , đó là List.class. Nó sẽ tàn phá nội bộ của một đối tư ợng Favorites nếu "type đen" List <String> .class và List <Integer> .class là hợp pháp và trả về cùng một tham chiếu đối tư ợng.

Không có cách giải quyết nào hoàn toàn thỏa đáng cho hạn chế thứ hai. Có một kỹ thuật đư ợc gọi là mã siêu loại đi một chặng đư ờng dài để giải quyết hạn chế, như ng kỹ thuật này có những hạn chế của riêng nó [Gafter07].

Loại mã thông báo đư ợc Yêu thích sử dụng là không bị ràng buộc: getFavorite và đặt Yêu thích chấp nhận bất kỳ đối tư ợng Lớp nào . Đôi khi bạn có thể cần phải giới hạn các loại có thể đư ợc chuyển cho một phư ơ ng thức. Điều này có thể đạt đư ợc với mã thông báo loại có giới hạn, đơ n giản là mã thông báo loại đặt một giới hạn vào loại có thể đư ợc biểu diễn, sử dụng tham số loại bị giới hạn (Mục 27) hoặc ký tự đại diện có giới hạn (Mục 28).

API chú thích (Mục 35) sử dụng rộng rãi các mã thông báo loại có giới hạn.
Ví dụ, đây là phư ơ ng pháp để đọc chú thích trong thời gian chạy. Phư ơ ng pháp này

Machine Translated by Google

đến từ giao diện AnnotatedElement , được thực hiện bởi các kiểu phản xạ đại diện cho các lớp, phương thức, trường và các phần tử chương trình khác:

```
public <T kéo dài chú thích>
     T getAnnotation (Loại <T> annotationType);
```

Đối số annotationType là một mã thông báo loại có giới hạn đại diện cho một loại annota tion. Phương thức trả về chú thích của phần tử thuộc kiểu đó, nếu nó có một hoặc null, nếu nó không có. Về bản chất, một phần tử được chú thích là một vùng chứa không đồng nhất an toàn về kiểu chữ có các khóa là các kiểu chú thích.

Giả sử bạn có một đối tượng kiểu Class <?> Và bạn muốn chuyển nó tới một phương thức yêu cầu mã thông báo kiểu bị giới hạn, chẳng hạn như getAnnotation. Bạn có thể truyền đối tượng đến Class <? mở rộng Annotation>, nhưng quá trình truyền này không được chọn, vì vậy nó sẽ tạo ra một cảnh báo thời gian biên dịch (Mục 24). May mắn thay, lớp Class cung cấp một phương thức thể hiện thực hiện kiểu ép kiểu này một cách an toàn (và động). Phương thức này được gọi là asSubclass, và nó truyền đối tượng Lớp mà nó được gọi để đại diện cho một lớp con của lớp được đại diện bởi đối số của nó. Nếu ép kiểu thành công, phương thức trả về đối số của nó; nếu nó không thành công, nó sẽ ném một ClassCastException.

Đây là cách bạn sử dụng phương thức asSubclass để đọc chú thích có kiểu không xác định tại thời điểm biên dịch. Phương pháp này biên dịch mà không có lỗi hoặc cảnh báo:

```
// Sử dụng asSubclass để truyền một cách an toàn tới mã thông báo loại bị giới
hạn static Annotation getAnnotation (phần tử AnnotatedElement,
                                          String annotationTypeName) { Class <?
     > AnnotationType = null; // Mã thông báo loại không bị giới hạn try
     {annotationType = Class.forName (annotationTypeName);

     } catch (Exception ex) {ném mới
          IllegalArgumentException (ex);

     } return element.getAnnotation
          ( annotationType.asSubclass (Annotation.class));
}
```

Tóm lại, việc sử dụng thông thường các generic, được minh họa bởi các API bộ sưu tập, hạn chế bạn ở một số lượng cố định của các tham số loại cho mỗi vùng chứa. Bạn có thể khắc phục hạn chế này bằng cách đặt tham số kiểu trên khóa thay vì vùng chứa. Bạn có thể sử dụng các đối tượng Lớp làm khóa cho các vùng chứa không đồng nhất an toàn kiểu như vậy. Một đối tượng Lớp được sử dụng theo kiểu này được gọi là mã thông báo kiểu. Bạn cũng có thể sử dụng loại khóa tùy chỉnh. Ví dụ, bạn có thể có kiểu DatabaseRow gửi một hàng cơ sở dữ liệu (vùng chứa) và một kiểu chung Cột <T> làm khóa của nó.

Machine Translated by Google

CHƯƠNG 6

Enums và chú thích

TRONG bản phát hành 1.5, hai họ kiểu tham chiếu đã được thêm vào ngôn ngữ: một mới loại lớp được gọi là kiểu enum và một loại giao diện mới được gọi là kiểu chú thích. Chương này thảo luận về các phương pháp hay nhất để sử dụng các họ kiểu mới này.

Mục 30: Sử dụng enum thay vì hằng số int

Loại liệt kê là loại có giá trị pháp lý bao gồm một tập hợp cố định của các dấu hiệu lừa đảo, chẳng hạn như các mùa trong năm, các hành tinh trong hệ mặt trời hoặc các bộ đồ trong một bộ bài chơi. Trước khi các kiểu enum được thêm vào ngôn ngữ, một mẫu common để đại diện cho các kiểu được liệt kê là khai báo một nhóm các hằng số int được đặt tên , một cho mỗi thành viên của kiểu:

```
// Mẫu int enum - thiếu nghiêm trọng! public static final
int APPLE_FUJI = 0; public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SMITH = 2;


public static final int ORANGE_NAVEL = 0; public
static final int ORANGE_TEMPLE = 1; public static
final int ORANGE_BLOOD = 2;
```

Kỹ thuật này, được gọi là int enum pattern, có nhiều thiếu sót. Nó không cung cấp gì về kiểu an toàn và ít về cách tiện lợi.
Trình biên dịch sẽ không phàn nàn nếu bạn chuyển một quả táo đến một phương thức mong đợi một quả cam, hãy so sánh quả táo với quả cam bằng toán tử == hoặc tệ hơn:

```
// Sốt táo có vị cam quýt ngon tuyệt! int i =
(APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

Machine Translated by Google

Lưu ý rằng tên của mỗi hằng số quả táo có tiền tố là APPLE_ và tên của mỗi hằng số màu cam có tiền tố là ORANGE_. Điều này là do Java không cung cấp không gian tên cho các nhóm int enum. Các tiền tố ngăn chặn xung đột tên khi hai nhóm int enum có các hằng số được đặt tên giống nhau.

Các chương trình sử dụng mẫu int enum rất dễ vỡ. Bởi vì int enums là hằng số thời gian biên dịch, chúng được biên dịch thành các máy khách sử dụng chúng. Nếu int được liên kết với một hằng số enum bị thay đổi, các máy khách của nó phải được biên dịch lại. Nếu không, chúng sẽ vẫn chạy, nhưng hành vi của chúng sẽ không xác định.

Không có cách nào dễ dàng để dịch hằng số int enum thành chuỗi có thể in được. Nếu bạn in một hằng số như vậy hoặc hiển thị nó từ trình gỡ lỗi, tất cả những gì bạn thấy là một con số, điều này không hữu ích lắm. Không có cách nào đáng tin cậy để lặp qua tất cả các hằng số int enum trong một nhóm, hoặc thậm chí để lấy kích thước của một nhóm int enum.

Bạn có thể gặp một biến thể của mẫu này, trong đó hằng số Chuỗi được sử dụng thay cho hằng số int . Biến thể này, được gọi là String enum pattern, thậm chí còn ít được mong đợi hơn. Mặc dù nó cung cấp các chuỗi có thể in được cho các hằng số của nó, nhưng nó có thể dẫn đến các vấn đề về hiệu suất vì nó dựa vào so sánh chuỗi. Tệ hơn nữa, nó có thể dẫn người dùng ngây thơ đến các hằng số chuỗi mã cứng thành mã máy khách thay vì sử dụng tên trường. Nếu hằng số chuỗi được mã hóa cứng như vậy chứa lỗi đánh máy, nó sẽ thoát khỏi sự phát hiện tại thời điểm biên dịch và dẫn đến lỗi trong thời gian chạy.

May mắn thay, kể từ phiên bản 1.5, ngôn ngữ này cung cấp một giải pháp thay thế tránh được những thiếu sót của các mẫu enum int và string , đồng thời cung cấp nhiều lợi ích bổ sung. Nó là (JLS, 8.9). Đây là cách nó trông ở dạng đơn giản nhất:

```
public enum Apple {FUJI, PIPPIN, GRANNY_SMITH}
public enum Orange {NAVEL, TEMPLE, BLOOD}
```

Bề ngoài, các kiểu enum này có thể giống với các kiểu của các ngôn ngữ khác, chẳng hạn như C, C ++ và C #, nhưng bề ngoài là lừa dối. Các kiểu enum của Java là các lớp chính thức, mạnh hơn nhiều so với các lớp tương tự của chúng trong các kiểu lan khác này, trong đó enum về cơ bản là các giá trị int .

Ý tưởng cơ bản đằng sau các kiểu enum của Java rất đơn giản: chúng là các lớp xuất một thể hiện cho mỗi hằng số thông qua một trường cuối cùng tĩnh công khai. Các kiểu enum có hiệu quả là cuối cùng, do không có hàm tạo nào có thể truy cập được. Vì các máy khách không thể tạo các trường hợp của kiểu enum cũng như không mở rộng nó, nên không thể có trường hợp nào ngoài các hằng số enum đã khai báo. Nói cách khác, các kiểu enum được kiểm soát bởi phiên bản (trang 6). Chúng là sự tổng quát của các hạt đơn (Mục 3), về cơ bản là các enum đơn nguyên tố. Đối với những độc giả quen thuộc với phần đầu tiên của cuốn sách này, các kiểu enum cung cấp hỗ trợ ngôn ngữ cho các loại enum pat tern typeafe [Bloch01, Item 21].

Machine Translated by Google

Enums cung cấp sự an toàn cho kiểu thời gian biên dịch. Nếu bạn khai báo một tham số thuộc loại Apple, bạn đư ợc đảm bảo rằng mọi tham chiếu đối tư ợng không rỗng đư ợc chuyển đến tham số là một trong ba giá trị Apple hợp lệ . Nỗ lực chuyển các giá trị không đúng kiểu sẽ dẫn đến lỗi thời gian biên dịch, cũng như cố gắng gán một sion expres của một kiểu enum cho một biến của kiểu enum khác hoặc sử dụng toán tử == để com pare các giá trị của các kiểu enum khác nhau.

Các kiểu enum với các hằng số đư ợc đặt tên giống nhau cùng tồn tại một cách hòa bình vì mỗi kiểu có không gian tên riêng. Bạn có thể thêm hoặc sắp xếp lại thứ tự các hằng số trong một kiểu enum mà không cần biên dịch lại các máy khách của nó vì các trư ờng xuất các hằng số cung cấp một lớp cách ly giữa một kiểu enum và các máy khách của nó: các giá trị hằng số không đư ợc biên dịch vào máy khách như chúng ở trong int enum mẫu. Cuối cùng, bạn có thể dịch enum thành các chuỗi có thể in đư ợc bằng cách gọi phư ơ ng thức toString của chúng .

Ngoài việc sửa chữa các thiếu sót của int enum, các kiểu enum cho phép bạn thêm các phư ơ ng thức và trư ờng tùy ý và triển khai các giao diện tùy ý. Chúng cung cấp các triển khai chất lư ợng cao của tất cả các phư ơ ng thức Đối tư ợng (Chư ơ ng 3), chúng bao hàm Có thể so sánh (Mục 12) và Có thể nối tiếp (Chư ơ ng 11), và dạng tuần tự của chúng đư ợc thiết kế để chịu đư ợc hầu hết các thay đổi đối với kiểu enum.

Vậy tại sao bạn lại muốn thêm các phư ơ ng thức hoặc trư ờng vào một kiểu enum? Để bắt đầu, bạn có thể muốn liên kết dữ liệu với các hằng số của nó. Ví dụ, các loại Táo và Cam của chúng tôi có thể đư ợc hư ởng lợi từ một phư ơ ng pháp trả về màu sắc của trái cây hoặc một phư ơ ng pháp trả về hình ảnh của nó. Bạn có thể tăng cư ờng một kiểu enum bằng bất kỳ phư ơ ng pháp nào có vẻ phù hợp. Một kiểu enum có thể bắt đầu cuộc sống như một tập hợp các hằng số enum đơ n giản và phát triển theo thời gian thành một dạng trừu tư ợng đầy đủ tính năng.

Để có một ví dụ hay về kiểu enum phong phú, hãy xem xét tám hành tinh trong hệ mặt trời của chúng ta. Mỗi hành tinh có khối lư ợng và bán kính, và từ hai thuộc tính này, bạn có thể tính toán lực hấp dẫn bề mặt của nó. Điều này cho phép bạn tính toán trọng lư ợng của một vật thể trên bề mặt hành tinh, dựa trên khối lư ợng của vật thể đó. Đây là cách enum này trông như thế nào. Các số trong ngoặc đơ n sau mỗi hằng số enum là các tham số đư ợc truyền cho hàm tạo của nó. Trong trư ờng hợp này, chúng là khối lư ợng và bán kính của hành tinh:

```
// Kiểu enum với dữ liệu và hành vi công khai
Hành tinh enum {MERCURY (3.302e + 23, 2.439e6),
      VENUS (4.869e + 24, 6.052e6), EARTH
      (5.975e + 24, 6.378e6), MARS (6.419e + 23,
      3.393e6), JUPITER (1.899e + 27, 7.149e7),
      SATURN (5.685e + 26, 6.027e7), URANUS
      (8.683e + 25, 2.556e7), NEPTUNE (1.024e +
      26, 2.477e7);
```

Machine Translated by Google


```
        khối lư ợng kép cuối cùng riêng tư ;              // Theo kilôgam bán
        kính kép cuối cùng riêng tư ;                     // Trong mét
        riêng cuối cùng double surfaceGravity; // Trong m / s ^ 2

        // Hằng số hấp dẫn chung tính bằng m ^ 3 / kg s ^ 2 riêng tĩnh cuối
        kép G = 6.67300E-11;

        // Constructor
        Planet (khối lư ợng gấp đôi, bán kính gấp đôi)
              {this.mass = mass; this.radius = bán kính;
              surfaceGravity = G * mass / (radius * bán
              kính);
        }

        public double mass ()                     {trả về khối lư ợng; }
        public double radius ()                   {bán kính trả về; }
        public double surfaceGravity () {return surfaceGravity; }

        public double surfaceWeight (double mass) {
              trả về khối lư ợng * surfaceGravity; // F = ma
        }
    }
```

Có thể dễ dàng viết một loại enum phong phú chẳng hạn như Planet. Để liên kết dữ liệu với hằng số enum, hãy khai báo các trư ờng thể hiện và viết một hàm tạo lấy dữ liệu và lư u trữ trong các trư ờng. Enums về bản chất là bất biến, vì vậy tất cả các trư ờng phải là trư ờng cuối cùng (Mục 15). Chúng có thể ở chế độ công khai, như ng tốt hơ n là đặt chúng ở chế độ riêng tư và cung cấp ngư ời truy cập công khai (Mục 14). Trong trư ờng hợp của Planet, nhà xây dựng cũng tính toán và lư u trữ trọng lực bề mặt, như ng đây chỉ là một sự tối ư u hóa. Nó có thể đư ợc tính toán lại từ khối lư ợng và bán kính mỗi khi nó đư ợc sử dụng bằng phư ơ ng pháp surfaceWeight , phư ơ ng pháp này lấy khối lư ợng của một vật thể và trả về trọng lư ợng của nó trên hành tinh đư ợc biểu thị bằng hằng số.

Mặc dù Planet enum rất đơ n giản, như ng nó lại có sức mạnh đáng kinh ngạc. Đây là một chư ơ ng trình ngắn lấy trọng lư ợng trái đất của một vật thể (trong một đơ n vị bất kỳ) và in ra một bảng đẹp về trọng lư ợng của vật thể trên tất cả tám hành tinh (trong cùng một đơ n vị):

```
    public class WeightTable {public
        static void main (String [] args) {double earthWeight
            = Double.parseDouble (args [0]); khối lư ợng kép = earthWeight /
            Planet.EARTH.surfaceGravity (); cho (Hành tinh p: Planet.values ())

                  System.out.printf ("Trọng lư ợng trên% s là% f% n",
                                          p, p.surfaceWeight (khối lư ợng));
        }
    }
```

Machine Translated by Google

Lư u ý rằng Planet, giống như tất cả các enum, có một phư ơ ng thức giá trị tĩnh trả về một mảng các giá trị của nó theo thứ tự chúng đư ợc khai báo. Cũng lư u ý rằng phư ơ ng thức toString trả về tên đã khai báo của mỗi giá trị enum, cho phép in dễ dàng bằng println và printf. Nếu không hài lòng với cách biểu diễn chuỗi này, bạn có thể thay đổi nó bằng cách ghi đè phư ơ ng thức toString . Đây là kết quả của việc chạy chư ơ ng trình WeightTable nhỏ của chúng tôi với đối số dòng lệnh 175:

```
Cân nặng trên MERCURY là 66.133672
Cân nặng trên VENUS là 158.383926
Cân nặng trên EARTH là 175.000000
Cân nặng trên MARS là 66.430699
Cân nặng trên JUPITER là 442.693902
Cân nặng trên SATURN là 186.464970
Cân nặng trên URANUS là 158.349709
Cân nặng trên NEPTUNE là 198.846116
```

Nếu đây là lần đầu tiên bạn thấy phư ơ ng thức printf của Java hoạt động, hãy lư u ý rằng khác với C ở chỗ bạn sử dụng % n trong đó bạn sử dụng \ n trong C.

Một số hành vi đư ợc liên kết với hằng số enum có thể chỉ cần đư ợc sử dụng từ bên trong lớp hoặc gói mà enum đư ợc định nghĩa. Các hành vi như vậy đư ợc thực hiện tốt nhất dư ới dạng phư ơ ng pháp riêng tư hoặc gói-riêng tư . Mỗi hằng số sau đó xe đi cùng với nó một tập hợp ẩn các hành vi cho phép lớp hoặc gói kết nối enum phản ứng thích hợp khi đư ợc hiển thị với hằng số. Cũng như với các lớp khác, trừ khi bạn có lý do thuyết phục để hiển thị một phư ơ ng thức enum cho các máy khách của nó, hãy khai báo nó là private hoặc nếu cần, package-private (Mục 13).

Nếu một enum nói chung là hữu ích, nó phải là một lớp cấp cao nhất; nếu việc sử dụng nó gắn liền với một lớp cấp cao nhất cụ thể, thì nó phải là một lớp thành viên của lớp cấp cao nhất đó (Mục 22). Ví dụ, enum java.math.RoundsMode đại diện cho một chế độ làm tròn cho các phân số thập phân. Các chế độ làm tròn này đư ợc sử dụng bởi lớp BigDecimal , như ng chúng cung cấp tính trừu tư ợng hữu ích về cơ bản không bị ràng buộc với Big Decimal. Bằng cách biến RoundingMode trở thành enum cấp cao nhất, các nhà thiết kế thư viện khuyến khích bất kỳ lập trình viên nào cần chế độ làm tròn sử dụng lại enum này, giúp tăng tính nhất quán giữa các API.

Các kỹ thuật đư ợc trình bày trong ví dụ Planet là đủ cho hầu hết các loại enum, như ng đôi khi bạn cần nhiều hơ n thế. Có các dữ liệu khác nhau đư ợc liên kết với mỗi hằng số Hành tinh , như ng đôi khi bạn cần phải liên kết hành vi ent khác nhau về cơ bản với mỗi hằng số. Ví dụ: giả sử bạn đang viết một kiểu enum để biểu diễn các phép toán trên một máy tính bốn chức năng cơ bản và bạn muốn

Machine Translated by Google

để cung cấp một phư ơ ng pháp để thực hiện phép toán số học đư ợc đại diện bởi mỗi con số. Một cách để đạt đư ợc điều này là bật giá trị của enum:

```
// Kiểu enum tự chuyển sang giá trị của nó - Thao tác enum công khai có vấn đề
{
      CỘNG, TRỪ, LẦN, CHIA;

      // Thực hiện op số học đư ợc biểu diễn bằng hằng số double này áp dụng
      (double x, double y) {switch (this) {case PLUS: return x + y; case
            MINUS: return x - y; case TIMES: return x * y; case DIVIDE:
                  return x / y;




            } ném AssertionError mới ("op không xác định: " + cái này);
      }
}
```

Mã này hoạt động, như ng không đẹp lắm. Nó sẽ không biên dịch mà không có câu lệnh ném vì về mặt kỹ thuật, phần cuối của phư ơ ng thức có thể truy cập đư ợc, mặc dù nó sẽ không bao giờ đạt đư ợc [JLS, 14.2.1]. Tệ hơ n nữa, mã rất dễ vỡ. Nếu bạn thêm một hằng số enum mới như ng quên thêm một trư ờng hợp tư ơ ng ứng vào công tắc, thì enum sẽ vẫn biên dịch, như ng nó sẽ không thành công trong thời gian chạy khi bạn cố gắng áp dụng thao tác mới.

May mắn thay, có một cách tốt hơ n để liên kết một hành vi khác với mỗi hằng enum: khai báo một phư ơ ng thức áp dụng trừu tư ợng trong kiểu enum và ghi đè nó bằng một phư ơ ng thức cụ thể cho mỗi hằng số trong một nội dung lớp cụ thể. Các meth ods như vậy đư ợc gọi là triển khai phư ơ ng thức cụ thể không đổi:

```
// Kiểu enum với các triển khai phư ơ ng thức cụ thể không đổi public enum
Operation {
      CỘNG {double apply (double x, double y) {return x + y;}},
      MINUS {double apply (double x, double y) {return x - y;}},
      TIMES {double apply (double x, double y) {return x * y;}},
      DIVIDE {double apply (double x, double y) {return x / y;}};

      Áp dụng kép trừu tư ợng (kép x, kép y);
}
```

Nếu bạn thêm một hằng số mới vào phiên bản thứ hai của Thao tác , không chắc bạn sẽ quên cung cấp một phư ơ ng thức áp dụng , vì phư ơ ng thức này ngay sau mỗi khai báo hằng số. Trong trư ờng hợp không chắc bạn quên, trình biên dịch sẽ nhắc bạn, vì các phư ơ ng thức trừu tư ợng trong kiểu enum phải đư ợc ghi đè bằng các phư ơ ng thức cụ thể trong tất cả các hằng của nó.

Machine Translated by Google

Việc triển khai phư ơ ng pháp cụ thể không đổi có thể đư ợc kết hợp với dữ liệu cụ thể không đổi. Ví dụ: đây là một phiên bản của Thao tác ghi đè phư ơ ng thức toString để trả về biểu tư ợng thư ờng đư ợc kết hợp với thao tác:

```
// Kiểu enum với các thân lớp cụ thể là hằng số và dữ liệu public enum
Operation {
      CỘNG ("+") {
            double apply (double x, double y) {return x + y; }
      },
      DẤU TRỪ("-") {
            double apply (double x, double y) {return x - y; }
      },
      TIMES ("*") {
            double apply (double x, double y) {return x * y; }
      },
      DIVIDE ("/")
            {double apply (double x, double y) {return x / y; }

      }; ký hiệu chuỗi cuối cùng riêng tư ;
      Hoạt động (Biểu tư ợng chuỗi) {this.symbol = biểu tư ợng; }
      @Override public String toString () {ký hiệu trả về; }

      Áp dụng kép trừu tư ợng (kép x, kép y);
}
```

Trong một số trư ờng hợp, ghi đè toString trong một enum rất hữu ích. Ví dụ: việc triển khai toString ở trên giúp dễ dàng in các biểu thức số học, như đư ợc trình bày trong chư ơ ng trình nhỏ này:

```
public static void main (String [] args) {
      double x = Double.parseDouble (args [0]); double y
      = Double.parseDouble (args [1]); for (Hoạt động op:
      Operation.values ())
            System.out.printf ("% f% s% f =% f% n", x, op,
                                    y, op.apply (x, y));
}
```

Chạy chư ơ ng trình này với 2 và 4 dư ới dạng đối số dòng lệnh sẽ tạo ra kết quả sau:

```
2,000000 + 4,000000 = 6,000000
2,000000 - 4,000000 = -2,000000 2,000000
* 4,000000 = 8,000000 2,000000 / 4,000000
= 0,500000
```

Machine Translated by Google

Các kiểu enum có một phư ơ ng thức valueOf (String) đư ợc tạo tự động để dịch tên của một hằng số thành chính hằng số đó. Nếu bạn ghi đè phư ơ ng thức toString trong một kiểu enum, hãy xem xét việc viết phư ơ ng thức fromString để dịch biểu diễn chuỗi tùy chỉnh trở lại enum tư ơ ng ứng. Đoạn mã sau (với tên kiểu đư ợc thay đổi thích hợp) sẽ thực hiện thủ thuật cho bất kỳ enum nào, miễn là mỗi hằng có một biểu diễn chuỗi duy nhất:

```
// Triển khai phư ơ ng thức fromString trên kiểu enum private static
final Map <String, Operation> stringToEnum
     = new HashMap <Chuỗi, Thao tác> (); static {//
Khởi tạo bản đồ từ tên hằng thành hằng số enum
     for (Hoạt động op: giá trị ())
          stringToEnum.put (op.toString (), op);
}
// Trả về Hoạt động đối với chuỗi hoặc null nếu chuỗi không hợp lệ Hoạt động tĩnh
công khai fromString (Ký hiệu chuỗi) {return stringToEnum.get (ký hiệu);

}
```

Lư u ý rằng các hằng số Hoạt động đư ợc đư a vào bản đồ stringToEnum từ một khối tĩnh chạy sau khi các hằng số đã đư ợc tạo. Cố gắng đặt mỗi hằng số vào bản đồ từ hàm tạo của chính nó sẽ gây ra lỗi compila tion. Đây là một điều tốt, vì nó sẽ gây ra NullPointerException nếu nó hợp pháp. Các hàm tạo enum không đư ợc phép truy cập vào các trư ờng tĩnh của enum, ngoại trừ các trư ờng hằng số thời gian biên dịch. Hạn chế này là cần thiết vì các trư ờng tĩnh này vẫn chư a đư ợc khởi tạo khi các hàm tạo chạy.

Một như ợc điểm của việc triển khai phư ơ ng thức cụ thể cho hằng số là chúng khiến việc chia sẻ mã giữa các hằng enum trở nên khó khăn hơ n. Ví dụ, hãy xem xét một enum đại diện cho các ngày trong tuần trong gói bảng lư ơ ng. Enum này có một phư ơ ng pháp tính lư ơ ng của một công nhân cho ngày đó dựa trên mức lư ơ ng cơ bản của công nhân (mỗi giờ) và số giờ làm việc trong ngày đó. Vào năm ngày trong tuần, bất kỳ thời gian nào làm việc vư ợt quá ca làm việc bình thư ờng sẽ đư ợc trả tiền làm thêm giờ; vào hai ngày cuối tuần, tất cả công việc đều phát sinh tiền làm thêm giờ. Với câu lệnh switch , thật dễ dàng để thực hiện phép tính này bằng cách áp dụng nhiều nhãn chữ hoa và chữ thư ờng cho mỗi đoạn mã trong số hai đoạn mã. Để ngắn gọn, mã trong ví dụ này sử dụng double, như ng lư u ý rằng double không phải là kiểu dữ liệu thích hợp cho ứng dụng trả lư ơ ng (Mục 48):

```
// Enum chuyển đổi giá trị của nó để chia sẻ mã - enum PayrollDay có vấn đề {

     THỨ HAI THỨ BA THỨ TƯ  THỨ NĂM THỨ SÁU,
     THỨ BẢY CHỦ NHẬT;
     private static final int HOURS_PER_SHIFT = 8;
```

Machine Translated by Google


```
        trả gấp đôi (double hoursWorked, double payRate) {double
            basePay = hoursWorked * payRate;

            tăng ca gấp đôi Thanh toán;         // Tính tiền làm thêm giờ
            chuyển đổi (cái này) {
                trư ờng hợp THỨ BẢY: trư ờng hợp CHỦ NHẬT:
                    thời gian làm thêmPay = giờ làm việc * payRate / 2;
                default: // Các ngày trong tuần
                    làm thêm giờPay = giờ làm việc <= HOURS_PER_SHIFT?
                        0: (giờ Làm việc - HOURS_PER_SHIFT) * payRate / 2; nghỉ;

            }

            trả lại basePay + làm thêm giờPay;
        }
    }
```

Không thể phủ nhận mã này là ngắn gọn, như ng nó rất nguy hiểm nếu bảo trì theo từng khía cạnh. Giả sử bạn thêm một phần tử vào enum, có thể là một giá trị đặc biệt để gửi lại một ngày nghỉ, như ng quên thêm một trư ờng hợp tư ơ ng ứng vào câu lệnh switch . Chư ơ ng trình sẽ vẫn biên dịch, như ng phư ơ ng thức trả lư ơ ng sẽ âm thầm trả cho ngư ời lao động số tiền tư ơ ng tự cho một ngày nghỉ phép như một ngày trong tuần bình thư ờng.

Để thực hiện tính toán lư ơ ng một cách an toàn với đề cập đến phư ơ ng pháp cụ thể không đổi, bạn sẽ phải nhân bản tính toán lư ơ ng làm thêm giờ cho mỗi hằng số hoặc chuyển tính toán thành hai phư ơ ng pháp trợ giúp (một cho các ngày trong tuần và một cho các ngày cuối tuần) và gọi trình trợ giúp thích hợp phư ơ ng pháp từ mỗi hằng số. Một trong hai cách tiếp cận sẽ dẫn đến một lư ợng lớn mã soạn sẵn, làm giảm đáng kể khả năng đọc và tăng khả năng mắc lỗi.

Có thể giảm bảng boilerplate bằng cách thay thế phư ơ ng thức Trừu tư ợng làm thêm giờ trên PayrollDay bằng một phư ơ ng pháp cụ thể thực hiện định lư ợng calcu làm thêm giờ cho các ngày trong tuần. Sau đó, chỉ những ngày cuối tuần sẽ phải ghi đè phư ơ ng thức. Như ng điều này sẽ có như ợc điểm tư ơ ng tự như câu lệnh switch: nếu bạn thêm một ngày khác mà không ghi đè phư ơ ng thức shiftPay , bạn sẽ âm thầm kế thừa phép tính ngày trong tuần.

Điều bạn thực sự muốn là buộc phải chọn chiến lư ợc trả lư ơ ng làm thêm giờ mỗi khi bạn thêm hằng số enum. May mắn thay, có một cách tốt đẹp để đạt đư ợc điều này. Ý tư ởng là chuyển việc tính toán lư ơ ng làm thêm giờ vào một enum lồng nhau riêng và chuyển một ví dụ của enum chiến lư ợc này cho phư ơ ng thức khởi tạo của enum PayrollDay . Sau đó, enum PayrollDay ủy quyền tính toán tiền làm thêm giờ cho enum chiến lư ợc, loại bỏ sự cần thiết của câu lệnh chuyển đổi hoặc phư ơ ng thức cụ thể của hằng số

Machine Translated by Google

triển khai trong PayrollDay. Mặc dù mẫu này ít ngắn gọn hơ n câu lệnh switch, như ng nó an toàn và linh hoạt hơ n:

```
// Mẫu chiến lư ợc enum PayrollDay
{MONDAY (PayType.WEEKDAY), THỨ BA
     (PayType.WEEKDAY), THỨ TƯ  (PayType.WEEKDAY), THỨ NĂM
     (PayType.WEEKDAY), THỨ SÁU (PayType.WEEKDAY), THỨ 7 (THỨ 7) , CHỦ NHẬT
     (PayType.WEEKEND);


     riêng cuối cùng PayType payType;
     PayrollDay (PayType payType) {this.payType = payType; }

     trả gấp đôi (double hoursWorked, double payRate) {return
          payType.pay (giờ làm việc, payRate);
     }
     // Chiến lư ợc enum gõ private
     enum PayType {WEEKDAY {double
          hoursPay (double hours,
               double payRate) {return hours <= HOURS_PER_SHIFT? 0:

                       (giờ - HOURS_PER_SHIFT) * payRate / 2;
               }
          },
          WEEKEND
               {tăng ca gấp đôi

               }

          }; private static final int HOURS_PER_SHIFT = 8;

          trừu tư ợng làm thêm giờ gấp đôiPay (giờ gấp đôi, trả lư ơ ng gấp đôi);

          trả gấp đôi (double hoursWorked, double payRate) {double
               basePay = hoursWorked * payRate; trả lại basePay + làm
               thêm giờPay (giờ làm việc, trả lư ơ ng);
          }
     }
}
```

Nếu các câu lệnh switch trên enums không phải là một lựa chọn tốt để thực hiện hành vi cụ thể của con stant trên enums, thì chúng có ích gì? Công tắc trên enum rất tốt để tăng cư ờng các loại enum bên ngoài với hành vi cụ thể không đổi. Ví dụ: giả sử enum Operation không nằm trong tầm kiểm soát của bạn và bạn

Machine Translated by Google

ư ớc gì nó có một phư ơ ng thức thể hiện để trả về nghịch đảo của mỗi phép toán. Bạn có thể mô phỏng hiệu ứng bằng phư ơ ng thức tĩnh sau:

```
// Bật một enum để mô phỏng một phư ơ ng thức bị thiếu public
static Thao tác nghịch đảo (Thao tác op) {switch (op) {

          case PLUS: trả lại Operation.MINUS; case
          MINUS: return Operation.PLUS; case TIMES: trả
          về Operation.DIVIDE; case DIVIDE: trả về
          Operation.TIMES; default: ném AssertionError                    "
          mới ("op không xác định:                                          + op);
     }
}
```

Nói chung, Enums có thể so sánh về hiệu suất với hằng số int . Một như ợc điểm nhỏ về hiệu suất của enum so với hằng số int là có chi phí không gian và thời gian để tải và khởi tạo các kiểu enum. Ngoại trừ trên các thiết bị sử dụng nhiều nguồn lực, chẳng hạn như điện thoại di động và máy nư ớng bánh mì, điều này khó có thể gây chú ý trong thực tế.

Vì vậy, khi nào bạn nên sử dụng enums? Bất cứ lúc nào bạn cần một bộ hằng số cố định. Tất nhiên, điều này bao gồm "các kiểu liệt kê tự nhiên", chẳng hạn như các hành tinh, các ngày trong tuần và các quân cờ. Như ng nó cũng bao gồm các bộ khác mà bạn biết tất cả các giá trị có thể có tại thời điểm biên dịch, chẳng hạn như các lựa chọn trên menu, mã hoạt động và cờ dòng lệnh. Không nhất thiết phải cố định tập hợp các hằng số trong kiểu enum. Tính năng enum đư ợc thiết kế đặc biệt để cho phép sự phát triển tư ơ ng thích nhị phân của các loại enum.

Tóm lại, ư u điểm của kiểu enum so với hằng số int là rất hấp dẫn. Enums dễ đọc hơ n, an toàn hơ n và mạnh mẽ hơ n. Nhiều enums không yêu cầu hàm tạo hoặc thành viên rõ ràng, như ng nhiều ngư ời khác đư ợc hư ởng lợi từ việc liên kết dữ liệu với mỗi hằng số và cung cấp các phư ơ ng thức mà hành vi của chúng bị ảnh hư ởng bởi dữ liệu này. Ít enums hơ n đư ợc hư ởng lợi từ việc kết hợp nhiều hành vi với một phư ơ ng pháp duy nhất. Trong trư ờng hợp tư ơ ng đối hiếm gặp này, ư u tiên các phư ơ ng thức cụ thể không đổi hơ n các phư ơ ng pháp enums chuyển đổi các giá trị của riêng chúng. Hãy xem xét mô hình enum chiến lư ợc nếu nhiều hằng số enum có chung các hành vi.

Machine Translated by Google

Mục 31: Sử dụng trư ờng phiên bản thay vì thứ tự

Nhiều enum đư ợc liên kết tự nhiên với một giá trị int duy nhất . Tất cả các enum đều có một phư ơ ng thức thứ tự , phư ơ ng thức này trả về vị trí số của mỗi hằng số enum trong kiểu của nó. Bạn có thể bị cám dỗ để lấy một giá trị int đư ợc liên kết từ thứ tự:

```
// Lạm dụng thứ tự để lấy một giá trị đư ợc liên kết - KHÔNG LÀM VIỆC NÀY công khai
enum Ensemble {
      SOLO, DUET, TRIO, QUARTET, QUINTET,
      SEXTET, SEPTET, OCTET, NONET, DECTET;

      public int numberOfMusicians () { return ordinal () + 1; }
}
```

Trong khi enum này hoạt động, nó là một cơ n ác mộng bảo trì. Nếu các hằng số đư ợc sắp xếp lại, phư ơ ng thức numberOfMusicians sẽ bị phá vỡ. Nếu bạn muốn thêm một hằng số sec ond enum đư ợc liên kết với một giá trị int mà bạn đã sử dụng, thì bạn đã không gặp may. Ví dụ, có thể rất hay khi thêm một hằng số cho tứ tấu kép , chẳng hạn như một bộ tám, bao gồm tám nhạc sĩ, như ng không có cách nào để làm điều đó.

Ngoài ra, bạn không thể thêm một hằng số cho một giá trị int mà không thêm hằng số cho tất cả các giá trị int xen vào . Ví dụ: giả sử bạn muốn thêm một đại diện liên tục phản đối một bộ tứ ba, bao gồm mư ời hai nhạc sĩ. Không có thuật ngữ stan dard cho một nhóm nhạc bao gồm mư ời một nhạc sĩ, vì vậy bạn buộc phải thêm một hằng số giả cho giá trị int không đư ợc sử dụng (11). Tốt nhất, điều này là xấu xí. Nếu nhiều giá trị int không đư ợc sử dụng, điều đó là không thực tế.

May mắn thay, có một giải pháp đơ n giản cho những vấn đề này. Không bao giờ lấy một giá trị đư ợc liên kết với một enum từ thứ tự của nó; thay vào đó lư u trữ nó trong một trư ờng cá thể:

```
public enum Ensemble {
      SOLO (1), DUET (2), TRIO (3), QUARTET (4), QUINTET (5),
      SEXTET (6), SEPTET (7), OCTET (8), DOUBLE_QUARTET (8),
      NONET (9), DECTET (10), TRIPLE_QUARTET (12);

      private cuối cùng int numberOfMusicians;
      Ensemble (int size) {this.numberOfMusicians = size; } public int
      numberOfMusicians () {return numberOfMusicians; }
}
```

Đặc tả Enum có điều này để nói về thứ tự: “Hầu hết các lập trình viên sẽ không sử dụng phư ơ ng pháp này. Nó đư ợc thiết kế để sử dụng bởi các cấu trúc dữ liệu dựa trên enum có mục đích chung như EnumSet và EnumMap. ” Trừ khi bạn đang viết một cấu trúc dữ liệu như vậy, bạn tốt nhất nên tránh hoàn toàn phư ơ ng pháp thứ tự .

Machine Translated by Google

Mục 32: Sử dụng EnumSet thay vì các trư ờng bit

Nếu các phần tử của một kiểu liệt kê đư ợc sử dụng chủ yếu trong các tập hợp, thì theo truyền thống là sử dụng mẫu int enum (Mục 30), gán sức mạnh khác nhau của 2 cho mỗi mục tiêu:

```
// Hằng số liệt kê trư ờng bit - OBSOLETE!
lớp công khai Văn bản {
      public static final int STYLE_BOLD = 1 << 0; // 1
      public static final int STYLE_ITALIC = 1 << 1; // 2
      public static final int STYLE_UNDERLINE = 1 << 2; // 4
      public static final int STYLE_STRIKETHROUGH = 1 << 3; // số 8

      // Tham số là bitwise HOẶC không hoặc nhiều hằng số STYLE_
      public void applyStyles (int styles) {...}
}
```

Biểu diễn này cho phép bạn sử dụng phép toán bitwise OR để kết hợp một số stants thành một tập hợp, đư ợc gọi là trư ờng bit:

```
text.applyStyles (STYLE_BOLD | STYLE_ITALIC);
```

Biểu diễn trư ờng bit cũng cho phép bạn thực hiện các hoạt động tập hợp chẳng hạn như liên hợp và giao nhau hiệu quả bằng cách sử dụng số học bit. Như ng các trư ờng bit có tất cả các ư u điểm của hằng số int enum và hơ n thế nữa. Nó thậm chí còn khó giải thích hơ n một chút hơ n một hằng số int enum đơ n giản khi nó đư ợc in dư ới dạng số. Cũng đó không phải là cách dễ dàng để lặp qua tất cả các phần tử đư ợc biểu thị bằng trư ờng bit.

Một số lập trình viên sử dụng enum ư u tiên cho hằng số int vẫn bám vào để sử dụng các trư ờng bit khi chúng cần truyền xung quanh các tập hợp hằng số. Không có lý do để làm điều này; một giải pháp thay thế tốt hơ n tồn tại. Gói java.util cung cấp Lớp EnumSet để biểu diễn hiệu quả các tập giá trị đư ợc rút ra từ một enum duy nhất thể loại. Lớp này thực hiện giao diện Đặt , cung cấp tất cả sự phong phú, kiểu an toàn và khả năng tư ơ ng tác mà bạn nhận đư ợc với bất kỳ triển khai Set nào khác . Như ng liên tục, mỗi EnumSet đư ợc biểu diễn dư ới dạng một vector bit. Nếu kiểu enum bên dư ới có sáu mư ơ i bốn phần tử trở xuống – và hầu hết đều có – toàn bộ EnumSet đư ợc đại diện với một trư ờng dài (trang 7), vì vậy hiệu suất của nó có thể so sánh với hiệu suất của trư ờng bit. Các hoạt động hàng loạt, chẳng hạn như removeAll và giữ lạiAll, đư ợc thực hiện bằng cách sử dụng số học thông minh bit, giống như bạn làm thủ công cho các trư ờng bit. Như ng bạn đư ợc cách nhiệt khỏi sự xấu xí và dễ xảy ra lỗi của việc xoay bit thủ công: EnumSet thực hiện công việc khó khăn cho bạn.

Machine Translated by Google

Đây là cách ví dụ trước đó trông như thế nào khi được sửa đổi để sử dụng enum thay vì trường bit. Nó ngắn hơn, rõ ràng hơn và an toàn hơn:

```
// EnumSet - một sự thay thế hiện đại cho các trường bit public
class Text {public enum Style {BOLD, ITALIC, UNDERLINE,
      STRIKETHROUGH}

      // Bất kỳ Bộ nào cũng có thể được chuyển vào, nhưng EnumSet rõ ràng là tốt
      nhất cho public void applyStyles (Set <Style> styles) {...}
}
```

Đây là mã máy khách chuyển một phiên bản EnumSet cho phương thức applyStyles . EnumSet cung cấp một tập hợp phong phú các nhà máy tĩnh để dễ dàng tạo tập hợp, một trong số đó được minh họa trong đoạn mã này:

```
text.applyStyles (EnumSet.of (Style.BOLD, Style.ITALIC));
```

Lưu ý rằng phương thức applyStyles nhận Set <Style> chứ không phải là Enum Set <Kiểu>. Mặc dù có vẻ như tất cả các máy khách sẽ chuyển một EnumSet cho phương thức, nhưng thực tiễn tốt là chấp nhận kiểu giao diện chứ không phải kiểu triển khai. Điều này cho phép một ứng dụng khách bất thường có thể vượt qua trong một số triển khai Set khác và không có bất lợi nào để nói đến (trang 190).

Tóm lại, chỉ vì một kiểu liệt kê sẽ được sử dụng trong các tập hợp, không có lý do gì để biểu diễn nó bằng các trường bit. Lớp EnumSet kết hợp tính chặt chẽ và hiệu suất của các trường bit với tất cả các ưu điểm của các kiểu enum được mô tả trong Mục 30. Một nhược điểm thực sự của EnumSet là nó không thể tạo ra EnumSet bất biến, nhưng điều này có thể sẽ được ghi nhớ trong một bản phát hành sắp tới. Trong thời gian chờ đợi, bạn có thể kết hợp EnumSet với Collections.unmodifiableSet, nhưng tính ngắn gọn và hiệu suất sẽ bị ảnh hưởng.

Machine Translated by Google

Mục 33: Sử dụng EnumMap thay vì lập chỉ mục thứ tự

Đôi khi bạn có thể thấy mã sử dụng phư ơ ng thức thứ tự (Mục 31) để lập chỉ mục vào một mảng. Ví dụ: hãy xem xét lớp đơ n giản này có nghĩa là đại diện cho một loại thảo mộc culi nary:

```
public class Herb
    {public enum Type {ANNUAL, PERENNIAL, BIENNIAL}

    Tên chuỗi cuối cùng riêng tư ;
    loại hình cuối cùng riêng tư ;

    Herb (Tên chuỗi, Kiểu loại) {this.name
        = name; this.type = loại;

    }

    @Override public String toString ()
        {return name;
    }
}
```

Bây giờ, giả sử bạn có một loạt các loại thảo mộc đại diện cho các loại cây trong vư ờn và bạn muốn liệt kê các loại cây này đư ợc sắp xếp theo loại (hàng năm, lâu năm hoặc hai năm một lần). Để làm điều này, bạn xây dựng ba bộ, một bộ cho mỗi loại và lặp lại qua gar den, đặt mỗi loại thảo mộc vào bộ thích hợp. Một số lập trình viên sẽ làm điều này bằng cách đặt các tập hợp vào một mảng đư ợc lập chỉ mục theo thứ tự của kiểu:

```
// Sử dụng ordinal () để lập chỉ mục một mảng - ĐỪNG LÀM ĐIỀU NÀY!
Herb [] vư ờn =   ...

Đặt <Herb> [] herbByType = // Đư ợc lập chỉ mục bởi Herb.Type.ordinal ()
    (Set <Herb> []) new Set [Herb.Type.values (). Length];
for (int i = 0; i <herbByType.length; i ++)
    herbByType [i] = new HashSet <Herb> ();

for (Herb h: garden)
    herbByType [h.type.ordinal ()]. add (h);

// In kết quả cho (int i
= 0; i <herbByType.length; i ++) {System.out.printf ("% s:
    % s% n", Herb.Type.values () [i], herbByType [ tôi]);

}
```

Machine Translated by Google

Kỹ thuật này hoạt động, như ng nó đầy rẫy các vấn đề. Bởi vì các mảng không tư ơ ng thích với generic (Mục 25), chư ơ ng trình yêu cầu ép kiểu không đư ợc kiểm tra và sẽ không biên dịch rõ ràng. Bởi vì mảng không biết chỉ mục của nó thay đổi gì, bạn phải gắn nhãn đầu ra theo cách thủ công. Như ng vấn đề nghiêm trọng nhất với kỹ thuật này là khi bạn truy cập vào một mảng đư ợc lập chỉ mục bởi thứ tự của enum, bạn có trách nhiệm sử dụng đúng giá trị int ; int không cung cấp kiểu an toàn của enums. Nếu bạn sử dụng sai giá trị, chư ơ ng trình sẽ âm thầm làm điều sai hoặc — nếu bạn may mắn — ném ArrayIndexOutOfBoundsException.

May mắn thay, có một cách tốt hơ n nhiều để đạt đư ợc hiệu quả tư ơ ng tự. Mảng đang phục vụ hiệu quả như một bản đồ từ enum đến một giá trị, vì vậy bạn cũng có thể sử dụng Bản đồ. Cụ thể hơ n, có một triển khai Bản đồ rất nhanh đư ợc thiết kế để sử dụng với các khóa enum, đư ợc gọi là java.util.EnumMap. Đây là cách chư ơ ng trình trông như thế nào khi nó đư ợc viết lại để sử dụng EnumMap:

```
// Sử dụng EnumMap để liên kết dữ liệu với enum Map <Herb.Type, Set
<Herb>> herbByType = new EnumMap <Herb.Type, Set <Herb>> (Herb.Type.class);
      for (Herb.Type t: Herb.Type.values ())

      herbByType.put (t, new HashSet <Herb> ()); for (Herb h:
garden) các loại thảo mộcByType.get (h.type) .add (h);

System.out.println (herbByType);
```

Chư ơ ng trình này ngắn hơ n, rõ ràng hơ n, an toàn hơ n và có tốc độ tư ơ ng đư ơ ng với phiên bản gốc. Không có dàn diễn viên không an toàn; không cần phải gắn nhãn đầu ra theo cách thủ công, vì các khóa bản đồ là các khóa tự động biết cách tự dịch sang các chuỗi có thể in đư ợc; và không có khả năng xảy ra lỗi trong tính toán chỉ số mảng. Lý do EnumMap có tốc độ tư ơ ng đư ơ ng với một mảng đư ợc lập chỉ mục theo thứ tự là EnumMap sử dụng một mảng như vậy trong nội bộ. Như ng nó ẩn chi tiết triển khai này khỏi ngư ời lập trình, kết hợp sự phong phú và an toàn kiểu của một Bản đồ với tốc độ của một mảng. Lư u ý rằng phư ơ ng thức khởi tạo EnumMap nhận đối tư ợng Class của kiểu khóa: đây là mã thông báo kiểu bị giới hạn, cung cấp thông tin kiểu chung thời gian chạy (Mục 29).

Bạn có thể thấy một mảng các mảng đư ợc lập chỉ mục (hai lần!) Theo thứ tự đư ợc sử dụng để biểu diễn ánh xạ từ hai giá trị enum. Ví dụ: chư ơ ng trình này sử dụng một mảng như vậy để ánh xạ hai giai đoạn thành sự chuyển pha (lỏng sang rắn đang đóng băng, lỏng sang khí đang sôi, v.v.):

```
// Sử dụng ordinal () để lập chỉ mục mảng - ĐỪNG LÀM VIỆC NÀY! public enum Phase {SOLID,
LIQUID, GAS; public enum Transition {

            MELT, FREEZE, BOIL, CONDENSE, SUBLIME, TIỀN GỬI;
```

Machine Translated by Google


```
        // Các hàng đư ợc lập chỉ mục theo src-ordinal, cols theo dst-ordinal
        private static final Transition [] [] TRANSITIONS = {{null,
                                TAN CHẢY,        SUBLIME},
              {FREEZE, null,                     SÔI LÊN },
              {DEPOSIT, CONDENSE, null}
        };

        // Trả về quá trình chuyển pha từ pha này sang pha khác Chuyển đổi tĩnh
        công khai từ (Phase src, Phase dst) {return TRANSITIONS [src.ordinal ()]
              [dst.ordinal ()];
        }
    }
}
```

Chư ơ ng trình này hoạt động và thậm chí có thể trông thanh lịch, như ng vẻ bề ngoài có thể lừa dối. Giống như ví dụ về khu vư ờn thảo mộc đơ n giản hơ n ở trên, trình biên dịch không có cách nào biết đư ợc mối quan hệ giữa thứ tự và chỉ số mảng. Nếu bạn mắc lỗi trong bảng chuyển đổi hoặc quên cập nhật nó khi bạn sửa đổi kiểu enum Giai đoạn hoặc Giai đoạn.Transition , chư ơ ng trình của bạn sẽ không thành công trong thời gian chạy. Lỗi có thể xảy ra dư ới dạng ArrayIndexOutOfBoundsException, một NullPointerExcep tion hoặc (tệ hơ n) hành vi có lỗi im lặng. Và kích thư ớc của bảng là bậc hai về số giai đoạn, ngay cả khi số mục nhập không rỗng nhỏ hơ n.

Một lần nữa, bạn có thể làm tốt hơ n nhiều với EnumMap. Bởi vì mỗi quá trình chuyển pha đư ợc lập chỉ mục bởi một cặp enum pha, tốt nhất bạn nên biểu diễn mối quan hệ dư ới dạng bản đồ từ enum này (pha nguồn) đến bản đồ từ enum thứ hai (pha đích) đến kết quả (chuyển pha). Hai pha liên quan đến chuyển đổi pha đư ợc ghi lại tốt nhất bằng cách liên kết dữ liệu với enum chuyển pha, sau đó đư ợc sử dụng để khởi tạo EnumMap lồng nhau:

```
// Sử dụng EnumMap lồng nhau để liên kết dữ liệu với cặp enum public enum Phase
{
    CHẤT RẮN, CHẤT LỎNG, CHẤT KHÍ;

    public enum Transition {
        MELT (SOLID, LIQUID), FREEZE (CHẤT LỎNG, CHẤT RẮN),
        BOIL (LIQUID, GAS), CONDENSE (GAS, LIQUID),
        SUBLIME (SOLID, GAS), DEPOSIT (GAS, SOLID);

        Giai đoạn cuối cùng src;
        Giai đoạn cuối cùng dst;

        Chuyển tiếp (Giai đoạn src, Giai đoạn dst)
            {this.src = src; this.dst = dst;

        }
```

Machine Translated by Google


```
            // Khởi tạo bản đồ chuyển pha private static final
            Map <Phase, Map <Phase, Transition >> m = new EnumMap <Phase, Map <Phase,
               Transition >> (Phase.class); static {for (Phase p: Phase.values ()) m.put
            (p, new EnumMap <Phase, Transition> (Phase.class));


                for (Transition trans: Transition.values ()) m.get
                   (trans.src) .put (trans.dst, trans);
            }

            Chuyển đổi tĩnh công khai từ (Giai đoạn src, Giai đoạn dst) {
                trả về m.get (src) .get (dst);
            }
        }
    }
```

Mã để khởi tạo bản đồ chuyển pha có thể trông hơ i phức tạp như ng nó không quá tệ. Loại bản đồ là Bản đồ <Pha, Bản đồ <Pha, Chuyển tiếp >>, có nghĩa là "bản đồ từ (nguồn) pha bản đồ từ (đích) đến pha chuyển tiếp". Vòng lặp đầu tiên trong khối khởi tạo tĩnh khởi tạo bản đồ bên ngoài để kết hợp ba bản đồ trống bên trong. Vòng lặp thứ hai trong khối khởi tạo các bản đồ bên trong bằng cách sử dụng thông tin nguồn và đích đư ợc cung cấp bởi mỗi hằng số truyền trạng thái.

Bây giờ, giả sử bạn muốn thêm một pha mới vào hệ thống: plasma hoặc khí ion hóa. Chỉ có hai quá trình chuyển đổi liên quan đến giai đoạn này: ion hóa, đư a một chất khí thành plasma; và khử ion, đư a plasma thành khí. Để cập nhật chư ơ ng trình dựa trên mảng, bạn sẽ phải thêm một hằng số mới vào Phase và hai hằng số vào Phase.Transition, đồng thời thay thế mảng chín phần tử ban đầu bằng một phiên bản mư ời sáu phần tử mới. Nếu bạn thêm quá nhiều hoặc quá ít elem vào mảng, hoặc đặt một phần tử không theo thứ tự, bạn sẽ gặp may: pro gram sẽ biên dịch, như ng nó sẽ không thành công trong thời gian chạy. Để cập nhật phiên bản dựa trên EnumMap , tất cả những gì bạn phải làm là thêm PLASMA vào danh sách các pha và IONIZE (GAS, PLASMA) và DEIONIZE (PLASMA, GAS) vào danh sách các chuyển đổi pha. Máy chủ chuyên nghiệp đảm nhận mọi thứ khác và bạn hầu như không có cơ hội để mắc lỗi.

Bên trong, bản đồ của bản đồ đư ợc thực hiện dư ới dạng một mảng của mảng, vì vậy bạn phải trả ít chi phí về không gian hoặc thời gian để có thêm sự rõ ràng, an toàn và dễ bảo trì.

Tóm lại, hiếm khi sử dụng thứ tự để lập chỉ mục mảng: hãy sử dụng EnumMap để thay thế. Nếu mối quan hệ mà bạn đang đại diện là đa chiều, hãy sử dụng EnumMap <..., EnumMap <... >>. Đây là một trư ờng hợp đặc biệt của nguyên tắc chung mà các lập trình viên ứng dụng hiếm khi sử dụng Enum.ordinal (Mục 31).

Machine Translated by Google

Tiết 34: Mô phỏng enums có thể mở rộng với giao diện

Trong hầu hết các khía cạnh, các kiểu enum vư ợt trội hơ n kiểu enum an toàn đư ợc mô tả trong ấn bản đầu tiên của cuốn sách này [Bloch01]. Về mặt nó, ngoại trừ một vấn đề liên quan đến khả năng mở rộng, có thể theo mẫu ban đầu như ng không đư ợc cấu trúc ngôn ngữ hỗ trợ. Nói cách khác, bằng cách sử dụng mẫu, có thể có một kiểu liệt kê mở rộng một kiểu khác; sử dụng tính năng ngôn ngữ, nó không phải là. Đây không phải là tai nạn. Đối với hầu hết các phần, khả năng mở rộng của enums hóa ra là một ý tư ởng tồi. Thật khó hiểu khi các phần tử của một kiểu mở rộng là các thể hiện của kiểu cơ sở và không phải ngư ợc lại. Không có cách nào tốt để liệt kê tất cả các phần tử của kiểu cơ sở và phần mở rộng của nó. Cuối cùng, khả năng mở rộng sẽ làm phức tạp nhiều khía cạnh của thiết kế và triển khai.

Điều đó nói rằng, có ít nhất một trư ờng hợp sử dụng hấp dẫn cho các kiểu liệt kê có thể mở rộng, đó là mã hoạt động, còn đư ợc gọi là mã opcodes. Opcode là một loại enumer ated có các phần tử đại diện cho các hoạt động trên một số máy, chẳng hạn như loại Hoạt động trong Mục 30, đại diện cho các chức năng trên một máy tính đơ n giản. Đôi khi ngư ời dùng API cung cấp các hoạt động của riêng họ là điều mong muốn, mở rộng một cách hiệu quả tập hợp các hoạt động do API cung cấp.

May mắn thay, có một cách hay để đạt đư ợc hiệu ứng này bằng cách sử dụng các loại enum. Ý tư ởng cơ bản là tận dụng lợi thế của thực tế là các kiểu enum có thể triển khai các mặt liên tùy ý bằng cách xác định một giao diện cho kiểu opcode và một enum là cách triển khai chuẩn của giao diện. Ví dụ: đây là phiên bản có thể mở rộng của Loại hoạt động từ Mục 30:

```
// Enum có thể mở rộng đư ợc mô phỏng sử dụng giao diện
public interface Hoạt động {double apply (double x, double
     y);
}

public enum BasicOperation triển khai hoạt động {
     CỘNG ("+") {
          public double apply (double x, double y) {return x + y; }
     },
     DẤU TRỪ("-") {
          public double apply (double x, double y) {return x - y; }
     },
     TIMES ("*") {
          public double apply (double x, double y) {return x * y; }
     },
     DIVIDE ("/")
          {public double apply (double x, double y) {return x / y; }
     };
```

Machine Translated by Google


```
        ký hiệu chuỗi cuối cùng riêng tư ;
        BasicOperation (Ký hiệu chuỗi) {
            this.symbol = biểu tư ợng;
        }
        @Override public String toString () {ký hiệu
            trả về;
        }
    }
```

Mặc dù kiểu enum (BasicOperation) không thể mở rộng, như ng kiểu giao diện (Hoạt động) thì có, và nó là kiểu giao diện đư ợc sử dụng để biểu diễn các hoạt động trong API. Bạn có thể xác định một kiểu enum khác thực hiện giao diện này và sử dụng các thể hiện của kiểu mới này thay cho kiểu cơ sở. Ví dụ: giả sử bạn muốn xác định một phần mở rộng cho kiểu hoạt động ở trên, bao gồm phép toán số dư và phép toán phần dư . Tất cả những gì bạn phải làm là viết một kiểu enum triển khai giao diện Hoạt động :

```
    // Phần mở rộng giả lập enum
    public enum ExtendedOperation thực hiện Thao tác {EXP ("^") {public
        double apply (double x, double y) {return Math.pow (x, y);


            }
        },
        REMAINDER ("%")
            {public double apply (double x, double y) {return x%
                y;
            }
        };

        ký hiệu chuỗi cuối cùng riêng tư ;
        ExtendedOperation (Ký hiệu chuỗi)
            {this.symbol = symbol;
        }
        @Override public String toString () {ký hiệu
            trả về;
        }
    }
```

Bạn có thể sử dụng các thao tác mới của mình ở bất cứ đâu bạn có thể sử dụng các toán tử cơ bản, miễn là các API đư ợc viết để lấy kiểu giao diện (Thao tác), chứ không phải để triển khai (BasicOperation). Lư u ý rằng bạn không phải khai báo phư ơ ng thức áp dụng trừu tư ợng trong enum như khi bạn làm trong một enum không thể mở rộng với các triển khai phư ơ ng thức dành riêng cho cá thể (trang 152). Điều này là do phư ơ ng thức trừu tư ợng (áp dụng) là một thành viên của giao diện (Hoạt động).

Machine Translated by Google

Không chỉ có thể chuyển một trư ờng hợp duy nhất của "enum mở rộng" bất kỳ nơ i nào mà "enum cơ sở" đư ợc mong đợi; có thể chuyển toàn bộ kiểu enum mở rộng và sử dụng các phần tử của nó ngoài hoặc thay cho các phần tử của kiểu cơ sở. Ví dụ, đây là phiên bản của chư ơ ng trình thử nghiệm trên trang 153 thực hiện tất cả các phép toán mở rộng đư ợc định nghĩa ở trên:

```
public static void main (String [] args) {
      double x = Double.parseDouble (args [0]); double y
      = Double.parseDouble (args [1]); kiểm tra
      (ExtendedOperation.class, x, y);

} private static <T kéo dài Enum <T> & Operation> void test (
            Lớp <T> opSet, double x, double y) {for (Hoạt
      động op: opSet.getEnumConstants ())
            System.out.printf ("% f% s% f =% f% n", x, op,
                                    y, op.apply (x, y));
}
```

Lư u ý rằng ký tự lớp cho kiểu hoạt động mở rộng (ExtendedOpera tion.class) đư ợc chuyển từ chính sang kiểm tra để mô tả tập hợp các toán hạng mở rộng. Chữ của lớp đóng vai trò như một mã thông báo kiểu bị giới hạn (Mục 29). Khai báo phức tạp đư ợc thừa nhận cho tham số opSet ((<T kéo dài Enum <T> & Opera tion> Class <T>) đảm bảo rằng đối tư ợng Lớp đại diện cho cả enum và kiểu con của Thao tác , đây chính xác là những gì đư ợc yêu cầu lặp lại mư ời ments và thực hiện các hoạt động liên quan đến mỗi một.

Một giải pháp thay thế thứ hai là sử dụng Bộ sư u tập <? Mở rộng Hoạt động>, là một loại ký tự đại diện có giới hạn (Mục 28), làm loại cho tham số opSet :

```
public static void main (String [] args) {
      double x = Double.parseDouble (args [0]); double y
      = Double.parseDouble (args [1]); kiểm tra
      (Arrays.asList (ExtendedOperation.values ()), x, y);
}

kiểm tra void tĩnh riêng (Bộ sư u tập <? Mở rộng Hoạt động> opSet,
            double x, double y) {for
      (Hoạt động op: opSet)
            System.out.printf ("% f% s% f =% f% n", x, op,
                                    y, op.apply (x, y));
}
```

Mã kết quả ít phức tạp hơ n một chút và phư ơ ng pháp kiểm tra linh hoạt hơ n một chút: nó cho phép ngư ời gọi kết hợp các hoạt động từ nhiều triển khai

Machine Translated by Google

các loại. Mặt khác, bạn từ bỏ khả năng sử dụng EnumSet (Mục 32) và EnumMap (Mục 33) trên các hoạt động đư ợc chỉ định, vì vậy bạn có thể tốt hơ n với mã thông báo loại bị giới hạn trừ khi bạn cần sự linh hoạt để kết hợp các hoạt động của nhiều loại triển khai .

Cả hai chư ơ ng trình trên sẽ tạo ra kết quả này khi chạy với các đối số dòng lệnh 2 và 4:

```
4,000000 ^ 2,000000 = 16,000000
4,000000% 2,000000 = 0,000000
```

Một như ợc điểm nhỏ của việc sử dụng các giao diện để mô phỏng các enum có thể mở rộng là việc triển khai không thể đư ợc kế thừa từ kiểu enum này sang kiểu enum khác. Trong trư ờng hợp ví dụ Hoạt động của chúng tôi , logic để lư u trữ và truy xuất ký hiệu liên quan đến một hoạt động đư ợc sao chép trong BasicOperation và ExtendedOperation. Trong trư ờng hợp này không thành vấn đề vì rất ít mã bị trùng lặp. Nếu có số lư ợng lớn hơ n các chức năng đư ợc chia sẻ, bạn có thể đóng gói nó trong một lớp trợ giúp hoặc một phư ơ ng thức trợ giúp tĩnh để loại bỏ sự trùng lặp mã.

Tóm lại, mặc dù bạn không thể viết một kiểu enum có thể mở rộng, như ng bạn có thể mô phỏng nó bằng cách viết một giao diện để đi với một kiểu enum cơ bản làm ẩn giao diện. Điều này cho phép khách hàng viết enums của riêng họ để triển khai giao diện. Sau đó, các enum này có thể đư ợc sử dụng ở bất cứ nơ i nào mà loại enum cơ bản có thể đư ợc sử dụng, giả sử các API đư ợc viết theo giao diện.

Machine Translated by Google

Mục 35: Ưu tiên chú thích cho các mẫu đặt tên

Trước khi phát hành 1.5, người ta thường sử dụng các mẫu đặt tên để chỉ ra rằng một số phần tử chương trình yêu cầu được xử lý đặc biệt bởi một công cụ hoặc khuôn khổ. Đối với bài kiểm tra, khung kiểm tra JUnit ban đầu yêu cầu người dùng chỉ định các phương pháp kiểm tra bằng cách bắt đầu tên của họ bằng bài kiểm tra ký tự [Beck04]. Công nghệ nique này hoạt động, nhưng nó có một số nhược điểm lớn. Đầu tiên, lỗi đánh máy có thể dẫn đến lỗi im lặng. Ví dụ: giả sử bạn vô tình đặt tên cho một phương pháp thử nghiệm là tsetSafetyOverride thay vì testSafetyOverride. JUnit sẽ không phàn nàn, nhưng nó cũng sẽ không thực hiện kiểm tra, dẫn đến cảm giác an toàn sai.

Một nhược điểm thứ hai của các mẫu đặt tên là không có cách nào để đảm bảo rằng chúng chỉ được sử dụng trên các phần tử chương trình thích hợp. Ví dụ: giả sử bạn gọi một lớp testSafetyMechanisms với hy vọng rằng JUnit sẽ tự động kiểm tra tất cả các phương thức của nó, bất kể tên của chúng là gì. Một lần nữa, JUnit sẽ không phàn nàn, nhưng nó cũng sẽ không thực hiện các bài kiểm tra.

Nhược điểm thứ ba của các mẫu đặt tên là chúng không cung cấp cách tốt để liên kết các giá trị tham số với các phần tử chương trình. Ví dụ: giả sử bạn muốn hỗ trợ một hạng mục kiểm tra chỉ thành công khi nó đưa ra một ngoại lệ cụ thể. Loại ngoại lệ về cơ bản là một tham số của thử nghiệm. Bạn có thể mã hóa tên loại ngoại lệ thành tên phương pháp thử nghiệm bằng cách sử dụng một số mẫu đặt tên phức tạp, nhưng điều này sẽ xấu và dễ hỏng (Mục 50). Trình biên dịch sẽ không có cách nào biết để kiểm tra xem chuỗi được cho là đặt tên cho một ngoại lệ có thực sự hoạt động hay không. Nếu lớp được đặt tên không tồn tại hoặc không phải là ngoại lệ, bạn sẽ không phát hiện ra cho đến khi bạn cố gắng chạy thử nghiệm.

Chú thích [JLS, 9.7] giải quyết tất cả những vấn đề này một cách độc đáo. Giả sử bạn muốn xác định một loại chú thích để chỉ định các bài kiểm tra đơn giản được chạy tự động và không thành công nếu chúng đưa ra một ngoại lệ. Đây là cách một loại chú thích như vậy, có tên là Thử nghiệm, có thể trông như thế nào:

```
// Khai báo kiểu chú thích điểm đánh dấu import
java.lang.annotation. *;

/
 ** * Cho biết rằng phương thức chú thích là một phương pháp thử nghiệm.
 * Chỉ sử dụng trên các phương thức tĩnh không tham số.
 * /
@Retention (RetentionPolicy.RUNTIME)
@Target (ElementType.METHOD) công
khai @interface Kiểm tra {}
```

Machine Translated by Google

Khai báo cho kiểu chú thích Kiểm tra bản thân nó đư ợc chú thích bằng chú thích Reten tion và Target . Chú thích như vậy trên khai báo loại chú thích đư ợc gọi là chú thích meta. @Retention (RetentionPolicy.RUNTIME) meta-annotation chỉ ra rằng các chú thích Kiểm tra nên đư ợc giữ lại trong thời gian chạy. Nếu không có nó, các chú thích kiểm tra sẽ ẩn đối với công cụ kiểm tra. Siêu chú thích @Tar get (ElementType.METHOD) chỉ ra rằng chú thích Kiểm tra là chỉ hợp pháp trên khai báo phư ơ ng thức: nó không thể đư ợc áp dụng cho khai báo lớp, trư ờng khai báo hoặc các phần tử chư ơ ng trình khác.

Lư u ý nhận xét phía trên khai báo Chú thích thử nghiệm có nội dung: “Chỉ sử dụng trên các phư ơ ng thức tĩnh không tham số. ” Sẽ thật tuyệt nếu trình biên dịch có thể thực thi hạn chế này, như ng nó không thể. Có những giới hạn về mức độ kiểm tra lỗi mà com piler có thể làm cho bạn ngay cả với chú thích. Nếu bạn đặt chú thích Kiểm tra trên khai báo một phư ơ ng thức thể hiện hoặc một phư ơ ng thức có một hoặc nhiều tham số, chư ơ ng trình thử nghiệm sẽ vẫn biên dịch, để nó cho công cụ thử nghiệm để xử lý lỗi prob lem trong thời gian chạy.

Đây là cách chú thích Kiểm tra trông như thế nào trong thực tế. Nó đư ợc gọi là ký hiệu chú thích đánh dấu, bởi vì nó không có tham số mà chỉ đơ n giản là "đánh dấu" phần tử đư ợc chú thích. Nếu lập trình viên đã viết sai chính tả Kiểm tra hoặc áp dụng chú thích Kiểm tra cho một phần tử gam pro khác với khai báo phư ơ ng thức, chư ơ ng trình sẽ không biên dịch:

```
// Chư ơ ng trình chứa chú thích điểm đánh dấu
lớp công khai Mẫu {
      @Test public static void m1 () {} // Kiểm tra sẽ vư ợt qua
      public static void m2 () {}
      @Test public static void m3 () {throw new       // Kiểm tra không thành công
            RuntimeException ("Boom");
      }
      public static void m4 () {}
      @Test public void m5 () {} // SỬ DỤNG KHÔNG HỢP LỆ: phư ơ ng thức nonstatic
      public static void m6 () {}
      @Test public static void m7 () {                // Kiểm tra không thành công
            ném RuntimeException mới ("Crash");
      }
      public static void m8 () {}
}
```

Lớp Mẫu có tám phư ơ ng thức tĩnh, bốn trong số đó đư ợc chú thích là các bài kiểm tra. Hai trong số này, m3 và m7, ném ngoại lệ và hai, m1 và m5, không. Như ng mà một trong những phư ơ ng thức chú thích không ném ngoại lệ, m5, là một ví dụ , vì vậy nó không phải là cách sử dụng hợp lệ của chú thích. Tóm lại, Mẫu chứa bốn các bài kiểm tra: một bài đạt, hai bài không đạt và một bài không hợp lệ. Bốn phư ơ ng pháp không đư ợc chú thích bằng Chú thích kiểm tra sẽ bị công cụ kiểm tra bỏ qua.

Machine Translated by Google

Các chú thích Kiểm tra không có ảnh hưởng trực tiếp đến ngữ nghĩa của lớp Mẫu . Họ chỉ phục vụ để cung cấp thông tin cho các chương trình quan tâm sử dụng.
Nói chung, chú thích không bao giờ thay đổi ngữ nghĩa của mã chú thích, nhưng cho phép nó được xử lý đặc biệt bằng các công cụ như trình chạy thử nghiệm đơn giản này:

```
// Chương trình xử lý chú thích điểm đánh dấu import
java.lang.reflect. *;

public class RunTests
    {public static void main (String [] args) ném Exception {int
        tests = 0; int qua = 0; Lớp testClass = Class.forName
        (args [0]); for (Phương thức m: testClass.getDeclaredMethods
        ()) {if (m.isAnnotationPresent (Test.class)) {tests ++;
        thử {m.invoke (null); đã qua ++; } catch
            (InvocationTargetException wrapExc) {Throwable exc =
                wrapExc.getCause (); System.out.println (m +
                "không thành công:" + exc); } catch (Exception
                    exc) {System.out.println ("INVALID @Test:
                                                                  "
                                                                    + m);
                }
            }
        }
        System.out.printf ("Đạt:% d, Không thành công:% d% n",
                                đã qua, các bài kiểm tra - đã thông qua);
    }
}
```

Công cụ chạy thử nghiệm lấy một tên lớp đủ điều kiện trên dòng lệnh và chạy tất cả các phương thức được chú thích bởi Kiểm thử của lớp một cách phản ánh, bằng cách gọi Method.invoke. Phương thức isAnnotationPresent cho công cụ biết phương thức nào sẽ chạy. Nếu một phương pháp kiểm tra ném một ngoại lệ, thì phương thức phản chiếu sẽ bao bọc nó trong một InvocationTargetException. Công cụ bắt được ngoại lệ này và in một báo cáo không thành công có chứa ngoại lệ ban đầu được đưa ra bởi phương pháp thử nghiệm, được trích xuất từ InvocationTargetException bằng phương thức getCause .

Nếu một nỗ lực để gọi một phương thức kiểm tra bằng phản chiếu ném ra bất kỳ ngoại lệ nào ngoài InvocationTargetException, thì nó cho biết việc sử dụng không hợp lệ annota thử nghiệm mà không được bắt tại thời điểm biên dịch. Việc sử dụng như vậy bao gồm chú thích của một phương thức phiên bản, một phương thức có một hoặc nhiều tham số, hoặc một phương thức không thể truy cập được. Khối bắt thứ hai trong trình chạy thử nghiệm bắt các lỗi sử dụng Thử nghiệm này

Machine Translated by Google

và in một thông báo lỗi thích hợp. Đây là kết quả đư ợc in nếu Chạy Thử nghiệm đư ợc chạy trên Mẫu:

```
public static void Sample.m3 () failed: RuntimeException: Boom INVALID @Test:
public void Sample.m5 () public static void Sample.m7 () failed:
RuntimeException: Crash Passed: 1, Failed: 3
```

Bây giờ, chúng ta hãy thêm hỗ trợ cho các thử nghiệm chỉ thành công nếu họ đư a ra một ngoại lệ. Chúng tôi sẽ cần một loại chú thích mới cho điều này:

```
// Kiểu chú thích có tham số import
java.lang.annotation. *; / ** * Cho biết rằng
phư ơ ng thức chú thích là một phư ơ ng thức thử
  nghiệm mà * phải ném ngoại lệ đư ợc chỉ định để thành công. * /


@Retention (RetentionPolicy.RUNTIME)
@Target (ElementType.METHOD)
public @interface ExceptionTest {
     Lớp <? mở rộng Exception> value ();
}
```

Loại tham số cho chú thích này là Class <? mở rộng Excep tion>. Loại ký tự đại diện này, phải thừa nhận, là một cái miệng. Trong tiếng Anh, nó có nghĩa là " đối tư ợng Lớp cho một số lớp mở rộng Ngoại lệ," và nó cho phép ngư ời dùng chú thích chỉ định bất kỳ loại ngoại lệ nào. Cách sử dụng này là một ví dụ về mã thông báo loại bị giới hạn (Mục 29). Đây là cách chú thích trông như thế nào trong thực tế. Lư u ý rằng các ký tự lớp đư ợc sử dụng làm giá trị cho tham số chú thích:

```
// Chư ơ ng trình chứa chú thích với tham số public class Sample2
{
     @ExceptionTest (ArithmeticException.class) public static
     void m1 () {// Kiểm tra phải chuyển int i = 0; i = i / i;


     }
     @ExceptionTest (ArithmeticException.class) public
     static void m2 () {// Nên fail (sai ngoại lệ) int [] a = new int [0]; int i
          = a [1];

     }
     @ExceptionTest (ArithmeticException.class) public
     static void m3 () {} // Không thành công (không có ngoại lệ)
}
```

Machine Translated by Google

Bây giờ, hãy sửa đổi công cụ chạy thử nghiệm để xử lý chú thích mới. Làm như vậy bao gồm thêm mã sau vào phương thức chính :

```
if (m.isAnnotationPresent (ExceptionTest.class)) {tests ++; thử
      {m.invoke (null); System.out.printf ("Kiểm tra% s không
      thành công: không có ngoại lệ% n", m);


      } catch (InvocationTargetException wrapEx) {Throwable exc =
            wrapEx.getCause (); Lớp <? mở rộng Exception> excType =
            m.getAnnotation (ExceptionTest.class) .value (); if
                  (excType.isInstance (exc)) {qua ++; } else {System.out.printf
            ("Kiểm tra% s không thành công: dự kiến% s, đã nhận% s% n", m,
                  excType.getName (), exc);




            }
      } catch (Exception exc)                                  "
            {System.out.println ("INVALID @Test:                + m);
      }
}
```

Mã này tương tự như mã mà chúng tôi đã sử dụng để xử lý các chú thích Kiểm tra , với một ngoại lệ: mã này trích xuất giá trị của tham số chú thích và sử dụng nó để kiểm tra xem ngoại lệ do kiểm tra đưa ra có đúng loại hay không. Không có phôi rõ ràng, do đó không có nguy cơ xảy ra với ClassCastException. Chương trình thử nghiệm được biên dịch đảm bảo rằng các tham số chú thích của nó đại diện cho các loại ngoại lệ hợp lệ, với một lưu ý: có thể các tham số chú thích hợp lệ tại thời điểm biên dịch nhưng một tệp lớp đại diện cho một loại ngoại lệ được chỉ định không còn xuất hiện trong thời gian chạy.
Trong trường hợp hy vọng hiếm gặp này, trình chạy thử nghiệm sẽ ném TypeNotPresentException.

Lấy ví dụ thử nghiệm ngoại lệ của chúng tôi thêm một bước nữa, có thể có một thử nghiệm vượt qua nếu nó ném bất kỳ một trong số các ngoại lệ đã chỉ định. Cơ chế chú thích có một cơ sở giúp dễ dàng hỗ trợ cách sử dụng này. Sup đặt ra, chúng tôi thay đổi kiểu tham số của chú thích ExceptionTest thành một mảng các đối tượng Lớp :

```
// Loại chú thích có tham số mảng @Retention
(RetentionPolicy.RUNTIME)
@Target (ElementType.METHOD) public
@interface ExceptionTest {
      Lớp <? mở rộng Ngoại lệ> [] value ();
}
```

Machine Translated by Google

Cú pháp cho các tham số mảng trong chú thích rất linh hoạt. Nó đư ợc tối ư u hóa cho các mảng một phần tử. Tất cả các chú thích ExceptionTest trư ớc đó vẫn hợp lệ với phiên bản tham số mảng mới của ExceptionTest và kết quả là các mảng sin gle-element. Để chỉ định một mảng nhiều phần tử, hãy bao quanh các phần tử bằng dấu ngoặc nhọn và phân tách chúng bằng dấu phẩy:

```
// Mã chứa chú thích có tham số mảng @ExceptionTest
({ IndexOutOfBoundsException.class, NullPointerException.class }) public
                        static void doublyBad () {List <String> list = new
ArrayList <String> ();


     // Đặc tả cho phép phư ơ ng thức này ném //
     IndexOutOfBoundsException hoặc NullPointerException list.addAll (5,
     null);
}
```

Việc sửa đổi công cụ chạy thử nghiệm để xử lý phiên bản mới của ExceptionTest. Mã này thay thế phiên bản gốc:

```
if (m.isAnnotationPresent (ExceptionTest.class)) {tests ++; thử
     {m.invoke (null); System.out.printf ("Kiểm tra% s không
     thành công: không có ngoại lệ% n", m); } catch (Throwable
          wrapExc) {Throwable exc = wrapExc.getCause (); Lớp <?
          mở rộng Ngoại lệ> [] excTypes =



               m.getAnnotation (ExceptionTest.class) .value ();
          int oldPassed = đã qua; for
          (Class <? expand Exception> excType: excTypes) {if (excType.isInstance
               (exc)) {pass ++; nghỉ;


               }

          } if (đã qua == oldPassed)
               System.out.printf ("Kiểm tra% s không thành công:% s% n", m, exc);
     }
}
```

Khung thử nghiệm đư ợc phát triển trong mục này chỉ là một món đồ chơ i, như ng rõ ràng nó thể hiện tính ư u việt của chú thích so với các mẫu đặt tên. Và nó chỉ làm xư ớc bề mặt của những gì bạn có thể làm với chú thích. Nếu bạn viết một công cụ yêu cầu ngư ời lập trình thêm thông tin vào tệp nguồn, hãy xác định một tập hợp thích hợp

Machine Translated by Google

của các loại chú thích. Đơn giản là không có lý do gì để sử dụng các mẫu đặt tên bây giờ mà chúng tôi có chú thích.

Điều đó nói rằng, ngoại trừ thợ công cụ, hầu hết các lập trình viên sẽ không có cần xác định các loại chú thích. Tuy nhiên, tất cả các lập trình viên nên sử dụng các loại chú thích được xác định trước do nền tảng Java cung cấp (Mục 36 và Mục 24). Ngoài ra, hãy cân nhắc sử dụng bất kỳ chú thích nào do IDE của bạn cung cấp hoặc phân tích tĩnh công cụ. Những chú thích như vậy có thể cải thiện chất lượng của thông tin chẩn đoán được cung cấp bởi các công cụ này. Tuy nhiên, xin lưu ý rằng những chú thích này vẫn chưa được sắp xếp theo tiêu chuẩn, vì vậy bạn sẽ có một số việc phải làm nếu chuyển đổi công cụ hoặc nếu một tiêu chuẩn nổi lên.

Machine Translated by Google

Mục 36: Thư ờng xuyên sử dụng chú thích Ghi đè

Khi các chú thích đư ợc thêm vào ngôn ngữ trong bản phát hành 1.5, một số loại chú thích đã đư ợc thêm vào các thư viện [JLS, 9.6.1]. Đối với các lập trình viên thông thư ờng, điều quan trọng nhất trong số này là Ghi đè. Chú thích này chỉ có thể đư ợc sử dụng trên khai báo phư ơ ng thức và nó chỉ ra rằng khai báo phư ơ ng thức đư ợc chú thích sẽ ghi đè một khai báo trong siêu kiểu. Nếu bạn thư ờng xuyên sử dụng chú thích này, nó sẽ bảo vệ bạn khỏi một lớp lớn các lỗi bất chính. Hãy xem xét chư ơ ng trình này, trong đó lớp Bigram đại diện cho một biểu đồ hay một cặp chữ cái có thứ tự:

```
// Bạn có thể phát hiện ra lỗi
không? public class Bigram
      {private cuối cùng char đầu
      tiên; char cuối cùng riêng tư  thứ
      hai; public Bigram (char first, char second)
            {this.first = first; this.second = giây;


      } public boolean bằng (Bigram b) {return
            b.first == first && b.second == second;

      } public int hashCode ()
            {return 31 * first + second;
      }

      public static void main (String [] args) {
            Đặt <Bigram> s = new HashSet <Bigram> (); for (int
            i = 0; i <10; i ++) for (char ch = 'a'; ch <= 'z';
                  ch ++) s.add (new Bigram (ch, ch));
                        System.out.println (s.size ());

      }
}
```

Chư ơ ng trình chính liên tục thêm hai mư ơ i sáu biểu đồ, mỗi biểu tư ợng bao gồm hai chữ cái viết thư ờng giống nhau, vào một tập hợp. Sau đó, nó in kích thư ớc của tập hợp. Bạn có thể mong đợi chư ơ ng trình in 26, vì các bộ không thể chứa các bản sao. Nếu bạn thử chạy chư ơ ng trình, bạn sẽ thấy rằng nó không in ra 26 mà là 260. Có vấn đề gì với nó?

Rõ ràng, tác giả của lớp Bigram có ý định ghi đè phư ơ ng thức bằng (Mục 8) và thậm chí nhớ ghi đè Mã băm song song (Mục 9).

Thật không may, lập trình viên không may của chúng tôi đã không thể ghi đè bằng, thay vào đó làm quá tải nó (Mục 41). Để ghi đè Object.equals, bạn phải xác định một phư ơ ng thức bằng có tham số thuộc loại Đối tư ợng, như ng tham số của Bigram là bằng

Machine Translated by Google

phư ơ ng thức không thuộc kiểu Đối tư ợng, vì vậy Bigram kế thừa phư ơ ng thức bằng từ Đối tư ợng. Điều này tư ơ ng đư ơ ng với các bài kiểm tra phư ơ ng pháp cho nhận dạng đối tư ợng, giống như toán tử == . Mỗi bản sao trong số mư ời bản sao của mỗi biểu đồ đều khác biệt với chín bản khác, vì vậy chúng đư ợc coi là không bằng nhau bởi Object.equals, điều này giải thích tại sao chư ơ ng trình in ra 260.

May mắn thay, trình biên dịch có thể giúp bạn tìm ra lỗi này, như ng chỉ khi bạn giúp trình biên dịch bằng cách nói với nó rằng bạn định ghi đè Object.equals. Để thực hiện việc này, hãy chú thích Bigram.equals bằng @Override, như đư ợc hiển thị bên dư ới:

```
@Override public boolean bằng (Bigram b) {
      trả về b.first == đầu tiên && b.second == thứ hai;
}
```

Nếu bạn chèn chú thích này và cố gắng biên dịch lại chư ơ ng trình, trình biên dịch sẽ tạo ra một thông báo lỗi như sau:

```
Bigram.java:10: method không ghi đè hoặc triển khai một phư ơ ng thức từ supertype
@Override public boolean = (Bigram b) {

      ^
```

Bạn sẽ ngay lập tức nhận ra mình đã làm gì sai, tự tát vào trán mình và thay thế cách triển khai tư ơ ng đư ơ ng bị hỏng bằng cách thực hiện đúng (Khoản 8):

```
@Override public boolean bằng (Object o) { if (! (O
      instanceof Bigram)) return false; Bigram b = (Bigram)
            o; trả về b.first == đầu tiên && b.second ==
      thứ hai;

}
```

Do đó, bạn nên sử dụng chú thích Ghi đè trên mọi khẩu phần decla phư ơ ng thức mà bạn tin tư ởng để ghi đè khai báo lớp cha. Có một ngoại lệ nhỏ đối với quy tắc này. Nếu bạn đang viết một lớp không có nhãn là trừu tư ợng và bạn tin rằng nó ghi đè một phư ơ ng thức trừu tư ợng, bạn không cần phải đặt chú thích Ghi đè lên phư ơ ng thức đó. Trong một lớp không đư ợc khai báo là trừu tư ợng, trình biên dịch sẽ phát ra một thông báo lỗi nếu bạn không ghi đè đư ợc một phư ơ ng thức siêu lớp trừu tư ợng. Tuy nhiên, bạn có thể muốn thu hút sự chú ý đến tất cả các phư ơ ng thức trong lớp của bạn ghi đè các phư ơ ng thức của lớp cha, trong trư ờng hợp đó, bạn cũng nên làm phiền các phư ơ ng thức này.

Các IDE hiện đại cung cấp một lý do khác để sử dụng Ghi đè chú thích một cách dễ dàng. Các IDE như vậy có kiểm tra tự động đư ợc gọi là kiểm tra mã. nếu bạn