

ORACLE®

Electronic content includes:

- 160 practice exam questions
- Fully customizable test engine



OCA Java SE 8 Programmer I Exam Guide (Exam 1Z0-808)

Complete Exam Preparation

ORACLE®
Certified Associate
Java SE 8 Programmer



Kathy Sierra, SCJP
Bert Bates, SCJP, OCA, OCP

Oracle
Press™

Machine Translated by Google

ORACLE®

Oracle Press™

OCA Java® SE 8 Programmer I Exam Guide

(Exam 1Z0-808)

**Kathy Sierra
Bert Bates**

McGraw-Hill Education is an independent entity from Oracle Corporation and is not affiliated with Oracle Corporation in any manner. This publication and digital content may be used in assisting students to prepare for the Oracle Certified Associate Java™ Programmer I exam. Neither Oracle Corporation nor McGraw-Hill Education warrants that use of this publication and digital content will ensure passing the relevant exam. Oracle® and/or Java™ are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.



New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Bản quyền © 2017 của McGraw-Hill Education. Đã đăng ký Bản quyền. Trừ khi được cho phép theo Đạo luật Bản quyền của Hoa Kỳ năm 1976, không một phần nào của sản phẩm này có thể được sao chép hoặc phân phối dưới bất kỳ hình thức nào hoặc bằng bất kỳ phương tiện nào, hoặc được lưu trữ trong cơ sở dữ liệu hoặc hệ thống truy xuất, mà không có sự cho phép trước bằng văn bản của nhà xuất bản, với ngoại lệ rằng danh sách chương trình có thể được nhập, lưu trữ và thực thi trong hệ thống máy tính, nhưng chúng không được sao chép để xuất bản.

ISBN: 978-1-26-001138-8 MHID:
1-26-001138-0.

Tài liệu trong sách điện tử này cũng xuất hiện trong phiên bản in của tên sách này: ISBN: 978-1-26-001136-4, MHID: 1-26-001136-4.

Chuyển đổi sách điện tử bằng mãMantra
Phiên bản 1.0

Tất cả các nhãn hiệu hàng hoá là thương hiệu của chủ sở hữu tương ứng của họ. Thay vì đặt biểu tượng nhãn hiệu sau mỗi lần xuất hiện tên đã đăng ký nhãn hiệu, chúng tôi chỉ sử dụng tên theo cách biên tập và vì lợi ích của chủ sở hữu nhãn hiệu, không có ý định vi phạm nhãn hiệu. Khi những ký hiệu như vậy xuất hiện trong cuốn sách này, chúng đã được in với chữ viết hoa ban đầu.

Sách điện tử McGraw-Hill Education có sẵn với số lượng giảm giá đặc biệt để sử dụng làm phí bảo hiểm và khuyến mãi bán hàng hoặc để sử dụng trong các chương trình đào tạo của công ty. Để liên hệ với đại diện, vui lòng truy cập trang Liên hệ với chúng tôi tại www.mhprofessional.com.

Oracle và Java là các nhãn hiệu đã đăng ký của Oracle Corporation và / hoặc các chi nhánh của nó. Tất cả các nhãn hiệu khác là tài sản của chủ sở hữu tương ứng và McGraw-Hill Education không yêu cầu quyền sở hữu khi đề cập đến các sản phẩm có chứa các nhãn hiệu này.

Màn hình hiển thị của các chương trình phần mềm Oracle có bản quyền đã được sao chép ở đây với sự cho phép của Tập đoàn Oracle và / hoặc các chi nhánh của nó.

Tập đoàn Oracle không đưa ra bất kỳ tuyên bố hoặc bảo đảm nào về tính chính xác, đầy đủ hoặc hoàn chỉnh của bất kỳ thông tin nào có trong Công việc này, và không chịu trách nhiệm về bất kỳ sai sót hoặc thiếu sót nào.

and is not responsible for any errors or omission(s).

ĐIỀU KHOẢN SỬ DỤNG

Đây là một tác phẩm có bản quyền và McGraw-Hill Education và những người cấp phép của nó bảo lưu mọi quyền đối với và đối với tác phẩm. Sử dụng làm việc này là tùy thuộc vào các điều khoản. Trừ khi được cho phép theo Đạo luật bản quyền năm 1976 và quyền lưu trữ và truy xuất một bản sao của tác phẩm, bạn không được dịch ngược, tháo rời, thiết kế đối chiếu, tái sản xuất, sửa đổi, tạo các tác phẩm phái sinh dựa trên, truyền tải, phân phối, phổ biến, bán, xuất bản hoặc cấp phép lại cho tác phẩm hoặc bất kỳ phần nào của nó mà không có sự đồng ý trước của McGraw-Hill Education. Bạn có thể sử dụng tác phẩm cho mục đích phi thương mại và cá nhân của bạn; mọi việc sử dụng tác phẩm khác đều bị nghiêm cấm.

Quyền sử dụng tác phẩm của bạn có thể bị chấm dứt nếu bạn không tuân thủ những điều kiện.

CÔNG VIỆC ĐƯỢC CUNG CẤP "NGUYÊN TRẠNG." McGRAW-HILL EDUCATION VÀ CÁC NHÀ CẤP PHÉP CỦA NÓ KHÔNG ĐẢM BẢO HOẶC BẢO ĐẢM VỀ SỰ CHÍNH XÁC, BỔ SUNG HOẶC HOÀN THÀNH HOẶC KẾT QUẢ CÓ ĐƯỢC KHI SỬ DỤNG CÔNG VIỆC, BAO GỒM BẤT KỲ THÔNG TIN NÀO CÓ THỂ ĐƯỢC TIẾP CẬN QUA CÔNG VIỆC BẢO ĐẢM, RỘ RÀNG HOẶC NGỤ Ý, BAO GỒM NHƯNG KHÔNG GIỚI HẠN CÁC BẢO ĐẢM NGẦU NHIÊN VỀ TÍNH KHẢ NĂNG HOẶC PHÙ HỢP VỚI MỤC ĐÍCH CỤ THỂ. McGraw-Hill Education và các nhà cấp phép của nó không đảm bảo hoặc đảm bảo rằng các chức năng có trong tác phẩm sẽ đáp ứng yêu cầu của bạn hoặc hoạt động của nó sẽ không bị gián đoạn hoặc không có lỗi. McGraw-Hill Education và người cấp phép của McGraw-Hill Education sẽ không chịu trách nhiệm pháp lý đối với bạn hoặc bất kỳ ai khác về bất kỳ sự không chính xác, sai sót hoặc thiếu sót nào, bất kể nguyên nhân, trong công việc hoặc bất kỳ thiệt hại nào phát sinh từ đó. McGraw-Hill Education không chịu trách nhiệm về nội dung của bất kỳ thông tin nào được truy cập thông qua tác phẩm. Trong mọi trường hợp, McGraw-Hill Education và / hoặc người cấp phép của McGraw-Hill Education sẽ không chịu trách nhiệm pháp lý đối với bất kỳ thiệt hại gián tiếp, ngẫu nhiên, đặc biệt, trường phạt, do hậu quả hoặc tương tự do việc sử dụng hoặc không thể sử dụng tác phẩm, ngay cả khi bất kỳ thiệt hại nào trong số họ đã được thông báo về khả năng xảy ra những thiệt hại đó. Giới hạn trách nhiệm này sẽ áp dụng cho bất kỳ khiếu nại hoặc nguyên nhân nào cho dù khiếu nại hoặc nguyên nhân đó phát sinh trong hợp đồng, vi phạm hay cách khác.

NHÀ ĐÓNG GÓP

Giới thiệu về tác giả

Kathy Sierra là nhà phát triển chính cho kỳ thi SCJP cho Java 5 và Java 6.

Kathy đã làm việc với tư cách là “huấn luyện viên bậc thầy” của Sun và vào năm 1997, thành lập JavaRanch.com (hiện có tên là Coderanch.com), trang web cộng đồng Java lớn nhất thế giới.

Những cuốn sách Java bán chạy nhất của cô đã giành được nhiều giải thưởng của Tạp chí Phát triển Phần mềm và cô là thành viên sáng lập của chương trình Nhà vô địch Java của Oracle.

Hiện nay, Kathy đang phát triển các chương trình đào tạo nâng cao trong nhiều lĩnh vực khác nhau (từ cưỡi ngựa đến lập trình máy tính), nhưng sợi dây liên kết tất cả các dự án của cô với nhau là giúp người học giảm tải nhận thức.

Bert Bates là nhà phát triển chính cho nhiều kỳ thi chứng chỉ Java của Sun, bao gồm SCJP cho Java 5 và Java 6. Bert cũng là một trong những nhà phát triển chính cho các kỳ thi OCA 7 và OCP 7 của Oracle và là người đóng góp cho Oracle's OCA 8 và OCP 8 các kỳ thi. Anh ấy là người điều hành diễn đàn trên Coderanch.com (trước đây là JavaRanch.com) và đã phát triển phần mềm trong hơn 30 năm (argh!). Bert là đồng tác giả của một số cuốn sách Java bán chạy nhất và anh ấy là thành viên sáng lập của chương trình Nhà vô địch Java của Oracle. Bây giờ cuốn sách đã hoàn thành, Bert dự định đi đánh vài quả bóng tennis xung quanh và một lần nữa bắt đầu cưỡi con ngựa Iceland xinh đẹp của mình, Eyrraros fra Gufudal-Fremri.

Giới thiệu về Nhóm đánh giá kỹ thuật

Đây là ấn bản thứ năm của cuốn sách mà chúng tôi đã soạn thảo. Phiên bản đầu tiên chúng tôi làm việc là dành cho Java 2. Sau đó, chúng tôi đã cập nhật sách cho SCJP 5, một lần nữa cho SCJP 6, một lần nữa cho các kỳ thi OCA 7 và OCP 7, và bây giờ là OCA 8.

Mỗi bước của con đường, chúng tôi thật may mắn khi có được JavaRanch.com tuyệt vời, trung tâm nhóm đánh giá kỹ thuật ở phía chúng tôi. Trong suốt 14 năm qua, chúng tôi đã “phát triển” cuốn sách nhiều hơn là viết lại nó. Nhiều phần từ nguyên tác của chúng tôi trên sách Java 2 vẫn còn nguyên vẹn. Trên

các trang tiếp theo, chúng tôi muốn ghi nhận các thành viên của các nhóm đánh giá kỹ thuật khác nhau, những người đã tiết kiệm thịt xông khói của chúng tôi trong những năm qua.

Giới thiệu về Nhóm đánh giá kỹ thuật Java 2

Johannes de Jong đã mãi là trưởng nhóm đánh giá kỹ thuật của chúng tôi. (Anh ấy kiên nhẫn hơn bất kỳ ba người nào mà chúng tôi biết.) Đối với cuốn sách Java 2, anh ấy đã lãnh đạo nhóm lớn nhất của chúng tôi từ trước đến nay. Xin gửi lời cảm ơn chân thành đến những tình nguyện viên sau đây, những người hiểu biết, siêng năng, kiên nhẫn và kén cá chọn canh!

Rob Ross, Nicholas Cheung, Jane Griscti, Ilja Preuss, Vincent Brabant, Kudret Serin, Bill Seipel, Jing Yi, Ginu Jacob George, Radiya, LuAnn Mazza, Anshu Mishra, Anandhi Navaneethakrishnan, Didier Varon, Mary McCartney, Harsha Pherwani, Abhishek Misra , và Suman Das.

Giới thiệu về Nhóm Đánh giá Kỹ thuật SCJP 5



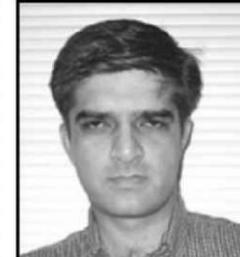
Andrew



Bill M.



Burk



Devender



Gian



Jef



Jeoren



Jim



Johannes



Kristin



Marcelo



Marilyn



Mark



Mikalai



Seema



Valentin

Chúng tôi không biết ai đã đốt nhiều dầu nhất lúc nửa đêm, nhưng chúng tôi có thể (và đã) đếm các chinh sửa của mọi người – vì vậy, theo thứ tự của hầu hết các chinh sửa được thực hiện, chúng tôi tự hào giới thiệu các Siêu sao của mình.

Các danh hiệu hàng đầu của chúng tôi thuộc về Kristin Stromberg – mỗi khi bạn nhìn thấy dấu chấm phẩy

đã sử dụng đúng cách, hãy ngả mũ trước Kristin. Tiếp theo là Burk Hufnagel , người đã sửa nhiều mã hơn chúng ta cần thừa nhận. Bill Mietelski và Gian Franco Casula đã mắc phải mọi lỗi mà chúng tôi đã ném vào họ – công việc tuyệt vời, các bạn! Deude Thareja đảm bảo rằng chúng tôi không sử dụng quá nhiều tiếng lóng, và Mark Spritzler vẫn giữ được sự hài hước. Mikalai Zaikin và Seema Manivannan đã thực hiện những cú bắt tuyệt vời trên mọi bước đường, và Marilyn de Queiroz và Valentin Crettaz đều có một màn trình diễn xuất sắc khác (cứu nguy cho chúng tôi một lần nữa).

Marcelo Ortega, Jef Cumps (một cựu chiến binh khác), Andrew Monkhouse và Jeroen Sterken đã làm tròn đội ngũ Siêu sao của chúng tôi – cảm ơn tất cả các bạn. Jim Yingst là thành viên của nhóm tạo đề thi Sun, và anh ấy đã giúp chúng tôi viết và xem lại một số câu hỏi hay hơn trong cuốn sách (bwa-ha-ha-ha).

Như thường lệ, mỗi khi bạn đọc một trang sạch, hãy cảm ơn những người đánh giá của chúng tôi và nếu bạn bắt lỗi, chắc chắn là do tác giả của bạn đã làm sai. Và ôi, một lời cảm ơn cuối cùng đến Johannes. Bạn thông tri, dude!

Giới thiệu về Nhóm Đánh giá Kỹ thuật SCJP 6



Fred



Marc P.



Marc W.



Mikalai



Christophe

Vì việc nâng cấp lên kỳ thi Java 6 giống như một cuộc phẫu thuật nhỏ, chúng tôi quyết định rằng nhóm đánh giá kỹ thuật cho bản cập nhật này cho cuốn sách cần phải có kiểu dáng tương tự. Để đạt được mục tiêu đó, chúng tôi đã chọn một đội ưu tú gồm các chuyên gia hàng đầu của JavaRanch để thực hiện bài đánh giá cho kỳ thi Java 6.

Lòng biết ơn vô hạn của chúng tôi dành cho Mikalai Zaikin. Mikalai đóng một vai trò rất lớn trong cuốn sách Java 5, và anh ấy đã quay lại để giúp chúng tôi một lần nữa cho ấn bản Java 6 này. Chúng tôi cần cảm ơn Volha, Anastasia và Daria vì đã cho chúng tôi mượn Mikalai. Nhận xét và chỉnh sửa của anh ấy đã giúp chúng tôi thực hiện những cải tiến lớn cho cuốn sách. Cảm ơn,

Mikalai!

Marc Peabody nhận được kudo đặc biệt vì đã giúp chúng tôi lập cú đúp! Ngoài việc giúp chúng tôi làm bài kiểm tra SCWCD mới của Sun, Marc đã giới thiệu một loạt các chỉnh sửa tuyệt vời cho cuốn sách này – bạn đã tiết kiệm được thịt xông khói của chúng tôi trong mùa đông này, Marc! (BTW, chúng tôi đã không biết được rằng Marc, Bryan Basham và Bert đều có chung niềm đam mê với Frisbee cuối cùng!)

Giống như một số người đánh giá của chúng tôi, Fred Rosenberger không chỉ tình nguyện dành nhiều thời gian kiểm duyệt thời gian của mình tại JavaRanch, anh ấy còn tìm thấy thời gian để giúp chúng tôi thực hiện cuốn sách này. Stacey và Olivia, bạn cảm ơn chúng tôi vì đã cho chúng tôi mượn Fred một thời gian.

Marc Weber kiểm duyệt tại một số diễn đàn bận rộn nhất của JavaRanch. Marc biết công cụ của mình và khám phá ra một số vấn đề thực sự lén lút được chôn giấu trong cuốn sách. Mặc dù chúng tôi thực sự đánh giá cao sự giúp đỡ của Marc, nhưng chúng tôi cần cảnh báo tất cả các bạn hãy coi chừng – anh ấy có một Phaser!

Cuối cùng, chúng tôi gửi lời cảm ơn đến Christophe Verre – nếu chúng tôi có thể tìm thấy anh ấy. Có vẻ như Christophe thực hiện nhiệm vụ kiểm duyệt JavaRanch của mình từ nhiều địa điểm khác nhau trên toàn cầu, bao gồm Pháp, Wales và gần đây nhất là Tokyo.

Đã hơn một lần, Christophe đã bảo vệ chúng tôi khỏi sự thiếu tổ chức của chính chúng tôi. Cảm ơn sự kiên nhẫn của bạn, Christophe! Điều quan trọng cần biết là những người này đều đã tặng danh hiệu người đánh giá của họ cho JavaRanch! Cộng đồng JavaRanch đang mắc nợ bạn.

Nhóm OCA 7 và OCP 7

Tác giả đóng góp



Tom



Jeanne

Kỳ thi OCA 7 chủ yếu là sự đóng gói lại hữu ích một số mục tiêu của kỳ thi SCJP 6. Mặt khác, kỳ thi OCP 7 giới thiệu một loạt các chủ đề hoàn toàn mới. Chúng tôi đã mời một số chuyên gia Java tài năng để giúp chúng tôi bao gồm một số chủ đề mới trong kỳ thi OCP 7. Cảm ơn và chúc mừng Tom McGinn vì công lao tuyệt vời của anh ấy trong việc tạo ra chương JDBC lớn. Một số nhà phê bình nói với chúng tôi rằng Tom đã làm một công việc đáng kinh ngạc trong việc truyền tải giọng điệu thân mật mà chúng tôi sử dụng trong suốt cuốn sách. Tiếp theo, cảm ơn Jeanne Boyarsky. Jeanne thực sự là một phụ nữ phục hưng trong dự án này. Cô ấy đã đóng góp vào một số chương OCP; cô ấy đã viết một số câu hỏi cho các kỳ thi thạc sĩ; cô thực hiện một số hoạt động quản lý dự án; và như thể vẫn chưa đủ, cô ấy là một trong những người đánh giá kỹ thuật năng nổ nhất của chúng tôi. Jeanne, chúng tôi không thể cảm ơn đủ. Chúng tôi gửi lời cảm ơn tới Matt Heimer vì công việc xuất sắc của anh ấy trong chương Đồng thời. Một chủ đề thực sự khó khăn được xử lý độc đáo! Cuối cùng, Roel De Nijs và Roberto Perillo đã có một số đóng góp tốt đẹp cho cuốn sách và giúp đỡ nhóm đánh giá kỹ thuật –cảm ơn các bạn!

Nhóm đánh giá kỹ thuật



Roel



Mikalai



Vijitha



Roberto

Roel, chúng ta có thể nói gì? Công việc của bạn như một người đánh giá kỹ thuật là tuyệt vời. Roel mắc quá nhiều lỗi kỹ thuật, khiến chúng tôi quay cuồng. Giữa sách in và tất cả tài liệu trong đĩa CD, chúng tôi ước tính rằng có hơn 1.500 trang "nội dung" ở đây. Nó rất lớn! Roel nghiên ngẫm hết trang này đến trang khác, không bao giờ mất tập trung và khiến cuốn sách này trở nên hay hơn theo nhiều cách. Cảm ơn bạn, Roel!

Ngoài những đóng góp khác của mình, Jeanne đã cung cấp một trong những đánh giá kỹ thuật kỹ lưỡng nhất mà chúng tôi nhận được. (Chúng tôi nghĩ rằng cô ấy đã gia nhập đội robot giết người của mình để giúp cô ấy!)

Có vẻ như sẽ không có cuốn sách K & B nào hoàn chỉnh nếu không có sự trợ giúp từ người cũ của chúng tôi bạn Mikalai Zaikin. Bằng cách nào đó, giữa việc giành được 812 chứng chỉ Java khác nhau, trở thành một người chồng và người cha (nhờ Volha, Anastasia, Daria và Ivan), và là một "người dân lý thuyết" [sic], Mikalai đã có những đóng góp đáng kể cho chất lượng của cuốn sách; chúng tôi rất vinh dự vì bạn đã giúp chúng tôi một lần nữa, Mikalai.

Tiếp theo, chúng tôi muốn cảm ơn Vijitha Kumara, người kiểm duyệt JavaRanch và chuyên gia đánh giá công nghệ phụ. Chúng tôi đã có nhiều người đánh giá giúp đỡ trong suốt quá trình dài viết cuốn sách này, nhưng Vijitha là một trong số ít người đã gắn bó với chúng tôi từ [Chương 1](#) trong suốt các kỳ thi thạc sĩ và đến Chương 15. Vijitha, cảm ơn sự giúp đỡ của bạn và kiên trì!

Cuối cùng, cảm ơn những người còn lại trong nhóm đánh giá của chúng tôi: Roberto Perillo (người cũng đã viết một số đề thi sát thủ), Jim Yingst (đây có phải là lần thứ tư của bạn?), Những người tái phạm khác: Fred Rosenberger, Christophe Verre, Devaka Cooray, Marc Peabody, và người mới Amit Ghorpade – cảm ơn, các bạn!

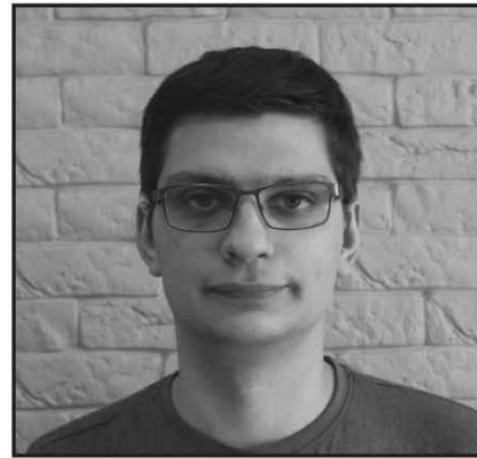
Giới thiệu về Nhóm Đánh giá Kỹ thuật OCA 8



Campbell



Fritz



Paweł



Vijitha



Tim



Roberto

Do "lỗi thí điểm" từ phía tác giả của bạn, các thành viên của nhóm đánh giá cho cuốn sách này đã được yêu cầu làm việc theo một lịch trình chặt chẽ hơn lý tưởng. Chúng tôi rất cảm ơn sáu nhà phê bình của chúng tôi cho cuốn sách này, những người đã hành động ngay trong một thông báo ngắn và một lần nữa đã có nhiều đóng góp tốt đẹp cho chất lượng của cuốn sách. Như đã trở thành "chuẩn mực", tất cả những người đánh giá của chúng tôi đều là người kiểm duyệt tại trang web cộng đồng Java tốt nhất trên toàn thế giới, Coderanch.com.

Đề cập đầu tiên của chúng tôi là về Campbell Ritchie, người kiểm duyệt Coderanch lâu năm, người đi bộ trên những ngọn đồi dốc và theo số lượng của chúng tôi, người đánh giá có nhiều kinh nghiệm nhất về điều này

phiên bản. Nói cách khác, Campbell là người tìm ra nhiều lỗi nhất. Trong số các mục tiêu theo đuổi khác, Campbell là một chuyên gia trong cả lĩnh vực bệnh lý học và lập trình, bởi vì. tại sao không? Mỗi khi bạn đọc một trang không có lỗi, hãy nghĩ đến Campbell và cảm ơn anh ấy.

Trong trận hòa ảo cho vị trí thứ hai trong cuộc thi "tìm thấy lỗi", chúng tôi giới thiệu Paweł Baczyński và Fritz Walraven. Paweł gọi Ba Lan là quê hương của mình và tự hào nhất về vợ, Ania, và các con, Szymek và Gabrysia. Anh ấy gần như tự hào về việc chúng tôi phải sử dụng một bàn phím đặc biệt để đánh vần tên của anh ấy một cách chính xác!

Fritz đến từ Hà Lan, và nếu chúng ta hiểu đúng, hãy cỗ vũ những đứa trẻ của anh ấy khi chúng thể hiện năng lực thể thao của mình ở các địa điểm thể thao khác nhau xung quanh Amersfoort. Fritz cũng đã tình nguyện tại một trại trẻ mồ côi ở Uganda (chúng tôi hy vọng anh ấy sẽ dạy những đứa trẻ đó Java!). Fritz, nếu chúng ta gặp trực tiếp, Bert sẽ thách đấu với bạn trong một trận đấu bóng bàn.

Chúng tôi muốn gửi lời cảm ơn đặc biệt đến Fritz và Paweł vì đã gắn bó với chúng tôi từ [Chương 1](#) đến kỳ thi thực hành thứ hai. Đó là một cuộc chạy marathon, và chúng tôi cảm ơn cả hai.

Tiếp theo, chúng tôi muốn cảm ơn người đánh giá trả lại Vijitha Kumara. Đúng vậy, anh ấy đã giúp chúng tôi trước đây và vậy mà anh ấy lại tình nguyện – wow! Khi Vijitha không đi du lịch hoặc đi bộ đường dài, anh ấy thích thư mà anh ấy gọi là “những thử nghiệm điên rồ”. Chúng tôi yêu bạn, Vijitha – đừng tự nỗ tung!

Tim Cooke đã ở đó khi chúng tôi cần anh ấy, khi bắt đầu quá trình và một lần nữa để giúp chúng tôi hoàn thiện mọi thứ. Chúng tôi thích Tim mặc dù anh ấy được biết là đã dành thời gian cho những “lập trình viên chức năng” bất chính. (Chúng tôi nghĩ đó là nơi anh ấy ở khi anh ấy mất tích trong một vài chương giữa cuốn sách.) Tim sống ở Ireland và bắt đầu lập trình khi còn nhỏ trên chiếc Amstrad CPC 464. Nice!

Cuối cùng, xin gửi lời cảm ơn đặc biệt đến một cựu binh khác của nhóm đánh giá kỹ thuật, Roberto Perillo. Cảm ơn bạn đã quay lại để giúp chúng tôi một lần nữa Roberto. Roberto là một người đàn ông của gia đình, rất thích dành thời gian cho con trai mình, Lorenzo. Khi Lorenzo đã đi ngủ, Roberto được biết đến là người chơi guitar hoặc cỗ vũ cho “Futebol Clube” São Paulo (chúng tôi nghĩ đây là một đội bóng đá).

Các bạn đều tuyệt vời. Cảm ơn sự trợ giúp tuyệt vời của bạn.

Đối với Andi

Đối với Bob

NỘI DUNG TẠI A KÍNH CƯỜNG LỰC

- 1 Khai báo và Kiểm soát Truy cập
- 2 Hướng đối tượng
- 3 nhiệm vụ
- 4 nhà điều hành
- 5 Kiểm soát dòng chảy và các ngoại lệ
- 6 Chuỗi, Mảng, Mảng, Danh sách, Ngày và Lambdas
- A Giới thiệu về Tải xuống
- Mục lục

NỘI DUNG

Sự nhìn nhận

Lời nói đầu

Giới thiệu

1 Khai báo và Kiểm soát Truy cập

Các tính năng và

lợi ích của Java Refresher của Java (Mục tiêu 1.5 của OCA)

Định danh và Từ khóa (Mục tiêu 1.2 và 2.1 của OCA)

Mã nhận dạng pháp

lý Quy ước mã Java của Oracle

Xác định các lớp (Mục tiêu OCA 1.2, 1.3, 1.4, 6.4 và 7.5)

Quy tắc khai báo tệp nguồn

Sử dụng lệnh javac và java

Sử dụng public static void main (String [] args)

Câu lệnh nhập khẩu và API Java

Báo cáo nhập khẩu tinh

Khai báo và sửa đổi lớp

Bài tập 1-1: Tạo lớp siêu trùm tượng và lớp bê tông

Giao diện

sử dụng lớp con (Mục tiêu 7.5 của OCA)

Khai báo giao diện Khai

báo hằng số giao diện Khai báo

các phương thức giao diện mặc định Khai

báo các phương thức giao diện tinh Khai

báo các thành viên lớp (Mục tiêu OCA 2.1, 2.2, 2.3, 4.1, 4.2, 6.2, 6.3, và 6.4)

Access Modifier

Nonaccess Thành viên Sửa đổi

thành viên Khai báo cấu trúc

Khai báo biến

Khai báo và sử dụng enums (Mục tiêu 1.2 của OCA)

Khai báo enums Tóm

tắt chứng nhận Cuộc khoan

✓ hai phút Hỏi và Đáp

Tự kiểm tra Câu trả lời tự

kiểm tra

2 Hướng đối tượng

Đóng gói (Mục tiêu OCA 6.1 và 6.5)

Tính kế thừa và tính đa hình (Mục tiêu của OCA 7.1 và 7.2)

Sự phát triển của sự thừa kế

Tính đa hình mối quan hệ IS-A và

HAS-A (Mục tiêu 7.2 của OCA)

Ghi đè / Quá tải (Mục tiêu 6.1 và 7.2 của OCA)

Các phương thức được ghi

đè Các phương thức được ghi đè

Truyền (Mục tiêu OCA 2.2 và 7.3)

Triển khai một giao diện (Mục tiêu 7.5 của OCA)

Java 8 – Hiện có Nhiều Ké thừa!

Các hình thức hoàn trả hợp pháp (Mục tiêu 2.2 và 6.1 của OCA)

Khai báo loại trả lại

Trả lại giá trị

Trình xây dựng và Trình tạo (Mục tiêu OCA 6.3 và 7.4)

Kiến thức cơ bản về xây dựng

Constructor Chaining

Quy tắc dành cho người xây dựng

Xác định xem một hàm tạo mặc định sẽ được tạo ra hay không

Trình tạo quá tải

Khởi khởi tạo (Mục tiêu OCA 1.2 và 6.3-ish)

Tin học (Mục tiêu 6.2 của OCA)

Các biến và phương thức tĩnh

Tóm tắt chứng nhận

✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

Câu trả lời tự kiểm tra

3 nhiệm vụ

Stack and Heap – Đánh giá nhanh

Văn bản, Bài tập và Biến (Mục tiêu OCA 2.1, 2.2 và 2.3)

Giá trị văn bản cho tất cả các toán tử

gán kiểu nguyên thủy Bài tập 3-1: Đức

kiểu nguyên thủy

Phạm vi (Mục tiêu 1.1 của OCA)

Phạm vi biến đổi

Khởi tạo biến (Mục tiêu OCA 2.1, 4.1 và 4.2)

Sử dụng một phần tử biến hoặc mảng chưa được khởi tạo và

Chưa giao

Nguyên bản và đối tượng cục bộ (ngăn xếp, tự động)

Chuyển các biến vào các phương thức (Mục tiêu 6.6 của OCA)

Truyền các biến tham chiếu đối tượng

Java có sử dụng ngữ nghĩa truyền qua giá trị không?

Thông qua việc thu gom rác từ

các biến nguyên thủy (Mục tiêu 2.4 của OCA)

Tổng quan về Quản lý Bộ nhớ và Thu thập Rác Tổng quan về Trình
thu thập Rác của Java Viết Mã Làm cho Đối tượng Đủ điều kiện để
Thu thập

Bài tập 3-2: Thí nghiệm thu gom rác

Tóm tắt chứng nhận



Khoan hai phút

Hỏi & Đáp Tự kiểm tra

Câu trả lời tự kiểm tra

4 nhà điều hành

Toán tử Java (Mục tiêu OCA 3.1, 3.2 và 3.3)

Phép toán chỉ định Toán

tử quan hệ Toán tử so

sánh Toán tử số học Toán

tử điều kiện Toán tử logic

Điều kiện và lặp lại

Tóm tắt chứng nhận

✓ Khoan hai phút

Hỏi & Đáp [Tự kiểm tra](#)

Câu trả lời tự kiểm tra

5 Kiểm soát dòng chảy và các ngoại lệ

Sử dụng câu lệnh if và switch (Mục tiêu 3.3 và 3.4 của OCA)

if-else Câu lệnh

chuyển đổi rẽ nhánh

[Bài tập 5-1: Tao câu lệnh switch-case](#)

Tạo cấu trúc vòng lặp (Mục tiêu của OCA 5.1, 5.2, 5.3, 5.4 và 5.5)

Sử dụng vòng lặp

while Sử dụng vòng

lặp do Sử dụng vòng

lặp for Sử dụng ngắn và tiếp

tục Câu lệnh không được gắn

nhãn Câu lệnh được gắn nhãn

[Bài tập 5-2: Tao ngoại lệ được gắn nhãn trong khi](#)

xử lý vòng lặp (Mục tiêu của OCA 8.1, 8.2, 8.3, 8.4 và 8.5)

Bắt một ngoại lệ Sử dụng thử và ném bắt Sử dụng

cuối cùng Tuyên truyền các ngoại lệ chưa được

thông báo [Bài tập 5-3: Tuyên truyền và bắt một](#)

[ngoại lệ Xác định](#) ngoại lệ Hệ thống phân cấp ngoại lệ Xử lý

toàn bộ Phân cấp ngoại lệ Ngoại lệ Khớp Tuyên bố ngoại lệ và

giao diện công cộng [Bài tập 5-4: Tạo ra một loại ngoại lệ](#)

cùng một Ngoại lệ Các lỗi và Ngoại lệ Chung (Mục tiêu 8.5

của OCA)

Trường hợp ngoại lệ đến từ ngoại
lệ JVM-Thrown Ngoại lệ ném có lập
trình Tóm tắt về các lỗi và ngoại lệ của
bài kiểm tra

Tóm tắt chứng nhận

✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

Câu trả lời tự kiểm tra

6 Chuỗi, Mảng, Danh sách, Ngày và Lambdas

Sử dụng String và StringBuilder (Mục tiêu 9.2 và 9.1 của OCA)

Lớp String Thông

tin quan trọng về chuỗi và bộ nhớ Các phương thức
quan trọng trong lớp String Lớp StringBuilder Các
phương thức quan trọng trong lớp StringBuilder

Làm việc với dữ liệu lịch (Mục tiêu 9.3 của OCA)

Các lớp của

năm nhà máy bắt biến

Sử dụng và Thao tác Ngày và Thời gian Định

dạng Ngày và Thời gian bằng Mảng (Mục tiêu 4.1
và 4.2 của OCA)

Khai báo một mảng Xây

dụng một mảng Khởi tạo

một mảng

Sử dụng ArrayLists và Wrappers (Mục tiêu 9.4 và 2.5 của OCA)

Khi nào sử dụng ArrayLists

Các phương thức ArrayList trong

Hành động Các phương pháp quan trọng trong Tự

động đóng hộp lớp ArrayList với ArrayLists Java

7 "Diamond" Syntax Advanced Encapsulation (OCA

Objective 6.5)

Đóng gói cho các biến tham chiếu

Sử dụng Lambdas đơn giản (Mục tiêu OCA 9.5)

Tóm tắt chứng nhận Cuộc

✓ khoan hai phút

Hỏi & Đáp Tự kiểm tra

Câu trả lời tự kiểm tra

A Giới thiệu về Tải xuống

[yêu cầu hệ thống](#)

[Tải xuống từ Trung tâm Truyền thông của McGraw-Hill Professional](#)

[Cài đặt phần mềm thi thực hành](#)

[Chạy phần mềm thi thực hành](#)

[Thực hành các tính năng của phần mềm thi](#)

[Xóa cài đặt](#)

[Cứu giúp](#)

[Hỗ trợ kỹ thuật](#)

[Khắc phục sự cố Windows 8](#)

[Hỗ trợ Nội dung Giáo dục McGraw-Hill](#)

[Mục lục](#)

SỰ NHÌN NHẬN

K

thể thao và Bert muốn cảm ơn những người sau:

- Tất cả những người cực kỳ chăm chỉ tại McGraw-Hill Education:
Tim Green (người đã gắn bó với chúng tôi 14 năm nay), Lisa McClain và LeeAnn Pickrell. Cảm ơn tất cả sự giúp đỡ của bạn cũng như rất nhanh nhẹ, kiên nhẫn, linh hoạt và chuyên nghiệp cũng như nhóm người tốt nhất mà chúng tôi có thể hy vọng được làm việc cùng.
- Tất cả bạn bè của chúng tôi tại Kraftur (và những người bạn khác có liên quan đến ngựa của chúng tôi) và đặc biệt nhất là Sherry, Steinar, Stina và các cô gái, Kacie, DJ, Jec, Leslie và David, Annette và Bruce, Lucy, Cait và Jennifer, Gabrielle, và Mary. Cảm ơn Pedro và Ely.
- Một số chuyên gia phần mềm và bạn bè đã giúp đỡ chúng tôi trong những ngày đầu: Tom Bender, Peter Loerincs, Craig Matthews, Leonard Coyne, Morgan Porter và Mike Kavenaugh.
- Dave Gustafson đã tiếp tục hỗ trợ, hiểu biết sâu sắc và huấn luyện.
- Người bạn và người bạn tuyệt vời của chúng tôi tại Oracle, Yvonne Prefontaine.
- Những người bạn và người thầy tuyệt vời của chúng ta: Simon Roberts, Bryan Basham và Kathy Collina.
- Stu, Steve, Burt và Marc Hedlund vì đã đưa một số niềm vui vào quá trình này.
- Đôi với Eden và Skyler, vì kinh hoàng rằng những người lớn - ngoài trường học - sẽ học chăm chỉ cho kỳ thi.
- Gửi tới Ông chủ của Coderanch Trail, Paul Wheaton, vì đã điều hành trang web cộng đồng Java tốt nhất trên Web, và gửi đến tất cả những người điều hành JavaRanch, er, Coderanch hào phóng và kiên nhẫn.
- Gửi đến tất cả các giảng viên Java trong quá khứ và hiện tại đã giúp học tập Java là một trải nghiệm thú vị, bao gồm (chỉ nêu tên một số) Alan Petersen, Jean Tordella, Georgianna Meagher, Anthony Orapallo, Jacqueline Jones,

James Cubeta, Teri Cubeta, Rob Weingruber, John Nyquist, Asok Perumainar, Steve Stelting, Kimberly Bobrow, Keith Ratliff, và anh chàng Java quan tâm và truyền cảm hứng nhất hành tinh, Jari Paukku.

- Những người bạn lông lá và lông lá của chúng ta: Eyra, Kara, Draumur, Vafi, Boi, Niki và Bokeh.
- Cuối cùng, gửi đến Eric Freeman và Beth Robson để bạn tiếp tục truyền cảm hứng.

LỜI NÓI ĐẦU

T

Mục tiêu chính của cuốn sách là giúp bạn chuẩn bị và vượt qua kỳ thi lấy chứng chỉ Lập trình viên OCA Java SE 8 của Oracle.

Cuốn sách này bám sát cả chiều rộng và chiều sâu của các đề thi thật. Ví dụ, sau khi đọc cuốn sách này, có thể bạn sẽ không trở thành một chuyên gia về 00, nhưng nếu bạn nghiên cứu tài liệu và làm tốt các Bài kiểm tra bản thân, bạn sẽ có hiểu biết cơ bản về 00 và bạn sẽ làm tốt kỳ thi. Sau khi hoàn thành cuốn sách này, bạn nên cảm thấy tự tin rằng bạn đã xem xét kỹ lưỡng tất cả các mục tiêu mà Oracle đã thiết lập cho các kỳ thi này.

Trong cuốn sách này

Cuốn sách này được sắp xếp thành sáu chương để tối ưu hóa việc học các chủ đề của kỳ thi OCA 8. Bất cứ khi nào có thể, chúng tôi tổ chức các chương để song song với các mục tiêu Oracle, nhưng đôi khi chúng tôi sẽ kết hợp các mục tiêu hoặc lặp lại một phần chúng để trình bày các chủ đề theo thứ tự phù hợp hơn với việc học tài liệu.

Trong mỗi chương

Chúng tôi đã tạo một tập hợp các thành phần của chương để thu hút sự chú ý của bạn đến các mục quan trọng, cung cấp các điểm quan trọng và cung cấp các gợi ý làm bài thi hữu ích. Hãy xem những gì bạn sẽ tìm thấy trong mỗi chương: Mỗi chương đều bắt

- Đầu với Mục tiêu Chứng nhận – những gì bạn cần biết để vượt qua phần trong bài kiểm tra về chủ đề của chương. Các tiêu đề Mục tiêu Chứng nhận xác định các mục tiêu trong chương, vì vậy bạn sẽ luôn biết mục tiêu khi bạn nhìn thấy nó!



Bài kiểm tra Ghi chú theo dõi kêu gọi sự chú ý đến thông tin và những cạm bẫy tiềm ẩn trong kỳ thi. Bởi vì chúng tôi là thành viên trong nhóm đã tạo ra các kỳ thi này, chúng tôi biết những gì bạn sắp phải trải qua!



- Trên Chú thích công việc thảo luận về các khía cạnh thực tế của các chủ đề chứng nhận có thể không xảy ra trong kỳ thi, nhưng điều đó sẽ hữu ích trong thế giới thực.
- Các bài tập được thực hiện xen kẽ trong các chương. Chúng giúp bạn nắm vững các kỹ năng có thể là một lĩnh vực trọng tâm trong kỳ thi.
Đừng chỉ đọc qua các bài tập; chúng là thực hành thực hành mà bạn có thể cảm thấy thoải mái khi hoàn thành. Vừa học vừa làm là một cách hiệu quả để tăng năng lực của bạn với một sản phẩm.
- Từ thanh bên của Lớp học mô tả các vấn đề thường xảy ra nhất trong bối cảnh lớp học đào tạo. Các thanh bên này cung cấp cho bạn góc nhìn có giá trị về các chủ đề liên quan đến chứng nhận và sản phẩm. Họ chỉ ra những sai lầm phổ biến và giải quyết các câu hỏi này sinh từ các cuộc thảo luận trong lớp.
- Tóm tắt Chứng nhận là một đánh giá ngắn gọn của chương và trình bày lại các điểm nổi bật liên quan đến kỳ thi.
- ✓ ■ Cuộc khoan hai phút ở cuối mỗi chương là danh sách kiểm tra các điểm chính của chương. Nó có thể được sử dụng để xem xét vào phút cuối.

Hỏi & Đáp Tự kiểm tra đưa ra các câu hỏi tương tự như những câu hỏi được tìm thấy trên kỳ thi chứng chỉ, bao gồm nhiều lựa chọn và câu hỏi kéo và thả giả. Câu trả lời cho những câu hỏi này, cũng như phần giải thích câu trả lời, có thể được tìm thấy ở cuối mỗi chương. Bằng cách làm Bài kiểm tra bản thân sau khi hoàn thành mỗi chương, bạn sẽ củng cố những gì bạn đã học được từ chương đó, đồng thời làm quen với cấu trúc của đề thi.

GIỚI THIỆU

Cơ quan

Cuốn sách này được tổ chức theo cách để phục vụ cho việc ôn tập chuyên sâu cho kỳ thi OCA 8, dành cho cả các chuyên gia Java có kinh nghiệm và những người đang trong giai đoạn đầu kinh nghiệm với các công nghệ Java. Mỗi chương bao gồm ít nhất một khía cạnh chính của kỳ thi, với sự nhấn mạnh về “lý do tại sao” cũng như “cách thực hiện” lập trình bằng ngôn ngữ Java.

Phần mềm luyện thi với hai đề thi gồm 80 câu hỏi có sẵn để tải xuống.

Cuốn sách này không phải là gì

Bạn sẽ không tìm thấy hướng dẫn học Java cho người mới bắt đầu trong cuốn sách này. Tất cả hơn 500 trang của cuốn sách này chỉ dành riêng để giúp bạn vượt qua kỳ thi. Nếu bạn là người mới làm quen với Java, chúng tôi khuyên bạn nên dành một ít thời gian để học những điều cơ bản. Bạn không nên bắt đầu với cuốn sách này cho đến khi bạn biết cách viết, biên dịch và chạy các chương trình Java đơn giản. Tuy nhiên, chúng tôi không giả định bất kỳ mức độ hiểu biết trước nào về các chủ đề riêng lẻ được đề cập. Nói cách khác, đối với bất kỳ chủ đề nhất định nào (được thúc đẩy hoàn toàn bởi các mục tiêu của kỳ thi thực tế), chúng tôi bắt đầu với giả định rằng bạn là người mới đối với chủ đề đó. Vì vậy, chúng tôi cho rằng bạn là người mới đối với các chủ đề riêng lẻ, nhưng chúng tôi cho rằng bạn không phải là người mới đối với Java.

Chúng tôi cũng không giả vờ vừa chuẩn bị cho bạn cho kỳ thi, vừa đồng thời khiến bạn trở thành một Java hoàn chỉnh. Đây là hướng dẫn ôn tập kỳ thi chứng chỉ và nó rất rõ ràng về nhiệm vụ của nó. Điều đó không có nghĩa là chuẩn bị cho kỳ thi sẽ không giúp bạn trở thành một lập trình viên Java tốt hơn! Ngược lại, ngay cả những nhà phát triển Java có kinh nghiệm nhất cũng thường tuyên bố rằng việc phải chuẩn bị cho kỳ thi lấy chứng chỉ khiến họ trở thành những lập trình viên thông thạo và hiểu biết hơn rất nhiều so với những gì họ đã có nếu không có sự nghiên cứu dựa trên kỳ thi.

Nội dung kỹ thuật số

Nội dung kỹ thuật số

Để biết thêm thông tin chi tiết về phần mềm thi thực hành, vui lòng xem tại [Phụ lục](#).

Một số con trỏ

Khi bạn đã đọc xong cuốn sách này, hãy dành một chút thời gian để xem xét kỹ lưỡng. Bạn có thể muốn quay lại cuốn sách nhiều lần và sử dụng tất cả các phương pháp mà cuốn sách cung cấp để xem lại tài liệu:

1. Đọc lại tất cả Cuộc tập trận kéo dài hai phút hoặc nhờ ai đó đó bạn. Bạn cũng có thể sử dụng các bài tập như một cách để ôn luyện nhanh trước kỳ thi. Bạn có thể muốn tạo một số thẻ flash từ thẻ chỉ mục 3×5 có chất liệu Two Minute Drill trên đó.
2. Đọc lại tất cả các ghi chú Xem bài thi. Hãy nhớ rằng những ghi chú này được viết bởi các tác giả đã giúp tạo ra bài kiểm tra. Họ biết những gì bạn nên mong đợi – và những gì bạn nên đề phòng.
3. Làm lại các bài kiểm tra bản thân. Làm các bài kiểm tra ngay sau khi bạn đọc chương là một ý tưởng hay vì các câu hỏi giúp củng cố những gì bạn vừa học. Tuy nhiên, tốt hơn hết là quay lại sau và làm tất cả các câu hỏi trong sách trong một lần ngồi. Giả vờ rằng bạn đang tham gia kỳ thi trực tiếp. (Bất cứ khi nào bạn làm Bài kiểm tra bản thân, hãy đánh dấu câu trả lời của bạn trên một mảnh giấy riêng biệt. Bằng cách đó, bạn có thể xem qua các câu hỏi nhiều lần tùy ý cho đến khi bạn cảm thấy thoải mái với tài liệu.)
4. Hoàn thành các bài tập. Các bài tập được thiết kế để bao gồm các chủ đề của kỳ thi và không có cách nào tốt hơn để làm quen với tài liệu này hơn là luyện tập. Hãy chắc chắn rằng bạn hiểu lý do tại sao bạn đang thực hiện từng bước trong mỗi bài tập. Nếu có điều gì bạn chưa rõ, hãy đọc lại phần đó trong chương.
5. Viết nhiều mã Java. Chúng tôi sẽ lặp lại lời khuyên này nhiều lần. Khi chúng tôi viết cuốn sách này, chúng tôi đã viết hàng trăm chương trình Java nhỏ để giúp chúng tôi thực hiện nghiên cứu của mình. Chúng tôi đã nghe từ hàng trăm ứng viên đã vượt qua kỳ thi, và hầu hết mọi trường hợp, những ứng viên đạt điểm cực cao trong kỳ thi đã viết rất nhiều mã trong quá trình học của họ. Thử nghiệm với các mẫu mã trong sách, tạo danh sách các lỗi trình biên dịch khủng khiếp – loại bỏ IDE của bạn, viết dòng lệnh và viết mã!

Giới thiệu về tài liệu trong sách

Kỳ thi OCA 8 được coi là một trong những kỳ thi khó nhất trong ngành CNTT, và chúng tôi có thể nói với bạn từ kinh nghiệm rằng rất nhiều thí sinh tham gia kỳ thi mà không chuẩn bị. Là lập trình viên, chúng ta có xu hướng chỉ học những gì chúng ta cần để hoàn thành dự án hiện tại của mình, với những thời hạn điên rồ mà chúng ta thường phải tuân theo.

Nhưng kỳ thi này cố gắng chứng minh sự hiểu biết hoàn toàn của bạn về ngôn ngữ Java, không chỉ những phần bạn đã quen thuộc trong công việc của mình.

Chỉ riêng kinh nghiệm sẽ hiếm khi giúp bạn vượt qua kỳ thi này với điểm đậu, bởi vì ngay cả những điều bạn nghĩ rằng bạn biết cũng có thể hoạt động khác một chút so với bạn tưởng tượng. Nó không đủ để có thể làm cho mã của bạn hoạt động chính xác; bạn phải hiểu các nguyên tắc cơ bản cốt lõi một cách sâu sắc và có đủ độ rộng để bao quát hầu hết mọi thứ có thể phát sinh trong quá trình sử dụng ngôn ngữ.

Ai quan tâm đến chứng nhận?

Người sử dụng lao động làm. Các công ty săn đầu người có. Lập trình viên làm. Vượt qua kỳ thi này chứng tỏ ba điều quan trọng đối với nhà tuyển dụng hiện tại hoặc tương lai: bạn thông minh; bạn biết làm thế nào để học tập và chuẩn bị cho một bài kiểm tra đầy thử thách; và hơn hết, bạn biết ngôn ngữ Java. Nếu nhà tuyển dụng có sự lựa chọn giữa ứng viên đã vượt qua kỳ thi và ứng viên không vượt qua kỳ thi, nhà tuyển dụng biết rằng lập trình viên được chứng nhận không phải mất thời gian để học ngôn ngữ Java.

Nhưng nó có nghĩa là bạn thực sự có thể phát triển phần mềm bằng Java? Không nhất thiết, nhưng đó là một khởi đầu thuận lợi. Để thực sự thể hiện khả năng phát triển của bạn (thay vì chỉ kiến thức về ngôn ngữ của bạn), bạn nên cân nhắc theo đuổi Kỳ thi dành cho nhà phát triển Java, nơi bạn được giao nhiệm vụ xây dựng chương trình, bắt đầu hoàn thành và gửi nó cho người đánh giá để đánh giá và ghi bàn.

Tham gia kỳ thi Lập trình viên

Trong một thế giới hoàn hảo, bạn sẽ được đánh giá về kiến thức thực sự của bạn về một chủ đề, không chỉ đơn giản là cách bạn trả lời một loạt các câu hỏi kiểm tra. Nhưng cuộc sống không hoàn hảo, và việc đánh giá kiến thức của mọi người trên cơ sở một đối mặt là không thực tế.

Đối với hầu hết các chứng chỉ của mình, Oracle đánh giá các ứng viên bằng máy tính-

Đối với hầu hết các chứng chỉ của mình, Oracle đánh giá các ứng viên bằng máy tính dịch vụ thử nghiệm dựa trên do Pearson VUE điều hành. Để ngăn cản việc ghi nhớ đơn giản, các kỳ thi Oracle đưa ra một bộ câu hỏi có khả năng khác nhau cho các ứng viên khác nhau. Trong quá trình phát triển của kỳ thi, hàng trăm câu hỏi được biên soạn và tinh chỉnh bằng cách sử dụng các trình thử nghiệm beta. Từ bộ sưu tập lớn này, các câu hỏi được tổng hợp từ từng mục tiêu và được tập hợp thành nhiều phiên bản khác nhau của kỳ thi.

Mỗi bài thi Oracle có một số câu hỏi cụ thể và thời lượng của bài thi được thiết kế rõ ràng rõ ràng. Thời gian còn lại luôn hiển thị ở góc của màn hình thử nghiệm. Nếu hết thời gian trong một kỳ thi, bài kiểm tra sẽ kết thúc và các câu trả lời không đầy đủ được tính là không chính xác.



Nhiều thí sinh có kinh nghiệm không quay lại và thay đổi câu trả lời trừ khi họ có lý do chính đáng để làm như vậy. Chỉ thay đổi câu trả lời khi bạn cảm thấy mình có thể đã đọc sai hoặc hiểu sai câu hỏi trong lần đầu tiên.

Sự cẩn thận có thể khiến bạn đoán lần thứ hai mọi câu trả lời và tự nói mình không biết câu trả lời đúng.

Sau khi hoàn thành bài kiểm tra, bạn sẽ nhận được e-mail từ Oracle cho bạn biết rằng kết quả của bạn đã có trên Web. Kể từ mùa đông năm 2017, kết quả của bạn có thể được tìm thấy tại certview.oracle.com. Nếu bạn muốn có một bản in chứng chỉ của mình, bạn phải đưa ra một yêu cầu cụ thể.

Dạng câu hỏi

Các kỳ thi Java của Oracle đưa ra các câu hỏi ở dạng trắc nghiệm.

Câu hỏi nhiều lựa chọn Trong các

phiên bản trước của kỳ thi, khi bạn gặp một câu hỏi trắc nghiệm, bạn không được biết có bao nhiêu câu trả lời đúng; nhưng với mỗi phiên bản của kỳ thi, các câu hỏi trở nên khó hơn, vì vậy ngày nay, mỗi câu hỏi trắc nghiệm cho bạn biết bao nhiêu câu trả lời để chọn. Các câu hỏi Tự kiểm tra ở cuối mỗi chương phù hợp với hình thức, từ ngữ và độ khó của đề thi thật, ngoại trừ hai trường hợp:

độ khó của đề thi thực, với hai ngoại lệ:

- Bất cứ khi nào chúng tôi có thể, câu hỏi của chúng tôi sẽ không cho bạn biết có bao nhiêu câu trả lời đúng tồn tại (chúng tôi sẽ nói "Chọn tất cả câu phù hợp"). Chúng tôi làm điều này để giúp bạn nắm vững tài liệu. Một số người làm bài thi hiểu biết có thể loại bỏ các câu trả lời sai khi biết số câu trả lời đúng. Cũng có thể, nếu bạn biết bao nhiêu câu trả lời đúng, hãy chọn những câu trả lời hợp lý nhất. Công việc của chúng tôi là giúp bạn tăng cường sức khỏe cho kỳ thi thực sự!
- Kỳ thi thực thường đánh số các dòng mã trong một câu hỏi. Đôi khi chúng tôi không đánh số các dòng mã – chủ yếu là để chúng tôi có không gian để thêm nhận xét ở những vị trí quan trọng. Trong bài kiểm tra thực tế, khi danh sách mã bắt đầu bằng dòng 1, điều đó có nghĩa là bạn đang xem toàn bộ tệp nguồn. Nếu danh sách mã bắt đầu ở số dòng lớn hơn 1, điều đó có nghĩa là bạn đang xem một phần tệp nguồn. Khi xem một phần tệp nguồn, hãy giả sử mã bạn không thấy là chính xác. (Ví dụ: trừ khi được nêu rõ ràng, bạn có thể giả định rằng một phần tệp nguồn sẽ có câu lệnh nhập và gói chính xác.)



Khi bạn thấy mình lúng túng khi trả lời các câu hỏi trắc nghiệm, hãy dùng giấy nháp (hoặc bảng trắng) để viết ra hai hoặc ba câu trả lời mà bạn cho là mạnh nhất, sau đó gạch chân câu trả lời mà bạn cho là đúng nhất. Dưới đây là một ví dụ về những gì giấy nháp của bạn có thể trông như thế nào khi bạn đã trải qua bài kiểm tra một lần:

- 21. B hoặc C
- 33. A hoặc C

Điều này cực kỳ hữu ích khi bạn đánh dấu câu hỏi và tiếp tục.

Sau đó, bạn có thể quay lại câu hỏi và ngay lập tức bắt đầu quá trình suy nghĩ của bạn ở nơi bạn đã dừng lại. Sử dụng kỹ thuật này để tránh phải đọc lại và suy nghĩ lại các câu hỏi. Bạn cũng sẽ cần sử dụng giấy nháp của mình trong các câu hỏi kịch bản phức tạp, dựa trên văn bản để tạo ra các hình ảnh trực quan để hiểu rõ hơn về câu hỏi. Kỹ thuật này đặc biệt hữu ích nếu bạn là người học trực quan.

Mẹo làm bài kiểm tra

Mẹo làm bài kiểm tra

Số lượng câu hỏi và tỷ lệ vượt qua cho mỗi kỳ thi có thể thay đổi. Luôn kiểm tra với Oracle trước khi làm bài kiểm tra, tại www.Oracle.com.

Bạn được phép trả lời các câu hỏi theo bất kỳ thứ tự nào và bạn có thể quay lại và kiểm tra câu trả lời của mình sau khi đã hoàn thành bài kiểm tra. Không có hình phạt nào cho những câu trả lời sai, vì vậy ít nhất bạn nên thử một câu trả lời hơn là không đưa ra một câu trả lời nào cả.

Một chiến lược tốt để làm bài kiểm tra là xem qua một lần và trả lời tất cả câu hỏi đến với bạn một cách nhanh chóng. Sau đó, bạn có thể quay lại và làm những việc khác. Trả lời một câu hỏi có thể giúp bạn nhớ lại cách trả lời câu hỏi trước một.

Hãy rất cẩn thận trên các ví dụ mã. Trước tiên, hãy kiểm tra các lỗi cú pháp: đếm dấu ngoặc nhọn, dấu chấm phẩy và dấu ngoặc đơn, sau đó đảm bảo có nhiều lỗi bên trái cũng như só bên phải. Tìm các lỗi viết hoa và các vấn đề cú pháp khác như vậy trước khi cố gắng tìm ra mã hoạt động.

Nhiều câu hỏi trong kỳ thi sẽ xoay quanh sự tinh tế của cú pháp. Bạn sẽ cần phải có kiến thức toàn diện về ngôn ngữ Java để thành công.

Điều này đưa chúng ta đến một vấn đề khác mà một số ứng viên đã báo cáo. Trung tâm khảo thí phải cung cấp cho bạn đầy đủ dụng cụ viết để bạn có thể giải quyết các vấn đề "trên giấy". Trong một số trường hợp, các trung tâm đã cung cấp không đủ bút lông và bảng xóa khá nhỏ và cồng kềnh để sử dụng hiệu quả. Chúng tôi khuyên bạn nên gọi điện trước và xác minh rằng bạn sẽ được cung cấp bảng trắng đủ lớn, bút lông đủ mịn và tẩy tốt. Những gì chúng tôi thực sự muốn khuyến khích là mọi người hãy phàn nàn với Oracle và Pearson VUE và yêu cầu họ cung cấp bút chì thực tế và ít nhất một tờ giấy trắng.

Mẹo học bài cho kỳ thi

Đầu tiên và quan trọng nhất, hãy dành cho mình nhiều thời gian để học tập. Java là một ngôn ngữ lập trình phức tạp và bạn không thể mong đợi nhồi nhét những gì bạn cần biết vào một buổi học duy nhất. Đây là lĩnh vực được học tốt nhất theo thời gian, bằng cách nghiên cứu một chủ đề và sau đó áp dụng kiến thức của bạn. Hãy xây dựng cho mình một lịch trình học tập và bám sát nó, nhưng phải hợp lý với những áp lực bạn đặt ra cho bản thân, đặc biệt nếu bạn đang học bên cạnh nhiệm vụ thường xuyên tại nơi làm việc.

Một kỹ thuật dễ sử dụng trong việc học tập để thi lấy chứng chỉ là nỗ lực 15 phút mỗi ngày. Đơn giản chỉ cần học tối thiểu 15 phút mỗi ngày. Nó

nỗ lực từng phút mỗi ngày. Đơn giản chỉ cần học tối thiểu 15 phút mỗi ngày. Đó là một cam kết nhỏ nhưng có ý nghĩa. Nếu bạn có một ngày mà bạn không thể tập trung, thì hãy từ bỏ vào lúc 15 phút. Nếu bạn có một ngày mà nó hoàn toàn dành cho bạn, hãy nghiên cứu lâu hơn. Miễn là bạn có nhiều "ngày trôi chảy" hơn, thì cơ hội thành công của bạn là rất tốt.

Chúng tôi thực sự khuyên bạn nên sử dụng thẻ flash khi chuẩn bị cho các kỳ thi lập trình viên. Thẻ flash chỉ đơn giản là một thẻ chỉ mục 3×5 hoặc 4×6 với một câu hỏi ở mặt trước và câu trả lời ở mặt sau. Bạn tự xây dựng các thẻ này khi xem qua một chương, nắm bắt bất kỳ chủ đề nào mà bạn cho rằng có thẻ cần nhiều thời gian ghi nhớ hoặc luyện tập hơn. Bạn có thể tự tìm hiểu chúng bằng cách đọc câu hỏi, suy nghĩ về câu trả lời, sau đó lật lại thẻ để xem bạn có đúng hay không. Hoặc bạn có thể nhờ người khác giúp bạn bằng cách giơ thẻ có câu hỏi mà bạn đang gặp phải và sau đó xác minh câu trả lời của bạn. Hầu hết các sinh viên của chúng tôi đều nhận thấy những điều này vô cùng hữu ích, đặc biệt là vì chúng rất dễ di chuyển nên trong khi bạn đang ở chế độ học tập, bạn có thể mang chúng đi khắp mọi nơi. Tuy nhiên, tốt nhất không nên sử dụng chúng khi đang lái xe, trừ những lúc đèn đỏ. Chúng tôi đã đưa chúng tôi đi khắp mọi nơi – văn phòng bác sĩ, nhà hàng, rạp hát, bạn có thể đặt tên cho nó.

Các nhóm nghiên cứu chứng chỉ là một nguồn tài liệu tuyệt vời khác, và bạn sẽ không tìm thấy một cộng đồng lớn hơn hoặc săn sàng hơn trên [Coderanch.com](#). Diễn đàn chứng nhận Big Moose Saloon. Nếu bạn có câu hỏi từ cuốn sách này hoặc bất kỳ câu hỏi thi thử nào khác mà bạn có thể đã vấp phải, đăng câu hỏi lên diễn đàn chứng nhận sẽ giúp bạn có câu trả lời trong hầu hết mọi trường hợp trong vòng một ngày – thường là trong vòng vài giờ.

Cuối cùng, chúng tôi khuyên bạn nên viết nhiều chương trình Java nhỏ! Trong quá trình viết cuốn sách này, chúng tôi đã viết hàng trăm chương trình nhỏ, và nếu bạn lắng nghe những gì các ứng viên thành công nhất nói (bạn biết đấy, những người đạt 98%), họ hầu như luôn báo cáo rằng họ đã viết rất nhiều mã.

Lên lịch kiểm tra của bạn

Bạn có thể mua phiếu dự thi của mình từ Oracle hoặc Pearson VUE. Truy cập [Oracle.com](#) (theo các liên kết đào tạo / chứng nhận) hoặc truy cập [PearsonVue.com](#) để biết chi tiết lịch thi và địa điểm của các trung tâm khảo thí.

Đến kỳ thi

Như với bất kỳ bài kiểm tra nào, bạn sẽ bị cám dỗ để nhồi nhét vào đêm hôm trước. Hãy chống lại sự cám dỗ đó. Bạn nên biết tài liệu vào thời điểm này, và nếu bạn còn lúng túng trong

sự cám dỗ. Bạn nên biết tài liệu vào thời điểm này, và nếu bạn lo lắng vào buổi sáng, bạn sẽ không nhớ những gì bạn đã học. Có được một giấc ngủ ngon.

Đến sớm cho kỳ thi của bạn; nó cung cấp cho bạn thời gian để thư giãn và xem xét các sự kiện chính. Tận dụng cơ hội để xem lại các ghi chú của bạn. Nếu bạn quá mệt mỏi với việc học, bạn thường có thể bắt đầu bài kiểm tra của mình sớm vài phút. Chúng tôi khuyên bạn không nên đến muộn. Bài kiểm tra của bạn có thể bị hủy hoặc bạn có thể không có đủ thời gian để hoàn thành bài kiểm tra.

Khi đến trung tâm kiểm tra, bạn sẽ cần cung cấp giấy tờ tùy thân có ảnh hợp lệ, hiện tại. Truy cập PearsonVue.com để biết chi tiết về các yêu cầu ID.

Họ chỉ muốn chắc chắn rằng bạn không gửi cho người hàng xóm giỏi về Java guru, người mà bạn đã trả tiền để làm bài kiểm tra cho bạn.

Ngoài một bộ não chứa đầy những dữ kiện, bạn không cần phải mang bất cứ thứ gì khác vào phòng thi. Trên thực tế, bộ não của bạn là tất cả những gì bạn được phép tham gia kỳ thi!

Tất cả các bài kiểm tra đều được đóng sách, có nghĩa là bạn không được mang theo bất kỳ tài liệu tham khảo nào bên mình. Bạn cũng không được phép mang bất kỳ ghi chú nào ra khỏi phòng thi. Quản trị viên kiểm tra sẽ cung cấp cho bạn một bảng đánh dấu nhỏ. Nếu được phép, chúng tôi khuyên bạn nên mang theo chai nước hoặc chai nước trái cây (gọi trước để biết chi tiết về những thứ được phép). Những bài kiểm tra này kéo dài và khó, và não của bạn hoạt động tốt hơn nhiều khi được cung cấp đủ nước. Về mặt hydrat hóa, cách tiếp cận lý tưởng là uống thường xuyên, từng ngụm nhỏ. Bạn cũng nên minh xem bạn sẽ được phép thực hiện bao nhiêu "khoảng nghỉ sinh học" trong suốt kỳ thi!

Để điện thoại trên xe. Nó sẽ chỉ thêm căng thẳng cho tình huống, vì họ không được phép vào phòng thi và đôi khi vẫn có thể nghe thấy nếu họ reo bên ngoài phòng. Ví, sách và các tài liệu khác phải được để lại cho người quản lý trước khi vào kỳ thi.

Khi vào phòng thi, bạn sẽ được thông báo sơ lược về phần mềm thi. Bạn có thể được yêu cầu hoàn thành một cuộc khảo sát. Thời gian bạn dành cho khảo sát không bị trừ vào thời gian kiểm tra thực tế của bạn – bạn cũng không có thêm thời gian nếu điền nhanh vào khảo sát. Ngoài ra, hãy nhớ các câu hỏi bạn nhận được trong kỳ thi sẽ không thay đổi tùy thuộc vào cách bạn trả lời các câu hỏi khảo sát. Sau khi bạn hoàn thành cuộc khảo sát, đồng hồ thực bắt đầu tích tắc và cuộc vui bắt đầu.

Phần mềm kiểm tra cho phép bạn tiến và lùi giữa các câu hỏi. Quan trọng nhất, có một hộp kiểm Mark trên màn hình – đây sẽ là một công cụ quan trọng, như được giải thích trong phần tiếp theo.

Kỹ thuật làm bài kiểm tra

Nếu không có kế hoạch tấn công, thí sinh có thể bị choáng ngợp bởi kỳ thi hoặc

Nếu không có kế hoạch tấn công, thí sinh có thể bị choáng ngợp trong kỳ thi hoặc trở nên lạc lõng và hết thời gian. Đôi với hầu hết các phần, nếu bạn cảm thấy thoải mái với tài liệu, thời gian được phân bổ là quá đủ để hoàn thành bài thi. Bí quyết là giữ cho thời gian không bị trôi đi trong bất kỳ một vấn đề cụ thể nào.

Mục tiêu rõ ràng của bạn là trả lời các câu hỏi một cách chính xác và nhanh chóng, nhưng các yếu tố khác có thể làm bạn mất tập trung. Dưới đây là một số mẹo để làm bài thi hiệu quả hơn.

Tăng kích thước thách thức

Đầu tiên, hãy lướt nhanh qua tất cả các câu hỏi trong đề thi. "Chọn" những câu hỏi dễ, trả lời ngay tại chỗ. Đọc ngắn gọn từng câu hỏi, chú ý loại câu hỏi và chủ đề. Theo nguyên tắc, hãy cố gắng dành ít hơn 25 phần trăm thời gian thử nghiệm của bạn trong lần vượt qua này.

Bước này cho phép bạn đánh giá phạm vi và độ phức tạp của kỳ thi, đồng thời giúp bạn xác định cách tăng tốc thời gian của mình. Nó cũng cung cấp cho bạn ý tưởng về nơi để tìm câu trả lời tiềm năng cho một số câu hỏi. Đôi khi cách diễn đạt của một câu hỏi có thể cho bạn biết manh mối hoặc khiến bạn suy nghĩ về một câu hỏi khác.

Nếu bạn không hoàn toàn tự tin vào câu trả lời của mình cho một câu hỏi, hãy trả lời câu hỏi đó, nhưng hãy chọn hộp Đánh dấu để gắn cờ câu hỏi đó để xem xét sau. Trong trường hợp bạn hết thời gian, ít nhất bạn đã cung cấp câu trả lời "phỏng đoán đầu tiên", thay vì để trống.

Thứ hai, quay lại toàn bộ bài kiểm tra, sử dụng thông tin chi tiết bạn có được từ lần xem đầu tiên. Ví dụ: nếu toàn bộ bài kiểm tra có vẻ khó, bạn sẽ biết tốt hơn là dành hơn một phút cho mỗi câu hỏi. Tạo nhịp độ với các cột mốc nhỏ – ví dụ: "Tôi cần trả lời 10 câu hỏi sau mỗi 15 phút".

Ở giai đoạn này, có lẽ bạn nên bỏ qua những câu hỏi tốn nhiều thời gian, đánh dấu chúng cho lần vượt qua tiếp theo. Cố gắng hoàn thành giai đoạn này trước khi bạn đạt 50 đến 60 phần trăm trong thời gian thử nghiệm.

Thứ ba, quay lại tất cả các câu hỏi bạn đã đánh dấu để xem xét, sử dụng nút Đánh giá Đã đánh dấu trong màn hình đánh giá câu hỏi. Bước này bao gồm việc xem xét lại tất cả các câu hỏi mà bạn không chắc chắn trong các lần vượt qua trước đó, cũng như giải quyết các câu hỏi mất thời gian mà bạn đã trì hoãn cho đến nay. Xem xét nhóm câu hỏi này cho đến khi bạn trả lời được tất cả.

Nếu bạn cảm thấy thoải mái hơn với câu hỏi đã đánh dấu trước đó, hãy bỏ đánh dấu Xem lại nút Đã đánh dấu ngay bây giờ. Nếu không, hãy để nó được đánh dấu. Làm theo cách của bạn để giải quyết các câu hỏi tốn thời gian ngay bây giờ, đặc biệt là những câu hỏi yêu cầu thủ công

các phép tính. Bỏ đánh dấu chúng khi bạn hài lòng với câu trả lời.

Đến cuối bước này, bạn đã trả lời mọi câu hỏi trong bài kiểm tra, mặc dù bạn có dễ dàng một số câu trả lời của mình. Nếu bạn hết thời gian ở bước tiếp theo, ít nhất bạn sẽ không bị mất điểm vì thiếu câu trả lời. Bạn đang ở trong trạng thái tuyệt vời nếu bạn vẫn còn 10 đến 20 phần trăm thời gian của mình.

Xem lại câu trả lời của bạn

Bây giờ bạn đang bay! Bạn đã trả lời tất cả các câu hỏi và bạn đã sẵn sàng để kiểm tra chất lượng. Thực hiện thêm một lần nữa (vâng, một lần nữa) qua toàn bộ bài kiểm tra, đọc lại ngắn gọn từng câu hỏi và câu trả lời của bạn.

Cẩn thận xem lại các câu hỏi để kiểm tra các câu hỏi "lừa". Hãy đặc biệt cảnh giác với những thứ có lựa chọn "Không biên dịch". Hãy tinh táo để tìm manh mối vào phút cuối. Bạn đã khá quen thuộc với hầu hết mọi câu hỏi tại thời điểm này và bạn có thể tìm thấy một vài manh mối mà bạn đã bỏ qua trước đây.

Vòng chung kết

Khi bạn tự tin với tất cả các câu trả lời của mình, hãy kết thúc bài kiểm tra bằng cách gửi nó để chấm điểm. Sau khi hoàn thành bài kiểm tra của mình, bạn sẽ nhận được e-mail từ Oracle cung cấp cho bạn một liên kết đến một trang có kết quả bài kiểm tra của bạn. Đối với văn bản này, bạn phải yêu cầu một chứng chỉ bản cứng cụ thể nếu không sẽ không được gửi cho bạn.

Kiểm tra lại

Nếu bạn không vượt qua kỳ thi, đừng nản lòng. Có gắng có thái độ tốt về trải nghiệm và sẵn sàng thử lại. Hãy xem xét bản thân có học thức hơn một chút. Bạn sẽ biết hình thức của bài kiểm tra tốt hơn một chút và bạn sẽ biết rõ về mức độ khó của các câu hỏi mà bạn sẽ gặp trong lần tới.

Nếu bạn trả lại nhanh chóng, có thể bạn sẽ nhớ một số câu hỏi mà bạn có thể đã bỏ qua. Điều này sẽ giúp bạn tập trung nỗ lực học tập vào đúng lĩnh vực.

Cuối cùng, hãy nhớ rằng các chứng chỉ Oracle rất có giá trị vì chúng rất khó để có được. Rốt cuộc, nếu ai đó có thể nhận được một cái, thì nó sẽ có giá trị gì? Cuối cùng, cần phải có thái độ tốt và học tập nhiều thì bạn mới có thể làm được!

Bản đồ mục tiêu

Bảng sau đây mô tả các mục tiêu của kỳ thi và nơi bạn sẽ tìm thấy các mục tiêu cho chúng trong cuốn sách. (Lưu ý: Chúng tôi đã tóm tắt một số mô tả bạn sẽ tìm thấy trên [Oracle.com](https://oracle.com) trang mạng.)

Lập trình viên Java SE 8 được chứng nhận của Oracle
(Bài kiểm tra 1Z0-808)

Exam Objective	Study Guide Coverage
Java Basics	
Define the scope of variables (1.1)	Chapter 3
Define the structure of a Java class (1.2)	Chapters 1 and 2
Create executable Java applications with a main method (1.3)	Chapter 1
Import other Java packages to make them accessible in your code (1.4)	Chapter 1
Compare and contrast features of Java (1.5)	Chapter 1
Working with Java Data Types	
Declare and initialize variables (2.1)	Chapters 1 and 3
Differentiate between object reference variables and primitive variables (2.2)	Chapters 1, 2, and 3
Know how to read or write to object fields (2.3)	Whole book
Explain an object's lifecycle (creation, "dereference," and garbage collection) (2.4)	Chapter 3
Use wrapper classes such as Boolean, Double, Integer (2.5)	Chapter 6
Using Operators and Decision Constructs	
Use Java operators (3.1)	Chapters 1 and 4
Test equality between Strings and other objects using == and equals() (3.2)	Chapters 1 and 4
Create if and if/else and ternary constructs (3.4)	Chapters 4 and 5
Use a switch statement (3.5)	Chapter 5
Creating and Using Arrays	
Declare, instantiate, initialize and use a one-dimensional array (4.1)	Chapters 3 and 6
Declare, instantiate, initialize and use multi-dimensional arrays (4.2)	Chapters 3 and 6
Using Loop Constructs	
Create and use while loops (5.1)	Chapter 5
Create and use for loops including the enhanced for loop (5.2)	Chapter 5
Create and use do/while loops (5.3)	Chapter 5
Compare loop constructs (5.4)	Chapter 5
Use break and continue (5.5)	Chapter 5

Exam Objective	Study Guide Coverage
Working with Methods and Encapsulation	
Create methods with arguments and return values (6.1)	Chapter 2
Apply the static keyword to methods and fields (6.2)	Chapters 1 and 2
Create and overload constructors (6.3)	Chapters 1 and 2
Apply access modifiers (6.4)	Chapter 1
Apply encapsulation principles to a class (6.5)	Chapters 2 and 6
Determine the effect upon object references and primitive values when they are passed into methods that change the values (6.6)	Chapter 3
Working with Inheritance	
Describe inheritance and its benefits (7.1)	Chapter 2
Develop code that demonstrates the use of polymorphism (7.2)	Chapter 2
Determine when casting is necessary (7.3)	Chapter 2
Use super and this to access objects and constructors (7.4)	Chapter 2
Use abstract classes and interfaces (7.5)	Chapters 1 and 2
Handling Exceptions	
Differentiate among checked exceptions, RuntimeExceptions, and Errors (8.1)	Chapter 5
Create a try-catch block and determine how exceptions alter normal program flow (8.2)	Chapter 5
Describe the advantages of Exception handling (8.3)	Chapter 5
Create and invoke a method that throws an exception (8.4)	Chapter 5
Recognize common exception classes (8.5)	Chapter 5
Working with Selected Classes from the Java API	
Manipulate data using the StringBuilder class (9.1)	Chapter 6
Create and manipulate Strings (9.2)	Chapter 6
Create and manipulate calendar data (9.3)	Chapter 6
Declare and use an ArrayList (9.4)	Chapter 6
Write a simple Lambda expression that consumes a Lambda Predicate expression (9.5)	Chapter 6



1

Khai báo và Kiểm soát truy cập

CHỨNG NHẬN

MỤC TIÊU

- Các tính năng và lợi ích của Java
- Định danh và Từ khóa • javac,
java, main () và Nhập • Khai báo các lớp và
giao diện
- Khai báo các thành viên trong lớp
- Khai báo hàm tạo và mảng • Tạo thành viên
lớp tĩnh
- Sử dụng enums

✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

W

e giả sử rằng vì bạn đang có kế hoạch trở nên được chứng nhận, bạn đã biết những điều cơ bản về Java. Nếu bạn hoàn toàn mới về ngôn ngữ, chương này – và phần còn lại của sách – sẽ gây nhầm lẫn; vì vậy hãy chắc chắn rằng bạn biết ít nhất những điều cơ bản về ngôn ngữ trước khi đi sâu vào cuốn sách này. Điều đó nói rằng, chúng tôi đang bắt đầu với một bản cập nhật ngắn gọn, cấp cao để đưa bạn trở lại với tâm trạng Java, trong trường hợp bạn đã vắng mặt một thời gian.

Trình làm mới Java

Một chương trình Java chủ yếu là một tập hợp các đối tượng nói chuyện với các đối tượng khác bằng cách gọi các phương thức của nhau. Mọi đối tượng thuộc một kiểu nhất định và kiểu đó được định nghĩa bởi một lớp hoặc một giao diện. Hầu hết các chương trình Java sử dụng một tập hợp các đối tượng thuộc nhiều kiểu khác nhau. Sau đây là danh sách một số thuật ngữ hữu ích cho ngôn ngữ hướng đối tượng (OO) này:

- **Mẫu Class** Một mẫu mô tả các loại trạng thái và hành vi mà các đối tượng thuộc loại của nó hỗ trợ.
- **Đối tượng** Trong thời gian chạy, khi Máy ảo Java (JVM) gặp từ khóa mới , nó sẽ sử dụng lớp thích hợp để tạo một đối tượng là một thể hiện của lớp đó. Đối tượng đó sẽ có trạng thái riêng và quyền truy cập vào tất cả các hành vi được định nghĩa bởi lớp của nó.
- **Trạng thái (các biến thể hiện)** Mỗi đối tượng (thể hiện của một lớp) sẽ có một tập hợp các biến thể hiện duy nhất của riêng nó như được định nghĩa trong lớp. Nói chung, các giá trị được gán cho các biến phiên bản của đối tượng tạo nên tiều bang.
- **Hành vi (các phương thức)** Khi một lập trình viên tạo một lớp, cô ấy sẽ tạo các phương thức cho lớp đó. Các phương thức là nơi lưu trữ logic của lớp và là nơi công việc thực sự được thực hiện. Chúng là nơi các thuật toán được thực thi và dữ liệu được thao tác.

Số nhận dạng và Từ khóa Tất cả các

thành phần Java mà chúng ta vừa nói đến – các lớp, biến và phương thức – các tên cần thiết. Trong Java, những tên này được gọi là số nhận dạng, và như bạn có thể mong đợi, có các quy tắc cho những gì cấu thành một mã định danh Java hợp pháp. Tuy nhiên, ngoài những gì hợp pháp, các lập trình viên Java (và Oracle) đã tạo ra các quy ước để đặt tên cho các phương thức, biến và lớp.

Giống như tất cả các ngôn ngữ lập trình, Java có một tập hợp các từ khóa cài sẵn. Những từ khóa này không được sử dụng làm định danh. Ở phần sau của chương này, chúng ta sẽ xem xét chi tiết về các quy tắc đặt tên, quy ước và các từ khóa Java này.

Di sản

Trung tâm của Java và các ngôn ngữ OO khác là khái niệm kế thừa, cho phép mã được xác định trong một lớp hoặc giao diện được sử dụng lại trong các lớp khác. Trong Java, bạn có thể định nghĩa một lớp cha tổng quát (trừu tượng hơn) và sau đó mở rộng nó với các lớp con cụ thể hơn. Lớp cha không biết gì về các lớp mà

ké thừa từ nó, nhưng tất cả các lớp con kế thừa từ lớp cha phải khai báo rõ ràng mỗi quan hệ kế thừa. Lớp con kế thừa từ lớp cha được tự động cung cấp các biến cá thể có thể truy cập và các phương thức được định nghĩa bởi lớp cha, nhưng lớp con cũng có thể tự do ghi đè các phương thức của lớp cha để xác định hành vi cụ thể hơn. Ví dụ, một lớp siêu xe có thể xác định các phương thức chung cho tất cả các loại ô tô, nhưng một lớp con Ferrari có thể ghi đè lên phương thức speed () đã được định nghĩa trong lớp Car .

Giao diện

Một người bạn đồng hành mạnh mẽ với sự kế thừa là việc sử dụng các giao diện. Các giao diện thường giống như một lớp cha trừu tượng 100 phần trăm xác định các phương thức mà một lớp con phải hỗ trợ, nhưng không phải cách chúng phải được hỗ trợ. Nói cách khác, ví dụ, một giao diện Động vật có thể khai báo rằng tất cả các lớp thực thi Động vật đều có phương thức eat () , nhưng giao diện Động vật không cung cấp bất kỳ logic nào cho phương thức eat () . Điều đó có nghĩa là tùy thuộc vào các lớp triển khai giao diện Động vật để xác định mã thực cho cách loại Động vật cụ thể đó hoạt động như thế nào khi phương thức eat () của nó được gọi. Lưu ý: Kể từ Java 8, các giao diện hiện có thể bao gồm các phương thức cụ thể, có thể kế thừa. Chúng ta sẽ nói nhiều hơn về vấn đề này khi chúng ta đi sâu vào 00 trong chương tiếp theo.

MỤC TIÊU XÁC NHẬN

Các tính năng và lợi ích của Java (Mục tiêu 1.5 của OCA)

1.5 So sánh và đối chiếu các tính năng và thành phần của Java như: tính độc lập của nền tảng, hướng đối tượng, tính đóng gói, v.v.

Có lẽ một chủ đề tuyệt vời để bắt đầu, trong phạm vi chính thức của chúng tôi về OCA 8, là thảo luận về những lợi ích khác nhau mà Java cung cấp cho các lập trình viên. Java hiện đã hơn 20 tuổi (wow!) Và vẫn là một trong những ngôn ngữ lập trình được yêu cầu nhiều nhất trên thế giới. Hơi khó hiểu khi có một ngôn ngữ được đặt tên tương tự, "JavaScript" (một cách triển khai tiêu chuẩn ECMA), cũng là một ngôn ngữ rất phổ biến. Java và JavaScript có một số khía cạnh chung, nhưng chúng không được nhầm lẫn; chúng khá khác biệt. Chúng ta hãy nhìn vào

một số lợi ích mà Java cung cấp cho các lập trình viên và so sánh chúng (khi thích hợp) với một số đối thủ cạnh tranh của Java. Lưu ý ở đây, nhiều lợi ích này dựa trên các chủ đề cực kỳ phức tạp. Những mô tả này không có nghĩa là chắc chắn, nhưng chúng đủ cho kỳ thi:

- Hướng đối tượng Khi các hệ thống phần mềm ngày càng lớn, chúng sẽ khó kiểm tra và nâng cao hơn. Trong vài thập kỷ qua, lập trình hướng đối tượng (OO) đã là phương pháp thiết kế phần mềm thống trị cho các hệ thống lớn, bởi vì các hệ thống OO được thiết kế tốt vẫn có thể kiểm tra và nâng cao được, ngay cả khi chúng phát triển thành các ứng dụng khổng lồ với hàng triệu dòng mã. Thiết kế OO cũng cung cấp một cách tự nhiên để suy nghĩ về cách các thành phần trong hệ thống nên được xây dựng và cách chúng tương tác với nhau. Các lớp, đối tượng, trạng thái hệ thống và hành vi trong hệ thống OO được thiết kế tốt dễ dàng ánh xạ về mặt khái niệm với các đối tác của chúng trong thế giới thực.
- Đóng gói Đóng gói là một khái niệm quan trọng trong lập trình OO. Tính năng đóng gói cho phép một thành phần phần mềm ẩn dữ liệu của nó khỏi các thành phần khác, bảo vệ dữ liệu không bị cập nhật mà không có sự chấp thuận hoặc biết của thành phần đó. Java làm cho việc đóng gói dễ dàng đạt được hơn nhiều so với các ngôn ngữ không phải OO.
- Quản lý bộ nhớ Không giống như một số đối thủ cạnh tranh của nó (C và C ++), Java cung cấp tính năng quản lý bộ nhớ tự động. Trong các ngôn ngữ không cung cấp tính năng quản lý bộ nhớ tự động, việc theo dõi bộ nhớ thông qua các con trỏ là khá phức tạp. Hơn nữa, theo dõi các lỗi liên quan đến quản lý bộ nhớ (thường được gọi là rò rỉ bộ nhớ) là một quá trình phổ biến, dễ xảy ra lỗi và tốn thời gian.
- Thư viện khổng lồ Java có một thư viện khổng lồ gồm các mã được viết sẵn, được kiểm tra tốt và được hỗ trợ tốt. Mã này dễ dàng đưa vào các ứng dụng Java của bạn và được ghi lại đầy đủ thông qua API Java. Trong suốt cuốn sách này, chúng ta sẽ khám phá một số thành viên được sử dụng nhiều nhất (và hữu ích nhất) trong thư viện lõi tiêu chuẩn của Java.
- Bảo mật theo thiết kế Khi mã Java đã biên dịch được thực thi, nó sẽ chạy bên trong Máy ảo Java (JVM). JVM cung cấp một "hộp cá" an toàn để mã Java của bạn chạy vào và JVM đảm bảo rằng các lập trình viên bất chính không thể viết mã Java sẽ gây ra sự cố trên máy của người khác khi nó chạy.
- Viết một lần, chạy ở mọi nơi (thực thi đa nền tảng) Một trong những mục tiêu

(phần lớn, nhưng không đạt được hoàn hảo) của Java là phần lớn mã Java bạn viết có thể chạy trên nhiều nền tảng, từ các thiết bị Internet-of Things (IoT) nhỏ bé, đến điện thoại, máy tính xách tay, máy chủ lớn.

Một cụm từ phổ biến khác để chỉ khả năng này chạy trên nhiều thiết bị là nền tảng chéo.

- Được gõ mạnh Một ngôn ngữ được gõ mạnh thường yêu cầu người lập trình phải khai báo rõ ràng các kiểu dữ liệu và đối tượng đang được sử dụng trong chương trình. Gõ mạnh cho phép trình biên dịch Java bắt được nhiều lỗi lập trình tiềm ẩn trước khi mã của bạn được biên dịch. Ở đầu kia của phổ là các ngôn ngữ được nhập động.

Các ngôn ngữ được gõ động có thể ít dài dòng hơn, viết mã ban đầu nhanh hơn và thường được ưa thích trong môi trường nơi các nhóm nhỏ và tạo mẫu nhanh là tiêu chuẩn. Nhưng các ngôn ngữ được đánh máy mạnh như Java trở thành ngôn ngữ riêng của chúng trong các cửa hàng phần mềm lớn với nhiều đội ngũ lập trình viên và nhu cầu về mã chất lượng sản xuất, có thể kiểm tra và đáng tin cậy hơn.

- Java đa luồng cung cấp các tính năng ngôn ngữ và API tích hợp sẵn cho phép các chương trình sử dụng nhiều quy trình của hệ điều hành (do đó, nhiều "lỗi") cùng một lúc. Khi các hệ thống phát triển để xử lý các vấn đề tính toán chuyên sâu hơn và tập dữ liệu lớn hơn, khả năng sử dụng tất cả các bộ xử lý cốt lõi của máy tính trở nên cần thiết. Lập trình đa luồng không bao giờ là đơn giản, nhưng Java cung cấp một bộ công cụ phong phú để làm cho nó dễ dàng nhất có thể.
- Tính toán phân tán Một cách khác để giải quyết các vấn đề lập trình lớn là phân phối khỏi lượng công việc trên nhiều máy. API Java cung cấp một số cách để đơn giản hóa các tác vụ liên quan đến tính toán phân tán.
Một ví dụ như vậy là tuần tự hóa, một quá trình trong đó một đối tượng Java được chuyển đổi thành một dạng di động. Các đối tượng được tuần tự hóa có thể được gửi đến các máy khác, được giải mã hóa, và sau đó được sử dụng như một đối tượng Java bình thường.

Một lần nữa, chúng tôi mới chỉ sơ lược về bề mặt của những chủ đề phức tạp này, nhưng nếu bạn hiểu những mô tả ngắn gọn này, bạn nên chuẩn bị để xử lý bất kỳ câu hỏi nào cho mục tiêu này.

Quá nhiều về lý thuyết, chúng ta hãy đi vào chi tiết.

MỤC TIÊU XÁC NHẬN

Số nhận dạng và Từ khóa (Mục tiêu OCA)

1.2 và 2.1)

1.2 Xác định cấu trúc của một lớp Java.

2.1 Khai báo và khởi tạo biến (bao gồm ép kiểu dữ liệu nguyên thủy).

Hãy nhớ rằng khi chúng tôi liệt kê một hoặc nhiều Mục tiêu Chúng nhận trong sách, như chúng tôi vừa làm, điều đó có nghĩa là phần sau đây bao gồm ít nhất một phần nào đó của mục tiêu đó. Một số mục tiêu sẽ được đề cập trong một số chương khác nhau, vì vậy bạn sẽ thấy cùng một mục tiêu ở nhiều nơi trong cuốn sách. Ví dụ: phần này bao gồm khai báo và định danh, nhưng việc sử dụng những thứ bạn khai báo chủ yếu được đề cập trong các chương sau.

Vì vậy, chúng ta sẽ bắt đầu với các mã định danh Java. Hai khía cạnh của số nhận dạng Java mà chúng tôi đề cập ở đây là

- Định danh hợp pháp Các quy tắc mà trình biên dịch sử dụng để xác định xem tên có hợp pháp hay không.
- Các quy ước về mã Java của Oracle Các khuyến nghị của Oracle về cách đặt tên các lớp, biến và phương thức. Chúng tôi thường tuân thủ các tiêu chuẩn này trong suốt cuốn sách, ngoại trừ khi chúng tôi đang cố gắng chỉ cho bạn cách mã hóa một câu hỏi kiểm tra khó. Bạn sẽ không được hỏi các câu hỏi về Quy ước mã Java, nhưng chúng tôi thực sự khuyên bạn nên sử dụng chúng.

Định danh pháp lý

Về mặt kỹ thuật, số nhận dạng hợp pháp chỉ được bao gồm các ký tự Unicode, số, ký hiệu tiền tệ và các ký tự kết nối (chẳng hạn như dấu gạch dưới).

Bài kiểm tra không đi sâu vào chi tiết về phạm vi nào của bộ ký tự Unicode đủ điều kiện là các chữ cái và chữ số. Vì vậy, chẳng hạn, bạn sẽ không cần biết rằng các chữ số Tây Tạng nằm trong khoảng từ \ u0420 đến \ u0f29. Dưới đây là các quy tắc bạn cần biết:

- Số nhận dạng phải bắt đầu bằng một chữ cái, một ký tự tiền tệ (\$) hoặc một ký tự kết nối chẳng hạn như dấu gạch dưới (_). Số nhận dạng không được bắt đầu bằng một chữ số!
- Sau ký tự đầu tiên, số nhận dạng có thể chứa bất kỳ sự kết hợp nào của các chữ cái, ký tự tiền tệ, ký tự kết nối hoặc số.
- Trên thực tế, không có giới hạn về số ký tự mà một mã định danh có thể chứa.

■ Bạn không thể sử dụng từ khóa Java làm số nhận dạng. [Bảng 1-1](#) liệt kê tất cả các từ khóa Java.

BẢNG 1-1

dưới đây đầy đủ các từ khóa Java (khẳng định được thêm vào 1.4, enum được thêm vào 1.5)

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

■ Định danh trong Java phân biệt chữ hoa chữ thường; foo và FOO là hai định danh khác nhau.

Sau đây là các ví dụ về số nhận dạng hợp pháp và bất hợp pháp. Đầu tiên là một số định danh pháp lý:

```
int _a;
int $c;
int _____2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

Những điều sau đây là bất hợp pháp (nhiệm vụ của bạn là nhận ra lý do tại sao):

```
int :b;
int -d;
int e#;
int .f;
int 7g;
```

Quy ước mã Java của Oracle

Oracle ước tính rằng trong suốt thời gian tồn tại của một đoạn mã tiêu chuẩn, 20% nỗ lực sẽ dành cho việc tạo và thử nghiệm mã ban đầu, và 80% nỗ lực sẽ dành cho việc bảo trì và nâng cao mã sau đó. Đồng ý và mã hóa một bộ tiêu chuẩn mã giúp giảm thiểu nỗ lực liên quan đến việc thử nghiệm, duy trì và cải tiến bất kỳ đoạn mã nào.

Oracle đã tạo ra một tập hợp các tiêu chuẩn mã hóa cho Java và xuất bản các tiêu chuẩn đó trong một tài liệu có tiêu đề khéo léo là "Các quy ước về mã Java", mà bạn có thể tìm thấy nếu bắt đầu tại java.oracle.com. Đó là một tài liệu tuyệt vời, ngắn gọn và dễ đọc và chúng tôi đánh giá cao.

Điều đó nói rằng, bạn sẽ thấy rằng nhiều câu hỏi trong bài kiểm tra không tuân theo quy ước mã vì những hạn chế trong công cụ kiểm tra được sử dụng để cung cấp kỳ thi quốc tế. Một trong những điều tuyệt vời về chứng chỉ Oracle là các kỳ thi được tổ chức thống nhất trên toàn thế giới.

Để đạt được điều đó, các danh sách mã mà bạn sẽ thấy trong kỳ thi thực thường khá chật chội và không tuân theo các tiêu chuẩn mã của Oracle. Để giúp bạn khó khăn hơn cho kỳ thi, chúng tôi thường trình bày danh sách mã có giao diện và cảm giác chật chội tương tự, thường chỉ thay đổi lè lưỡi của chúng tôi hai khoảng trống so với tiêu chuẩn của Oracle là bốn.

Chúng ta cũng sẽ ghép các dấu ngoặc nhọn vào nhau một cách không tự nhiên, và đôi khi chúng ta sẽ đặt một số câu lệnh trên cùng một dòng. ouch! Ví dụ:

```

1. class Wombat implements Runnable {
2.     private int i;
3.     public synchronized void run() {
4.         if (i%5 != 0) { i++; }
5.         for(int x=0; x<5; x++, i++)
6.             { if (x > 1) Thread.yield(); }
7.         System.out.print(i + " ");
8.     }
9.     public static void main(String[] args) {
10.        Wombat n = new Wombat();
11.        for(int x=100; x>0; --x) { new Thread(n).start(); }
12.    } }
```

Hãy xem xét bản thân đã được cảnh báo trước – bạn sẽ thấy rất nhiều danh sách mã, câu hỏi giả và đề thi thực sự là bệnh hoạn và khó hiểu này. Không ai muốn bạn viết mã của mình như thế này – không phải nhà tuyển dụng của bạn, không phải đồng nghiệp của bạn, không phải chúng tôi, không phải Oracle và không phải nhóm tạo bài kiểm tra! Mã như thế này chỉ được tạo ra để các khái niệm phức tạp có thể được kiểm tra trong một công cụ kiểm tra toàn cầu. Tiêu chuẩn duy nhất được tuân thủ nhiều nhất có thể trong kỳ thi thực là tiêu chuẩn đặt tên. Dưới đây là các tiêu chuẩn đặt tên mà Oracle đề xuất và chúng tôi sử dụng trong bài kiểm tra cũng như trong hầu hết các cuốn sách:

- Lớp và giao diện Chữ cái đầu tiên phải được viết hoa và nếu một số từ được liên kết với nhau để tạo thành tên, thì chữ cái đầu tiên của các từ bên trong phải là chữ hoa (một định dạng đôi khi được gọi là

"CamelCase"). Đối với các lớp, tên thường phải là danh từ. Dưới đây là một số ví dụ:

Chú ý
Tài khoản
PrintWriter

Đối với giao diện, tên thường phải là tính từ, như sau:

Runnable
Serializable

- Phương thức Chữ cái đầu tiên phải là chữ thường, sau đó sử dụng các quy tắc CamelCase bình thường. Ngoài ra, tên thường phải là cặp động từ-danh từ. Ví dụ:

getBalance
doCalculation
setCustomerName

- Biến Giống như các phương thức, nên sử dụng định dạng CamelCase, nhưng bắt đầu bằng một chữ cái thường. Oracle đề xuất những cái tên ngắn gọn, có ý nghĩa, nghe có vẻ hợp với chúng ta. Vài ví dụ:

buttonWidth
accountBalance
myString

- Hằng số Hằng số Java được tạo bằng cách đánh dấu các biến tĩnh và biến cuối cùng. Chúng phải được đặt tên bằng chữ hoa với các ký tự gạch dưới làm dấu phân cách:

MIN_HEIGHT

MỤC TIÊU XÁC NHẬN

Xác định các lớp (Mục tiêu OCA 1.2, 1.3, 1.4, 6.4 và 7.5)

1.2 Xác định cấu trúc của một lớp Java.

1.3 Tạo các ứng dụng Java thực thi bằng một phương thức chính; chạy một chương trình Java từ dòng lệnh; bao gồm cả đầu ra bảng điều khiển. (sic)

1.4 Nhập các gói Java khác để làm cho chúng có thể truy cập được trong mã của bạn.

6.4 Áp dụng công cụ sửa đổi quyền truy cập.

7.5 Sử dụng các lớp và giao diện trừu tượng.

Khi bạn viết mã bằng Java, bạn đang viết các lớp hoặc giao diện. Bên trong các lớp đó, như bạn biết, là các biến và phương thức (cộng với một số thứ khác).

Cách bạn khai báo các lớp, phương thức và biến ảnh hưởng đáng kể đến hành vi của mã của bạn. Ví dụ, một phương thức công khai có thể được truy cập từ mã chạy ở bất kỳ đâu trong ứng dụng của bạn. Tuy nhiên, hãy đánh dấu phương thức đó là riêng tư và nó biến mất khỏi radar của mọi người (ngoại trừ lớp mà nó được khai báo).

Đối với mục tiêu này, chúng tôi sẽ nghiên cứu các cách mà bạn có thể khai báo và sửa đổi (hoặc không) một lớp. Bạn sẽ thấy rằng chúng tôi bao gồm các công cụ sửa đổi ở mức độ cực kỳ chi tiết và mặc dù chúng tôi biết bạn đã quen thuộc với chúng, nhưng chúng tôi đang bắt đầu lại từ đầu. Hầu hết các lập trình viên Java nghĩ rằng họ biết tất cả các bước hoạt động như thế nào, nhưng khi nghiên cứu kỹ hơn, họ thường phát hiện ra rằng chúng không (ít nhất là không đến mức độ cần thiết cho kỳ thi). Sự khác biệt tinh tế ở khắp mọi nơi, vì vậy bạn cần hoàn toàn chắc chắn rằng bạn hoàn toàn vững chắc về mọi thứ trong mục tiêu của phần này trước khi tham gia kỳ thi.

Quy tắc khai báo tệp nguồn

Trước khi chúng ta đi sâu vào khai báo lớp, hãy xem xét nhanh các quy tắc được liên kết với khai báo lớp, câu lệnh nhập và câu lệnh gói trong tệp nguồn:

- Chỉ có thể có một lớp công khai cho mỗi tệp mã nguồn.
- Chú thích có thể xuất hiện ở đầu hoặc cuối của bất kỳ dòng nào trong tệp mã nguồn; chúng độc lập với bất kỳ quy tắc định vị nào được thảo luận ở đây.
- Nếu có một lớp công khai trong một tệp, tên của tệp phải khớp với tên của lớp công khai. Ví dụ: một lớp được khai báo là lớp công khai Dog {} phải nằm trong tệp mã nguồn có tên Dog.java.
- Nếu lớp là một phần của gói, câu lệnh gói phải là dòng đầu tiên trong tệp mã nguồn, trước bất kỳ câu lệnh nhập nào có thể có mặt.

- Nếu có các câu lệnh nhập , chúng phải đi giữa câu lệnh gói (nếu có) và khai báo lớp. Nếu không có câu lệnh gói , thì (các) câu lệnh nhập phải là (các) dòng đầu tiên trong tệp mã nguồn. Nếu không có gói hoặc câu lệnh nhập , khai báo lớp phải là dòng đầu tiên trong tệp mã nguồn.
- câu lệnh nhập và gói áp dụng cho tất cả các lớp trong tệp mã nguồn. Nói cách khác, không có cách nào để khai báo nhiều lớp trong một tệp và đặt chúng trong các gói khác nhau hoặc sử dụng các lần nhập khác nhau.
- Một tệp có thể có nhiều hơn một lớp không công khai .
- Tệp không có lớp công khai có thể có tên không khớp với bất kỳ lớp nào trong tệp.

Sử dụng lệnh javac và java

Trong cuốn sách này, chúng ta sẽ nói về việc gọi các lệnh javac và java khoảng 1000 lần. Mặc dù trong thế giới thực , bạn có thể sẽ sử dụng môi trường phát triển tích hợp (IDE) hầu hết thời gian, nhưng bạn có thể thấy một số câu hỏi trong bài kiểm tra sử dụng dòng lệnh để thay thế, vì vậy chúng ta sẽ xem xét các khái niệm cơ bản. (Nhân tiện, chúng tôi KHÔNG sử dụng IDE khi viết cuốn sách này. Chúng tôi vẫn có một chút ưu tiên đối với dòng lệnh trong khi nghiên cứu cho kỳ thi; tất cả các IDE đều cố gắng hết sức để "hữu ích" và đôi khi họ sẽ sửa lỗi của bạn những vấn đề mà không nói cho bạn biết. Công việc đó thật tốt, nhưng có lẽ không quá tuyệt vời khi bạn đang học để thi lấy chứng chỉ!)

Biên dịch với javac Lệnh

javac được sử dụng để gọi trình biên dịch của Java. Bạn có thể chỉ định nhiều tùy chọn khi chạy javac. Ví dụ: có các tùy chọn để tạo thông tin gỡ lỗi hoặc cảnh báo trình biên dịch. Đây là tổng quan về cấu trúc của javac:

```
javac [tùy chọn] [tệp nguồn]
```

Có các tùy chọn dòng lệnh bổ sung được gọi là @argfiles, nhưng chúng hiếm khi được sử dụng và bạn sẽ không cần phải nghiên cứu chúng để làm bài kiểm tra. Cả [tùy chọn] và [tệp nguồn] đều là các phần tùy chọn của lệnh và cả hai đều cho phép nhiều mục nhập. Sau đây là các lệnh javac hợp pháp:

```
javac -help
javac -version Foo.java Bar.java
```

Lời gọi đầu tiên không biên dịch bất kỳ tệp nào, nhưng in ra một bản tóm tắt các tùy chọn hợp lệ. Lời gọi thứ hai chuyển cho trình biên dịch một tùy chọn (-version, in phiên bản của trình biên dịch bạn đang sử dụng) và chuyển cho trình biên dịch hai tệp .java để biên dịch (Foo.java và Bar.java). Bất cứ khi nào bạn chỉ định nhiều tùy chọn và / hoặc tệp, chúng phải được phân tách bằng dấu cách.

Khởi chạy ứng dụng với java Lệnh java
được sử dụng để gọi Máy ảo Java (JVM). Đây là cấu trúc cơ bản của lệnh:

```
java [tùy chọn] lớp [args]
```

Các phần [options] và [args] của lệnh java là tùy chọn và chúng cả hai đều có thể có nhiều giá trị. Bạn phải chỉ định chính xác một tệp lớp để thực thi và lệnh java giả sử bạn đang nói về tệp .class , vì vậy bạn không chỉ định phần mở rộng .class trên dòng lệnh. Đây là một ví dụ:

```
java -showversion MyClass x 1
```

Lệnh này có thể được hiểu là “Cho tôi xem phiên bản JVM đang được sử dụng, sau đó khởi chạy tệp có tên MyClass.class và gửi cho nó hai đối số Chuỗi có giá trị là x và 1”. Hãy xem đoạn mã sau:

```
public class MyClass {
    public static void main(String[] args) {
        System.out.println(args[0] + " " + args[1]);
    }
}
```

Nó được biên dịch và sau đó được gọi như sau:

```
java MyClass x 1
```

Đầu ra sẽ là

```
x 1
```

Chúng ta sẽ đi sâu vào các mảng sau, nhưng bây giờ đủ để biết args đó – giống như tất cả các mảng – sử dụng chỉ mục dựa trên 0. Nói cách khác, đối số dòng lệnh đầu tiên được gán cho args [0], đối số thứ hai được gán cho args [1], v.v.

Sử dụng public static void main (String [] args)

Việc sử dụng phương thức main () được ngụ ý trong hầu hết các câu hỏi của kỳ thi và trong kỳ thi OCA, phương thức này được đề cập cụ thể. Đối với .0001 phần trăm bạn không biết, main () là phương thức mà JVM sử dụng để bắt đầu thực thi một chương trình Java.

Trước hết, điều quan trọng là bạn phải biết rằng việc đặt tên cho một phương thức là main () không cung cấp cho nó các siêu cường mà chúng ta thường liên kết với main (). Theo như trình biên dịch và JVM có liên quan, phiên bản duy nhất của main () với siêu năng lực là main () có chữ ký này:

```
public static void main (String [] args)
```

Các phiên bản khác của main () với các chữ ký khác là hoàn toàn hợp pháp, nhưng chúng được coi như các phương thức bình thường. Có một số tính linh hoạt trong việc khai báo phương thức main () "đặc biệt" (phương thức được sử dụng để khởi động ứng dụng Java): thứ tự của các bở ngữ của nó có thể được thay đổi một chút; mảng Chuỗi không nhất thiết phải được đặt tên là args; và nó có thể được khai báo bằng cú pháp varargs. Sau đây là tất cả các khai báo pháp lý cho main "đặc biệt" ():

```
static public void main(String[] args)
public static void main(String... x)
static public void main(String bang_a_gong[] )
```

Đối với kỳ thi OCA 8, điều quan trọng duy nhất mà bạn cần biết là main () đó có thể bị quá tải. Chúng tôi sẽ trình bày chi tiết về quá tải trong chương tiếp theo.

Câu lệnh nhập khẩu và API Java

Có một số lớp Java gazillion trên thế giới. API Java có hàng nghìn lớp và cộng đồng Java đã viết phần còn lại. Chúng ta sẽ đi ra ngoài và tranh luận rằng tất cả các lập trình viên Java ở khắp mọi nơi đều sử dụng kết hợp các lớp mà họ đã viết và các lớp mà các lập trình viên khác đã viết. Giả sử chúng ta đã tạo như sau:

tiếp theo:

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

Đây là một lớp hoàn toàn hợp pháp, nhưng hóa ra, một trong những lớp được sử dụng phổ biến nhất trong Java API cũng được đặt tên là ArrayList, hoặc có vẻ như.. Tên thực của phiên bản API là java.util.ArrayList. Đó là tên đầy đủ của nó.

Việc sử dụng các tên đúp điều kiện là điều giúp các nhà phát triển Java đảm bảo rằng hai phiên bản của một lớp như ArrayList không bị nhầm lẫn. Vì vậy, bây giờ hãy nói rằng tôi muốn sử dụng lớp ArrayList từ API:

```
public class MyClass {
    public static void main(String[] args) {
        java.util.ArrayList<String> a =
            new java.util.ArrayList<String>();
    }
}
```

(Trước hết, hãy tin tưởng chúng tôi về cú pháp <Chuỗi> ; chúng ta sẽ làm điều đó sau.) Mặc dù điều này là hợp pháp, nhưng nó cũng có RẤT NHIỀU lần nhấn phím. Vì các lập trình viên của chúng tôi về cơ bản là lười biếng (ở đó, chúng tôi đã nói rồi), chúng tôi thích sử dụng các lớp của người khác RẤT NHIỀU VÀ chúng tôi ghét nhập. Nếu chúng ta có một chương trình lớn, chúng ta có thể sử dụng ArrayLists nhiều lần. nhập báo cáo để giải cứu! Thay vì mã trước đó, lớp của chúng tôi có thể trông như thế này:

```
import java.util.ArrayList;
public class MyClass {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
    }
}
```

Chúng ta có thể diễn giải câu lệnh nhập như sau: "Trong Java API có một gói được gọi là 'use', và trong gói đó là một lớp có tên là 'ArrayList'.

Bất cứ khi nào bạn nhìn thấy từ 'ArrayList' trong lớp này, đó chỉ là viết tắt của: 'java.util.ArrayList'. " (Lưu ý: Còn nhiều gói hơn nữa!) Nếu bạn là một lập trình viên C, bạn có thể nghĩ rằng câu lệnh import tương tự như #include. Không hẳn vậy. Tất cả những gì một câu lệnh nhập Java làm là giúp bạn tiết kiệm một số thao tác nhập. Đó là nó.

Như chúng ta vừa ngụ ý, một gói thường có nhiều lớp. Việc nhập khẩu

tuyên bố cung cấp một khả năng tiết kiệm phím bấm khác. Giả sử bạn muốn sử dụng một vài lớp khác nhau từ gói `java.util` : `ArrayList` và `TreeSet`. Bạn có thể thêm một ký tự đại diện (*) vào câu lệnh nhập của mình , nghĩa là “Nếu bạn thấy một tham chiếu đến một lớp mà bạn không chắc chắn, bạn có thể xem qua toàn bộ gói cho lớp đó,” như sau:

```
import java.util.*;
public class MyClass {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        TreeSet<String> t = new TreeSet<String>();
    }
}
```

Khi trình biên dịch và JVM nhìn thấy mã này, họ sẽ biết cách xem qua `java.util` cho `ArrayList` và `TreeSet`. Đối với bài kiểm tra, điều cuối cùng bạn cần nhớ về việc sử dụng câu lệnh nhập trong các lớp học của mình là bạn có thể tự do trộn và kết hợp. Có thể nói điều này:

```
ArrayList<String> a = new ArrayList<String>();
java.util.ArrayList<String> a2 = new java.util.ArrayList<String>();
```

Báo cáo nhập khẩu tĩnh

Bạn đọc thân mến, Chúng tôi thực sự gặp khó khăn trong việc đưa cuộc thảo luận về nhập tĩnh này vào đâu. Ở góc độ học tập, đây có lẽ không phải là địa điểm lý tưởng, nhưng ở góc độ tham khảo, chúng tôi nghĩ nó có lý. Khi bạn đang tìm hiểu tài liệu lần đầu tiên, bạn có thể bối rối bởi một số ý tưởng trong phần này. Nếu đúng như vậy, chúng tôi xin lỗi. Đặt một ghi chú dính trên trang này và khoanh tròn lại sau khi bạn học xong [Chương 3](#). Mặt khác, khi bạn đã qua giai đoạn học tập và bạn đang sử dụng cuốn sách này làm tài liệu tham khảo, chúng tôi nghĩ nên đặt phần này ở đây sẽ khá hữu ích. Böyle giờ, chuyển sang nhập tĩnh.

Đôi khi các lớp sẽ chứa các thành viên tĩnh. (Chúng ta sẽ nói nhiều hơn về tĩnh các thành viên lớp sau đó, nhưng vì chúng ta đang nói về chủ đề nhập nên chúng tôi nghĩ rằng chúng ta sẽ chuyển nhập tĩnh ngay bây giờ.) Các thành viên lớp tĩnh có thể tồn tại trong các lớp bạn viết và trong rất nhiều lớp trong Java API.

Như chúng tôi đã nói trước đó, cuối cùng, các câu lệnh nhập giá trị duy nhất có là chúng tiết kiệm việc nhập và chúng có thể làm cho mã của bạn dễ đọc hơn. Trong Java 5 (một thời gian dài trước đây), câu lệnh nhập đã được cải tiến để cung cấp khả năng giảm số lần gõ phím thậm chí còn lớn hơn, mặc dù một số người cho rằng điều này xảy ra ở

chi phí của khả năng đọc. Tính năng này được gọi là nhập tĩnh. Nhập tĩnh có thể được sử dụng khi bạn muốn "lưu nhập" trong khi sử dụng các thành viên tĩnh của một lớp. (Bạn có thể sử dụng tính năng này trên các lớp trong API và trên các lớp của riêng bạn.) Đây là ví dụ "trước và sau" sử dụng một vài thành viên lớp tĩnh được cung cấp bởi một lớp thường được sử dụng trong Java API, `java.lang.Integer`. Ví dụ này cũng sử dụng một thành viên tĩnh mà bạn đã sử dụng hàng nghìn lần, có thể bạn chưa bao giờ suy nghĩ nhiều về nó; trường ngoài trong lớp Hệ thống .

Trước khi nhập tĩnh:

```
public class TestStatic {
    public static void main(String[] args) {
        System.out.println(Integer.MAX_VALUE);
        System.out.println(Integer.toHexString(42));
    }
}
```

Sau khi nhập tĩnh:

```
import static java.lang.System.out; // 1
import static java.lang.Integer.*; // 2
public class TestStaticImport {
    public static void main(String[] args) {
        out.println(MAX_VALUE); // 3
        out.println(toHexString(42)); // 4
    }
}
```

Cả hai lớp đều tạo ra cùng một đầu ra:

```
2147483647
2a
```

Hãy xem điều gì đang xảy ra trong đoạn mã sử dụng tính năng nhập tĩnh:

1. Mặc dù tính năng này thường được gọi là "nhập tĩnh", cú pháp PHẢI là nhập tĩnh theo sau là tên đủ điều kiện của thành viên tĩnh mà bạn muốn nhập hoặc ký tự đại diện. Trong trường hợp này, chúng tôi đang thực hiện nhập tĩnh trên đối tượng `System` class `out` .
2. Trong trường hợp này, chúng tôi có thể muốn sử dụng một số thành viên tĩnh của lớp `java.lang.Integer` . Câu lệnh nhập tĩnh này sử dụng ký tự đại diện để nói, "Tôi muốn nhập tĩnh **TẤT CẢ** các thành viên tĩnh trong lớp này."

3. Vậy giờ chúng ta cuối cùng cũng thấy lợi ích của tính năng nhập tĩnh! Chúng tôi không phải nhập Hệ thống trong System.out.println! Ô! Thứ hai, chúng tôi không phải nhập Số nguyên trong Integer.MAX_VALUE. Vì vậy, trong dòng mã này, chúng tôi có thể sử dụng một phím tắt cho một phương thức tĩnh VÀ một hằng số.
4. Cuối cùng, chúng ta thực hiện thêm một phím tắt nữa, lần này cho một phương thức trong Integer lớp.

Chúng tôi đã có một chút mỉa mai về tính năng này, nhưng chúng tôi không phải là những người duy nhất. Chúng tôi không tin rằng việc lưu một vài lần nhấn phím có thể làm cho mã khó đọc hơn một chút, nhưng đủ các nhà phát triển đã yêu cầu nó được thêm vào ngôn ngữ.

Dưới đây là một số quy tắc để sử dụng nhập tĩnh:

- Bạn phải nói nhập tĩnh; bạn không thể nói nhập tĩnh.
- Cần thận với các thành viên tĩnh được đặt tên không rõ ràng . Ví dụ: nếu bạn thực hiện nhập tĩnh cho cả lớp Integer và lớp Long , việc tham chiếu đến MAX_VALUE sẽ gây ra lỗi trình biên dịch, vì cả Integer và Long đều có hằng số MAX_VALUE và Java sẽ không biết bạn đang đề cập đến MAX_VALUE nào .
- Bạn có thể thực hiện nhập tĩnh trên các tham chiếu đối tượng tĩnh , hằng số (hãy nhớ chúng là tĩnh và cuối cùng) và các phương thức tĩnh .



Như bạn đã thấy, khi sử dụng nhập và nhập câu lệnh tĩnh, đôi khi bạn có thể sử dụng ký tự đại diện * để thực hiện một số tìm kiếm đơn giản cho mình. (Bạn có thể tìm kiếm trong một gói hoặc trong một lớp.)

Như bạn đã thấy trước đó, nếu bạn muốn "tìm kiếm tên lớp trong gói java.util", bạn có thể nói như sau:

```
import java.util.*; // ok to search the java.util package
```

Tương tự, nếu bạn muốn "tìm kiếm thông qua lớp java.lang.Integer cho các thành viên tĩnh", bạn có thể nói như sau:

```
import static java.lang.Integer.*; // ok to search the  
// java.lang.Integer class
```

Nhưng bạn không thể tạo các tìm kiếm rộng hơn. Ví dụ: bạn KHÔNG THỂ sử dụng phép nhập để tìm kiếm thông qua toàn bộ API Java:

```
import java.*; // Legal, but this WILL NOT search across packages.
```

Khai báo và sửa đổi lớp

Các khai báo lớp mà chúng ta sẽ thảo luận trong phần này được giới hạn cho các lớp cấp cao nhất. Ngoài các lớp cấp cao nhất, Java cung cấp cho một loại lớp khác được gọi là các lớp lồng nhau hoặc các lớp bên trong. Các lớp học nội bộ được bao gồm trong kỳ thi OCP, nhưng không có trong kỳ thi OCA. Khi bạn trở thành ứng viên OCP, bạn sẽ thích tìm hiểu về các lớp học bên trong. Không, thực sự. Nghiêm túc.

Đoạn mã sau là một khai báo lớp bare-bone:

```
lớp MyClass { }
```

Mã này biên dịch tốt, nhưng bạn cũng có thể thêm các sửa đổi trước khi khai báo lớp. Nói chung, các bổ ngữ được chia thành hai loại:

- Công cụ sửa đổi quyền truy cập (công khai, được bảo vệ, riêng tư)
- Các công cụ sửa đổi không truy cập (bao gồm cả nghiêm ngặt, cuối cùng và trừu tượng)

Trước tiên, chúng tôi sẽ xem xét các công cụ sửa đổi quyền truy cập, vì vậy bạn sẽ học cách hạn chế hoặc cho phép quyền truy cập vào một lớp bạn tạo. Kiểm soát truy cập trong Java hơi phức tạp vì có bốn kiểm soát truy cập (cấp độ truy cập) nhưng chỉ có ba công cụ sửa đổi truy cập. Mức kiểm soát truy cập thứ tư (được gọi là quyền truy cập mặc định hoặc gói) là những gì bạn nhận được khi không sử dụng bất kỳ công cụ sửa đổi quyền truy cập nào. Nói cách khác, mọi lớp, phương thức và biến thể hiện mà bạn khai báo đều có điều khiển truy cập, cho dù bạn nhập rõ ràng một hay không. Mặc dù tất cả bốn điều khiển truy cập (có nghĩa là cả ba phần bổ trợ) đều hoạt động cho hầu hết các khai báo phương thức và biến, một lớp có thể được khai báo chỉ với quyền truy cập công khai hoặc mặc định ; hai cấp độ kiểm soát truy cập khác không có ý nghĩa đối với một lớp, như bạn sẽ thấy.



Java là một ngôn ngữ tập trung vào gói; các nhà phát triển cho rằng điều đó tốt

tổ chức và phạm vi tên, bạn sẽ đặt tất cả các lớp của mình thành các gói.

Họ đã đúng, và bạn nên làm như vậy. Hãy tưởng tượng cơ bản này: Ba lập trình viên khác nhau, trong cùng một công ty nhưng làm việc trên các phần khác nhau của một dự án, viết một lớp tên là Tiện ích. Nếu ba lớp Tiện ích đó chưa được khai báo trong bất kỳ gói rõ ràng nào và nằm trong classpath, bạn sẽ không có cách nào để nói với trình biên dịch hoặc JVM mà bạn đang cố gắng tham chiếu đến ba lớp. Oracle khuyến nghị các nhà phát triển sử dụng các tên miền đảo ngược, được nối với tên bộ phận và / hoặc tên dự án. Ví dụ: nếu tên miền của bạn là geeksanonymous.com và bạn đang làm việc trên mã máy khách cho chương trình TwelvePoint0Steps, bạn sẽ đặt tên gói của mình giống như com.geeksanonymous.steps.client. Điều đó về cơ bản sẽ thay đổi tên lớp của bạn thành com.geeksanonymous.steps.client.Utilities. Bạn vẫn có thể có xung đột tên trong công ty của mình nếu bạn không đưa ra phương án đặt tên của riêng mình, nhưng bạn được đảm bảo không va chạm với các lớp được phát triển bên ngoài công ty của bạn (giả sử chúng tuân theo quy ước đặt tên của Oracle và nếu chúng không , Chà, Những điều Thực sự Xấu có thể xảy ra).

Quyền truy cập lớp học

Nó có nghĩa là gì để truy cập một lớp? Khi chúng ta nói mã từ một lớp (lớp A) có quyền truy cập vào lớp khác (lớp B), điều đó có nghĩa là lớp A có thể thực hiện một trong ba điều:

- Tạo một thể hiện của lớp B.
- Mở rộng lớp B (nói cách khác, trở thành một lớp con của lớp B).
- Truy cập các phương thức và biến nhất định trong lớp B, tùy thuộc vào quyền kiểm soát truy cập của các phương thức và biến đó.

Trên thực tế, quyền truy cập có nghĩa là khả năng hiển thị. Nếu lớp A không thể nhìn thấy lớp B, cấp độ truy cập của các phương thức và biến trong lớp B sẽ không thành vấn đề; lớp A sẽ không có bất kỳ cách nào để truy cập các phương thức và biến đó.

Quyền truy cập mặc định Một lớp có quyền truy cập mặc định không có phần bổ trợ nào đứng trước nó trong khai báo! Đó là quyền kiểm soát truy cập mà bạn nhận được khi không nhập sửa đổi trong khai báo lớp. Hãy coi quyền truy cập mặc định là quyền truy cập mức gói , bởi vì lớp có quyền truy cập mặc định chỉ có thể được nhìn thấy bởi các lớp trong cùng một gói.

Ví dụ: nếu lớp A và lớp B nằm trong các gói khác nhau và lớp A có quyền truy cập mặc định, thì lớp B sẽ không thể tạo một thể hiện của lớp A hoặc thậm chí khai báo một biến hoặc kiểu trả về của lớp A. Trên thực tế, lớp B phải giả vờ rằng lớp A thậm chí không tồn tại hoặc trình biên dịch sẽ phản nàn. Nhìn vào phần sau

tệp nguồn:

```
package cert;
class Beverage { }
```

Now look at the second source file:

```
package exam.stuff;
import cert.Beverage;
class Tea extends Beverage { }
```

Như bạn có thể thấy, lớp cha (Đồ uống) nằm trong một gói khác với lớp con (Trà). Câu lệnh nhập ở đầu tệp Trà đang cố gắng (bắt chéo ngón tay) để nhập loại Đồ uống . Tệp Đồ uống biên dịch tốt, nhưng khi chúng tôi cố gắng biên dịch tệp Trà , chúng tôi nhận được một cái gì đó như sau:

```
Can't access class cert.Beverage. Class or interface must be public, in same package, or an accessible member class.
```

```
import cert.Beverage;
```

(Lưu ý: Vì nhiều lý do khác nhau, các thông báo lỗi mà chúng tôi hiển thị trong suốt cuốn sách này có thể không khớp với thông báo lỗi bạn nhận được. Điều lo lắng, điểm thực sự là hiểu khi nào bạn có thể gặp phải một lỗi nào đó.)

Tea sẽ không biên dịch vì lớp cha của nó, Beverage, có quyền truy cập mặc định và là trong một gói khác. Bạn có thể làm một trong hai điều để làm cho việc này hoạt động. Bạn có thể đặt cả hai lớp trong cùng một gói hoặc bạn có thể khai báo Đồ uống là công khai, như phần tiếp theo mô tả.

Khi bạn thấy một câu hỏi có logic phức tạp, hãy nhớ xem các công cụ sửa lỗi quyền truy cập trước. Bằng cách đó, nếu bạn phát hiện vi phạm quyền truy cập (ví dụ: một lớp trong gói A đang cố gắng truy cập một lớp mặc định trong gói B), bạn sẽ biết mã sẽ không biên dịch để bạn không phải bận tâm làm việc Hợp lý. Nó không phải là nếu bạn không có bất cứ điều gì tốt hơn để làm với thời gian của bạn trong khi làm bài kiểm tra.

Chỉ cần chọn câu trả lời "Biên dịch không thành công" và chuyển sang câu hỏi tiếp theo.

Quyền truy cập công khai

Một khai báo lớp với từ khóa public cho phép tất cả các lớp từ tất cả các gói truy cập vào lớp công khai . Nói cách khác, tất cả các lớp trong Vũ trụ Java (JU) đều có quyền truy cập vào một lớp công khai . Tuy nhiên, đừng quên rằng nếu một lớp công khai mà bạn đang cố gắng sử dụng nằm trong một gói khác với lớp bạn đang viết, bạn vẫn cần phải nhập lớp công khai .

Trong ví dụ từ phần trước, chúng tôi có thể không muốn đặt

lớp con trong cùng một gói với lớp cha. Để làm cho mã hoạt động, chúng ta cần thêm từ khóa public vào trước khai báo lớp cha (Đò uống) , như sau:

```
chứng chỉ gói;  
Đò uống hạng công cộng {}
```

Điều này thay đổi lớp Đò uống để nó sẽ hiển thị cho tất cả các lớp trong tất cả các gói. Lớp bây giờ có thể được khởi tạo từ tất cả các lớp khác và bất kỳ lớp nào giờ đây đều có thể tự do đổi với lớp con (mở rộng từ) nó – trừ khi, nghĩa là, lớp đó cũng được đánh dấu bằng bô trơ nonaccess cuối cùng. Đọc tiếp.

Các công cụ sửa đổi lớp khác (Nonaccess) Bạn có thể

sửa đổi khai báo lớp bằng cách sử dụng từ khoá final, abstract, hoặc precisionfp. Các công cụ sửa đổi này bổ sung cho bất kỳ điều khiển truy cập nào trên lớp, vì vậy, ví dụ, bạn có thể khai báo một lớp là công khai và cuối cùng. Nhưng không phải lúc nào bạn cũng có thể kết hợp các công cụ sửa đổi nonaccess. Ví dụ: bạn có thể tự do sử dụng nghiêm ngặt kết hợp với cuối cùng , nhưng bạn không bao giờ được đánh dấu một lớp là cuối cùng và trừu tượng. Bạn sẽ thấy lý do tại sao trong hai phần tiếp theo.

Bạn sẽ không cần biết nghiêm ngặt hoạt động như thế nào, vì vậy chúng tôi chỉ tập trung vào việc sửa đổi một lớp dưới dạng cuối cùng hoặc trừu tượng. Đối với bài kiểm tra, bạn chỉ cần biết rằng nghiêm ngặt là một từ khóa và có thể được sử dụng để sửa đổi một lớp hoặc một phương thức, nhưng không bao giờ là một biến. Đánh dấu một lớp là nghiêm ngặt có nghĩa là bất kỳ mã phương thức nào trong lớp sẽ tuân thủ nghiêm ngặt các quy tắc tiêu chuẩn IEEE 754 cho dấu chấm động. Nếu không có công cụ sửa đổi đó, các dấu chấm động được sử dụng trong các phương thức có thể hoạt động theo cách phụ thuộc vào nền tảng. Nếu bạn không khai báo một lớp là nghiêm ngặt, bạn vẫn có thể nhận được hành vi nghiêm ngặt trên cơ sở từng phương thức bằng cách khai báo một phương thức là nghiêm ngặt. Nếu bạn chưa biết tiêu chuẩn IEEE 754, bây giờ không phải là lúc để tìm hiểu nó. Như họ nói, bạn có những con cá lớn hơn để chiến.

Lớp học cuối cùng

Khi được sử dụng trong khai báo lớp, từ khóa cuối cùng có nghĩa là lớp không thể được phân lớp. Nói cách khác, không có lớp nào khác có thể mở rộng (ké thừa từ) một lớp cuối cùng và bất kỳ nỗ lực nào để làm như vậy sẽ dẫn đến lỗi trình biên dịch.

Vậy tại sao bạn lại chấm điểm cuối cùng của lớp? Rốt cuộc, điều đó không vi phạm khái niệm thừa kế toàn bộ OO? Bạn chỉ nên tạo một lớp cuối cùng nếu bạn cần đảm bảo tuyệt đối rằng không có phương thức nào trong lớp đó sẽ bị ghi đè. Nếu bạn phụ thuộc sâu vào việc triển khai một số

thì việc sử dụng cuối cùng sẽ mang lại cho bạn sự bảo mật mà không ai có thể thay đổi việc triển khai bên dưới bạn.

Bạn sẽ nhận thấy nhiều lớp trong thư viện lõi Java là cuối cùng. Ví dụ, lớp String không thể được phân lớp. Hãy tưởng tượng sự tàn phá nếu bạn không thể đảm bảo một đối tượng String sẽ hoạt động như thế nào trên bất kỳ hệ thống nhất định nào mà ứng dụng của bạn đang chạy! Nếu các lập trình viên được tự do mở rộng lớp String (và do đó thay thế các cá thể lớp con String mới của họ nơi các cá thể java.lang.String được mong đợi), thì nền văn minh - như chúng ta đã biết - có thể sụp đổ. Vì vậy, hãy sử dụng cuối cùng để an toàn, nhưng chỉ khi bạn chắc chắn rằng lớp cuối cùng của bạn đã thực sự nói tất cả những gì cần phải nói trong các phương thức của nó. Về bản chất, việc đánh dấu một lớp cuối cùng có nghĩa là lớp của bạn không bao giờ có thể được cải thiện dựa trên, hoặc thậm chí là chuyên biệt, bởi một lập trình viên khác.

Có một lợi ích khi có các lớp học không chính thức trong trường hợp này: Hãy tưởng tượng rằng bạn tìm sự cố với một phương thức trong một lớp bạn đang sử dụng, nhưng bạn không có mã nguồn. Vì vậy, bạn không thể sửa đổi nguồn để cải thiện phương thức, nhưng bạn có thể mở rộng lớp và ghi đè phương thức trong lớp con mới của mình và thay thế lớp con ở mọi nơi mà lớp cha ban đầu được mong đợi. Tuy nhiên, nếu lớp học là cuối cùng, bạn đang mắc kẹt.

Hãy sửa đổi ví dụ Đồ uống của chúng tôi bằng cách đặt từ khóa cuối cùng trong khai báo:

```
package cert;
public final class Beverage {
    public void importantMethod() { }
}
```

Bây giờ chúng ta hãy thử biên dịch lớp con Tea :

```
package exam.stuff;
import cert.Beverage;
class Tea extends Beverage { }
```

We get an error—something like this:

```
Can't subclass final classes: class
cert.Beverage class Tea extends Beverage{
1 error
```

Trong thực tế, hầu như bạn sẽ không bao giờ học lớp cuối cùng. Một lớp cuối cùng xóa bỏ một lợi ích chính của OO – khả năng mở rộng. Trừ khi bạn gặp vấn đề nghiêm trọng về an toàn hoặc bảo mật, hãy giả sử rằng một ngày nào đó lập trình viên khác sẽ cần mở rộng lớp học của bạn. Nếu bạn không làm vậy, lập trình viên tiếp theo buộc phải duy trì mã của bạn sẽ săn lùng bạn và <chèn thứ thực sự đáng sợ>.

xuống và <chèn thứ thực sự đáng sợ>.

Các lớp trừu tượng Một lớp trừu tượng không bao giờ có thể được khởi tạo. Mục đích duy nhất của nó, sứ mệnh trong cuộc sống, *raison d'être*, là được mở rộng (phân lớp). (Tuy nhiên, lưu ý rằng bạn có thể biên dịch và thực thi một lớp trừu tượng, miễn là bạn không cố tạo một thể hiện của nó.) Tại sao phải tạo một lớp nếu bạn không thể tạo các đối tượng từ nó? Bởi vì lớp có thể quá trừu tượng. Ví dụ, hãy tưởng tượng bạn có một Class Car có các phương thức chung chung cho tất cả các loại xe.

Nhưng bạn không muốn bất kỳ ai thực sự tạo ra một đối tượng Car trừu tượng chung chung. Làm thế nào họ sẽ khởi tạo trạng thái của nó? Nó sẽ có màu gì? Có bao nhiêu chỗ ngồi? Mã lực? Dẫn động bốn bánh? Hay quan trọng hơn, nó sẽ cư xử như thế nào? Nói cách khác, các phương pháp sẽ được thực hiện như thế nào?

Không, bạn cần lập trình viên để khởi tạo các loại ô tô thực tế như BMWBoxster và SubaruOutback. Chúng tôi dám cá rằng chủ sở hữu Boxster sẽ cho bạn biết chiếc xe của anh ấy làm được những điều mà Subaru có thể làm "chỉ có trong mơ". Hãy xem lớp trừu tượng sau :

```
abstract class Car {
    private double price;
    private String model;
    private String year;
    public abstract void goFast();
    public abstract void goUpHill();
    public abstract void impressNeighbors();
    // Additional, important, and serious code goes here
}
```

Mã trước đó sẽ biên dịch tốt. Tuy nhiên, nếu bạn cố gắng khởi tạo Xe trong một nội dung mã khác, bạn sẽ gặp lỗi trình biên dịch như sau:

```
AnotherClass.java:7: class Car is an abstract
class. It can't be instantiated.
    Car x = new Car();
1 error
```

Lưu ý rằng các phương thức được đánh dấu trừu tượng kết thúc bằng dấu chấm phẩy thay vì dấu ngoặc nhọn.

Tìm các câu hỏi có khai báo phương thức kết thúc bằng dấu chấm phẩy, thay vì dấu ngoặc nhọn. Nếu phương thức nằm trong một lớp - trái ngược với một giao diện - thì cả phương thức và lớp phải được đánh dấu là trừu tượng. Bạn có thể nhận được một câu hỏi hỏi cách bạn có thể sửa một mẫu mã bao gồm một phương pháp

kết thúc bằng dấu chấm phẩy nhưng không có phần bỏ trợ trừu tượng trên lớp hoặc phương thức. Trong trường hợp đó, bạn có thể đánh dấu phương thức và lớp trừu tượng hoặc thay đổi dấu chấm phẩy thành mã (như một cặp dấu ngoặc nhọn). Hãy nhớ rằng nếu bạn thay đổi một phương thức từ trừu tượng thành nonabstract, đừng quên thay đổi dấu chấm phẩy ở cuối khai báo phương thức thành một cặp ngoặc nhọn!

Chúng ta sẽ xem xét các phương thức trừu tượng chi tiết hơn ở phần sau trong mục tiêu này, nhưng hãy luôn nhớ rằng nếu ngay cả một phương thức duy nhất là trừu tượng, thì toàn bộ lớp phải được khai báo là trừu tượng. Một phương pháp trừu tượng làm hỏng toàn bộ nhóm. Tuy nhiên, bạn có thể đặt các phương thức nonabstract trong một lớp trừu tượng . Ví dụ: bạn có thể có các phương thức có triển khai không được thay đổi từ Loại xe sang Loại xe, chẳng hạn như getColor () hoặc setPrice (). Bằng cách đặt các phương thức nonabstract trong một lớp trừu tượng , bạn cung cấp cho tất cả các lớp con cụ thể (cụ thể có nghĩa là không trừu tượng) các triển khai phương thức kế thừa. Tin tốt là các lớp con cụ thể có thể kế thừa chức năng và chỉ cần triển khai các phương thức xác định hành vi cụ thể của lớp con.

(Nhân tiện, nếu bạn cho rằng chúng tôi đã lạm dụng raison d'être trước đó, đừng gửi một email. Chúng tôi muốn thấy bạn viết nó thành một cuốn sách cấp chứng chỉ lập trình viên.)

Mã hóa với các loại lớp trừu tượng (bao gồm các giao diện, sẽ được thảo luận ở phần sau của bài viết này chương) cho phép bạn tận dụng tính đa hình và mang lại cho bạn mức độ linh hoạt và khả năng mở rộng cao nhất. Bạn sẽ tìm hiểu thêm về tính đa hình trong [Chương 2](#).

Bạn không thể đánh dấu một lớp là trừu tượng và cuối cùng. Chúng có ý nghĩa gần như trái ngược nhau. Một lớp trừu tượng phải được phân lớp con, trong khi lớp cuối cùng không được phân lớp con. Nếu bạn thấy sự kết hợp của các bối rối trừu tượng và cuối cùng được sử dụng cho khai báo lớp hoặc phương thức, thì mã sẽ không biên dịch.

BÀI TẬP 1-1

Tạo một lớp siêu trừu tượng và lớp con cụ thể

Bài tập sau sẽ kiểm tra kiến thức của bạn về các lớp công khai, mặc định, cuối cùng và trừu tượng . Tạo một lớp cha trừu tượng có tên là Fruit và một lớp con cụ thể có tên Apple. Lớp cha phải thuộc về một gói được gọi là food và lớp con có thể thuộc về gói mặc định (có nghĩa là nó không được đưa vào một gói một cách rõ ràng). Đặt lớp cha ở chế độ công khai và đặt lớp con mặc định truy cập.

1. Tạo lớp cha như sau:

```
thực phẩm gói;  
public abstract class Fruit {/* bất kỳ mã nào bạn muốn */}
```

2. Tạo lớp con trong một tệp riêng biệt như sau:

```
nhập khẩu thực phẩm. lớp  
Apple mở rộng Fruit {/* bất kỳ mã nào bạn muốn */}
```

3. Tạo một thư mục gọi là food ngoài thư mục trong cài đặt đường dẫn lớp của bạn.

4. Có gắng biên dịch hai tệp. Nếu bạn muốn sử dụng lớp Apple , hãy đảm bảo rằng bạn đặt tệp Fruit.class trong thư mục con thực phẩm .

MỤC TIÊU XÁC NHẬN

Sử dụng các giao diện (Mục tiêu 7.5 của OCA)

7.6 Sử dụng các lớp và giao diện trừu tượng.

Khai báo giao diện

Nói chung, khi bạn tạo một giao diện, bạn đang xác định một hợp đồng cho những gì một lớp có thể làm, mà không cần nói bất cứ điều gì về cách lớp sẽ làm điều đó.

Lưu ý: Đối với Java 8, bây giờ bạn cũng có thể mô tả cách thực hiện, nhưng bạn thường không làm như vậy. Cho đến khi chúng ta đi đến các tính năng liên quan đến giao diện mới của Java 8 – các phương thức mặc định và tĩnh – chúng ta sẽ thảo luận về các giao diện từ quan điểm truyền thống, một lần nữa, định nghĩa một hợp đồng cho những gì một lớp có thể làm.

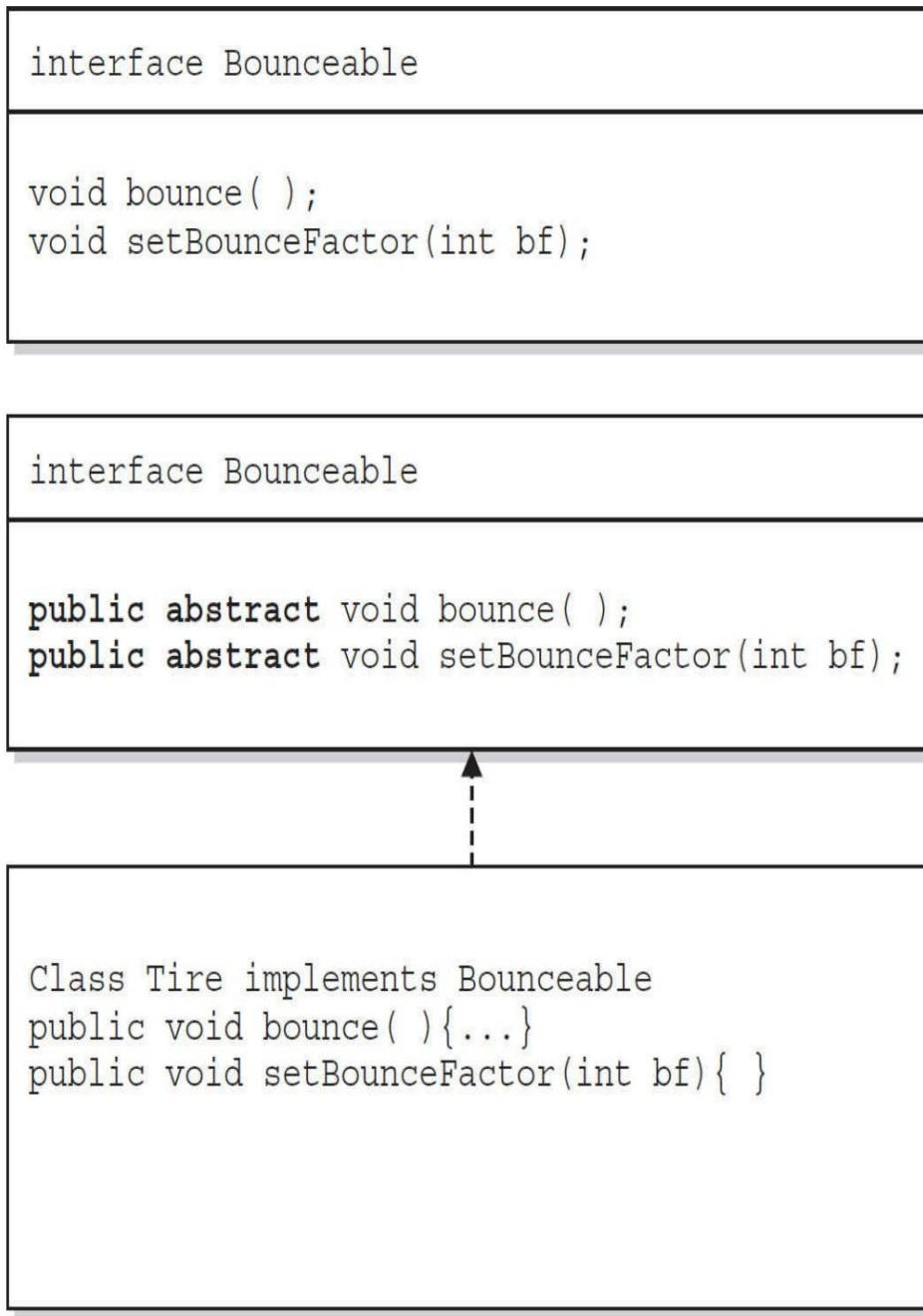
Giao diện là một hợp đồng. Ví dụ, bạn có thể viết một giao diện Có thể trả lại, có nội dung thực tế, "Đây là giao diện Có thể trả lại. Bất kỳ loại lớp cụ thể nào triển khai giao diện này đều phải đồng ý viết mã cho các phương thức bounce () và setBounceFactor () . "

Bằng cách xác định một giao diện cho Bounceable, bất kỳ lớp nào muốn được coi là một thứ có thể trả lại có thể chỉ cần triển khai giao diện Bounceable và cung cấp mã cho hai phương thức của giao diện.

Các giao diện có thể được thực hiện bởi bất kỳ lớp nào, từ bất kỳ cây kế thừa nào. Điều này cho phép bạn lấy các lớp hoàn toàn khác nhau và tạo cho chúng một đặc điểm chung.

Ví dụ: bạn có thể muốn cả Quả bóng và Lốp đều có hành vi này, nhưng Bóng và Lốp không chia sẻ bất kỳ mối quan hệ kế thừa nào; Bóng mở rộng Đồ chơi

trong khi Tire chỉ mở rộng java.lang.Object. Nhưng bằng cách làm cho cả Ball và Tyre triển khai Bounceable, bạn đang nói rằng Ball và Tyre có thể được coi là "Những thứ có thể bị trả lại", trong Java có nghĩa là "Những thứ mà bạn có thể gọi nảy () và setBounceFactor () các phương pháp." [Hình 1-1](#) minh họa mối quan hệ giữa các giao diện và các lớp.



HÌNH 1-1 Mối quan hệ giữa các giao diện và các lớp

Hãy nghĩ về một giao diện truyền thống như một lớp trừu tượng 100% . Giống như một lớp trừu tượng , một giao diện xác định các phương thức trừu tượng có dạng sau:

```
abstract void bounce(); // Ends with a semicolon rather than
// curly braces
```

Nhưng mặc dù một lớp trừu tượng có thể định nghĩa cả trừu tượng và nonabstract các phương thức, một giao diện thường chỉ có các phương thức trừu tượng . Một cách khác, giao diện khác với các lớp trừu tượng là giao diện có rất ít tính linh hoạt trong cách khai báo các phương thức và biến được định nghĩa trong giao diện.

Các quy tắc này rất nghiêm ngặt:

- Các phương thức giao diện hoàn toàn là công khai và trừu tượng, trừ khi được khai báo là mặc định hoặc tĩnh. Nói cách khác, bạn không cần phải thực sự nhập các sửa đổi công khai hoặc trừu tượng trong khai báo phương thức, nhưng phương thức vẫn luôn là công khai và trừu tượng.
- Tất cả các biến được định nghĩa trong một giao diện phải là công khai, tĩnh và cuối cùng - nói cách khác, các giao diện chỉ có thể khai báo các hằng số, không phải các biến cá thể.
- Các phương thức giao diện không thể được đánh dấu cuối cùng, nghiêm ngặt hoặc gốc. (Thông tin thêm về các bồ ngữ này ở phần sau của chương.)
- Một giao diện có thể mở rộng một hoặc nhiều giao diện khác.
- Một giao diện không thể mở rộng bất cứ thứ gì ngoài một giao diện khác.
- Một giao diện không thể triển khai một giao diện hoặc lớp khác.
- Một giao diện phải được khai báo với giao diện từ khóa.
- Các kiểu giao diện có thể được sử dụng đa hình (xem [Chương 2](#) để biết thêm chi tiết).

Sau đây là một khai báo giao diện pháp lý:

giao diện trừu tượng công khai Có thể cuộn {}

Gõ vào công cụ sửa đổi trừu tượng được coi là thừa; các giao diện là hoàn toàn trừu tượng cho dù bạn nhập trừu tượng hay không. Bạn chỉ cần biết rằng cả hai khai báo này đều hợp pháp và giống hệt nhau về chức năng:

giao diện trừu tượng công khai Có thể cuộn {} giao
diện công khai Có thể cuộn {}

Công cụ sửa đổi công khai là bắt buộc nếu bạn muốn giao diện có quyền truy cập công khai thay vì mặc định.

Chúng tôi đã xem xét khai báo giao diện, nhưng bây giờ chúng tôi sẽ xem xét kỹ các phương thức trong một giao diện:

```
public interface Bounceable {
    public abstract void bounce();
    public abstract void setBounceFactor(int bf);
}
```

Tuy nhiên, việc nhập các sửa đổi công khai và trừu tượng trên các phương thức là không cần thiết, vì tất cả các phương thức giao diện đều là công khai và trừu tượng. Với quy tắc đó, bạn có thể thấy rằng đoạn mã sau hoàn toàn tương đương với giao diện trước đó:

```
public interface Bounceable {
    void bounce(); // No modifiers
    void setBounceFactor(int bf); // No modifiers
}
```

Bạn phải nhớ rằng tất cả các phương thức giao diện không được khai báo mặc định hoặc tĩnh là công khai và trừu tượng bất kể những gì bạn thấy trong định nghĩa giao diện.

Tìm kiếm các phương thức giao diện được khai báo với bất kỳ sự kết hợp nào của công khai, trừu tượng hoặc không có sửa đổi. Ví dụ, năm khai báo phương thức sau đây, nếu được khai báo trong các giao diện riêng của chúng, là hợp pháp và giống hệt nhau!

```
void bounce();
public void bounce();
abstract void bounce();
public abstract void bounce();
abstract public void bounce();
```

Các khai báo phương thức giao diện sau sẽ không biên dịch:

```
final void bounce(); // final and abstract can never be used
                     // together, and abstract is implied
private void bounce(); // interface methods are always public
protected void bounce(); // (same as above)
```

Khai báo các hằng số giao diện

Bạn được phép đặt các hằng số trong một giao diện. Bằng cách đó, bạn đảm bảo rằng bất kỳ lớp nào triển khai giao diện sẽ có quyền truy cập vào cùng một hằng số. Qua

bất kỳ lớp nào triển khai giao diện sẽ có quyền truy cập vào cùng một hằng số. Bằng cách đặt các hằng số ngay trong giao diện, bất kỳ lớp nào thực thi giao diện đều có quyền truy cập trực tiếp vào các hằng số, giống như thể lớp đó đã kế thừa chúng.

Bạn cần nhớ một quy tắc chính cho các hằng số giao diện. Họ phải luôn luôn

công khai tinh cuối cùng

Vì vậy, điều đó nghe có vẻ đơn giản, phải không? Rốt cuộc, hằng số giao diện không khác với bất kỳ hằng số có thể truy cập công khai nào khác, vì vậy rõ ràng chúng phải được khai báo công khai, tinh và cuối cùng. Nhưng trước khi bạn lướt qua phần còn lại của cuộc thảo luận này, hãy nghĩ về hàm ý: Bởi vì các hằng số giao diện được định nghĩa trong một giao diện, chúng không cần phải được khai báo là công khai, tinh hoặc cuối cùng. Chúng phải công khai, tinh và cuối cùng, nhưng bạn không thực sự phải khai báo chúng theo cách đó. Cũng giống như các phương thức giao diện luôn công khai và trừu tượng cho dù bạn có nói như vậy trong mã hay không, bất kỳ biến nào được xác định trong giao diện phải là – và mặc nhiên là – một hằng số công khai. Xem liệu bạn có thể phát hiện ra sự cố với đoạn mã sau hay không (giả sử hai tệp riêng biệt):

```
interface Foo {
    int BAR = 42;
    void go();
}

class Zap implements Foo {
    public void go() {
        BAR = 27;
    }
}
```

Bạn không thể thay đổi giá trị của một hằng số! Khi giá trị đã được chỉ định, giá trị không bao giờ có thể được sửa đổi. Việc gán xảy ra trong chính giao diện (nơi hằng được khai báo), vì vậy lớp thực thi có thể truy cập và sử dụng nó, nhưng dưới dạng giá trị chỉ đọc. Vì vậy, phép gán BAR = 27 sẽ không biên dịch.



Tìm kiếm các định nghĩa giao diện xác định các hằng số, nhưng không sử dụng các công cụ sửa đổi bắt buộc một cách rõ ràng. Ví dụ: tất cả những điều sau đây đều giống hệt nhau:

```

public int x = 1;           // Looks non-static and non-final,
                           // but isn't!
int x = 1;                 // Looks default, non-final,
                           // non-static, but isn't!
static int x = 1;          // Doesn't show final or public
final int x = 1;           // Doesn't show static or public
public static int x = 1;    // Doesn't show final
public final int x = 1;     // Doesn't show static
static final int x = 1;     // Doesn't show public
public static final int x = 1; // what you get implicitly

```

Bất kỳ sự kết hợp nào của các công cụ sửa đổi bắt buộc (nhưng ngầm định) đều hợp pháp, cũng như không sử dụng công cụ sửa đổi nào cả! Trong bài kiểm tra, bạn có thể mong đợi thấy những câu hỏi mà bạn sẽ không thể trả lời chính xác trừ khi bạn biết, ví dụ: một biến giao diện là cuối cùng và không bao giờ có thể được cung cấp giá trị bởi lớp triển khai (hoặc bất kỳ loại nào khác).

Khai báo các phương thức giao diện mặc định

Kể từ Java 8, các giao diện có thể bao gồm các phương thức có thể kế thừa * với các triển khai cụ thể. (* Định nghĩa chặt chẽ về "kế thừa" đã trở nên hơi mờ nhạt với Java 8; chúng ta sẽ nói thêm về kế thừa trong [Chương 2](#).) Các phương thức cụ thể này được gọi là các phương thức mặc định. Trong chương tiếp theo, chúng ta sẽ nói nhiều về các quy tắc khác nhau liên quan đến 00 bị ảnh hưởng do các phương thức mặc định. Vậy giờ chúng ta sẽ chỉ đề cập đến các quy tắc khai báo đơn giản:

- các phương thức mặc định được khai báo bằng cách sử dụng từ khóa `default`. Từ khóa `default` chỉ có thể được sử dụng với chữ ký phương thức giao diện, không phải chữ ký phương thức lớp. các phương thức mặc định là công khai theo định nghĩa và công cụ sửa đổi là tùy chọn.
- các phương thức mặc định không thể được đánh dấu là riêng tư, được bảo vệ, tịnh, cuối cùng hoặc trứu tượng.
- các phương thức mặc định phải có một phần thân phương thức cụ thể.

Dưới đây là một số ví dụ về các phương pháp mặc định hợp pháp và bất hợp pháp:

```

interface TestDefault {
    default int m1(){return 1;} // legal
    public default void m2(){;} // legal
    static default void m3(){;} // illegal: default cannot be marked static
    default void m4();           // illegal: default must have a method body
}

```

Khai báo các phương thức giao diện tĩnh

Kể từ Java 8, các giao diện có thể bao gồm các phương thức tĩnh với các triển khai cụ thể. Như với các phương thức mặc định của giao diện, có những hàm ý 00 mà chúng ta sẽ thảo luận trong [Chương 2](#). Bây giờ, chúng ta sẽ tập trung vào những điều cơ bản về khai báo và sử dụng các phương thức giao diện tĩnh :

- các phương thức giao diện tĩnh được khai báo bằng cách sử dụng từ khóa static .
- các phương thức giao diện tĩnh là công khai theo mặc định và công cụ sửa đổi công khai là tùy chọn. các phương thức giao diện tĩnh không thể được đánh dấu là riêng tư, được bảo vệ, cuối cùng hoặc trừu tượng.
- các phương thức giao diện tĩnh phải có một phần thân phương thức cụ thể.
- Khi gọi một phương thức giao diện tĩnh , kiểu của phương thức (tên giao diện) PHẢI được bao gồm trong lời gọi.

Dưới đây là một số ví dụ về các phương thức giao diện tĩnh hợp pháp và bất hợp pháp và sử dụng:

```

interface StaticIface {
    static int m1(){ return 42; }          // legal
    public static void m2(){ ; }           // legal
    // final static void m3(){ ; }         // illegal: final not allowed
    // abstract static void m4(){ ; }       // illegal: abstract not allowed
    // static void m5();                  // illegal: needs a method body
}

public class TestSIF implements StaticIface {
    public static void main(String[] args) {
        System.out.println(StaticIface.m1()); // legal: m1()'s type
                                            // must be included
        new TestSIF().go();
        // System.out.println(m1());        // illegal: reference to interface
                                            // is required
    }
    void go() {
        System.out.println(StaticIface.m1()); // also legal from an instance
    }
}

```

sản xuất đầu ra này:

42
42

Như chúng ta đã nói trước đó, chúng ta sẽ quay lại thảo luận về các phương thức mặc định và phương thức tĩnh cho các giao diện trong [Chương 2](#).

MỤC TIÊU XÁC NHẬN

Khai báo các thành viên trong lớp (Mục tiêu của OCA 2.1, 2.2, 2.3, 4.1, 4.2, 6.2, 6.3 và 6.4)

- 2.1 Khai báo và khởi tạo biến (bao gồm ép kiểu dữ liệu nguyên thủy).
- 2.2 Phân biệt giữa biến tham chiếu đối tượng và biến nguyên thủy.
- 2.3 Biết cách đọc hoặc ghi vào các trường đối tượng.
- 4.1 Khai báo, khởi tạo, khởi tạo và sử dụng mảng một chiều.
- 4.2 Khai báo, khởi tạo, khởi tạo và sử dụng mảng đa chiều. (sic)
- 6.2 Áp dụng từ khóa static cho các phương thức và trường.
- 6.3 Tạo và nạp chồng các hàm tạo; bao gồm cả tác động đến các hàm tạo mặc định.

(sic)

6.4 Áp dụng công cụ sửa đổi quyền truy cập.

Chúng tôi đã xem xét ý nghĩa của việc sử dụng một công cụ sửa đổi trong khai báo lớp và bây giờ chúng ta sẽ xem xét ý nghĩa của việc sửa đổi một phương thức hoặc khai báo biến.

Các phương thức và biến thể hiện (phi địa phương) được gọi chung là các thành viên. Bạn có thể sửa đổi một thành viên bằng cả sửa đổi truy cập và không truy cập, và bạn có nhiều công cụ sửa đổi để lựa chọn (và kết hợp) hơn là khi bạn khai báo một lớp.

Truy cập công cụ sửa đổi

Bởi vì các thành viên phương thức và biến thường được cấp quyền kiểm soát truy cập theo cùng một cách, chúng tôi sẽ đề cập đến cả hai trong phần này.

Trong khi một lớp chỉ có thể sử dụng hai trong bốn cấp độ kiểm soát truy cập (mặc định hoặc public), các thành viên có thể sử dụng cả bốn:

- công cộng
- được bảo vệ
- mặc định
- riêng

Bảo vệ mặc định là những gì bạn nhận được khi không nhập công cụ sửa đổi quyền truy cập vào khai báo thành viên. Các loại kiểm soát truy cập mặc định và được bảo vệ có hành vi gần như giống hệt nhau, ngoại trừ một điểm khác biệt mà chúng tôi sẽ đề cập sau.

Lưu ý: Kể từ Java 8, từ mặc định CŨNG có thể được sử dụng để khai báo các phương thức nhất định trong giao diện. Khi được sử dụng trong khai báo phương thức của giao diện, mặc định có một ý nghĩa khác với những gì chúng ta đang mô tả trong phần còn lại của chương này.

Điều quan trọng là bạn phải biết kiểm soát truy cập bên trong và bên ngoài cho kỳ thi. Sẽ có khá nhiều câu hỏi nơi kiểm soát truy cập đóng một vai trò nào đó. Một số câu hỏi kiểm tra một số khái niệm về kiểm soát truy cập cùng một lúc, vì vậy nếu không biết một phần nhỏ của kiểm soát truy cập có thể đồng nghĩa với việc bạn làm hỏng toàn bộ câu hỏi.

Điều gì có nghĩa là khi mã trong một lớp có quyền truy cập vào một thành viên của lớp khác?

Hiện tại, hãy bỏ qua bất kỳ sự khác biệt nào giữa các phương thức và biến.

Nếu lớp A có quyền truy cập vào một thành viên của lớp B, điều đó có nghĩa là thành viên của lớp B được hiển thị cho lớp A. Khi một lớp không có quyền truy cập vào thành viên khác, trình biên dịch sẽ tát bạn vì cố gắng truy cập vào thứ mà bạn không. thậm chí

trình biên dịch sẽ tắt bạn vì cố gắng truy cập một cái gì đó mà bạn thậm chí không được biết là tồn tại!

Bạn cần hiểu hai vấn đề truy cập khác nhau:

- Liệu mã phương thức trong một lớp có thể truy cập vào một thành viên của lớp khác hay không
- Liệu một lớp con có thể kế thừa một thành viên của lớp cha của nó hay không

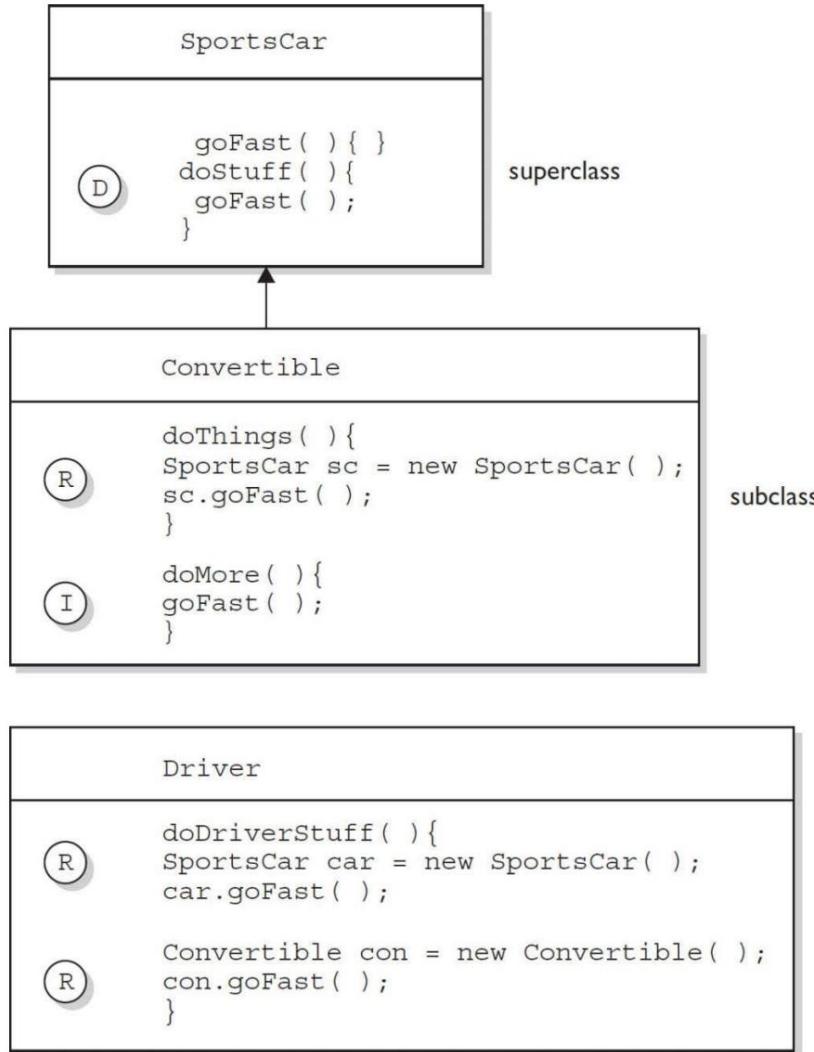
Kiểu truy cập đầu tiên xảy ra khi một phương thức trong một lớp cố gắng truy cập vào một phương thức hoặc một biến của lớp khác, sử dụng toán tử dấu chấm (.) để gọi một phương thức hoặc truy xuất một biến. Ví dụ:

```
class Zoo {
    public String coolMethod() {
        return "Wow baby";
    }
}
class Moo {
    public void useAZoo() {
        Zoo z = new Zoo();
        // If the preceding line compiles Moo has access
        // to the Zoo class
        // But... does it have access to the coolMethod()?
        System.out.println("A Zoo says, " + z.coolMethod());
        // The preceding line works because Moo can access the
        // public method
    }
}
```

Kiểu truy cập thứ hai xoay quanh việc, nếu có, các thành viên của lớp cha một lớp con có thể truy cập thông qua kế thừa. Chúng tôi không xem liệu lớp con có thể gọi một phương thức trên một thể hiện của lớp cha hay không (ví dụ về kiểu truy cập đầu tiên). Thay vào đó, chúng tôi đang xem xét liệu lớp con có kế thừa một thành viên của lớp cha của nó hay không. Hãy nhớ rằng, nếu một lớp con kế thừa một thành viên, nó chính xác như thể lớp con thực sự khai báo chính thành viên đó. Nói cách khác, nếu một lớp con kế thừa một thành viên, thì lớp con đó có thành viên đó. Đây là một ví dụ:

```
class Zoo {  
    public String coolMethod() {  
        return "Wow baby";  
    }  
}  
class Moo extends Zoo {  
    public void useMyCoolMethod() {  
        // Does an instance of Moo inherit the coolMethod()  
        System.out.println("Moo says, " + this.coolMethod());  
        // The preceding line works because Moo can inherit the  
        // public method  
        // Can an instance of Moo invoke coolMethod() on an  
        // instance of Zoo?  
  
        Zoo z = new Zoo();  
        System.out.println("Zoo says, " + z.coolMethod());  
        // coolMethod() is public, so Moo can invoke it on a Zoo  
        // reference  
    }  
}
```

[Hình 1-2](#) so sánh một lớp kế thừa một thành viên của lớp khác và truy cập một thành viên của lớp khác bằng cách sử dụng tham chiếu của một thể hiện của lớp đó.



Three ways to access a method:

- (D) Invoking a method declared in the same class
- (R) Invoking a method using a reference of the class
- (I) Invoking an inherited method

HÌNH 1-2

sự kế thừa so với toán tử dấu chấm đối với quyền truy cập của thành viên

Phần lớn kiểm soát truy cập (cả hai loại) tập trung vào việc liệu hai lớp liên quan nằm trong các gói giống nhau hay khác nhau. Tuy nhiên, đừng quên rằng nếu bản thân lớp A không thể được truy cập bởi lớp B, thì không thành viên nào trong lớp A có thể được truy cập bởi lớp B.

Bạn cần biết ảnh hưởng của các kết hợp khác nhau của quyền truy cập lớp và thành viên (chẳng hạn như một lớp mặc định với một biến công khai). Để tìm ra điều này, trước tiên

nhìn vào cấp độ truy cập của lớp. Nếu bản thân lớp đó sẽ không hiển thị với lớp khác, thì không thành viên nào trong số các thành viên sẽ được hiển thị, ngay cả khi thành viên được khai báo là công khai. Khi bạn đã xác nhận rằng lớp được hiển thị, thì việc xem xét các cấp độ truy cập trên từng thành viên là rất hợp lý.

Thành viên công khai

Khi một phương thức hoặc thành viên biến được khai báo là public, điều đó có nghĩa là tất cả các lớp khác, bất kể chúng thuộc về gói nào, đều có thể truy cập thành viên đó (giả sử rằng bản thân lớp đó là hiển thị).

Xem tệp nguồn sau:

```
package book;
import cert.*; // Import all classes in the cert package
class Goo {
    public static void main(String[] args) {
        Sludge o = new Sludge();
        o.testIt();
    }
}
```

Now look at the second file:

```
package cert;
public class Sludge {
    public void testIt() { System.out.println("sludge"); }
}
```

Như bạn có thể thấy, Goo và Bùn ở trong các gói khác nhau. Tuy nhiên, Goo có thể gọi phương thức trong S mud mà không có vấn đề gì vì cả lớp S mud và phương thức testIt () của nó đều được đánh dấu là công khai.

Đối với một lớp con, nếu một thành viên của lớp cha của nó được khai báo là công khai, lớp con kế thừa thành viên đó bất kể cả hai lớp có trong cùng một gói hay không:

```
package cert;
public class Roo {
    public String doRooThings() {
        // imagine the fun code that goes here
        return "fun";
    }
}
```

Lớp Roo khai báo thành viên doRooThings () là công khai. Vì vậy, nếu chúng ta thực hiện một

lớp con của Roo, bất kỳ mã nào trong lớp con Roo đó đều có thể gọi phương thức doRooThings () kế thừa của chính nó .

Lưu ý trong đoạn mã sau rằng phương thức doRooThings () được gọi mà không cần phải mở đầu nó bằng một tham chiếu:

```
package notcert; // Not the package Roo is in
import cert.Roo;
class Cloo extends Roo {
    public void testCloo() {
        System.out.println(doRooThings());
    }
}
```

Hãy nhớ rằng, nếu bạn thấy một phương thức được gọi (hoặc một biến được truy cập) mà không có toán tử dấu chấm (.), Điều đó có nghĩa là phương thức hoặc biến đó thuộc về lớp mà bạn thấy mã đó. Nó cũng có nghĩa là phương thức hoặc biến đang được truy cập ngầm bằng cách sử dụng tham chiếu này . Vì vậy, trong đoạn mã trước, lời gọi đến doRooThings () trong lớp Cloo cũng có thể được viết dưới dạng this.doRooThings () . Tham chiếu này luôn đề cập đến đối tượng hiện đang thực thi – nói cách khác, đối tượng đang chạy mã nơi bạn nhìn thấy tham chiếu này . Bởi vì tham chiếu này là ẩn, bạn không cần phải viết trước mã truy cập thành viên của mình với nó, nhưng nó sẽ không ảnh hưởng gì. Một số lập trình viên bao gồm nó để làm cho mã dễ đọc hơn cho các lập trình viên Java mới (hoặc không phải).

Bên cạnh việc có thể tự gọi phương thức doRooThings () , mã từ một số lớp khác có thể gọi doRooThings () trên một cá thể Cloo , như sau:

```
package notcert;
class Toon {
    public static void main(String[] args) {
        Cloo c = new Cloo();
        System.out.println(c.doRooThings()); // No problem; method
                                            // is public
    }
}
```

Thành viên riêng

Các thành viên được đánh dấu là riêng tư không thể được truy cập bằng mã trong bất kỳ lớp nào khác với lớp mà thành viên riêng tư đã được khai báo. Hãy thực hiện một thay đổi nhỏ đối với lớp Roo từ ví dụ trước:

```

package cert;
public class Roo {
    private String doRooThings() {
        // imagine the fun code that goes here, but only the Roo
        // class knows
        return "fun";
    }
}

```

Phương thức doRooThings () hiện là private nên không lớp nào khác có thể sử dụng nó. Nếu chúng tôi cố gắng gọi phương thức từ bất kỳ lớp nào khác, chúng tôi sẽ gặp rắc rối:

```

package notcert;
import cert.Roo;
class UseARoo {
    public void testIt() {
        Roo r = new Roo(); //So far so good; class Roo is public
        System.out.println(r.doRooThings()); // Compiler error!
    }
}

```

Nếu chúng tôi cố gắng biên dịch UseARoo, chúng tôi gặp lỗi trình biên dịch như sau:

```

không thể tìm thấy biểu
tượng ký hiệu: method doRooThings ()

```

Như thể phương thức doRooThings () không tồn tại, và liên quan đến bất kỳ mã nào bên ngoài lớp Roo , điều này là đúng. Một thành viên riêng là ẩn đối với bất kỳ mã nào bên ngoài lớp riêng của thành viên đó.

Điều gì về một lớp con cố gắng kế thừa một thành viên riêng của lớp cha của nó? Khi một thành viên được khai báo là private, một lớp con không thể kế thừa nó. Đối với bài kiểm tra, bạn cần nhận ra rằng một lớp con không thể nhìn thấy, sử dụng hoặc thậm chí nghĩ về các thành viên riêng tư của lớp cha của nó. Tuy nhiên, bạn có thể khai báo một phương thức so khớp trong lớp con. Nhưng bắt kể nó trông như thế nào, nó không phải là một phương pháp ghi đè! Nó chỉ đơn giản là một phương thức trùng tên với một phương thức private (mà bạn không nên biết về nó) trong lớp cha.

Các quy tắc ghi đè không áp dụng, vì vậy bạn có thể làm cho phương thức mới được khai báo- nhưng chỉ-xảy-ra-khớp này khai báo các ngoại lệ mới hoặc thay đổi kiểu trả về hoặc làm bất kỳ điều gì khác mà bạn muốn nó làm.

```

package cert;
public class Roo {
    private String doRooThings() {
        // imagine the fun code that goes here, but no other class
        // will know
        return "fun";
    }
}

```

Phương thức doRooThings () hiện đã bị giới hạn đối với tất cả các lớp con, ngay cả những lớp trong cùng một gói với lớp cha:

```

package cert;                                // Cloo and Roo are in the same package
class Cloo extends Roo {                      // Still OK, superclass Roo is public
    public void testCloo() {
        System.out.println(doRooThings()); // Compiler error!
    }
}

```

Nếu chúng tôi cố gắng biên dịch lớp con Cloo, trình biên dịch rất vui khi tạo ra một lỗi một cái gì đó như thế này:

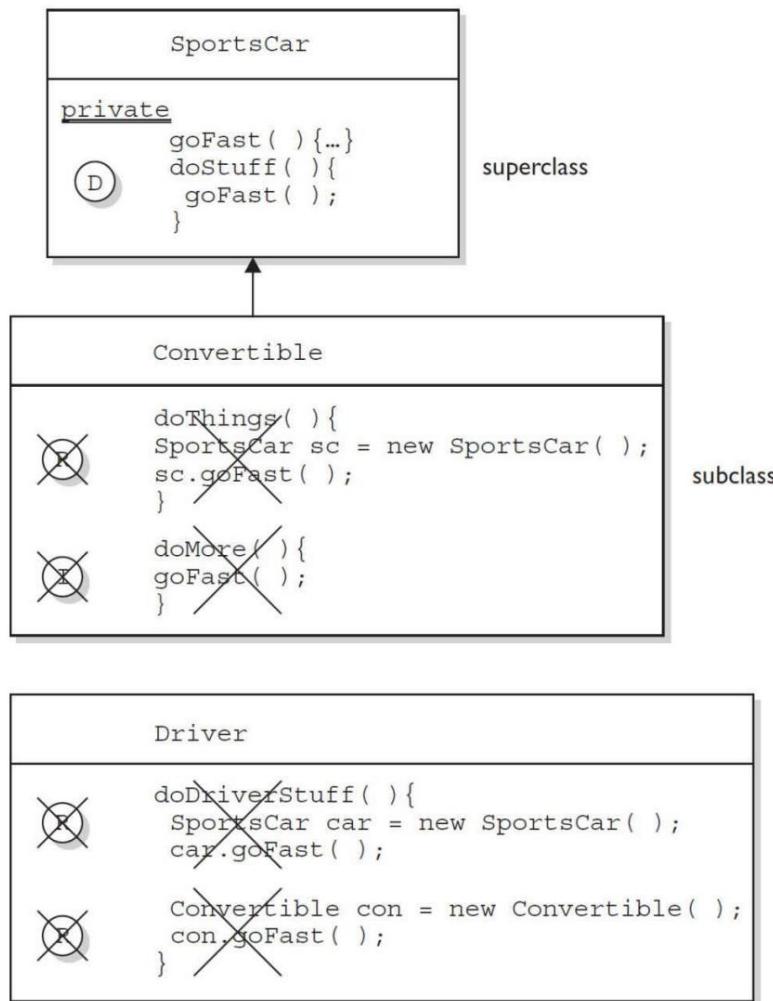
```

%javac Cloo.java
Cloo.java:4: Undefined method: doRooThings()
    System.out.println(doRooThings());
1 error

```

Một phương thức private có thể bị ghi đè bởi một lớp con không? Đó là một câu hỏi thú vị, nhưng câu trả lời là không. Vì lớp con, như chúng ta đã thấy, không thể kế thừa một phương thức private , do đó, nó không thể ghi đè phương thức – việc ghi đè phụ thuộc vào tính kế thừa. Chúng tôi sẽ trình bày chi tiết hơn về hàm ý của điều này sau phần này cũng như trong [Chương 2](#), nhưng hiện tại, chỉ cần nhớ rằng không thể ghi đè phương thức được đánh dấu là private . [Hình 1-3](#) minh họa tác động của các công cụ sửa đổi công cộng và riêng tư đối với các lớp từ các gói giống nhau hoặc khác nhau.

The effect of private access control



Three ways to access a method:

- (D) Invoking a method declared in the same class
- (R) Invoking a method using a reference of the class
- (I) Invoking an inherited method

HÌNH 1-3 Mô hình lớp và cách truy cập công cộng và tư nhân

Thành viên được bảo vệ và mặc định

Lưu ý: Chỉ xin nhắc lại, trong một số phần tiếp theo, khi chúng ta sử dụng từ "mặc định", chúng ta đang nói về kiểm soát truy cập. Chúng tôi KHÔNG nói về loại phương thức giao diện Java 8 mới có thể được khai báo là mặc định.

Các cấp độ kiểm soát truy cập được bảo vệ và mặc định gần như giống hệt nhau, nhưng với

một sự khác biệt quan trọng. Một thành viên mặc định chỉ có thể được truy cập nếu lớp truy cập thành viên đó thuộc cùng một gói, trong khi một thành viên được bảo vệ có thể được truy cập (qua kế thừa) bởi một lớp con ngay cả khi lớp con nằm trong một gói khác. Hãy xem hai lớp sau:

```
package certification;
public class OtherClass {
    void testIt() { // No modifier means method has default
                    // access
        System.out.println("OtherClass");
    }
}
```

Trong một tệp mã nguồn khác, bạn có như sau:

```
package somethingElse;
import certification.OtherClass;
class AccessClass {
    static public void main(String[] args) {
        OtherClass o = new OtherClass();
        o.testIt();
    }
}
```

Như bạn có thể thấy, phương thức `testIt()` trong tệp đầu tiên có quyền truy cập mặc định (nghĩ ở cấp độ gói). Cũng lưu ý rằng lớp `OtherClass` nằm trong một gói khác với `AccessClass`. `AccessClass` có thể sử dụng phương thức `testIt()` không? Nó sẽ gây ra lỗi trình biên dịch? Daniel sẽ kết hôn với Francesca? Giữ nguyên.

Không tìm thấy phương thức nào phù hợp với `testIt()`
trong chứng nhận lớp.`OtherClass.o.testIt();`

Từ các kết quả trước đó, bạn có thể thấy rằng `AccessClass` không thể sử dụng Phương thức `OtherClass testIt()` vì `testIt()` có quyền truy cập mặc định và `AccessClass` không nằm trong cùng một gói với `OtherClass`. Vì vậy, `AccessClass` không thể nhìn thấy nó, trình biên dịch phàn nàn và chúng tôi không biết Daniel và Francesca là ai là.

Hành vi mặc định và được bảo vệ chỉ khác nhau khi chúng ta nói về các lớp con. Nếu từ khóa bảo vệ được sử dụng để định nghĩa một thành viên, bất kỳ lớp con nào của lớp khai báo thành viên có thể truy cập nó thông qua kế thừa. Không quan trọng nếu lớp cha và lớp con nằm trong các gói khác nhau; thành viên lớp cha được bảo vệ vẫn hiển thị với lớp con (mặc dù chỉ hiển thị trong một

theo cách, như chúng ta sẽ thấy một chút sau). Điều này trái ngược với hành vi mặc định, hành vi này không cho phép một lớp con truy cập vào một thành viên của lớp cha trừ khi lớp con nằm trong cùng một gói với lớp cha.

Trong khi quyền truy cập mặc định không mở rộng bất kỳ sự cân nhắc đặc biệt nào đối với các lớp con (bạn đang ở trong gói hoặc bạn không ở trong gói), thì công cụ sửa đổi được bảo vệ tôn trọng mối quan hệ cha-con, ngay cả khi lớp con chuyển đi (và tham gia một gói mới). Vì vậy, khi bạn nghĩ về quyền truy cập mặc định, hãy nghĩ đến giới hạn gói.

Không có ngoại lệ. Nhưng khi bạn nghĩ rằng được bảo vệ, hãy nghĩ đến gói + trẻ em. Một lớp có thành viên được bảo vệ đang đánh dấu thành viên đó có quyền truy cập mức gói cho tất cả các lớp, nhưng với một ngoại lệ đặc biệt cho các lớp con bên ngoài gói.

Nhưng điều đó có nghĩa là gì đối với một lớp con bên ngoài gói có quyền truy cập vào một thành viên của lớp cha (cha)? Nó có nghĩa là lớp con kế thừa thành viên. Tuy nhiên, điều đó không có nghĩa là lớp con ngoài gói có thể truy cập thành viên bằng cách sử dụng tham chiếu đến một thể hiện của lớp cha. Nói cách khác, bảo vệ = kế thừa. Được bảo vệ không có nghĩa là lớp con có thể coi thành viên của lớp cha được bảo vệ như nó là công khai. Vì vậy, nếu lớp con bên ngoài gói nhận được một tham chiếu đến lớp cha (ví dụ: bằng cách tạo một thể hiện của lớp cha ở đâu đó trong mã của lớp con), thì lớp con không thể sử dụng toán tử dấu chấm trên tham chiếu lớp cha để truy cập thành viên được bảo vệ. . . Đối với một lớp con ngoài gói, một thành viên được bảo vệ cũng có thể là mặc định (hoặc thậm chí là riêng tư), khi lớp con đang sử dụng tham chiếu đến lớp cha. Lớp con chỉ có thể nhìn thấy thành viên được bảo vệ thông qua kế thừa.

Bạn đang bối rối? Hãy chờ đợi và tất cả sẽ trở nên rõ ràng hơn với phần tiếp theo lô ví dụ mã.

Chi tiết được bảo vệ

Chúng ta hãy xem xét một biến cá thể được bảo vệ (hãy nhớ, một biến cá thể là một thành viên) của một lớp cha.

```
package certification;
public class Parent {
    protected int x = 9; // protected access
}
```

Đoạn mã trước khai báo biến x là được bảo vệ. Điều này làm cho biến có thể truy cập được đối với tất cả các lớp khác bên trong gói chứng nhận, cũng như có thể được kế thừa bởi bất kỳ lớp con nào bên ngoài gói.

Bây giờ, hãy tạo một lớp con trong một gói khác và cố gắng sử dụng

biến x (mà lớp con kế thừa):

```
package other;                                // Different package
import certification.Parent;
class Child extends Parent {
    public void testIt() {
        System.out.println("x is " + x); // No problem; Child
                                         // inherits x
    }
}
```

Mã trước đó biên dịch tốt. Tuy nhiên, lưu ý rằng lớp Con đang truy cập biến được bảo vệ thông qua kế thừa. Hãy nhớ rằng bất cứ khi nào chúng ta nói về một lớp con có quyền truy cập vào thành viên của lớp cha, chúng ta có thể đang nói về lớp con kế thừa thành viên đó, không chỉ đơn giản là truy cập thành viên thông qua tham chiếu đến một thể hiện của lớp cha (cách mà bất kỳ lớp nào khác sẽ truy cập nó). Xem điều gì sẽ xảy ra nếu lớp con Con (bên ngoài gói của lớp cha) cố gắng truy cập vào một biến được bảo vệ bằng cách sử dụng tham chiếu lớp cha :

```
package other;
import certification.Parent;
class Child extends Parent {
    public void testIt() {
        System.out.println("x is " + x);           // No problem; Child
                                                   // inherits x
        Parent p = new Parent();                  // Can we access x using
                                                   // the p reference?
        System.out.println("X in parent is " + p.x); // Compiler error!
    }
}
```

Trình biên dịch rất vui khi cho chúng tôi thấy vấn đề:

```
%javac -d . other/Child.java
other/Child.java:9: x has protected access in certification.Parent
System.out.println("X in parent is " + p.x);
                                         ^
1 error
```

Cho đến nay, chúng tôi đã thiết lập rằng một thành viên được bảo vệ cơ bản có gói cấp hoặc quyền truy cập mặc định vào tất cả các lớp ngoại trừ các lớp con. Chúng tôi đã thấy rằng các lớp con bên ngoài gói có thể kế thừa một thành viên được bảo vệ. Cuối cùng, chúng tôi đã thấy rằng các lớp con bên ngoài gói không thể sử dụng tham chiếu lớp cha để

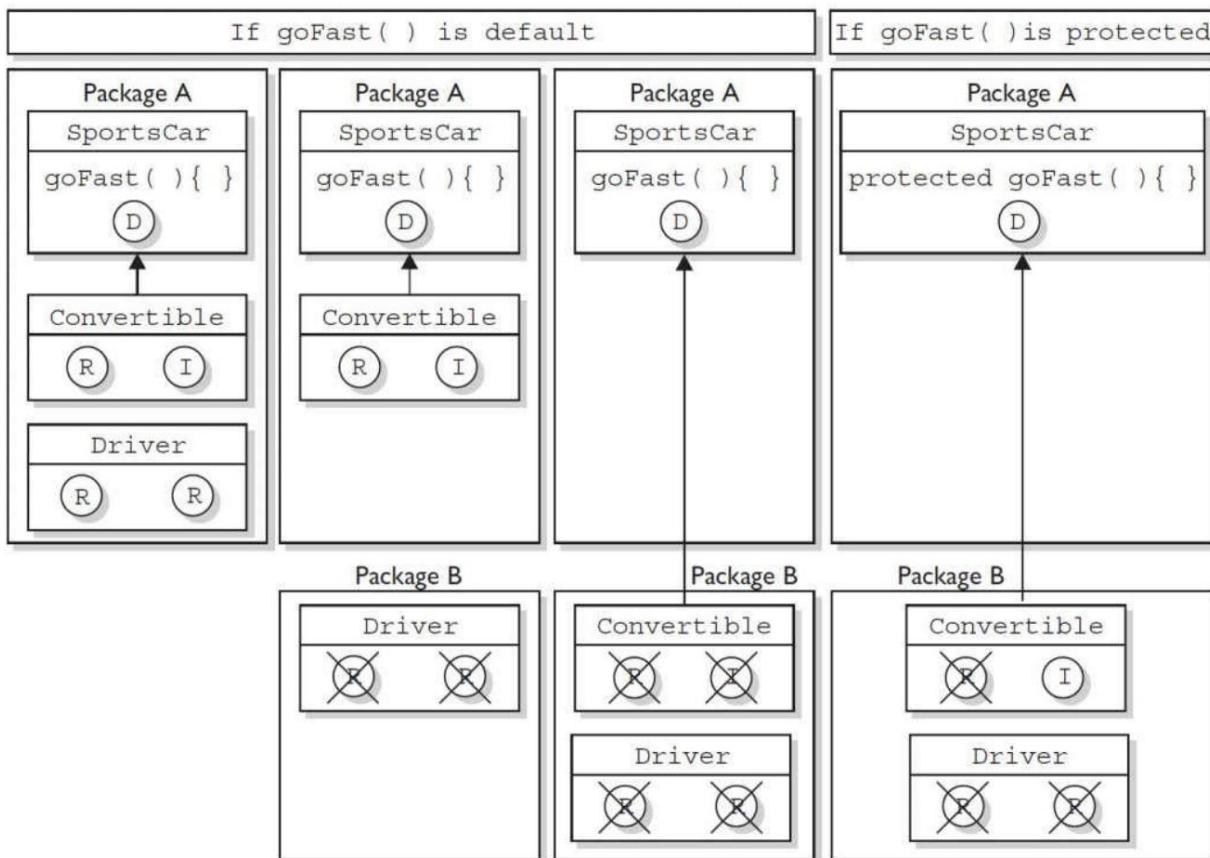
truy cập một thành viên được bảo vệ. Đối với một lớp con bên ngoài gói, thành viên được bảo vệ chỉ có thể được truy cập thông qua kế thừa.

Nhưng vẫn còn một vấn đề nữa mà chúng tôi chưa xem xét: Thành viên được bảo vệ trông như thế nào đối với các lớp khác đang cố gắng sử dụng lớp con-bên ngoài-gói để truy cập thành viên lớp siêu được bảo vệ kế thừa của lớp con? Ví dụ: sử dụng các lớp Parent / Child trước đây của chúng tôi, điều gì sẽ xảy ra nếu một số lớp khác -Neighbor, chẳng hạn - trong cùng một gói với Child (lớp con) có tham chiếu đến một cá thể Con và muốn truy cập vào biến thành viên x? Nói cách khác, thành viên được bảo vệ đó hoạt động như thế nào khi lớp con đã kế thừa nó?

Nó có duy trì trạng thái được bảo vệ để các lớp trong gói Child có thể nhìn thấy nó không?

Không! Khi lớp con-ngoài-gói kế thừa thành viên được bảo vệ, thành viên đó (như được lớp con kế thừa) sẽ trở thành riêng tư đối với bất kỳ mã nào bên ngoài lớp con, ngoại trừ các lớp con của lớp con. Vì vậy, nếu class Neighbor khởi tạo một đối tượng Child, thì ngay cả khi class Neighbor nằm trong cùng một gói với class Child, class Neighbor sẽ không có quyền truy cập vào biến x được kế thừa (nhưng được bảo vệ) của Child's. [Hình 1-4](#) minh họa ảnh hưởng của quyền truy cập được bảo vệ đối với các lớp và lớp con trong các gói giống nhau hoặc khác nhau.

Chà! Điều đó kết thúc được bảo vệ, công cụ sửa đổi bị hiểu lầm nhiều nhất trong Java. Một lần nữa, nó chỉ được sử dụng trong những trường hợp rất đặc biệt, nhưng bạn có thể tin tưởng vào việc nó hiển thị trong bài kiểm tra. Böyle giờ chúng ta đã đe dọa cập đến công cụ sửa đổi được bảo vệ, chúng ta sẽ chuyển sang quyền truy cập thành viên mặc định, một miếng bánh so với được bảo vệ.

**Key:**

D `goFast(){}
doStuff(){}
goFast();`

Where `goFast` is Declared in the same class.

R

```
doThings() {
    SportsCar sc = new SportsCar();
    sc.goFast();
}
```

Invoking `goFast()` using a Reference to the class in which `goFast()` was declared.

I

```
doMore() {
    goFast();
}
```

Invoking the `goFast()` method Inherited from a superclass.

HÌNH 1-4 Mô hình của quyền truy cập được bảo vệ

Chi tiết măc định

Hãy bắt đầu với hành vi măc định của một thành viên trong lớp cha. Chúng tôi sẽ sửa đổi thành viên x của Phụ huynh để đặt nó thành măc định.

```
package certification;
public class Parent {
    int x = 9; // No access modifier, means default
                // (package) access
}
```

Lưu ý rằng chúng tôi đã không đặt một công cụ sửa đổi truy cập trước biến x.

Hãy nhớ rằng nếu bạn không nhập công cụ sửa đổi truy cập trước khai báo lớp hoặc thành viên, thì điều khiển truy cập là mặc định, có nghĩa là cấp độ gói. Bây giờ chúng ta sẽ cố gắng truy cập thành viên mặc định từ lớp Con mà chúng ta đã thấy trước đó.

Khi chúng tôi cố gắng biên dịch tệp Child.java , chúng tôi gặp lỗi như sau:

```
Child.java:4: Undefined variable: x
        System.out.println("Variable x is " + x);
1 error
```

Trình biên dịch đưa ra lỗi tương tự như khi một thành viên được khai báo là private.

Lớp con Con (trong một gói khác với lớp cha mẹ) không thể nhìn thấy hoặc sử dụng thành viên lớp cha mặc định x! Bây giờ, điều gì về quyền truy cập mặc định cho hai lớp trong cùng một gói?

```
package certification;
public class Parent{
    int x = 9; // default access
}
```

Và trong lớp thứ hai, bạn có những thứ sau:

```
package certification;
class Child extends Parent{
    static public void main(String[] args) {
        Child sc = new Child();
        sc.testIt();
    }
    public void testIt() {
        System.out.println("Variable x is " + x); // No problem;
    }
}
```

Tệp nguồn trước đó biên dịch tốt, và lớp Con chạy và hiển thị giá trị của x. Chỉ cần nhớ rằng các thành viên mặc định chỉ hiển thị với các lớp con nếu các lớp con đó nằm trong cùng một gói với lớp cha.

Biến cục bộ và công cụ sửa đổi quyền truy cập

Biến cục bộ và công cụ sửa đổi quyền truy cập

Các công cụ sửa đổi quyền truy cập có thể được áp dụng cho các biến cục bộ không? KHÔNG!

Không bao giờ có trường hợp nào mà công cụ sửa đổi quyền truy cập có thể được áp dụng cho một biến cục bộ, vì vậy hãy chú ý đến mã như sau:

```
class Foo {
    void doStuff() {
        private int x = 7;
        this.doMore(x);
    }
}
```

Bạn có thể chắc chắn rằng bất kỳ biến cục bộ nào được khai báo với một công cụ sửa đổi truy cập sẽ không biên dịch. Trên thực tế, chỉ có một công cụ sửa đổi có thể được áp dụng cho các biến cục bộ – cuối cùng.

Điều đó làm điều đó cho cuộc thảo luận của chúng tôi về công cụ sửa đổi quyền truy cập thành viên. [Bảng 1-2](#) hiển thị tất cả các kết hợp của quyền truy cập và khả năng hiển thị; bạn thực sự nên dành một chút thời gian cho nó. Tiếp theo, chúng ta sẽ đi sâu vào các công cụ sửa đổi (nonaccess) khác mà bạn có thể áp dụng cho các khai báo thành viên.

BẢNG 1-2 Khả năng quyền truy cập vào các thành viên trong lớp

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	<i>Yes, through inheritance</i>	No	No
From any nonsubclass class outside the package	Yes	No	No	No

Nonaccess Member Modifier

Chúng ta đã thảo luận về quyền truy cập thành viên, đề cập đến việc liệu mã từ một lớp có thể gọi một phương thức (hoặc truy cập một biến thể hiện) từ một lớp khác hay không. Điều đó vẫn để lại một lượng lớn các bổ trợ khác mà bạn có thể sử dụng trên các khai báo thành viên.

Hai điều bạn đã quen thuộc – cuối cùng và tóm tắt – vì chúng tôi đã áp dụng

chúng vào khai báo lớp trước đó trong chương này. Nhưng chúng ta vẫn phải xem nhanh qua tạm thời, đồng bộ hóa, bản địa, nghiêm ngặt, và sau đó là một cái nhìn dài về Big One, tĩnh, ở phần sau của chương.

Trước tiên, chúng ta sẽ xem xét các công cụ sửa đổi được áp dụng cho các phương thức, sau đó là xem xét các công cụ sửa đổi được áp dụng cho các biến phiên bản. Chúng ta sẽ kết thúc phần này với việc xem xét cách static hoạt động khi được áp dụng cho các biến và phương thức.

Phương pháp cuối cùng

Từ khóa cuối cùng ngăn một phương thức bị ghi đè trong một lớp con và thường được sử dụng để thực thi chức năng API của một phương thức. Ví dụ, lớp Thread có một phương thức gọi là `isAlive()` để kiểm tra xem một luồng có còn hoạt động hay không. Tuy nhiên, nếu bạn mở rộng lớp Thread, thực sự không có cách nào để bạn có thể tự mình triển khai chính xác phương thức này (đối với một điều là nó sử dụng mã gốc), vì vậy các nhà thiết kế đã tạo ra nó cuối cùng. Cũng như bạn không thể phân lớp con của lớp String (vì chúng ta cần có thể tin tưởng vào hành vi của một đối tượng String), bạn không thể ghi đè nhiều phương thức trong thư viện lớp lõi. Hạn chế không thể bị ghi đè này cung cấp sự an toàn và bảo mật, nhưng bạn nên sử dụng nó một cách hết sức thận trọng. Việc ngăn chặn một lớp con ghi đè một phương thức sẽ ngăn chặn nhiều lợi ích của OO, bao gồm khả năng mở rộng thông qua tính đa hình. Một khai báo phương thức cuối cùng điển hình trông giống như sau:

```
class SuperClass{
    public final void showSample() {
        System.out.println("One thing.");
    }
}
```

Việc mở rộng SuperClass là hợp pháp, vì lớp này không được đánh dấu là cuối cùng, nhưng chúng tôi không thể ghi đè phương thức cuối cùng `showSample()`, vì đoạn mã sau có gắng thực hiện:

```
class SubClass extends SuperClass{
    public void showSample() { // Try to override the final
        // superclass method
        System.out.println("Another thing.");
    }
}
```

Có gắng biên dịch mã trước đó cho chúng ta một cái gì đó như sau:

```
%javac FinalTest.java
FinalTest.java:5: The method void showSample() declared in class
SubClass cannot override the final method of the same signature
declared in class SuperClass.
Final methods cannot be overridden.
    public void showSample() { }
1 error
```

Đối số cuối cùng Đối

số phương thức là khai báo biến xuất hiện ở giữa các dấu ngoặc đơn trong khai báo phương thức. Một khai báo phương thức đi kèm với nhiều đối số trông như thế này:

```
public Record getRecord (int fileNumber, int recNumber) {}
```

Các đối số của phương thức về cơ bản giống như các biến cục bộ. Trong ví dụ trước, cả hai biến fileNumber và recNumber sẽ tuân theo tất cả các quy tắc được áp dụng cho các biến cục bộ. Điều này có nghĩa là họ cũng có thể có công cụ sửa đổi cuối cùng:

```
public Record getRecord (int fileNumber, end int recNumber) {}
```

Trong ví dụ này, biến recNumber được khai báo là cuối cùng, tất nhiên, có nghĩa là nó không thể được sửa đổi trong phương thức. Trong trường hợp này, "đã sửa đổi" có nghĩa là gán lại một giá trị mới cho biến. Nói cách khác, tham số cuối cùng phải giữ nguyên giá trị như đối số có khi nó được truyền vào phương thức.

Phương pháp trừu tượng

Một phương thức trừu tượng là một phương thức đã được khai báo (dưới dạng trừu tượng) nhưng không được thực thi. Nói cách khác, phương thức không chứa mã chức năng. Và nếu bạn nhớ lại từ phần trước "Các lớp trừu tượng", một khai báo phương thức trừu tượng thậm chí không có dấu ngoặc nhọn cho vị trí mã triển khai, mà thay vào đó đóng bằng dấu chấm phẩy. Nói cách khác, nó không có thân phương thức. Bạn đánh dấu một phương thức trừu tượng khi bạn muốn buộc các lớp con cung cấp việc triển khai. Ví dụ: nếu bạn viết một lớp trừu tượng Car với phương thức goUpHill (), bạn có thể muốn buộc mỗi loại con của Car xác định hành vi goUpHill () của riêng nó, cụ thể cho loại ô tô cụ thể đó.

```
công khai trừu tượng void showSample();
```

Lưu ý rằng phương thức trừu tượng kết thúc bằng dấu chấm phẩy thay vì dấu ngoặc nhọn. Sẽ là bất hợp pháp nếu chỉ có một phương thức trừu tượng trong một lớp không được khai báo rõ ràng là trừu tượng! Nhìn vào lớp bất hợp pháp sau:

```
public class IllegalClass{
    public abstract void doIt();
}
```

The preceding class will produce the following error if you try to compile it:

```
IllegalClass.java:1: class IllegalClass must be declared
abstract.
It does not define void doIt() from class IllegalClass.
public class IllegalClass{
1 error
```

Tuy nhiên, bạn có thể có một lớp trừu tượng không có phương thức trừu tượng . Các ví dụ sau sẽ biên dịch tốt:

```
public abstract class LegalClass{
    void goodMethod() {
        // lots of real implementation code here
    }
}
```

Trong ví dụ trước, goodMethod () không trừu tượng. Ba manh mỗi khác nhau cho bạn biết nó không phải là một phương pháp trừu tượng :

- Phương thức này không được đánh dấu là trừu tượng.
- Khai báo phương thức bao gồm dấu ngoặc nhọn, trái ngược với kết thúc bằng dấu chấm phẩy. Nói cách khác, phương thức có một thân phương thức.
- Phương thức này có thể cung cấp mã triển khai thực tế bên trong dấu ngoặc nhọn.

Bất kỳ lớp nào mở rộng một lớp trừu tượng phải thực hiện tất cả các phương thức trừu tượng của lớp cha, trừ khi lớp con cũng trừu tượng. Quy tắc là: Lớp con cụ thể đầu tiên của một lớp trừu tượng phải thực hiện tất cả các phương thức trừu tượng của lớp cha.

Concrete chỉ có nghĩa là nonabstract, vì vậy nếu bạn có một lớp trừu tượng mở rộng một lớp trừu tượng khác, lớp con trừu tượng không cần cung cấp các triển khai cho các phương thức trừu tượng được kế thừa . Tuy nhiên, dù sớm hay muộn, ai đó sẽ tạo ra một lớp con nonabstract (nói cách khác, một lớp

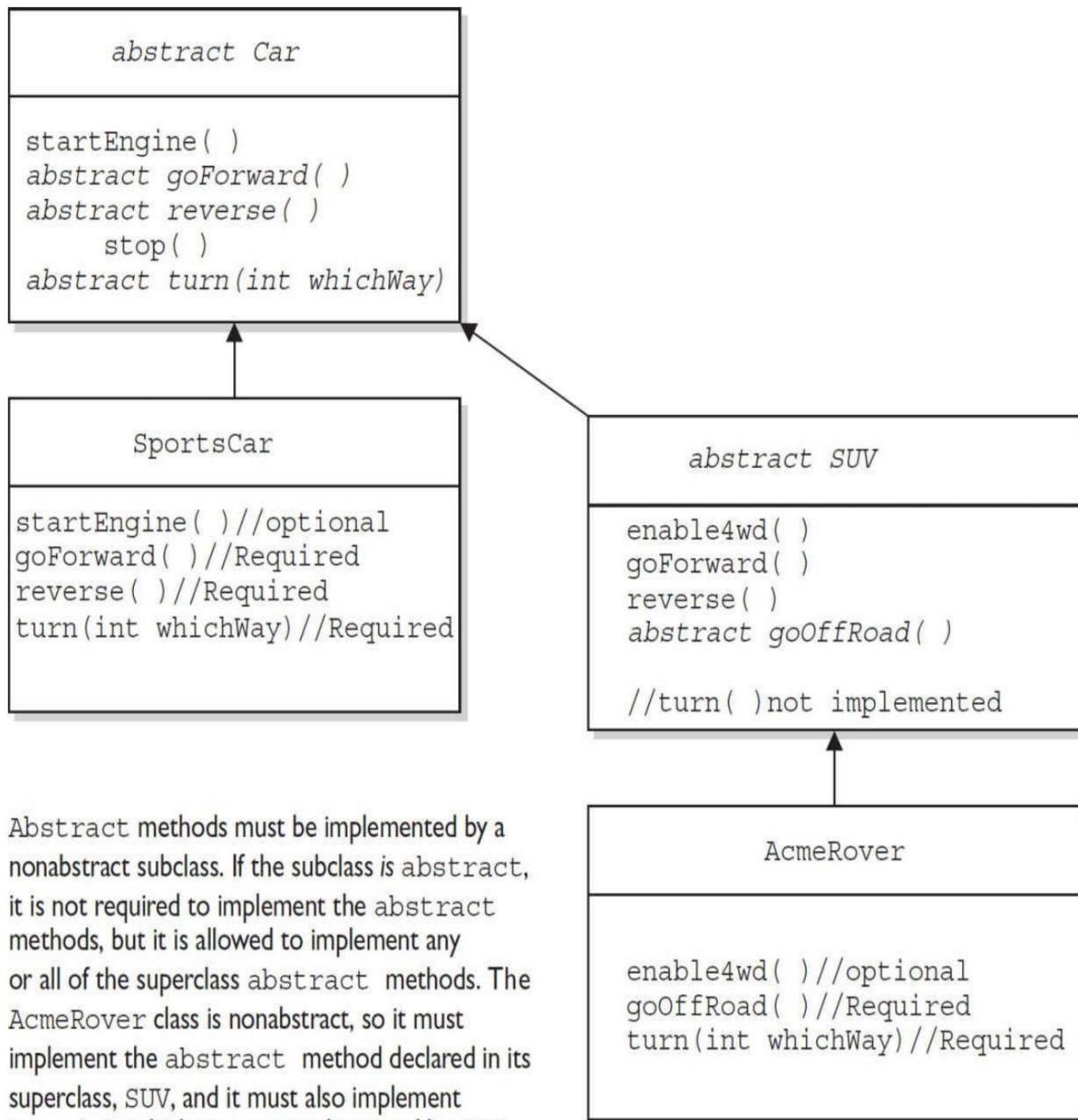
có thể được khởi tạo), và lớp con đó sẽ phải triển khai tất cả các phương thức trừu tượng từ cây kế thừa. Ví dụ sau minh họa một cây kế thừa với hai lớp trừu tượng và một lớp cụ thể:

```
public abstract class Vehicle {
    private String type;
    public abstract void goUpHill(); // Abstract method
    public String getType() {           // Non-abstract method
        return type;
    }
}

public abstract class Car extends Vehicle {
    public abstract void goUpHill(); // Still abstract
    public void doCarThings() {
        // special car code goes here
    }
}

public class Mini extends Car {
    public void goUpHill() {
        // Mini-specific going uphill code
    }
}
```

Vậy class Mini có bao nhiêu phương thức ? Số ba. Nó kế thừa cả hai phương thức getType () và doCarThings () vì chúng là công khai và cụ thể (nonabstract). Nhưng vì goUpHill () là trừu tượng trong lớp cha Phương tiện và không bao giờ được triển khai trong lớp Xe (vì vậy nó vẫn là trừu tượng), nó có nghĩa là lớp Mini – như lớp cụ thể đầu tiên bên dưới Xe – phải triển khai phương thức goUpHill () . Nói cách khác, lớp Mini không thể chuyển buck (của việc triển khai phương thức trừu tượng) cho lớp tiếp theo xuống cây kế thừa, nhưng lớp Car có thể, vì Car, giống như Vehicle, là trừu tượng. [Hình 1-5](#) minh họa các tác động của công cụ sửa đổi trừu tượng lên các lớp con cụ thể và trừu tượng .



HÌNH 1-5

Hình ảnh minh họa công cụ sửa đổi trừu tượng lên các lớp con cụ thể và trừu tượng

Tìm kiếm các lớp cụ thể không cung cấp triển khai phương thức cho các phương thức trừu tượng của lớp cha. Mã sau sẽ không biên dịch:

```

public abstract class A {
    abstract void foo();
}
class B extends A {
    void foo(int I) { }
}

```

Lớp B sẽ không biên dịch vì nó không triển khai phương thức trừu tượng kế thừa foo(). Mặc dù phương thức foo(int I) trong lớp B có thể dường như là một triển khai của phương thức trừu tượng của lớp cha, nhưng nó chỉ đơn giản là một phương thức được nạp chồng (một phương thức sử dụng cùng một mã định danh, nhưng các đối số khác nhau), vì vậy nó không đáp ứng các yêu cầu để triển khai phương thức trừu tượng của lớp cha. Chúng ta sẽ xem xét sự khác biệt giữa quá tải và ghi đè chi tiết trong [Chương 2](#).

Một phương thức không bao giờ có thể được đánh dấu là trừu tượng và cuối cùng, hoặc vừa trừu tượng vừa riêng tư. Hãy nghĩ về nó – các phương thức trừu tượng phải được thực hiện (về cơ bản có nghĩa là bị ghi đè bởi một lớp con), trong khi các phương thức cuối cùng và riêng tư không bao giờ được ghi đè bởi một lớp con. Hay nói cách khác, một chỉ định trừu tượng có nghĩa là lớp cha không biết gì về cách các lớp con sẽ hoạt động trong phương thức đó, trong khi một chỉ định cuối cùng có nghĩa là lớp cha biết mọi thứ về cách tất cả các lớp con (tuy nhiên chúng có thể be) nên cư xử theo phương pháp đó. Các bộ ngữ trừu tượng và cuối cùng hầu như đối lập nhau. Bởi vì các phương thức private thậm chí không thể được nhìn thấy bởi một lớp con (chứ đừng nói đến kế thừa), chúng cũng không thể bị ghi đè, vì vậy chúng cũng không thể được đánh dấu là trừu tượng.

Cuối cùng, bạn cần biết rằng – đối với các lớp cấp cao nhất – công cụ sửa đổi trừu tượng không bao giờ có thể được kết hợp với công cụ sửa đổi tĩnh. Chúng tôi sẽ đề cập đến các phương thức tĩnh ở phần sau trong mục tiêu này, nhưng bây giờ chỉ cần nhớ rằng những điều sau đây sẽ là bất hợp pháp:

```
truu tuuong static void doStuff();
```

Và nó sẽ cung cấp cho bạn một lỗi mà bây giờ sẽ quen thuộc:

```
MyClass.java:2: sự kết hợp bất hợp pháp của các bộ ngữ: truu tuuong và tĩnh truu tuuong
tinh void doStuff();
```

Phương pháp đồng bộ hóa

Từ khóa được đồng bộ hóa chỉ ra rằng một phương pháp chỉ có thể được truy cập bởi một

chủ đề tại một thời điểm. Khi bạn đang nghiên cứu về OCP 8 của mình, bạn sẽ nghiên cứu kỹ từ khóa được đồng bộ hóa , nhưng hiện tại ... tất cả những gì chúng tôi quan tâm là biết rằng công cụ sửa đổi được đồng bộ hóa chỉ có thể được áp dụng cho các phương thức – không phải biến, không phải lớp, chỉ các phương pháp. Một khai báo được đồng bộ hóa điển hình trông giống như sau:

```
công khai đồng bộ hóa Bản ghi truy xuấtUserInfo (int id) {}
```

Bạn cũng nên biết rằng công cụ sửa đổi được đồng bộ hóa có thể được đối sánh với bất kỳ cấp độ nào trong số bốn cấp độ kiểm soát truy cập (có nghĩa là nó có thể được ghép nối với bất kỳ từ khóa nào trong số ba từ khóa công cụ sửa đổi truy cập).

Phương pháp bản địa

Công cụ sửa đổi gốc chỉ ra rằng một phương thức được triển khai trong mã phụ thuộc vào nền tảng, thường ở C. Bạn không cần biết cách sử dụng các phương thức gốc cho bài kiểm tra, ngoài việc biết rằng phương thức gốc là một công cụ sửa đổi (do đó là một từ khóa dành riêng) và native chỉ có thể được áp dụng cho các phương thức – không phải lớp, không phải biến, chỉ là phương thức. Lưu ý rằng phần thân của một phương thức gốc phải là dấu chấm phẩy (;) (giống như các phương thức trừu tượng), cho biết rằng việc triển khai bị bỏ qua.

Các phương thức nghiêm

ngặt trước đó Chúng ta đã xem xét việc sử dụng nghiêm ngặt như một công cụ sửa đổi lớp, nhưng ngay cả khi bạn không khai báo một lớp dưới dạng nghiêm ngặt, bạn vẫn có thể khai báo một phương thức riêng lẻ dưới dạng nghiêm ngặt. Hãy nhớ rằng, nghiêm ngặt buộc các dấu chấm động (và bất kỳ phép toán dấu phẩy động nào) phải tuân theo tiêu chuẩn IEEE 754. Với nghiêm ngặt, bạn có thể dự đoán cách các dấu chấm động của bạn sẽ hoạt động bất kể nền tảng cơ bản mà JVM đang chạy. Nhược điểm là nếu nền tảng cơ bản có khả năng hỗ trợ độ chính xác cao hơn, thì phương pháp nghiêm ngặt sẽ không thể tận dụng được lợi thế của nó.

Bạn sẽ muốn học IEEE 754 nếu bạn cần thứ gì đó để giúp bạn ngã ngũ. Tuy nhiên, đối với bài kiểm tra, bạn không cần biết bất cứ điều gì về nghiêm ngặt ngoài những gì nó được sử dụng - rằng nó có thể sửa đổi khai báo lớp hoặc phương thức và một biến không bao giờ có thể được khai báo là nghiêm ngặt.

Các phương thức có danh sách đối số biến (varargs)

Java cho phép bạn tạo các phương thức có thể nhận nhiều đối số.

Tùy thuộc vào nơi bạn nhìn, bạn có thể nghe thấy khả năng này được gọi là "danh sách đối số có độ dài thay đổi", "đối số biến", "varargs", "varargs" hoặc

"Danh sách đối số có độ dài thay đổi", "đối số có thể thay đổi", "varargs", "varargs" hoặc yêu thích cá nhân của chúng tôi (từ bộ phận giải mã), "tham số độ hiém có thể thay đổi". Tất cả chúng đều giống nhau, và chúng tôi sẽ sử dụng thuật ngữ "kỳ đà" từ đây trở đi.

Như một chút thông tin cơ bản, chúng tôi muốn làm rõ cách chúng tôi sẽ sử dụng các thuật ngữ "Đối số" và "tham số" trong suốt cuốn sách này:

- **đối số** Những thứ bạn chỉ định giữa các dấu ngoặc đơn khi bạn gọi một phương thức:

```
doStuff("a", 2); // invoking doStuff, so "a" & 2 are
                  // arguments
```

- **tham số** Những thứ trong chữ ký của phương thức chỉ ra những gì phương thức phải nhận khi nó được gọi:

```
void doStuff(String s, int a) { } // we're expecting two
                                  // parameters:
                                  // String and int
```

Hãy xem lại các quy tắc khai báo cho kỳ đà:

- **Kiểu var-arg** Khi bạn khai báo một tham số var-arg, bạn phải chỉ định kiểu của (các) đối số mà tham số này của phương thức của bạn có thể nhận được. (Đây có thể là kiểu nguyên thủy hoặc kiểu đối tượng.)
- Cú pháp cơ bản Để khai báo một phương thức sử dụng tham số var-arg, bạn theo kiểu có dấu chấm lửng (...), dấu cách, sau đó là tên của mảng sẽ chứa các tham số nhận được.
- Các tham số khác Việc có các tham số khác trong một phương thức sử dụng var-arg là hợp pháp.
- Giới hạn của Var-arg var-arg phải là tham số cuối cùng trong chữ ký của phương thức và bạn chỉ có thể có một var-arg trong một phương thức.
- Hãy xem xét một số khai báo var-arg hợp pháp và bất hợp pháp:

Hợp pháp:

```
void doStuff(int... x) { }           // expects from 0 to many ints
                                      // as parameters
void doStuff2(char c, int... x) { }   // expects first a char,
                                      // then 0 to many ints
void doStuff3(Animal... animal) { }  // 0 to many Animals
```

Không hợp lệ:

```
void doStuff4(int x...) { }           // bad syntax
void doStuff5(int... x, char... y) { } // too many var-args
void doStuff6(String... s, byte b) { } // var-arg must be last
```

Khai báo hàm tạo

Trong Java, các đối tượng được xây dựng. Mỗi khi bạn tạo một đối tượng mới, ít nhất một hàm tạo được gọi. Mỗi lớp đều có một phương thức khởi tạo, mặc dù nếu bạn không tạo nó một cách rõ ràng, trình biên dịch sẽ tạo một phương thức khởi tạo cho bạn. Có rất nhiều quy tắc liên quan đến các hàm tạo, và chúng tôi đang lưu phần thảo luận chi tiết của mình cho [Chương 2](#).

Bây giờ, chúng ta hãy tập trung vào các quy tắc khai báo cơ bản. Đây là một ví dụ đơn giản:

```
class Foo {
    protected Foo() { }           // this is Foo's constructor
    protected void Foo() { }     // this is a badly named, but legal, method
}
```

Điều đầu tiên cần chú ý là các hàm tạo trông rất giống các phương thức. Một sự khác biệt chính là một phương thức khởi tạo không bao giờ có thể có kiểu trả về. bao giờ hết! Tuy nhiên, khai báo hàm tạo có thể có tất cả các công cụ sửa đổi truy cập thường và chúng có thể nhận các đối số (bao gồm cả varargs), giống như các phương thức. NGUYÊN TẮC LỚN khác cần hiểu về các hàm tạo là chúng phải có cùng tên với lớp mà chúng được khai báo. Các trình xây dựng không thể được đánh dấu là tĩnh (xét cho cùng, chúng được liên kết với việc khởi tạo đối tượng) và chúng không thể được đánh dấu là cuối cùng hoặc trừu tượng (vì chúng không thể được ghi đè). Dưới đây là một số khai báo về hàm tạo hợp pháp và bất hợp pháp:

```

class Foo2 {
    // legal constructors
    Foo2() { }
    private Foo2(byte b) { }
    Foo2(int x) { }
    Foo2(int x, int... y) { }
    // illegal constructors
    void Foo2() { } // it's a method, not a constructor
    Foo() { } // not a method or a constructor
    Foo2(short s); // looks like an abstract method
    static Foo2(float f) { } // can't be static
    final Foo2(long x) { } // can't be final
    abstract Foo2(char c) { } // can't be abstract
    Foo2(int... x, int t) { } // bad var-arg syntax
}

```

Khai báo biến

Có hai loại biến trong Java:

- Nguyên thủy Một nguyên thủy có thể là một trong tám kiểu: char, boolean, byte, short, int, long, double, hoặc float. Khi một nguyên thủy đã được khai báo, kiểu nguyên thủy của nó không bao giờ có thể thay đổi, mặc dù trong hầu hết các trường hợp, giá trị của nó có thể thay đổi.
- Các biến tham chiếu Một biến tham chiếu được sử dụng để tham chiếu đến (hoặc truy cập) một đối tượng. Một biến tham chiếu được khai báo là một kiểu cụ thể và kiểu đó không bao giờ có thể thay đổi được. Một biến tham chiếu có thể được sử dụng để tham chiếu đến bất kỳ đối tượng nào của kiểu được khai báo hoặc của kiểu con của kiểu đã khai báo (kiểu tương thích). Chúng ta sẽ nói nhiều hơn về việc sử dụng một biến tham chiếu để tham chiếu đến một kiểu con trong [Chương 2](#), khi chúng ta thảo luận về tính đa hình.

Khai báo Nguyên thủy và Dải nguyên thủy Các biến nguyên

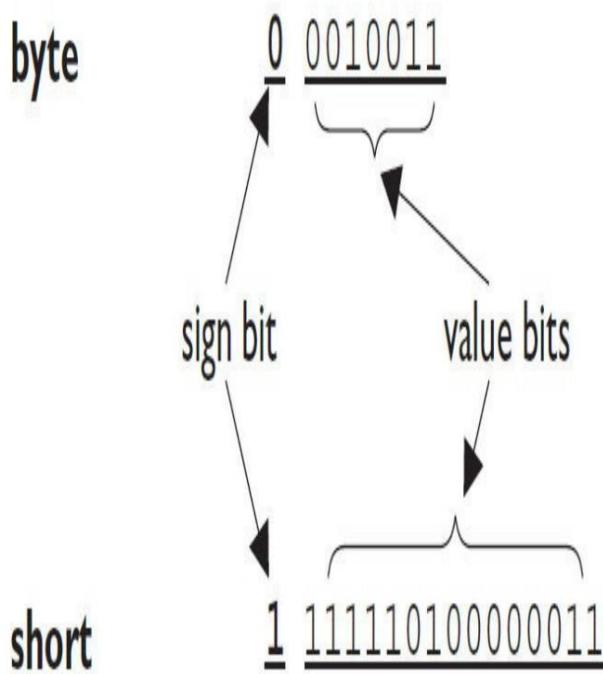
thủy có thể được khai báo dưới dạng biến lớp (tĩnh), biến thể hiện, tham số phương thức hoặc biến cục bộ. Bạn có thể khai báo một hoặc nhiều kiểu nguyên thủy, cùng kiểu nguyên thủy, trong một dòng duy nhất. Trong [Chương 3](#), chúng ta sẽ thảo luận về các cách khác nhau mà chúng có thể được khởi tạo, nhưng bây giờ chúng ta sẽ để lại cho bạn một vài ví dụ về khai báo biến nguyên thủy:

```
byte b;  
boolean myBooleanPrimitive;  
int x, y, z; // declare three int primitives
```

Trong các phiên bản trước của kỳ thi, bạn cần biết cách tính phạm vi cho tất cả các nguyên thủy Java. Đối với bài kiểm tra hiện tại, bạn có thể bỏ qua một số chi tiết đó, nhưng điều quan trọng vẫn là hiểu rằng đối với các kiểu số nguyên, chuỗi từ nhỏ đến lớn là byte, short, int và long, và số nhân đôi lớn hơn số float.

Bạn cũng sẽ cần biết rằng các kiểu số (cả kiểu số nguyên và dấu phẩy động) đều có dấu và điều đó ảnh hưởng đến phạm vi của chúng như thế nào. Đầu tiên, chúng ta hãy xem lại các khái niệm.

Tất cả sáu kiểu số trong Java đều được tạo thành từ một số byte 8 bit nhất định và có dấu, có nghĩa là chúng có thể là số âm hoặc số dương. Bit ngoài cùng bên trái (chữ số có nghĩa nhất) được sử dụng để biểu thị dấu hiệu, trong đó 1 có nghĩa là âm và 0 có nghĩa là dương, như trong [Hình 1-6](#). Phần còn lại của các bit đại diện cho giá trị, sử dụng ký hiệu bổ sung của hai.



sign bit: 0 = positive

1 = negative

value bits:

byte: 7 bits can represent 2^7 or

128 different values:

0 thru 127 -or- -128 thru -1

short: 15 bits can represent

2^{15} or 32768 values:

0 thru 32767 -or- -32768 thru -1

HÌNH 1-6 dấu cho một byte

BẢNG 1-3 m vi số nguyên thủy

Type	Bits	Bytes	Minimum Range	Maximum Range
byte	8	1	-2^7	$2^7 - 1$
short	16	2	-2^{15}	$2^{15} - 1$
int	32	4	-2^{31}	$2^{31} - 1$
long	64	8	-2^{63}	$2^{63} - 1$
float	32	4	n/a	n/a
double	64	8	n/a	n/a

Bảng 1-3 cho thấy các loại nguyên thủy với kích thước và phạm vi của chúng. [Hình 1-6](#)).

cho thấy rằng với một byte chẳng hạn, có thể có 256 số (hoặc 2 trong số này là số âm và nửa - 1 là số dương. Phạm vi dương nhỏ hơn một so với phạm vi âm vì số 0 được lưu trữ dưới dạng số nhị phân dương . Chúng tôi sử dụng công thức $-2^{(\text{bit} - 1)}$ để tính phạm vi âm và sử dụng $2^{(\text{bit} - 1)} - 1$ cho phạm vi dương. Một lần nữa, nếu bạn biết hai cột đầu tiên của bảng này, bạn sẽ có phong độ tốt cho kỳ thi.

Phạm vi của số dấu phẩy động rất phức tạp để xác định, nhưng may mắn là bạn không cần biết những điều này trong kỳ thi (mặc dù bạn được mong đợi biết rằng một kép chứa 64 bit và một số float 32).

Không có phạm vi giá trị boolean ; một boolean chỉ có thể đúng hoặc sai.

Nếu ai đó hỏi bạn về độ sâu bit của boolean, hãy nhìn thẳng vào mắt họ và nói, "Điều đó phụ thuộc vào máy ảo." Họ sẽ rất ấn tượng.

Kiểu char (một ký tự) chứa một ký tự Unicode 16 bit.

Mặc dù bộ ASCII mở rộng được gọi là ISO Latin-1 chỉ cần 8 bit (256 ký tự khác nhau), nhưng cần có một dải lớn hơn để biểu diễn các ký tự được tìm thấy trong các ngôn ngữ khác ngoài tiếng Anh. Các ký tự Unicode thực sự được biểu diễn bằng các số nguyên 16 bit không dấu, có nghĩa là 2^{16} các giá trị có thể có, từ 0 đến 65535 ($2^{16} - 1$). Bạn sẽ học trong [Chương 3](#) rằng vì một char thực sự là một số nguyên loại, nó có thể được gán cho bất kỳ loại số nào đủ lớn để chứa 65535 (có nghĩa là bất kỳ thứ gì lớn hơn số ngắn; mặc dù cả ký tự và ký tự ngắn đều là loại 16 bit, hãy nhớ rằng số ngắn sử dụng 1 bit để biểu thị dấu, vì vậy ít số dương hơn có thể chấp nhận được trong thời gian ngắn).

Khai báo các biến tham chiếu

Các biến tham chiếu có thể được khai báo dưới dạng biến tĩnh, biến thẻ hiện, tham số phương thức hoặc biến cục bộ. Bạn có thể khai báo một hoặc nhiều biến tham chiếu, cùng kiểu, trong một dòng. Trong [Chương 3](#), chúng ta sẽ thảo luận về các cách khác nhau mà chúng có thể được khởi tạo, nhưng bây giờ chúng ta sẽ để lại cho bạn một vài ví dụ về khai báo biến tham chiếu:

```
Object o;
Dog myNewDogReferenceVariable;
String s1, s2, s3; // declare three String vars.
```

Biến thẻ hiện

Các biến cá thể được định nghĩa bên trong lớp, nhưng bên ngoài bất kỳ phương thức nào và chỉ được khởi tạo khi lớp được khởi tạo. Biến cá thể là các trường thuộc về mỗi đối tượng duy nhất. Ví dụ: đoạn mã sau xác định các trường (biến phiên bản) cho tên, chức danh và người quản lý cho các đối tượng nhân viên:

```
class Employee {
    // define fields (instance variables) for employee instances
    private String name;
    private String title;
    private String manager;
    // other code goes here including access methods for private
    // fields
}
```

Lớp Nhân viên trước đó nói rằng mỗi cá thể nhân viên sẽ biết tên riêng, chức danh và người quản lý. Nói cách khác, mỗi cá thể có thể có các giá trị riêng biệt cho ba trường đó. Đối với kỳ thi, bạn cần biết rằng các biến phiên bản

- Có thể sử dụng bất kỳ cấp độ truy cập nào trong số bốn cấp độ truy cập (có nghĩa là chúng có thể được đánh dấu bằng bất kỳ công cụ sửa đổi quyền truy cập nào trong số ba cấp độ truy cập)
- Có thể được đánh dấu cuối cùng
- Có thể được đánh dấu thoáng qua
- Không thể được đánh dấu là trừu tượng
- Không thể được đánh dấu là đã đồng bộ hóa
- Không thể được đánh dấu nghiêm ngặt
- Không thể được đánh dấu là gốc

- Không thể được đánh dấu là tĩnh vì sau đó chúng sẽ trở thành các biến lớp

Chúng tôi đã đề cập đến các tác động của việc áp dụng kiểm soát truy cập cho các biến cá thể (nó hoạt động theo cách tương tự như đối với các phương thức thành viên). Một chút sau trong chương này, chúng ta sẽ xem xét ý nghĩa của việc áp dụng công cụ sửa đổi cuối cùng hoặc tạm thời cho một biến cá thể. Tuy nhiên, trước tiên, chúng ta sẽ xem xét nhanh sự khác biệt giữa biến cá thể và biến cục bộ. [Hình 1-7](#) so sánh cách mà các sửa đổi có thể được áp dụng cho các phương thức so với các biến.

Local Variables	Variables (nonlocal)	Methods
final	final public protected private static transient volatile	final public protected private static abstract synchronized strictfp native

HÌNH 1-7

các công cụ sửa đổi trên các biến so với các phương thức

Biến cục bộ (Tự động / Ngăn xếp / Phương pháp)

Biến cục bộ là một biến được khai báo trong một phương thức. Điều đó có nghĩa là biến không chỉ được khởi tạo bên trong phương thức mà còn được khai báo bên trong phương thức.

Cũng giống như biến cục bộ bắt đầu hoạt động bên trong phương thức, nó cũng bị hủy khi phương thức hoàn thành. Các biến cục bộ luôn nằm trên ngăn xếp, không phải trong đống.

(Chúng ta sẽ nói thêm về ngăn xếp và đống trong [Chương 3](#).) Mặc dù giá trị của biến có thể được chuyển vào, chẳng hạn, một phương thức khác sau đó lưu giá trị trong một biến thể hiện, bản thân biến đó chỉ tồn tại trong phạm vi của phương pháp.

Chỉ cần đừng quên rằng trong khi biến cục bộ nằm trên ngăn xếp, nếu biến là một tham chiếu đối tượng, thì bản thân đối tượng sẽ vẫn được tạo trên heap. Không có cái gọi là đối tượng ngăn xếp, chỉ có một biến ngăn xếp. Bạn sẽ thường nghe thấy các lập trình viên sử dụng cụm từ “đối tượng cục bộ”, nhưng ý nghĩa thực sự của chúng là “biến tham chiếu được khai báo cục bộ”. Vì vậy, nếu bạn nghe một lập trình viên sử dụng biểu thức đó, bạn sẽ biết rằng anh ta quá lười biếng để diễn đạt nó một cách chính xác về mặt kỹ thuật. Bạn có thể nói với anh ấy rằng chúng tôi đã nói điều đó - trừ khi anh ấy biết chúng tôi sống ở đâu.

Khai báo biến cục bộ không thể sử dụng hầu hết các công cụ sửa đổi có thể được áp dụng cho các biến cá thể, chẳng hạn như công khai (hoặc các công cụ sửa đổi truy cập khác), tạm thời, biến động, trừu tượng hoặc tĩnh, nhưng như chúng ta đã thấy trước đó, các biến cục bộ có thể được đánh dấu cuối cùng. Và như bạn sẽ học trong [Chương 3](#) (nhưng đây là bản xem trước), trước khi một biến cục bộ có thể được sử dụng, nó phải được khởi tạo với một giá trị. Ví dụ:

```
class TestServer {
    public void logIn() {
        int count = 10;
    }
}
```

Thông thường, bạn sẽ khởi tạo một biến cục bộ trong cùng dòng mà bạn khai báo nó, mặc dù bạn vẫn có thể cần gán lại nó sau này trong phương thức. Điều quan trọng là phải nhớ rằng một biến cục bộ phải được khởi tạo trước khi bạn cố gắng sử dụng nó. Trình biên dịch sẽ từ chối bất kỳ mã nào cố gắng sử dụng một biến cục bộ chưa được gán giá trị vì – không giống như các biến phiên bản – các biến cục bộ không nhận được giá trị mặc định.

Một biến cục bộ không thể được tham chiếu trong bất kỳ mã nào bên ngoài phương thức mà nó được khai báo. Trong ví dụ mã trước, sẽ không thể tham chiếu đến

số lượng biến ở bất kỳ nơi nào khác trong lớp ngoại trừ trong phạm vi của phương thức `logIn()`. Một lần nữa, điều đó không có nghĩa là giá trị của số đếm không thể được chuyển ra khỏi phương thức để bắt đầu một cuộc sống mới. Nhưng biến chứa giá trị đó, `count`, không thể được truy cập sau khi phương thức hoàn tất, vì đoạn mã bất hợp pháp sau đây chứng minh:

```
class TestServer {
    public void logIn() {
        int count = 10;
    }
    public void doSomething(int i) {
        count = i; // Won't compile! Can't access count outside
                   // method logIn()
    }
}
```

Có thể khai báo một biến cục bộ có cùng tên với một thẻ hiện Biến đổi. Nó được gọi là bóng mờ, như đoạn mã sau thẻ hiện:

```
class TestServer {
    int count = 9;           // Declare an instance variable named count
    public void logIn() {
        int count = 10;      // Declare a local variable named count
        System.out.println("local variable count is " + count);
    }
    public void count() {
        System.out.println("instance variable count is " + count);
    }
    public static void main(String[] args) {
        new TestServer().logIn();
        new TestServer().count();
    }
}
```

The preceding code produces the following output:

```
local variable count is 10
instance variable count is 9
```

Tại sao trên Trái đất (hoặc hành tinh bạn chọn) bạn muốn làm điều đó? Thông thường, bạn sẽ không. Nhưng một trong những lý do phổ biến hơn là đặt tên một tham số trùng tên với biến cá thể mà tham số sẽ được gán.

Đoạn mã (sai) sau đây đang cố gắng đặt giá trị của biến phiên bản bằng cách sử dụng

Đoạn mã sau (sai) đang cố gắng đặt giá trị của một biến phiên bản bằng cách sử dụng một tham số:

```
class Foo {
    int size = 27;
    public void setSize(int size) {
        size = size; // ??? which size equals which size???
    }
}
```

Vì vậy, bạn đã quyết định rằng – để có thể đọc được tổng thể – bạn muốn đặt tên cho tham số giống với tên của biến cá thể mà giá trị của nó dành cho, nhưng làm cách nào để giải quyết xung đột đặt tên? Sử dụng từ khóa này. Từ khóa this always, always, luôn cập đến đối tượng hiện đang chạy. Đoạn mã sau cho thấy điều này đang hoạt động:

```
class Foo {
    int size = 27;
    public void setSize(int size) {
        this.size = size; // this.size means the current object's
                          // instance variable, size. The size
                          // on the right is the parameter
    }
}
```

Khai báo mảng Trong Java,

mảng là các đối tượng lưu trữ nhiều biến cùng kiểu hoặc các biến là tất cả các lớp con của cùng một kiểu. Mảng có thể chứa các tham chiếu nguyên thủy hoặc đối tượng, nhưng bản thân mảng sẽ luôn là một đối tượng trên heap, ngay cả khi mảng được khai báo là chứa các phần tử nguyên thủy. Nói cách khác, không có cái gọi là mảng nguyên thủy, nhưng bạn có thể tạo một mảng nguyên thủy.

Đối với kỳ thi, bạn cần biết ba điều:

- Cách tạo biến tham chiếu mảng (khai báo)
- Cách tạo một đối tượng mảng (cấu trúc)
- Cách điền vào mảng với các phần tử (khởi tạo)

Đối với mục tiêu này, bạn chỉ cần biết cách khai báo một mảng; ổn bao gồm việc xây dựng và khởi tạo mảng trong [Chương 5](#).



Mảng hiệu quả, nhưng nhiều khi bạn sẽ muốn sử dụng một trong các loại Bộ sưu tập từ `java.util` (bao gồm `HashMap`, `ArrayList` và `TreeSet`).

Các lớp bộ sưu tập cung cấp nhiều cách linh hoạt hơn để truy cập một đối tượng (để chèn, xóa, đọc, v.v.) và, không giống như mảng, có thể mở rộng hoặc hợp đồng động khi bạn thêm hoặc xóa các phần tử. Có nhiều loại Bộ sưu tập cho nhiều nhu cầu. Bạn có cần phân loại nhanh không? Một nhóm các đối tượng không có bản sao? Một cách để truy cập một cặp tên-giá trị? Java cung cấp nhiều kiểu Bộ sưu tập để giải quyết những tình huống này, nhưng kiểu Bộ sưu tập duy nhất trong bài kiểm tra là `ArrayList` và [Chương 5](#) sẽ thảo luận chi tiết hơn về `ArrayList`.

Mảng được khai báo bằng cách nêu rõ loại phần tử mà mảng sẽ giữ (một đối tượng hoặc một nguyên thủy), theo sau là dấu ngoặc vuông ở hai bên của mã định danh. Khai báo một mảng nguyên thủy:

```
int [] key;           // Square brackets before name (recommended)
int key [];          // Square brackets after name (legal but less
                     // readable)
```

Khai báo một mảng tham chiếu đối tượng:

```
Thread[] threads;   // Recommended
Thread threads []; // Legal but less readable
```



Khi khai báo một tham chiếu mảng, bạn phải luôn đặt dấu ngoặc mảng ngay sau kiểu được khai báo, thay vì đặt sau mã định danh (tên biến). Bằng cách đó, bất kỳ ai đọc mã đều có thể dễ dàng nhận ra rằng, ví dụ, khóa là một tham chiếu đến một đối tượng mảng `int`, không phải là một nguyên nguyên `int`.

Chúng ta cũng có thể khai báo mảng nhiều chiều, trên thực tế, là mảng mảng. Điều này có thể được thực hiện theo cách sau:

```
String [][] [] OccupantName;
String [] managerName [];
```

Ví dụ đầu tiên là mảng ba chiều (một mảng các mảng), và ví dụ thứ hai là mảng hai chiều. Lưu ý trong ví dụ thứ hai, chúng ta có

một dấu ngoặc vuông trước tên biến và một dấu sau. Điều này hoàn toàn hợp pháp đối với trình biên dịch, một lần nữa chứng minh rằng chỉ vì nó hợp pháp không có nghĩa là nó đúng.



Việc đưa kích thước của mảng vào khai báo của bạn là không hợp pháp. Có, chúng tôi biết bạn có thể làm điều đó bằng một số ngôn ngữ khác, đó là lý do tại sao bạn có thể thấy một hoặc hai câu hỏi bao gồm mã tương tự như sau:

```
int [5] điểm;
```

Mã trước đó sẽ không biên dịch. Hãy nhớ rằng, JVM không phân bổ không gian cho đến khi bạn thực sự khởi tạo đối tượng mảng. Đó là khi kích thước các vấn đề.

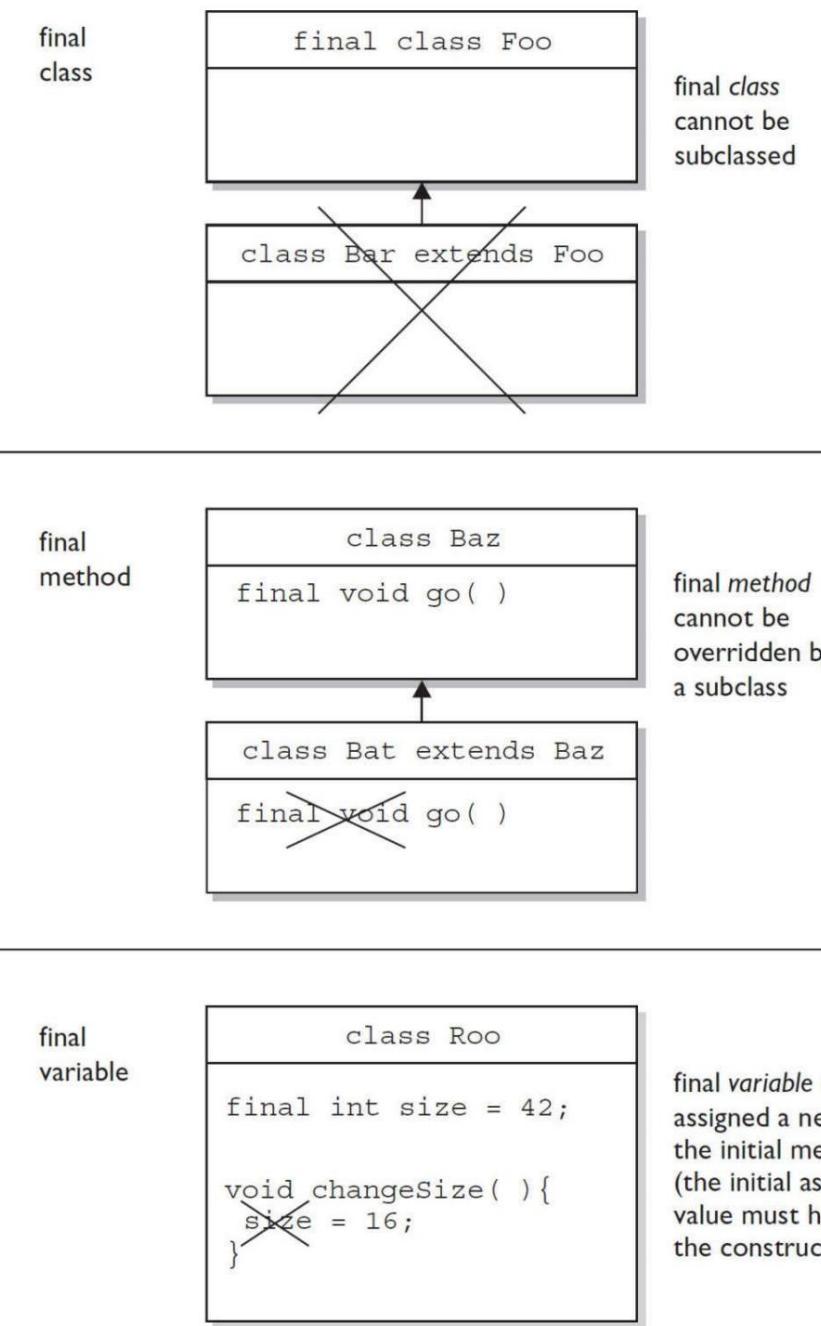
Trong [Chương 6](#), chúng ta sẽ dành nhiều thời gian thảo luận về mảng, cách khởi tạo và sử dụng chúng và cách xử lý mảng đa chiều. hãy chú ý theo dõi!

Các biến cuối cùng

Việc khai báo một biến với từ khóa cuối cùng sẽ không thể gán lại biến đó sau khi nó đã được khởi tạo với một giá trị rõ ràng (lưu ý rằng chúng tôi đã nói "rõ ràng" chứ không phải "mặc định"). Đối với nguyên thủy, điều này có nghĩa là một khi biến được gán một giá trị, thì giá trị đó không thể thay đổi được. Ví dụ: nếu bạn gán 10 cho biến int x, thì x sẽ ở lại 10, mãi mãi. Vì vậy, đó là đơn giản cho các nguyên thủy, nhưng điều đó có nghĩa là gì khi có một biến tham chiếu đối tượng cuối cùng ?

Một biến tham chiếu được đánh dấu là cuối cùng không bao giờ có thể được gán lại để tham chiếu đến một đối tượng khác. Dữ liệu bên trong đối tượng có thể được sửa đổi, nhưng không thể thay đổi biến tham chiếu. Nói cách khác, tham chiếu cuối cùng vẫn cho phép bạn sửa đổi trạng thái của đối tượng mà nó tham chiếu đến, nhưng bạn không thể sửa đổi biến tham chiếu để biến nó tham chiếu đến một đối tượng khác. Ghi cái này vào: không có đối tượng cuối cùng , chỉ có tài liệu tham khảo cuối cùng . Chúng tôi sẽ giải thích điều này chi tiết hơn trong [Chương 3](#).

Bây giờ chúng ta đã đề cập đến cách mà công cụ sửa đổi cuối cùng có thể được áp dụng cho các lớp, phương thức và biến. [Hình 1-8](#) nêu rõ những điểm chính và sự khác biệt của các ứng dụng khác nhau của cuối cùng.



HÌNH 1-8 của cuối cùng lên các biến, phương thức và lớp

Biến tạm thời

Nếu bạn đánh dấu một biến cá thể là tạm thời, bạn đang yêu cầu JVM bỏ qua (bỏ qua) biến này khi bạn cố gắng tuần tự hóa đối tượng chứa nó.

Serialization là một trong những tính năng thú vị nhất của Java; nó cho phép bạn lưu (đôi khi được gọi là "làm phẳng") một đối tượng bằng cách viết trạng thái của nó (nói cách khác, giá trị của

biến cá thể) cho một loại luồng I / O đặc biệt. Với tuần tự hóa, bạn có thể lưu một đối tượng vào một tệp hoặc thậm chí gửi nó qua một dây để tái tạo (deserializing) ở đâu kia trong một JVM khác. Chúng tôi rất vui khi tuần tự hóa được thêm vào kỳ thi kể từ Java 5, nhưng chúng tôi rất buồn khi nói rằng kể từ Java 7, tuần tự hóa không còn được đưa vào kỳ thi nữa.

Các biến số dễ bay hơi

Công cụ sửa đổi biến động cho JVM biết rằng một luồng truy cập vào biến phải luôn紧跟 chiểu bản sao riêng của biến đó với bản sao chính trong bộ nhớ. Nói gì cơ? Đừng lo lắng về nó. Đối với kỳ thi, tất cả những gì bạn cần biết về biến động là nó tồn tại.



Công cụ sửa đổi dễ bay hơi cũng có thể được áp dụng cho các nhà quản lý dự án!

Các biến và phương thức tĩnh

Lưu ý: Thảo luận về static trong phần này KHÔNG bao gồm phương pháp giao diện tĩnh mới đã được thảo luận trước đó trong chương này. Bạn không chỉ thích cách những người Java 8 sử dụng lại các thuật ngữ Java quan trọng sao?

Công cụ sửa đổi tĩnh được sử dụng để tạo các biến và phương thức sẽ tồn tại độc lập với bất kỳ trường hợp nào được tạo cho lớp. Tất cả các thành viên tĩnh tồn tại trước khi bạn tạo một thể hiện mới của một lớp và sẽ chỉ có một bản sao của một thành viên tĩnh bất kể số lượng các thể hiện của lớp đó. Nói cách khác, tất cả các trường hợp của một lớp nhất định đều có chung giá trị cho bất kỳ biến tĩnh nào đã cho. Chúng tôi sẽ trình bày rất chi tiết về các thành viên tĩnh trong chương tiếp theo.

Những thứ bạn có thể đánh dấu là tĩnh:

- Phương pháp
- Biến
- Một lớp được lồng trong một lớp khác, nhưng không nằm trong một phương thức (không phải trên Kỳ thi OCA 8)
- Khởi khởi tạo

Những thứ bạn không thể đánh dấu là tĩnh:

- Các hàm tạo (không có ý nghĩa gì; một hàm tạo chỉ được sử dụng để tạo các phiên bản)
- Các lớp (trừ khi chúng được lồng vào nhau)
- Các giao diện (trừ khi chúng được lồng vào nhau)
- Phương pháp các lớp học bên trong địa phương (không phải trong kỳ thi OCA 8)
- Các phương thức lớp bên trong và các biến phiên bản (không có trong kỳ thi OCA 8)
- Biến cục bộ

MỤC TIÊU XÁC NHẬN

Khai báo và sử dụng enums (Mục tiêu 1.2 của OCA)

1.2 Xác định cấu trúc của một lớp Java.

Lưu ý: Trong quá trình tạo cuốn sách này, Oracle đã điều chỉnh một số mục tiêu cho kỳ thi OCA. Chúng tôi không chắc chắn 100 phần trăm rằng chủ đề enums có được đưa vào kỳ thi OCA, nhưng chúng tôi đã quyết định rằng tốt hơn là an toàn hơn là xin lỗi, vì vậy chúng tôi khuyên các thí sinh OCA nên học phần này. Trong mọi trường hợp, bạn có thể gặp phải việc sử dụng enums trong mã Java mà bạn đọc, vì vậy việc tìm hiểu về chúng sẽ mang lại hiệu quả bất kể.

Khai báo enums

Java cho phép bạn hạn chế một biến chỉ có một trong một số giá trị được xác định trước- nói cách khác, một giá trị từ danh sách được liệt kê. (Đáng ngạc nhiên, các mục trong danh sách được liệt kê được gọi là enums.)

Sử dụng enum có thể giúp giảm thiểu các lỗi trong mã của bạn. Ví dụ: hãy tưởng tượng bạn đang tạo một ứng dụng thành lập cà phê thương mại và trong ứng dụng quán cà phê của mình, bạn có thể muốn giới hạn các lựa chọn CoffeeSize của mình ở LỚN, LỚN và HƠN THẾ NỮA . Nếu bạn để một đơn hàng LARGE hoặc GRANDE đến, nó có thể gây ra lỗi. enums để giải cứu. Với khai báo đơn giản sau, bạn có thể đảm bảo rằng trình biên dịch sẽ ngăn bạn gán bất kỳ thứ gì cho CoffeeSize ngoại trừ BIG, HUGE hoặc OVERWHELMING:

```
enum CoffeeSize {BIG, HUGE, OVERWHELMING};
```

Kể từ đó, cách duy nhất để có được CoffeeSize sẽ là với một tuyên bố như sau:

```
CoffeeSize cs = CoffeeSize.BIG;
```

Không bắt buộc các hằng số enum phải viết hoa hoàn toàn, nhưng vay mượn từ Quy ước mã Oracle rằng các hằng số được đặt tên bằng chữ hoa, đó là một ý kiến hay.

Các thành phần cơ bản của một enum là các hằng số của nó (nghĩa là LỚN, LỚN VÀ HƠN THẾ NỮA), mặc dù trong một phút, bạn sẽ thấy rằng có thể có nhiều thứ hơn nữa đối với một enum. enums có thể được khai báo như một lớp riêng biệt của riêng chúng hoặc như một thành viên của lớp; tuy nhiên, chúng không được khai báo trong một phương thức!

Đây là một ví dụ khai báo một enum bên ngoài một lớp:

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING } // this cannot be
                                                // private or protected
class Coffee {
    CoffeeSize size;
}
public class CoffeeTest1 {
    public static void main(String[] args) {
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;           // enum outside class
    }
}
```

Mã trước có thể là một phần của một tệp (hoặc nói chung, các lớp enum có thể tồn tại trong tệp riêng của chúng như CoffeeSize.java). Nhưng hãy nhớ rằng, trong trường hợp này tệp phải được đặt tên là CoffeeTest1.java vì đó là tên của lớp công khai trong tệp. Điểm mấu chốt cần nhớ là một enum không nằm trong một lớp có thể được khai báo chỉ với công cụ sửa đổi công khai hoặc mặc định, giống như một lớp không bên trong. Đây là một ví dụ về khai báo một enum bên trong một lớp:

Điểm mấu chốt cần rút ra từ những ví dụ này là enum có thể được khai báo là lớp riêng của chúng hoặc nằm trong một lớp khác và cú pháp để truy cập các thành viên của enum phụ thuộc vào nơi enum được khai báo.

Điều sau KHÔNG hợp pháp:

```
public class CoffeeTest1 {
    public static void main(String[] args) {
        enum CoffeeSize { BIG, HUGE, OVERWHELMING } // WRONG! Cannot
                                                       // declare enums
                                                       // in methods
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;
    }
}
```

Để làm cho bạn hiểu hơn, các nhà thiết kế ngôn ngữ Java đã làm tùy chọn đặt dấu chấm phẩy ở cuối khai báo enum (khi không có khai báo nào khác cho enum này sau):

```
public class CoffeeTest1 {
    enum CoffeeSize { BIG, HUGE, OVERWHELMING }; // <-semicolon
                                                       // is optional here
    public static void main(String[] args) {
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;
    }
}
```

Vì vậy, những gì được tạo ra khi bạn tạo một enum? Điều quan trọng nhất để hãy nhớ rằng enums không phải là chuỗi hoặc int! Mỗi giá trị CoffeeSize được liệt kê thực sự là một ví dụ của CoffeeSize. Nói cách khác, BIG thuộc loại CoffeeSize. Hãy nghĩ về enum như một loại lớp trông giống như sau:

```

// conceptual example of how you can think
// about enums
class CoffeeSize {
    public static final CoffeeSize BIG =
        new CoffeeSize("BIG", 0);
    public static final CoffeeSize HUGE =
        new CoffeeSize("HUGE", 1);
    public static final CoffeeSize OVERWHELMING =
        new CoffeeSize("OVERWHELMING", 2);

    CoffeeSize(String enumName, int index) {
        // stuff here
    }
    public static void main(String[] args) {
        System.out.println(CoffeeSize.BIG);
    }
}

```

Lưu ý rằng mỗi giá trị được liệt kê, BIG, HUGE và OVERWHELMING, là như thế nào một ví dụ của loại CoffeeSize. Chúng được biểu diễn dưới dạng tĩnh và cuối cùng, trong thế giới Java, được coi là một hằng số. Cũng lưu ý rằng mỗi giá trị enum biết chỉ mục hoặc vị trí của nó – nói cách khác, thứ tự mà các giá trị enum được khai báo rất quan trọng. Bạn có thể nghĩ về kiểu enum CoffeeSize tồn tại trong một mảng kiểu CoffeeSize và như bạn sẽ thấy trong chương sau, bạn có thể lặp lại các giá trị của một enum bằng cách gọi phương thức giá trị () trên bất kỳ kiểu enum nào. (Đừng lo lắng về điều đó trong chương này.)

Khai báo hàm tạo, phương thức và biến trong một enum Vì enum thực sự là một lớp đặc biệt, bạn có thể làm được nhiều việc hơn là chỉ liệt kê các giá trị hằng được liệt kê. Bạn có thể thêm các hàm tạo, biến cá thể, phương thức và một thứ thực sự kỳ lạ được gọi là một thành phần riêng.

Để hiểu tại sao bạn có thể cần nhiều hơn trong enum của mình, hãy nghĩ về tình huống này: Hãy tưởng tượng bạn muốn biết kích thước thực tế, tính bằng ounce, ánh xạ tới từng trong ba hằng số CoffeeSize . Ví dụ, bạn muốn biết rằng LỚN là 8 ounce, HUGE là 10 ounce, và OVERWHELMING là một con số khổng lồ 16 ounce.

Bạn có thể tạo một số loại bảng tra cứu bằng cách sử dụng một số cấu trúc dữ liệu khác, nhưng đó sẽ là một thiết kế kém và khó bảo trì. Cách đơn giản nhất là coi các giá trị enum của bạn (BIG, HUGE và OVERWHELMING) là các đối tượng, mỗi đối tượng có thể có các biến phiên bản của riêng nó. Sau đó, bạn có thể gán các giá trị đó tại thời điểm các enum được khởi tạo bằng cách chuyển một giá trị cho hàm tạo enum . Điều này cần một

giải thích ít, nhưng trước tiên hãy xem đoạn mã sau:

```

enum CoffeeSize {
    // 8, 10 & 16 are passed to the constructor
    BIG(8), HUGE(10), OVERWHELMING(16);
    CoffeeSize(int ounces) {      // constructor
        this.ounces = ounces;
    }

    private int ounces;           // an instance variable
    public int getOunces() {
        return ounces;
    }
}

class Coffee {
    CoffeeSize size;            // each instance of Coffee has an enum

    public static void main(String[] args) {
        Coffee drink1 = new Coffee();
        drink1.size = CoffeeSize.BIG;

        Coffee drink2 = new Coffee();
        drink2.size = CoffeeSize.OVERWHELMING;

        System.out.println(drink1.size.getOunces()); // prints 8
        for(CoffeeSize cs: CoffeeSize.values())
            System.out.println(cs + " " + cs.getOunces());
    }
}

```

sản xuất:

```

số 8
LỚN 8
KHÔNG LÒ 10
HƯỚNG DẪN 16

```

Lưu ý: Mọi enum đều có một phương thức tĩnh, giá trị (), trả về một mảng các giá trị của enum theo thứ tự chúng được khai báo.

Các điểm chính cần nhớ về các hàm tạo enum là

- Bạn KHÔNG BAO GIỜ có thể gọi trực tiếp một hàm tạo enum .

Phương thức khởi tạo enum được gọi tự động, với các đối số bạn xác định sau

giá trị không đổi. Ví dụ, BIG (8) gọi phương thức khởi tạo CoffeeSize nhận một int, chuyển int theo nghĩa đen 8 cho phương thức khởi tạo. (Tất nhiên, đằng sau hậu trường, bạn có thể tưởng tượng rằng BIG cũng được chuyển cho hàm tạo, nhưng chúng ta không cần phải biết – hoặc quan tâm – về các chi tiết.)

- Bạn có thể xác định nhiều hơn một đối số cho hàm tạo và bạn có thể nạp chồng các hàm tạo enum, giống như bạn có thể nạp chồng cho một hàm tạo lớp bình thường. Chúng ta sẽ thảo luận chi tiết hơn về các hàm tạo trong [Chương 2](#). Để khởi tạo CoffeeSize với cả số ounce và kiểu nắp, bạn sẽ chuyển hai đối số cho hàm tạo là BIG (8, "A"), có nghĩa là bạn có một hàm tạo trong CoffeeSize nhận cả int và String.

Và, cuối cùng, bạn có thể xác định một cái gì đó thực sự kỳ lạ trong một enum trông giống như một lớp bên trong tên danh. Nó được biết đến như một phần thân của lớp cụ thể và bạn sử dụng nó khi bạn cần một hằng số cụ thể để ghi đè một phương thức được định nghĩa trong enum.

Hãy tưởng tượng tình huống này: Bạn muốn enums có hai phương pháp – một phương thức cho ounce và một cho mã nắp (một Chuỗi). Vậy giờ hãy tưởng tượng rằng hầu hết các kích thước cà phê sử dụng cùng một mã nắp, "B", nhưng kích thước HƠN THẾ NỮA sử dụng loại "A". Bạn có thể xác định một phương thức getLidCode () trong enum CoffeeSize trả về "B", nhưng sau đó bạn cần một cách để ghi đè nó cho VƯỢT QUA. Bạn không muốn thực hiện một số mã if / then khó duy trì trong phương thức getLidCode (), vì vậy cách tiếp cận tốt nhất có thể là bằng cách nào đó để hằng số OVERWHELMING ghi đè phương thức getLidCode () .

Điều này có vẻ lạ, nhưng bạn cần hiểu các quy tắc khai báo cơ bản:

```
enum CoffeeSize {
    BIG(8),
    HUGE(10),
```

```

OVERWHELMING(16) {
    // start a code block that defines
    // the "body" for this constant

    public String getLidCode() { // override the method
        // defined in CoffeeSize
        return "A";
    }
}; // the semicolon is REQUIRED when more code follows

CoffeeSize(int ounces) {
    this.ounces = ounces;
}

private int ounces;

public int getOunces() {
    return ounces;
}
public String getLidCode() { // this method is overridden
    // by the OVERWHELMING constant

    return "B"; // the default value we want to
    // return for CoffeeSize constants
}
}
}

```

TÓM TẮT CHỨNG NHẬN

Sau khi tiếp thu tài liệu trong chương này, bạn sẽ làm quen với một số đặc tính của ngôn ngữ Java. Bạn cũng có thể cảm thấy bối rối về lý do tại sao bạn lại muốn tham gia kỳ thi này ngay từ đầu. Đó là điều bình thường vào thời điểm này. Nếu bạn nghe thấy chính mình hỏi, "Tôi đang nghĩ gì?" chỉ cần nằm xuống cho đến khi nó đi qua. Chúng tôi muốn nói với bạn rằng mọi thứ trở nên dễ dàng hơn. rằng đây là chương khó nhất và nó sẽ xuống dốc từ đây.

Hãy xem lại ngắn gọn những điều bạn cần biết cho bài kiểm tra:

Sẽ có nhiều câu hỏi liên quan đến từ khóa một cách gián tiếp, vì vậy hãy đảm bảo bạn có thể xác định đâu là từ khóa và đâu là từ khóa.

Bạn cần hiểu các quy tắc liên quan đến việc tạo mã định danh hợp pháp và các quy tắc liên quan đến khai báo mã nguồn, bao gồm cả việc sử dụng các câu lệnh gói và nhập .

Bạn đã học cú pháp cơ bản cho các chương trình dòng lệnh java và javac .

Bạn đã học về khi nào main () có siêu cường và khi nào thì không.

Chúng tôi đã trình bày những kiến thức cơ bản về nhập và nhập các câu lệnh tĩnh . Thật hấp dẫn khi nghĩ rằng có nhiều thứ cho họ hơn là tiết kiệm một chút công việc nhập, nhưng không có.

Bây giờ bạn đã hiểu rõ về kiểm soát truy cập vì nó liên quan đến các lớp, các phương thức và các biến. Bạn đã xem cách các công cụ sửa đổi quyền truy cập (công khai, được bảo vệ và riêng tư) xác định quyền kiểm soát truy cập của một lớp hoặc thành viên.

Bạn đã học được rằng các lớp trừu tượng có thể chứa cả trừu tượng và nonabstract nhưng nếu cả một phương thức đơn lẻ được đánh dấu là trừu tượng, thì lớp đó phải được đánh dấu là trừu tượng. Đừng quên rằng một lớp con cụ thể (nonabstract) của một lớp trừu tượng phải cung cấp các triển khai cho tất cả các phương thức trừu tượng của lớp cha, nhưng một lớp trừu tượng không phải triển khai các phương thức trừu tượng từ lớp cha của nó. Một lớp con trừu tượng có thể "chuyển buck" cho lớp con cụ thể đầu tiên.

Chúng tôi đã đề cập đến việc triển khai giao diện. Hãy nhớ rằng các giao diện có thể mở rộng một giao diện khác (thậm chí nhiều giao diện) và bất kỳ lớp nào triển khai một giao diện phải triển khai tất cả các phương thức từ tất cả các giao diện trong cây kế thừa của giao diện mà lớp đó đang triển khai.

Bạn cũng đã xem xét các công cụ sửa đổi khác, bao gồm tĩnh, cuối cùng, trừu tượng, đồng bộ hóa, v.v. Bạn đã biết cách một số bỏ ngữ không bao giờ có thể được kết hợp trong một khai báo, chẳng hạn như trộn trừu tượng với cuối cùng hoặc riêng tư.

Hãy nhớ rằng không có đối tượng cuối cùng trong Java. Một biến tham chiếu được đánh dấu là cuối cùng không bao giờ có thể thay đổi được, nhưng đối tượng mà nó tham chiếu đến có thể được sửa đổi. Bạn đã thấy rằng áp dụng cuối cùng cho các phương thức có nghĩa là một lớp con không thể ghi đè chúng và khi được áp dụng cho một lớp, lớp cuối cùng không thể được phân lớp.

Các phương thức có thể được khai báo với một tham số var-arg (có thể lấy từ 0 đến nhiều đối số của kiểu đã khai báo), nhưng bạn chỉ có thể có một var-arg cho mỗi phương thức và nó phải là tham số cuối cùng của phương thức.

Đảm bảo rằng bạn đã quen với các kích thước tương đối của các nguyên mẫu số.

Hãy nhớ rằng mặc dù giá trị của các biến không phải biến cuối cùng có thể thay đổi, nhưng kiểu của biến tham chiếu không bao giờ có thể thay đổi.

Bạn cũng đã học được rằng mảng là các đối tượng chứa nhiều biến của cùng loại. Mảng cũng có thể chứa các mảng khác.

Hãy nhớ những gì bạn đã học về các biến và phương thức tĩnh, đặc biệt là các thành viên tĩnh là perclass chứ không phải per-instance. Đừng quên rằng một phương thức tĩnh không thể truy cập trực tiếp vào một biến thể hiện từ lớp mà nó nằm trong vì nó không có tham chiếu rõ ràng đến bất kỳ

thể hiện của lớp.

Cuối cùng, chúng tôi đã bao phủ enums. Một enum là một cách an toàn và linh hoạt để triển khai các hằng số. Bởi vì chúng là một loại lớp đặc biệt, enum có thể được khai báo rất đơn giản hoặc chúng có thể khá phức tạp - bao gồm các thuộc tính như phương thức, biến, hàm tạo và một loại đặc biệt của lớp bên trong được gọi là một phần thân lớp cụ thể.

Trước khi vượt qua bài kiểm tra thực hành, hãy dành một chút thời gian cho bài kiểm tra có tên lục quan sau đây là "Cuộc tập trận trong hai phút". Hãy thường xuyên quay lại bài tập đặc biệt này khi bạn xem qua cuốn sách này và đặc biệt là khi bạn đang thực hiện bài tập nhồi nhét vào phút cuối. Bởi vì – và đây là lời khuyên mà bạn ước gì mẹ đã cho bạn trước khi bạn lên đường vào đại học – đó không phải là những gì bạn biết, mà là khi bạn biết.

Đối với kỳ thi, biết những gì bạn không thể làm với ngôn ngữ Java cũng quan trọng như biết những gì bạn có thể làm. Hãy thử các câu hỏi mẫu! Chúng rất giống với độ khó và cấu trúc của đề thi thật và sẽ giúp bạn mở rộng tầm mắt về mức độ khó của đề thi. Đừng lo lắng nếu bạn hiểu sai nhiều thứ. Nếu bạn thấy chủ đề mà bạn yêu, hãy dành nhiều thời gian hơn để xem xét và nghiên cứu. Nhiều lập trình viên cần hai hoặc ba lần xem qua một chương (hoặc một mục tiêu cá nhân) trước khi họ có thể trả lời các câu hỏi một cách tự tin.

✓ KHOAN HAI PHÚT

Hãy nhớ rằng trong chương này, khi chúng ta nói về các lớp, chúng ta đang đề cập đến các lớp không bên trong, nói cách khác, các lớp cấp cao nhất .

Các tính năng và lợi ích của Java (Mục tiêu 1.5 của OCA)

- Mặc dù Java cung cấp nhiều lợi ích cho các lập trình viên, nhưng bạn nên nhớ rằng Java hỗ trợ lập trình hướng đối tượng nói chung, đóng gói, quản lý bộ nhớ tự động, một API lớn (thư viện), các tính năng bảo mật tích hợp, khả năng tương thích đa nền tảng, gõ mạnh, đa luồng và máy tính phân tán.

Định danh (Mục tiêu 2.1 của OCA)

- Số nhận dạng có thể bắt đầu bằng một chữ cái, một dấu gạch dưới hoặc một ký tự tiền tệ.

- Sau ký tự đầu tiên, số nhận dạng cũng có thể bao gồm các chữ số.
- Số nhận dạng có thể có độ dài bất kỳ.

Tệp Java thực thi và main () (Mục tiêu 1.3 OCA)

- Bạn có thể biên dịch và thực thi các chương trình Java bằng cách sử dụng các chương trình dòng lệnh javac và java tương ứng. Cả hai chương trình đều hỗ trợ nhiều tùy chọn dòng lệnh.
- Các phiên bản duy nhất của phương thức main () có quyền hạn đặc biệt là những phiên bản có ký hiệu phương thức tương đương với public static void main (String [] args).
- main () có thể bị quá tải.

Nhập khẩu (Mục tiêu 1.4 của OCA)

- Công việc duy nhất của câu lệnh nhập là lưu các lần gõ phím.
- Bạn có thể sử dụng dấu hoa thị (*) để tìm kiếm nội dung của một gói.
- Mặc dù được gọi là "nhập tinh", cú pháp là nhập tinh..
- Bạn có thể nhập các lớp API và / hoặc các lớp tùy chỉnh.

Quy tắc khai báo tệp nguồn (Mục tiêu 1.2 của OCA)

- Một tệp mã nguồn chỉ có thể có một lớp công khai .
- Nếu tệp nguồn chứa một lớp công khai , tên tệp phải khớp với tên lớp công khai .
- Một tệp chỉ có thể có một câu lệnh gói , nhưng nó có thể có nhiều lần nhập.
- Câu lệnh gói (nếu có) phải là dòng đầu tiên (noncomment) trong tệp nguồn.
- Các câu lệnh nhập (nếu có) phải đứng sau câu lệnh gói (nếu có) và trước phần khai báo lớp đầu tiên.
- Nếu không có câu lệnh gói , câu lệnh nhập phải là câu lệnh đầu tiên (noncomment) trong tệp nguồn. gói và câu lệnh nhập áp dụng cho tất cả các lớp trong tệp.
-

- Một tệp có thể có nhiều hơn một lớp không công khai.
- Các tệp không có lớp công khai không có giới hạn đặt tên.

Công cụ sửa đổi quyền truy cập lớp (Mục tiêu OCA 6.4)

- Có ba công cụ sửa đổi quyền truy cập: công khai, được bảo vệ và riêng tư.
- Có bốn cấp độ truy cập: công khai, được bảo vệ, mặc định và riêng tư.
- Các lớp chỉ có thể có quyền truy cập công khai hoặc mặc định.
- Một lớp có quyền truy cập mặc định chỉ có thể được nhìn thấy bởi các lớp trong cùng một gói.
- Một lớp có quyền truy cập công khai có thể được nhìn thấy bởi tất cả các lớp từ tất cả các gói.
- Khả năng hiển thị của lớp xoay quanh việc liệu mã trong một lớp có thể
 - Tạo một phiên bản của một lớp khác
 - Mở rộng (hoặc lớp con) một lớp khác
 - Truy cập các phương thức và biến của lớp khác

Công cụ sửa đổi lớp (Nonaccess) (Mục tiêu OCA 1.2, 7.1 và 7.5)

- Các lớp cũng có thể được sửa đổi với cuối cùng, trừu tượng hoặc nghiêm ngặt.
- Một lớp không thể vừa là cuối cùng vừa là trừu tượng.
- Một lớp cuối cùng không thể được phân lớp.
- Một lớp trừu tượng không thể được khởi tạo.
- Một phương thức trừu tượng duy nhất trong một lớp có nghĩa là cả lớp phải trừu tượng.
- Một lớp trừu tượng có thể có cả phương thức trừu tượng và phương thức nonabstract.
- Lớp cụ thể đầu tiên để mở rộng một lớp trừu tượng phải triển khai tất cả các phương thức trừu tượng của nó .

Triển khai giao diện (Mục tiêu 7.5 của OCA)

- Thông thường, các giao diện là các hợp đồng cho những gì một lớp có thể làm, nhưng chúng không nói gì về cách mà lớp phải làm điều đó.
- Các giao diện có thể được thực hiện bởi bất kỳ lớp nào từ bất kỳ cây kế thừa nào.
- Thông thường, một giao diện giống như một lớp trừu tượng 100% và hoàn toàn là

trừu tượng cho dù bạn có nhập công cụ sửa đổi trừu tượng vào khai báo hay không.

- Thông thường các giao diện chỉ có các phương thức trừu tượng .
- Theo mặc định, các phương thức giao diện là công khai và thường là trừu tượng – việc khai báo rõ ràng các bối ngữ này là tùy chọn.
- Các giao diện có thể có các hằng số, các hằng số này luôn ngầm định là công khai, tĩnh và cuối cùng.
- Các khai báo hằng số giao diện về công khai, tĩnh và cuối cùng là tùy chọn trong bất kỳ sự kết hợp nào.
- Kể từ Java 8, các giao diện có thể có các phương thức cụ thể được khai báo là mặc định hoặc tĩnh.

Lưu ý: Phần này sử dụng một số khái niệm mà chúng tôi CHƯA đề cập. Đừng hoảng sợ: khi bạn đã đọc hết cuốn sách, phần này sẽ có ý nghĩa như một tài liệu tham khảo.

- Một lớp triển khai nonabstract hợp pháp có các thuộc tính sau: Nó cung cấp các triển khai
 - cụ thể cho các phương thức của giao diện.
 - Nó phải tuân theo tất cả các quy tắc ghi đè hợp pháp cho các phương pháp mà nó triển khai.
 - Nó không được khai báo bất kỳ ngoại lệ mới đã được kiểm tra nào cho một phương thức triển khai.
 - Nó không được khai báo bất kỳ ngoại lệ đã kiểm tra nào rộng hơn các ngoại lệ được khai báo trong phương thức giao diện.
 - Nó có thể khai báo ngoại lệ thời gian chạy trên bất kỳ triển khai phương thức giao diện nào bất kể khai báo giao diện.
 - Nó phải duy trì chữ ký chính xác (cho phép trả về hiệp phương sai) và kiểu trả về của các phương thức mà nó thực hiện (nhưng không phải khai báo các ngoại lệ của giao diện).
- Một lớp thực hiện một giao diện tự nó có thể là trừu tượng.
- Một lớp thực thi trừu tượng không phải triển khai các phương thức giao diện (nhưng lớp con cụ thể đầu tiên thì phải).
- Một lớp chỉ có thể mở rộng một lớp (không có đa kế thừa), nhưng nó có thể triển khai nhiều giao diện.
- Các giao diện có thể mở rộng một hoặc nhiều giao diện khác.
- Giao diện không thể mở rộng một lớp hoặc triển khai một lớp hoặc giao diện.

- Khi thực hiện bài kiểm tra, hãy xác minh rằng giao diện và khai báo lớp là hợp pháp trước khi xác minh logic mã khác.

Công cụ sửa đổi quyền truy cập thành viên (Mục tiêu OCA 6.4)

- Các phương thức và biến cá thể (phi địa phương) được gọi là "thành viên".
- Các thành viên có thể sử dụng tất cả bốn cấp độ truy cập: công khai, được bảo vệ, mặc định và riêng tư.
- Quyền truy cập thành viên có hai hình thức:
 - Mã trong một lớp có thể truy cập một thành viên của lớp khác.
 - Một lớp con có thể kế thừa một thành viên của lớp cha của nó.
- Nếu một lớp không thể được truy cập, các thành viên của nó không thể được truy cập.
- Xác định khả năng hiển thị của lớp trước khi xác định khả năng hiển thị của thành viên. các thành viên công cộng có thể được truy cập bởi tất cả các lớp khác, ngay cả trong các gói khác.
- Nếu một thành viên của lớp cha là công khai, lớp con sẽ kế thừa nó – bất kể gói nào.
- Các thành viên được truy cập mà không có toán tử dấu chấm (.) Phải thuộc cùng một lớp đây. luôn tham chiếu đến đối tượng hiện đang thực thi. `this.aMethod`
- () giống như chỉ gọi `aMethod ()`. thành viên riêng chỉ có thể được truy cập bằng mã trong cùng một lớp. các thành viên riêng tư không hiển thị đối với các lớp con, vì vậy các thành viên riêng tư không thể được kế thừa.
- Các thành viên mặc định và được bảo vệ chỉ khác nhau khi các lớp con có liên quan:
 - Các thành viên mặc định chỉ có thể được truy cập bởi các lớp trong cùng một gói. Các thành viên được bảo vệ có thể được truy cập bởi các lớp khác trong cùng một gói, cộng với các lớp con, bất kể gói nào. bảo vệ = gói + trẻ em (trẻ em có nghĩa là lớp con).
 -
 - Đối với các lớp con bên ngoài gói, thành viên được bảo vệ chỉ có thể được truy cập thông qua kế thừa; một lớp con bên ngoài gói không thể truy cập thành viên được bảo vệ bằng cách sử dụng tham chiếu đến

cá thể siêu lớp. (Nói cách khác, kế thừa là cơ chế duy nhất để một lớp con bao gồm ngoài gói truy cập vào thành viên được bảo vệ của lớp cha của nó.)

- Một thành viên được bảo vệ được kế thừa bởi một lớp con từ một gói khác không thể truy cập vào bất kỳ lớp nào khác trong gói lớp con, ngoại trừ các lớp con của chính lớp con đó.

Biến cục bộ (Mục tiêu của OCA 2.1 và 6.4)

- Các khai báo biến cục bộ (phương thức, tự động hoặc ngăn xếp) không thể có bối cảnh truy cập.
- Cuối cùng là công cụ sửa đổi duy nhất có sẵn cho các biến cục bộ.
- Các biến cục bộ không nhận giá trị mặc định, vì vậy chúng phải được khởi tạo trước sử dụng.

Các bổ sung khác – Thành viên (Mục tiêu 7.1 và 7.5 của OCA)

- các phương thức cuối cùng không thể được ghi đè trong một lớp con.
- các phương thức trừu tượng được khai báo với một chữ ký, một kiểu trả về và một mệnh đề ném tùy chọn, nhưng chúng không được triển khai. các phương thức trừu tượng kết thúc bằng dấu chấm phẩy – không có dấu ngoặc nhọn.
- Ba cách để phát hiện một phương pháp nonabstract:
 - Phương thức này không được đánh dấu là trừu tượng.
 - Phương thức có dấu ngoặc nhọn.
 - Phương thức MIGHT có mã giữa các dấu ngoặc nhọn.
- Lớp nonabstract (cụ thể) đầu tiên để mở rộng một lớp trừu tượng phải triển khai tất cả các phương thức trừu tượng của lớp trừu tượng.
- Công cụ sửa đổi được đồng bộ hóa chỉ áp dụng cho các phương thức và khôi mã. các phương pháp được đồng bộ hóa có thể có bất kỳ kiểm soát truy cập nào và cũng có thể được đánh dấu là cuối cùng.
- các phương thức trừu tượng phải được thực hiện bởi một lớp con, vì vậy chúng phải có thể kế thừa được. Vì lý do đó
 - các phương thức trừu tượng không thể là private.
 - các phương thức trừu tượng không thể là cuối cùng.

- Công cụ sửa đổi gốc chỉ áp dụng cho các phương thức.
- Công cụ sửa đổi nghiêm ngặt chỉ áp dụng cho các lớp và phương thức.

Phương pháp với kỳ đà (Mục tiêu 1.2 của OCA)

- Các phương thức có thể khai báo một tham số chấp nhận từ 0 đến nhiều đối số, một phương thức được gọi là var-arg.
- Một tham số var-arg được khai báo với kiểu cú pháp ... name; ví dụ: doStuff (int ... x) {}.
- Một phương thức var-arg chỉ có thể có một tham số var-arg.
- Trong các phương thức có tham số bình thường và var-arg, var-arg phải đứng sau cùng.

Người xây dựng (Mục tiêu OCA 1.2 và 6.3)

- Các trình xây dựng phải có cùng tên với lớp
- Các hàm tạo có thể có các đối số, nhưng chúng không thể có kiểu trả về.
- Trình tạo có thể sử dụng bất kỳ công cụ sửa đổi quyền truy cập nào (thậm chí là riêng tư!).

Khai báo biến (Mục tiêu OCA 2.1)

- Các biến phiên bản có thể
 - Có bất kỳ kiểm soát truy cập nào
 - Được đánh dấu cuối cùng hoặc tạm thời
- Các biến phiên bản không được trừu tượng, đồng bộ hóa, nguyên bản hoặc nghiêm ngặt.
- Việc khai báo một biến cục bộ trùng tên với một biến thể hiện là hợp pháp; điều này được gọi là "bóng tối". biến cuối cùng có các thuộc tính sau: không thể gán lại biến cuối cùng khi đã gán giá trị. biến tham chiếu cuối cùng không thể tham chiếu đến một đối tượng khác khi đối
 - tượng đã được gán cho biến cuối cùng . các biến cuối cùng phải được khởi tạo trước khi
 - phương thức khởi tạo hoàn thành.
-
- Không có thứ gọi là vật thể cuối cùng . Một tham chiếu đối tượng được đánh dấu

cuối cùng KHÔNG có nghĩa là bản thân đối tượng không thể thay đổi.

- Công cụ sửa đổi tạm thời chỉ áp dụng cho các biến cá thể.
- Công cụ sửa đổi biến động chỉ áp dụng cho các biến cá thể.

Khai báo mảng (Mục tiêu của OCA 4.1 và 4.2)

- Mảng có thể chứa các đối tượng hoặc nguyên thủy, nhưng bản thân mảng luôn là một đối tượng.
- Khi bạn khai báo một mảng, dấu ngoặc có thể ở bên trái hoặc bên phải của tên biến.
- Không bao giờ hợp pháp khi bao gồm kích thước của một mảng trong khai báo.
- Một mảng các đối tượng có thể chứa bất kỳ đối tượng nào vượt qua kiểm tra IS-A (hoặc instanceof) cho kiểu đã khai báo của mảng. Ví dụ: nếu Horse mở rộng Động vật, thì một đối tượng Ngựa có thể đi vào mảng Động vật .

Các biến và phương thức tĩnh (Mục tiêu 6.2 của OCA)

- Chúng không bị ràng buộc với bất kỳ phiên bản cụ thể nào của một lớp.
- Không cần thể hiện lớp để sử dụng các thành viên tĩnh của lớp hoặc giao diện.
- Chỉ có một bản sao của một biến / lớp tĩnh và tất cả các trường hợp đều chia sẻ nó.
- các phương thức tĩnh không có quyền truy cập trực tiếp vào các thành viên không có tĩnh.

enums (Mục tiêu 1.2 của OCA)

- Một enum chỉ định một danh sách các giá trị hằng được gán cho một kiểu.
- Một enum KHÔNG phải là một chuỗi hoặc một int; kiểu của hằng số enum là kiểu enum . Ví dụ, SUMMER và FALL thuộc kiểu enum Season.
- Một enum có thể được khai báo bên ngoài hoặc bên trong một lớp, nhưng KHÔNG phải trong một phương thức.
- Một enum được khai báo bên ngoài một lớp KHÔNG được đánh dấu tĩnh, cuối cùng, trừu tượng, bảo vệ hoặc riêng tư.
- enums có thể chứa các hàm tạo, phương thức, biến và các thành viên của hằng số.
- hằng số enum có thể gửi các đối số đến phương thức khởi tạo enum , bằng cách sử dụng

cú pháp BIG (8), trong đó int nghĩa đen 8 được chuyển đến phương thức khởi tạo

- enum . các hàm tạo enum có thể có các đối số và có thể được nạp chồng. Các hàm tạo
- enum KHÔNG BAO GIỜ có thể được gọi trực tiếp trong mã. Chúng luôn được gọi tự động khi một enum được khởi tạo.
- Dấu chấm phẩy ở cuối khai báo enum là tùy chọn. Đây là hợp pháp:
 - enum Foo {ONE, TWO, THREE} enum Foo {ONE, TWO, THREE};
 - MyEnum.values () trả về một mảng các giá trị của MyEnum .

TỰ KIỂM TRA

Các câu hỏi sau đây sẽ giúp bạn đo lường sự hiểu biết của bạn về tài liệu được trình bày trong chương này. Đọc kỹ tất cả các lựa chọn, vì có thể có nhiều hơn một câu trả lời đúng. Chọn tất cả các câu trả lời đúng cho mỗi câu hỏi. Giữ tập trung.

Nếu bạn gặp khó khăn với những điều này lúc đầu, đừng đánh bại bản thân. Hãy tích cực. Hãy lặp lại những câu khẳng định hay ho với bản thân như "Tôi đủ thông minh để hiểu về enums" và "OK, để anh chàng khác biết enums tốt hơn tôi, nhưng tôi cá rằng anh ta không thể <chèn thứ gì đó mà bạn giỏi> như tôi."

1. Điều nào là đúng? (Chọn tất cả các áp dụng.)

- A. "X mở rộng Y" đúng nếu và chỉ khi X là một lớp và Y là một giao diện B. "X mở rộng Y" đúng nếu và chỉ khi X là một giao diện và Y là một lớp
- C. "X mở rộng Y" là đúng nếu X và Y là cả hai lớp hoặc cả hai giao diện
- D. "X mở rộng Y" là đúng cho tất cả các kết hợp X và Y là các lớp và / hoặc giao diện

2. Đưa ra:

```

class Rocket {
    private void blastOff() { System.out.print("bang "); }
}
public class Shuttle extends Rocket {
    public static void main(String[] args) {
        new Shuttle().go();
    }
    void go() {
        blastOff();
        // Rocket.blastOff(); // line A
    }
    private void blastOff() { System.out.print("sh-bang "); }
}

```

Đó là sự thật? (Chọn tất cả các áp dụng.)

- A. Khi mã là viết tắt, đầu ra là bang B. Khi mã là viết tắt, đầu ra là sh-bang C. Khi mã là viết tắt, quá trình biên dịch không thành công D.
- Nếu dòng A được bỏ ghi chú, đầu ra là bang bang E. Nếu dòng A không có ghi chú, đầu ra là sh-bang bang F. Nếu dòng A không được ghi chú, quá trình biên dịch không thành công.

3. Cho rằng cú pháp của vòng lặp for là đúng và đã cho:

```

import static java.lang.System.*;
class _ {
    static public void main(String[] __A_V_) {
        String $ = "";
        for(int x=0; ++x < __A_V_.length; ) // for loop
            $ += __A_V_[x];
        out.println($);
    }
}

```

Và dòng lệnh:

```
java _ - MỘT .
```

Kết quả là gì?

- A. -A
- B. A.

C. -A.

D. _A.

E. __A.

F. Biên dịch không thành

công G. Một ngoại lệ được đưa ra trong thời gian chạy

4. Cho:

```
1. enum Animals {  
2.     DOG("woof"), CAT("meow"), FISH("bubble");  
3.     String sound;  
4.     Animals(String s) { sound = s; }  
5. }  
6. class TestEnum {  
7.     static Animals a;  
8.     public static void main(String[] args) {  
9.         System.out.println(a.DOG.sound + " " + a.FISH.sound);  
10.    }  
11. }
```

Kết quả là gì?

A. gâu gâu

B. Nhiều lỗi biên dịch C. Biên

dịch không thành công do lỗi ở dòng 2 D. Biên

dịch không thành công do lỗi ở dòng 3 E. Biên

dịch không thành công do lỗi ở dòng 4 F. Biên

dịch không thành công do lỗi ở dòng 9

5. Cho hai tệp:

```
1. package pkgA;
2. public class Foo {
3.     int a = 5;
4.     protected int b = 6;
5.     public int c = 7;
6. }

3. package pkgB;
4. import pkgA.*;
5. public class Baz {
6.     public static void main(String[] args) {
7.         Foo f = new Foo();
8.         System.out.print(" " + f.a);
9.         System.out.print(" " + f.b);
10.        System.out.println(" " + f.c);
11.    }
12. }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

A. 5 6 7

B. 5 theo sau là một ngoại lệ C.

Biên dịch không thành công với một lỗi trên dòng 7

D. Biên dịch không thành công với một lỗi trên dòng

E. Biên dịch không thành công với một lỗi trên

dòng 9 F. Biên dịch không thành công với một lỗi trên dòng 10

6. Cho:

```
1. public class Electronic implements Device
   { public void doIt() { } }
2.
3. abstract class Phone1 extends Electronic { }
4.
5. abstract class Phone2 extends Electronic
   { public void doIt(int x) { } }
6.
7. class Phone3 extends Electronic implements Device
   { public void doStuff() { } }
8.
9. interface Device { public void doIt(); }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

A. Biên dịch thành công B.

Biên dịch không thành công với lỗi ở dòng 1 C.

Biên dịch không thành công với lỗi ở dòng 3 D.

Biên dịch không thành công với lỗi ở dòng 5 E.

Biên dịch không thành công với lỗi ở dòng 7 F.

Biên dịch không thành công với một lỗi trên dòng 9

7. Cho:

```
4. class Announce {  
5.     public static void main(String[] args) {  
6.         for(int __x = 0; __x < 3; __x++) ;  
7.         int #lb = 7;  
8.         long [] x [5];  
9.         Boolean [] ba[];  
10.    }  
11. }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

A. Biên dịch thành công B.

Biên dịch không thành công với lỗi ở dòng 6 C.

Biên dịch không thành công với lỗi ở dòng 7 D.

Biên dịch không thành công với lỗi ở dòng 8 E.

Biên dịch không thành công với lỗi ở dòng 9

8. Cho:

```
3. public class TestDays {  
4.     public enum Days { MON, TUE, WED };  
5.     public static void main(String[] args) {  
6.         for(Days d : Days.values() )  
7.             ;  
8.         Days [] d2 = Days.values();  
9.         System.out.println(d2[2]);  
10.    }  
11. }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

A. TUE

B. WED

C. Đầu ra không thể đoán trước

được D. Biên dịch không thành công do lỗi ở

dòng 4 E. Biên dịch không thành công do lỗi ở

dòng 6 F. Biên dịch không thành công do lỗi ở

dòng 8 G. Biên dịch không thành công do lỗi ở dòng 9

9. Cho:

```
4. public class Frodo extends Hobbit {  
5.     public static void main(String[] args) {  
6.         int myGold = 7;  
7.         System.out.println(countGold(myGold, 6));  
8.     }  
9. }  
10. class Hobbit {  
11.     int countGold(int x, int y) { return x + y; }  
12. }
```

Kết quả là gì?

A. 13

B. Biên dịch không thành công do nhiều lỗi

C. Biên dịch không thành công do lỗi ở dòng

6 D. Biên dịch không thành công do lỗi ở dòng

7 E. Biên dịch không thành công do lỗi ở dòng 11

10. Cho:

```
interface Gadget {  
    void doStuff();  
}  
abstract class Electronic {  
    void getPower() { System.out.print("plug in "); }  
}  
public class Tablet extends Electronic implements Gadget {  
    void doStuff() { System.out.print("show book "); }  
    public static void main(String[] args) {  
        new Tablet().getPower();  
        new Tablet().doStuff();  
    }  
}
```

Đó là sự thật? (Chọn tất cả các áp dụng.)

A. Lớp Tablet KHÔNG biên dịch B. Giao diện

Tiện ích sẽ KHÔNG biên dịch C. Đầu ra sẽ được

cắm vào sách hiển thị D. Lớp trừu tượng

Electronic sẽ KHÔNG biên dịch E. Lớp Tablet KHÔNG THỂ vừa

mở rộng và triển khai

11. Cho rằng lớp Integer nằm trong gói java.lang và đã cho:

```
1. // insert code here  
2. class StatTest {  
3.     public static void main(String[] args) {  
4.         System.out.println(Integer.MAX_VALUE);  
5.     }  
6. }
```

Cái nào, được chèn độc lập ở dòng 1, biên dịch? (Chọn tất cả các áp dụng.)

A. nhập tĩnh java.lang;

B. nhập tĩnh java.lang.Integer;

C. nhập tĩnh java.lang.Integer. *;

D. nhập tĩnh java.lang.Integer. *;

E. nhập tĩnh java.lang.Integer.MAX_VALUE;

F. Không có câu lệnh nào ở trên là cú pháp nhập hợp lệ

12. Cho:

```
interface MyInterface {  
    // insert code here  
}
```

Những dòng mã nào – được chèn độc lập tại chèn mã ở đây – sẽ biên dịch? (Chọn tất cả các áp dụng.)

- A. public static m1 () {}
- B. khoảng trống mặc định m2 () {}
- C. trừu tượng int m3 ();
- D. cuối cùng ngắn m4 () {return 5;}
- E. mặc định dài m5 ();
- F. static void m6 () {}

13. Câu nào đúng? (Chọn tất cả các áp dụng.)

- A. Java là ngôn ngữ lập trình được đánh kiểu động B. Java cung cấp khả năng kiểm soát chi tiết bộ nhớ thông qua việc sử dụng con trỏ
- C. Java cung cấp cho các lập trình viên khả năng tạo các đối tượng tốt gói lại
- D. Java cung cấp cho các lập trình viên khả năng gửi các đối tượng Java từ một máy này sang máy khác
- E. Java là một triển khai của tiêu chuẩn ECMA F. Khả năng đóng gói của Java cung cấp khả năng bảo mật chính của nó cơ chế

CÂU TRẢ LỜI TỰ KIỂM TRA

1. đúng.
 - A không chính xác vì các lớp triển khai giao diện, chúng không mở rộng họ.
 - B không chính xác vì giao diện chỉ "ké thừa từ" các giao diện khác.
 - D là không chính xác dựa trên các quy tắc trước đó. (Mục tiêu 7.5 của OCA)
2. và F đúng. Vì Rocket.blastOff () là private nên nó không thể bị ghi đè và nó sẽ ẩn đối với Class Shuttle.
 - A, C, D và E không chính xác dựa trên những điều trên. (Mục tiêu OCA 6.4)

3. B đúng. Câu hỏi này đang sử dụng hợp lệ (nhưng không phù hợp và kỳ lạ) số nhận dạng, nhập tĩnh, main () và logic tăng trước. (Lưu ý: Bạn có thẻ nhận được cảnh báo trình biên dịch khi biên dịch mã này.)
 A, C, D, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 1.2 của OCA)
4. A đúng; enums có thẻ có hàm tạo và biến.
 B, C, D, E và F không chính xác; những dòng này đều sử dụng đúng cú pháp. (Mục tiêu 1.2 của OCA)
5. D và E đúng. Biến a có quyền truy cập mặc định, vì vậy nó không thẻ được truy cập từ bên ngoài gói. Biến b có quyền truy cập được bảo vệ trong pkgA.
 A, B, C và F không chính xác dựa trên thông tin trên. (OCA Mục tiêu 1.4 và 6.5)
6. A đúng; tất cả những điều này đều là khai báo hợp pháp.
 B, C, D, E và F không chính xác dựa trên thông tin trên. (Mục tiêu 7.5 của OCA)
7. C và D đúng. Tên biến không được bắt đầu bằng # và khai báo mảng không được bao gồm kích thước mà không có phần thuyết minh. Phần còn lại của mã là hợp lệ.
 A, B và E không chính xác dựa trên những điều trên. (Mục tiêu 2.1 của OCA)
8. B đúng. Mọi enum đều đi kèm với một phương thức static values () trả về một mảng các giá trị của enum theo thứ tự mà chúng được khai báo trong enum.
 A, C, D, E, F và G không chính xác dựa trên thông tin trên.
(Mục tiêu OCP 1.2)
9. D đúng. Phương thức countGold () không thẻ được gọi từ một static định nghĩa bài văn.
 A, B, C và E không chính xác dựa trên thông tin trên. (Mục tiêu 6.2 của OCA)
10. A đúng. Theo mặc định, các phương thức của giao diện là công khai nên Phương thức Tablet.doStuff cũng phải ở chế độ công khai. Phần còn lại của mã là hợp lệ.
 B, C, D và E là không chính xác dựa trên các điều trên. (Mục tiêu 7.5 của OCA)
11. C và E là cú pháp chính xác cho nhập tĩnh. Dòng 4 không sử dụng nhập tĩnh, vì vậy mã cũng sẽ biên dịch mà không cần nhập.

A, B, D và F là không chính xác dựa trên các điều trên. (Mục tiêu 1.4 của OCA)

12. B C và F đều đúng. Kể từ Java 8, các giao diện có thể có mặc định và các phương thức tĩnh.

A, D và E không chính xác. A không có kiểu trả về; D không thể có thân phương thức; và E cần một thân phương thức. (Mục tiêu 7.5 của OCA)

13. C và D đúng.

A không chính xác vì Java là ngôn ngữ được nhập tĩnh. B không chính xác vì nó không cung cấp con trỏ. E không chính xác vì JavaScript là một triển khai của tiêu chuẩn ECMA, không phải Java. F không chính xác vì việc sử dụng bytecode và JVM cung cấp các cơ chế bảo mật chính của Java.



2

Hướng đối tượng

CHỨNG NHẬN

MỤC TIÊU

- Mô tả đóng gói • Thực hiện kế thừa • Sử dụng các mối quan hệ IS-A và HAS-A (OCP) • Sử dụng đa hình • Sử dụng ghi đè và ghi đè • Hiểu cách truyền • Sử dụng giao diện

- Hiểu và sử dụng các loại trả lại • Phát triển trình xây dựng • Sử dụng các thành viên tĩnh

✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

Being a Oracle Certified Associate (OCA) 8 có nghĩa là bạn phải thông thạo các khía cạnh hướng đối tượng của Java. Bạn phải mơ về hệ thống phân cấp thừa kế; sức mạnh của tính đa hình phải chảy qua bạn; và tính đóng gói phải trở thành bản chất thứ hai đối với bạn.

(Các mẫu khớp nối, liên kết, bố cục và thiết kế sẽ trở thành bánh mì và bơ của bạn khi bạn là Chuyên gia được chứng nhận Oracle [OCP] 8.) Chương này sẽ chuẩn bị cho bạn tất cả các mục tiêu hướng đối tượng và các câu hỏi bạn sẽ gặp trên thi. Chúng tôi đã nghe nói về nhiều lập trình viên Java có kinh nghiệm, những người chưa thực sự trở nên thông thạo với các công cụ hướng đối tượng mà Java

cung cấp, vì vậy chúng tôi sẽ bắt đầu từ đầu.

MỤC TIÊU XÁC NHÂN

Đóng gói (Mục tiêu OCA 6.1 và 6.5)

6.1 Tạo phương thức với đối số và giá trị trả về; bao gồm các phương thức được nạp chèo.

6.5 Áp dụng các nguyên tắc đóng gói cho một lớp.

Hãy tưởng tượng bạn đã viết mã cho một lớp học và hàng tá lập trình viên khác từ công ty của bạn đều viết các chương trình sử dụng lớp học của bạn. Bây giờ hãy tưởng tượng rằng sau này, bạn không thích cách lớp này hoạt động, bởi vì một số biến phiên bản của nó đang được đặt (bởi các lập trình viên khác từ trong mã của họ) thành các giá trị mà bạn không lường trước được. Mã của họ gây ra lỗi trong mã của bạn.

(Thư giãn, đây chỉ là giả thuyết.) Chà, nó là một chương trình Java, vì vậy bạn sẽ có thể đưa ra một phiên bản mới hơn của lớp, mà chúng có thể thay thế trong chương trình của mình mà không cần thay đổi bất kỳ mã nào của riêng chúng.

Kịch bản này nêu bật hai trong số những hứa hẹn / lợi ích của ngôn ngữ hướng đối tượng (OO): tính linh hoạt và khả năng bảo trì. Nhưng những lợi ích đó không tự động đến. Bạn cần làm gì đó. Bạn phải viết các lớp và mã của mình theo cách hỗ trợ tính linh hoạt và khả năng bảo trì. Vậy nếu Java hỗ trợ OO thì sao? Nó không thể thiết kế mã của bạn cho bạn. Ví dụ: hãy tưởng tượng bạn đã tạo lớp của mình với các biến cá thể công khai và những lập trình viên khác đang đặt trực tiếp các biến cá thể đó, như đoạn mã sau minh họa:

```
public class BadOO {
    public int size;
    public int weight;
    ...
}

public class ExploitBadOO {
    public static void main (String [] args) {
        BadOO b = new BadOO();
        b.size = -5; // Legal but bad!!
    }
}
```

Và bây giờ bạn đang gặp rắc rối. Bạn sẽ thay đổi lớp học theo cách như thế nào?

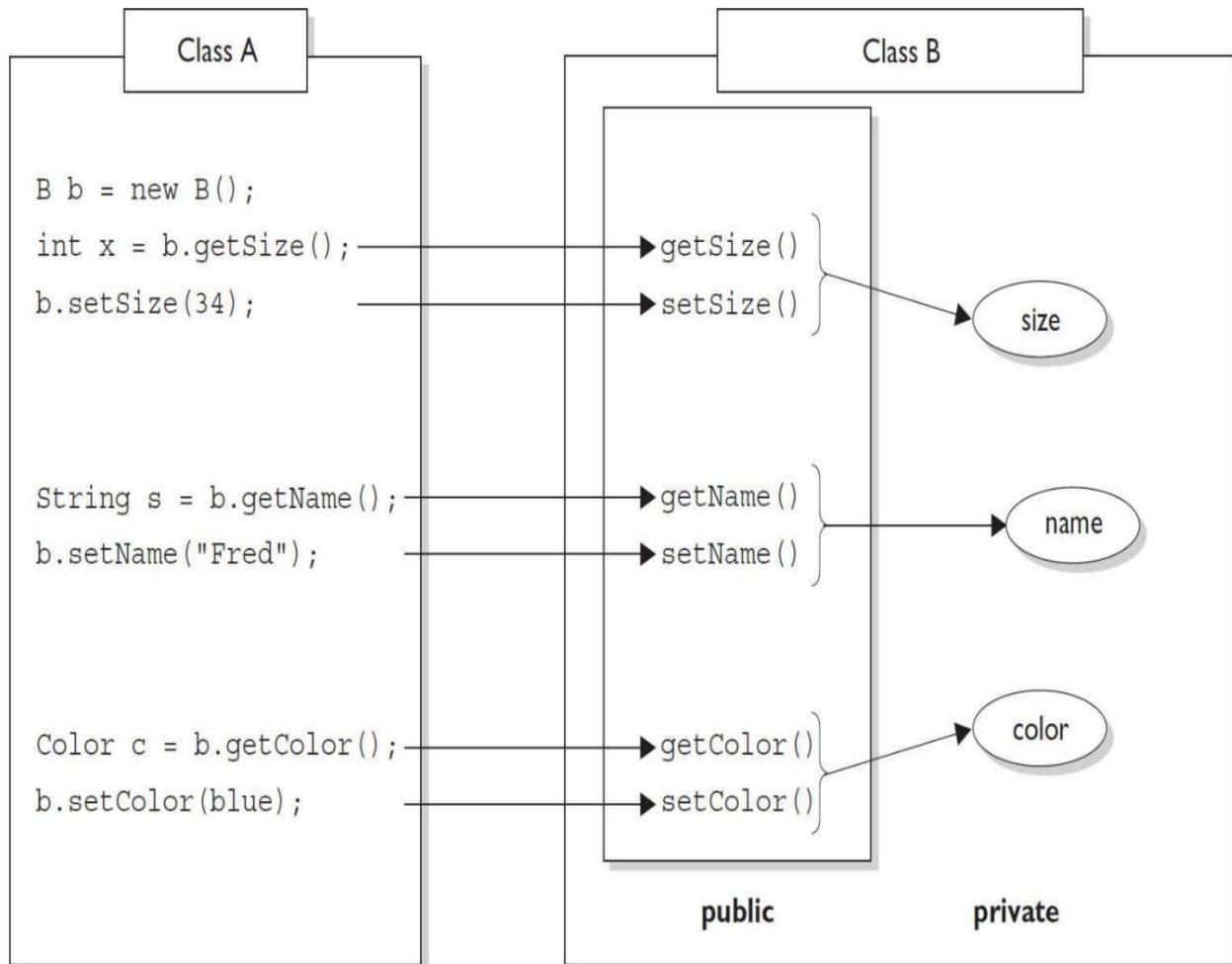
điều đó cho phép bạn xử lý các vấn đề xảy ra khi ai đó thay đổi biến kích thước thành một giá trị gây ra sự cố? Lựa chọn duy nhất của bạn là quay lại và viết mã phương thức để điều chỉnh kích thước (ví dụ: phương thức `setSize (int a)`) và sau đó cách ly biến kích thước bằng một công cụ sửa đổi quyền truy cập riêng tư. Nhưng ngay sau khi bạn thực hiện thay đổi đó với mã của mình, bạn sẽ phá vỡ mã của những người khác!

Khả năng thực hiện các thay đổi trong mã triển khai của bạn mà không phá mã của những người khác sử dụng mã của bạn là lợi ích chính của việc đóng gói. Bạn muốn ẩn các chi tiết triển khai đằng sau một giao diện lập trình công khai. Theo giao diện, chúng tôi có nghĩa là tập hợp các phương thức có thể truy cập mà mã của bạn cung cấp cho mã khác để gọi – nói cách khác, là API của mã của bạn. Bằng cách ẩn chi tiết triển khai, bạn có thể làm lại mã phương thức của mình (có thể cũng thay đổi cách các biến được sử dụng bởi lớp của bạn) mà không buộc thay đổi mã gọi phương thức đã thay đổi của bạn.

Nếu bạn muốn khả năng bảo trì, tính linh hoạt và khả năng mở rộng (và tất nhiên, bạn do), thiết kế của bạn phải bao gồm tính đóng gói. Làm thế nào để bạn làm điều đó?

- Giữ ẩn các biến cá thể (với một công cụ sửa đổi quyền truy cập, thường là riêng tư).
- Tạo các phương thức của trình truy cập công khai và buộc mã gọi sử dụng các phương thức đó thay vì truy cập trực tiếp vào biến cá thể. Những phương thức được gọi là trình truy cập này cho phép người dùng trong lớp của bạn đặt giá trị của một biến hoặc nhận giá trị của một biến.
- Đổi với các phương thức truy cập này, hãy sử dụng quy ước đặt tên phổ biến nhất của set `<SomeProperty>` và lấy `<SomeProperty>`.

[Hình 2-1](#) minh họa ý tưởng rằng việc đóng gói buộc người gọi mã của chúng ta phải đi qua các phương thức thay vì truy cập trực tiếp vào các biến.



Class A cannot access Class B instance variable data without going through getter and setter methods. Data is marked private; only the accessor methods are public.

HÌNH 2-1 Mô hình của đóng gói

Chúng tôi gọi các phương thức truy cập là getters và setters, mặc dù một số thích thuật ngữ huyền thoại người truy cập và người đột biến. (Cá nhân chúng tôi không thích từ "đột biến".) Bất kể bạn gọi chúng là gì, chúng là các phương thức mà các lập trình viên khác phải trải qua để truy cập các biến cá thể của bạn. Chúng trông đơn giản và có thể bạn đã sử dụng chúng mãi mãi:

```

public class Box {
    // hide the instance variable; only an instance
    // of Box can access it
    private int size;
    // Provide public getters and setters
    public int getSize() {
        return size;
    }

    public void setSize(int newSize) {
        size = newSize;
    }
}

```

Đợi tí. Mã trước hữu ích như thế nào? Nó thậm chí không thực hiện bất kỳ xác nhận hoặc xử lý nào. Có thể có lợi gì khi có bộ cài và bộ định tuyến không thêm chức năng? Vấn đề là, bạn có thể thay đổi quyết định sau đó và thêm nhiều mã hơn vào các phương thức của mình mà không làm hỏng API của bạn. Ngay cả khi hôm nay bạn không nghĩ mình thực sự cần xác thực hoặc xử lý dữ liệu, thì thiết kế OO tốt sẽ quyết định rằng bạn có kế hoạch cho tương lai. Để an toàn, hãy buộc mã gọi đi qua các phương thức của bạn thay vì chuyển trực tiếp đến các biến phiên bản. Luôn luôn. Sau đó, bạn có thể tự do làm lại các triển khai phương pháp của mình sau này mà không phải chịu sự phẫn nộ của hàng tá lập trình viên biết bạn sống ở đâu.

Lưu ý: Trong [Chương 6](#), chúng ta sẽ xem lại chủ đề về tính đóng gói vì nó áp dụng cho các biến cá thể cũng là các biến tham chiếu. Nó phức tạp hơn bạn nghĩ, vì vậy hãy chú ý theo dõi! (Ngoài ra, chúng tôi sẽ đợi đến [Chương 6](#) để thử thách bạn với các câu hỏi giả lập theo chủ đề đóng gói.)



Để ý đoạn mã có vẻ như đang hỏi về hoạt động của một phương thức, khi vấn đề thực sự là thiếu tính đóng gói. Hãy xem ví dụ sau và xem liệu bạn có thể tìm ra điều gì đang xảy ra không:

```

class Foo {
    public int left = 9;
    public int right = 3;
    public void setLeft(int leftNum) {
        left = leftNum;
        right = leftNum/3;
    }
    // lots of complex test code here
}

```

Bây giờ hãy xem xét câu hỏi này: Giá trị của bên phải luôn luôn bằng một phần ba giá trị của bên trái? Có vẻ như nó sẽ xảy ra, cho đến khi bạn nhận ra rằng người dùng của lớp Foo không cần sử dụng phương thức setLeft () ! Họ có thể chỉ cần đi thẳng đến các biến cá thể và thay đổi chúng thành bất kỳ giá trị int tùy ý nào.

MỤC TIÊU XÁC NHẬN

Tính kế thừa và tính đa hình (Mục tiêu của OCA 7.1 và 7.2)

7.1 Mô tả thừa kế và lợi ích của nó.

7.2 Phát triển mã thể hiện việc sử dụng tính đa hình; bao gồm ghi đè và kiểu đối tượng so với kiểu tham chiếu (sic).

Sự kế thừa có ở khắp mọi nơi trong Java. Thật an toàn khi nói rằng hầu như (gần như?) Không thể viết ngay cả chương trình Java nhỏ nhất mà không sử dụng tính năng thừa kế. Để khám phá chủ đề này, chúng ta sẽ sử dụng toán tử instanceof , mà chúng ta sẽ thảo luận chi tiết hơn trong [Chương 4](#). Bây giờ, chỉ cần nhớ rằng instanceof trả về true nếu biến tham chiếu đang được kiểm tra thuộc loại được so sánh với. Mã này

```

class Test {
    public static void main(String [] args) {
        Test t1 = new Test();
        Test t2 = new Test();
        if (!t1.equals(t2))
            System.out.println("they're not equal");
        if (t1 instanceof Object)
            System.out.println("t1's an Object");
    }
}

```

tạo ra đầu ra này:

chúng không bằng t1
là một Đối tượng

Phương pháp bằng đó đến từ đâu? Biến tham chiếu t1 là của nhập Test và không có phương thức bằng trong lớp Test . Hay là có? Bài kiểm tra if thứ hai hỏi liệu t1 có phải là một thể hiện của lớp Object hay không, và vì nó (sẽ sớm có thêm điều đó), nên kiểm tra if thành công.

Chờ đã. làm sao t1 có thể là một thể hiện của kiểu Đối tượng, khi chúng ta vừa nói rằng nó là của loại Kiểm tra? Tôi chắc rằng bạn đang đi trước chúng ta ở đây, nhưng hóa ra mọi lớp trong Java đều là một lớp con của lớp Object (tất nhiên là ngoại trừ chính lớp Object).

Nói cách khác, mọi lớp bạn từng sử dụng hoặc từng viết sẽ kế thừa từ Đối tượng lớp. Bạn sẽ luôn có một phương thức bằng , một phương thức sao chép , thông báo, chờ và những phương thức khác có sẵn để sử dụng. Bất cứ khi nào bạn tạo một lớp, bạn sẽ tự động kế thừa tất cả các phương thức của lớp Đối tượng .

Tại sao? Ví dụ , hãy xem xét phương thức bằng đó . Những người tạo ra Java một cách chính xác giả định rằng các lập trình viên Java sẽ rất phổ biến khi muốn so sánh các trường hợp của các lớp của họ để kiểm tra sự bình đẳng. Nếu class Object không có phương thức bằng , bạn phải tự viết một phương thức – bạn và mọi lập trình viên Java khác. Phương thức bằng một trong đó đã được kế thừa hàng tỷ lần.

(Công bằng mà nói, số bằng cũng đã được ghi đè hàng tỷ lần, nhưng chúng tôi đang vượt lên chính mình.)

Sự phát triển của sự kế thừa

Cho đến Java 8, khi chủ đề kế thừa được thảo luận, nó thường xoay quanh các lớp con kế thừa các phương thức từ các lớp cha của chúng. Mặc dù sự đơn giản hóa này không bao giờ đúng hoàn toàn, nhưng nó đã trở nên ít đúng hơn với các tính năng mới có sẵn trong Java 8. Như bảng sau cho thấy, bây giờ có thể

các tính năng có sẵn trong Java 8. Như bảng sau cho thấy, bây giờ có thể kế thừa các phương thức cụ thể từ các giao diện. Đây là một thay đổi lớn. Đối với phần còn lại của chương, khi chúng ta nói về kế thừa nói chung, chúng ta sẽ có xu hướng sử dụng các thuật ngữ "kiểu con" và "kiểu siêu" để thừa nhận rằng cả hai lớp và giao diện đều cần được tính đến. Chúng ta sẽ có xu hướng sử dụng các thuật ngữ "lớp con" và "lớp mẹ" khi chúng ta thảo luận về một ví dụ cụ thể đang được thảo luận.

Tính kế thừa là một khía cạnh quan trọng của hầu hết các chủ đề mà chúng ta sẽ thảo luận trong chương này, vì vậy hãy chuẩn bị cho RẤT NHIỀU cuộc thảo luận về sự tương tác giữa siêu kiểu và kiểu phụ!

Khi nghiên cứu bảng sau, bạn sẽ nhận thấy rằng giao diện Java 8 có thể chứa hai loại phương thức cụ thể, tĩnh và mặc định. Chúng ta sẽ thảo luận về những bổ sung quan trọng này sau trong chương này.

[Bảng 2-1](#) tóm tắt các phần tử của các lớp và giao diện liên quan đến tính kế thừa.

BẢNG 2-1 Phần tử kế thừa của các lớp và giao diện

Elements of Types	Classes	Interfaces
Instance variables	Yes	Not applicable
Static variables	Yes	Only constants
Abstract methods	Yes	Yes
Instance methods	Yes	Java 8, default methods
Static methods	Yes	Java 8, inherited no, accessible yes
Constructors	No	Not applicable
Initialization blocks	No	Not applicable

Đối với bài kiểm tra, bạn sẽ cần biết rằng bạn có thể tạo các mối quan hệ kế thừa trong Java bằng cách mở rộng một lớp hoặc bằng cách triển khai một giao diện. Cũng cần hiểu rằng hai lý do phổ biến nhất để sử dụng kế thừa là

- Để thúc đẩy việc sử dụng lại mã
- Để sử dụng tính đa hình

Hãy bắt đầu với việc tái sử dụng. Một cách tiếp cận thiết kế phổ biến là tạo một phiên bản khá chung chung của một lớp với mục đích tạo ra nhiều lớp con chuyên biệt hơn kế thừa từ đó. Ví dụ:

```
class GameShape {
    public void displayShape() {
        System.out.println("displaying shape");
    }
    // more code
}

class PlayerPiece extends GameShape {
    public void movePiece() {
        System.out.println("moving game piece");
    }
    // more code
}

public class TestShapes {
    public static void main (String[] args) {
        PlayerPiece shape = new PlayerPiece();
        shape.displayShape();
        shape.movePiece();
    }
}
```

kết quả đầu ra:

```
hiển thị mảnh trò chơi
di chuyển hình dạng
```

Lưu ý rằng lớp PlayerPiece kế thừa phương thức displayShape () chung từ lớp GameShape ít chuyên dụng hơn và cũng thêm phương thức riêng của nó, movePiece (). Tái sử dụng mã thông qua kế thừa có nghĩa là các phương thức có chức năng chung – chẳng hạn như displayShape (), có thể áp dụng cho nhiều loại hình dạng khác nhau trong trò chơi – không cần phải thực hiện lại. Điều đó có nghĩa là tất cả các lớp con chuyên biệt của GameShape đều được đảm bảo có các khả năng của lớp con tổng quát hơn. Bạn không muốn phải viết lại mã displayShape () trong mỗi thành phần chuyên biệt của trò chơi trực tuyến.

Nhưng bạn biết điều đó. Bạn đã trải qua nỗi đau của mã trùng lặp khi bạn thực hiện thay đổi ở một nơi và phải theo dõi tất cả những nơi khác có mã giống (hoặc rất giống) đó tồn tại.

mã tương tự (hoặc rất giống) tồn tại.

Cách sử dụng kế thừa thứ hai (và có liên quan) là cho phép các lớp của bạn được truy cập đa hình - một khả năng được cung cấp bởi các giao diện, nhưng chúng ta sẽ làm điều đó sau một phút. Giả sử bạn có một lớp GameLauncher muốn lặp qua danh sách các loại đối tượng GameShape khác nhau và gọi displayShape () trên mỗi đối tượng đó. Tại thời điểm bạn viết lớp này, bạn không biết mọi loại con GameShape có thể có mà bất kỳ ai khác sẽ viết.

Và bạn chắc chắn không muốn phải làm lại mã của mình chỉ vì ai đó đã quyết định tạo hình xác sáu tháng sau đó.

Điều tuyệt vời về tính đa hình ("nhiều dạng") là bạn có thể xử lý bất kỳ lớp nào của GameShape dưới dạng GameShape. Nói cách khác, bạn có thể viết mã trong lớp GameLauncher của mình với nội dung "Tôi không quan tâm bạn là loại đối tượng nào miễn là bạn kế thừa từ (mở rộng) GameShape. Và theo như tôi biết, nếu bạn mở rộng GameShape, thì chắc chắn bạn đã có một phương thức displayShape () , vì vậy tôi biết tôi có thể gọi nó. "

Hãy tưởng tượng bây giờ chúng ta có hai lớp con chuyên biệt mở rộng Lớp GameShape , PlayerPeces và TilePiece:

```
class GameShape {
    public void displayShape() {
        System.out.println("displaying shape");
    }
    // more code
}

class PlayerPiece extends GameShape {
    public void movePiece() {
        System.out.println("moving game piece");
    }
    // more code
}

class TilePiece extends GameShape {
    public void getAdjacent() {
        System.out.println("getting adjacent tiles");
    }
    // more code
}
```

Bây giờ hãy tưởng tượng một lớp thử nghiệm có một phương thức với kiểu đối số được khai báo là GameShape, có nghĩa là nó có thể nhận bất kỳ loại GameShape nào. Nói cách khác, bất kỳ lớp con nào của GameShape đều có thể được chuyển cho một phương thức có đối số kiểu

GameShape. Mã này

```
public class TestShapes {
    public static void main (String[] args) {
        PlayerPiece player = new PlayerPiece();
        TilePiece tile = new TilePiece();
        doShapes(player);
        doShapes(tile);
    }

    public static void doShapes(GameShape shape) {
        shape.displayShape();
    }
}
```

kết quả đầu ra:

```
hiển thị hình dạng
hiển thị hình dạng
```

Điểm mấu chốt là phương thức doShapes () được khai báo với một GameShape nhưng có thể được truyền vào bất kỳ kiểu con nào (trong ví dụ này là lớp con) của GameShape. Sau đó, phương thức có thể gọi bất kỳ phương thức nào của GameShape mà không cần quan tâm đến kiểu lớp thời gian chạy thực tế của đối tượng được truyền cho phương thức. Tuy nhiên, có những hàm ý. Phương thức doShapes () chỉ biết rằng các đối tượng là một loại GameShape vì đó là cách tham số được khai báo. Và việc sử dụng một biến tham chiếu được khai báo là kiểu GameShape – bất kể biến đó là tham số phương thức, biến cục bộ hay biến cá thể – có nghĩa là chỉ có thể gọi các phương thức của GameShape trên đó. Các phương thức bạn có thể gọi trên một tham chiếu hoàn toàn phụ thuộc vào kiểu được khai báo của biến, bất kể đối tượng thực tế là gì, mà tham chiếu đang tham chiếu đến. Điều đó có nghĩa là bạn không thể sử dụng biến GameShape để gọi, chẳng hạn như phương thức getAdjacent () ngay cả khi đối tượng được truyền vào thuộc loại TilePiece. (Chúng ta sẽ thấy điều này một lần nữa khi chúng ta xem xét các giao diện.)

Mối quan hệ IS-A và HAS-A

Lưu ý: Kể từ Mùa đông 2017, bài thi OCA 8 sẽ không hỏi bạn trực tiếp về mối quan hệ IS-A và HAS-A. Nhưng hiểu được mối quan hệ IS-A và HAS-A sẽ giúp ích cho các thí sinh OCA 8 với nhiều câu hỏi trong đề thi.

LÀ MỘT

Trong OO, khái niệm IS-A dựa trên sự kế thừa (hoặc triển khai giao diện). IS-A là một cách nói, "Thứ này là một loại của thứ kia." Ví dụ, Mustang là một loại Ngựa, vì vậy theo thuật ngữ OO, chúng ta có thể nói, "Mustang IS-A Horse." Xe Subaru IS-A. Bông cải xanh LÀ-Một loại rau (không phải là một loại rau rất thú vị, nhưng nó vẫn có giá trị). Bạn thể hiện mối quan hệ IS-A trong Java thông qua các từ khóa mở rộng (để kế thừa lớp) và thực hiện (để triển khai giao diện).

```
public class Car {
    // Cool Car code goes here
}

public class Subaru extends Car {
    // Important Subaru-specific stuff goes here
    // Don't forget Subaru inherits accessible Car members which
    // can include both methods and variables.
}
```

Xe hơi là một loại Phương tiện, vì vậy cây kế thừa có thể bắt đầu từ Hạng xe như sau:

```
public class Vehicle { ... }
public class Car extends Vehicle { ... }
public class Subaru extends Car { ... }
```

Theo thuật ngữ OO, bạn có thể nói như sau:

Xe là lớp siêu cấp của Xe.
Car là lớp con của Vehicle.
Xe hơi là siêu phẩm của Subaru.
Subaru là lớp con của Xe.
Xe kế thừa từ Xe.
Subaru kế thừa từ cả Xe và Xe.
Subaru có nguồn gốc từ Xe hơi.
Xe có nguồn gốc từ Xe.
Subaru có nguồn gốc từ Xe.
Subaru là một loại phụ của cả Xe và Xe hơi.

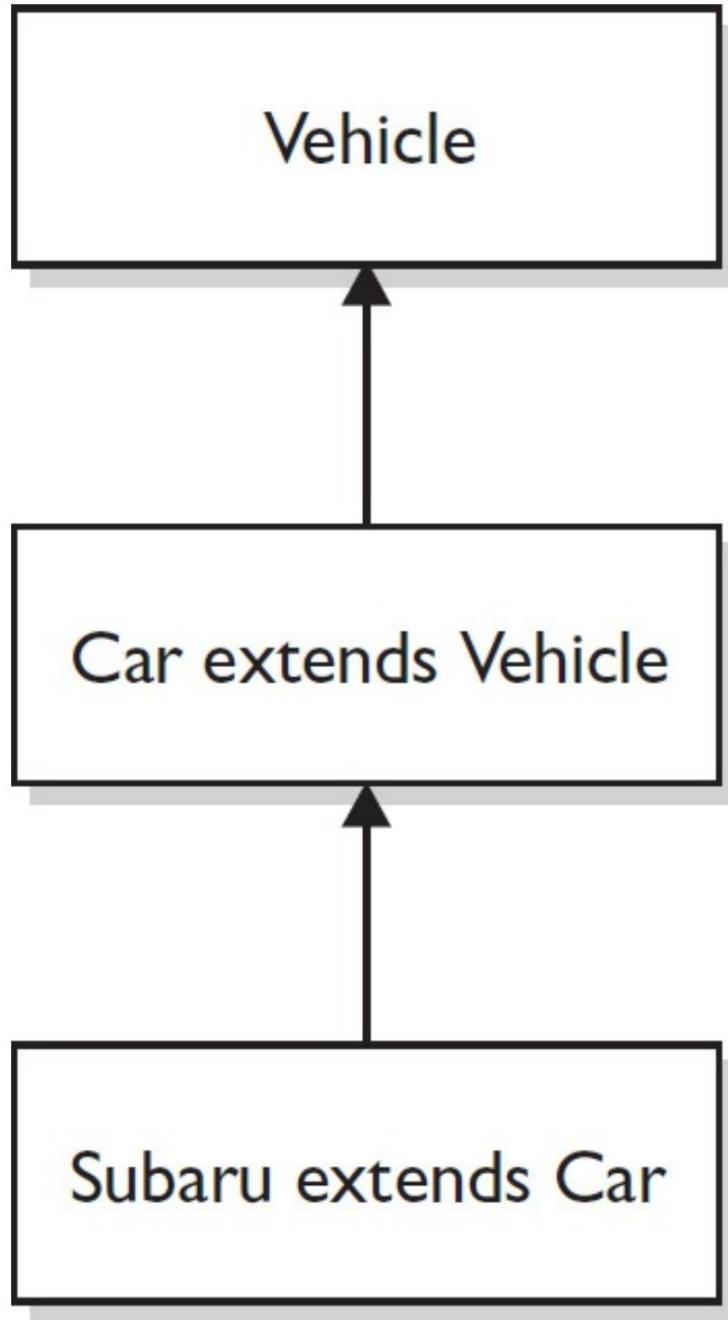
Quay trở lại mối quan hệ IS-A của chúng tôi, các câu sau là đúng:

"Xe mở rộng Xe" có nghĩa là "Xe IS-A Phương tiện".
"Subaru mở rộng Car" có nghĩa là "Subaru IS-A Car."

Và chúng ta cũng có thể nói:

"Subaru IS-A Vehicle"

bởi vì một lớp được cho là "một loại" bất cứ thứ gì ở xa hơn trong cây kế thừa của nó. Nếu biểu thức (Foo instanceof Bar) là true, thì lớp Foo IS-A Bar, ngay cả khi Foo không trực tiếp mở rộng Bar, nhưng thay vào đó mở rộng một số lớp khác là lớp con của Bar. [Hình 2-2](#) minh họa cây kế thừa cho Xe cộ, Xe hơi và Subaru. Các mũi tên di chuyển từ lớp con sang lớp cha. Nói cách khác, mũi tên của một lớp hướng về lớp mà từ đó nó mở rộng.



HÌNH 2-2 Mô hình thừa kế cho Xe, Xe hơi, Subaru

CÓ MỘT

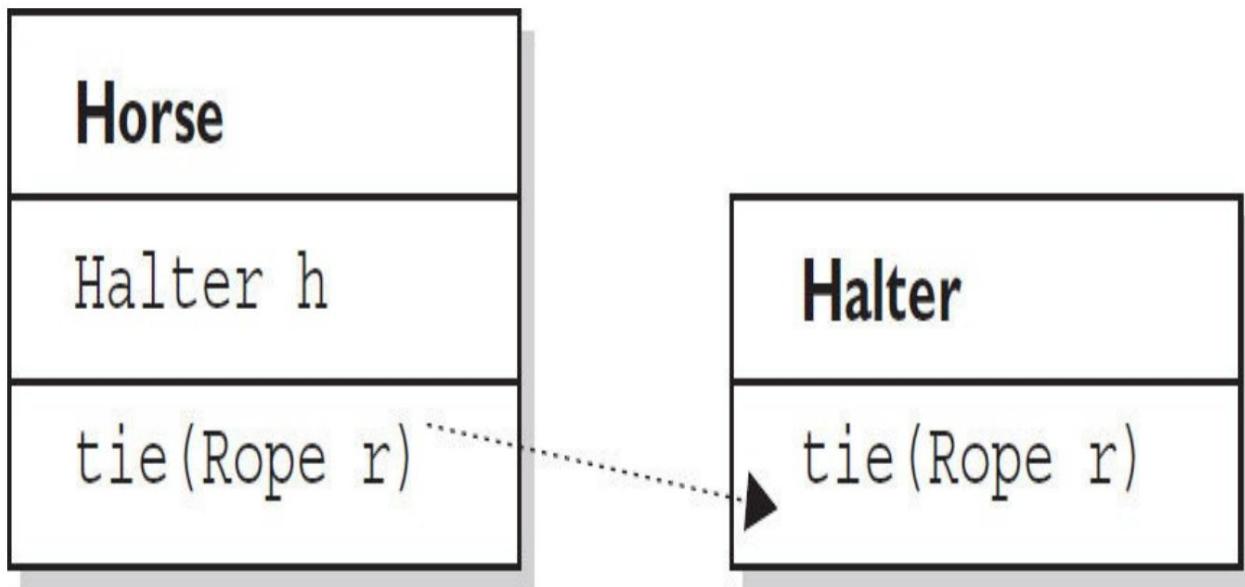
Mỗi quan hệ HAS-A dựa trên việc sử dụng, thay vì kế thừa. Nói cách khác, lớp A HAS-A B nếu mã trong lớp A có tham chiếu đến một thể hiện của lớp B. Ví dụ, bạn có thể nói như sau:

Một con ngựa là một con vật. Một con ngựa HAS-A Halter.

Mã có thể trông như thế này:

```
public class Animal { }
public class Horse extends Animal {
    private Halter myHalter;
}
```

Trong mã này, lớp Horse có một biến thể hiện của loại Halter (dây là một phần của thiết bị bạn có thể có nếu bạn có ngựa), vì vậy bạn có thể nói rằng "Horse HAS-A Halter." Nói cách khác, Horse có một tham chiếu đến Halter. Mã Horse có thể sử dụng tham chiếu Halter đó để gọi các phương thức trên Halter và nhận hành vi Halter mà không cần có mã (phương thức) liên quan đến Halter trong chính lớp Horse .
[Hình 2-3](#) minh họa mối quan hệ HAS-A giữa Horse và Halter.



Horse class has a Halter, because Horse declares an instance variable of type Halter.
When code invokes tie() on a Horse instance, the Horse invokes tie() on the Horse object's Halter instance variable.

HÌNH 2-3

quan hệ giữa Horse và Halter

Mỗi quan hệ HAS-A cho phép bạn thiết kế các lớp tuân theo các thông lệ tốt của OO bằng cách không có các lớp nguyên khôi thực hiện một số việc khác nhau. Các lớp (và các đối tượng kết quả của chúng) nên là các chuyên gia. Như bạn của chúng tôi, Andrew nói, "Các lớp học chuyên biệt thực sự có thể giúp giảm thiểu lỗi." Lớp càng chuyên biệt, càng có nhiều khả năng bạn có thể sử dụng lại lớp đó trong các ứng dụng khác. Nếu bạn đặt tất cả mã liên quan đến Halter trực tiếp vào lớp Horse , bạn sẽ kết thúc

sao chép mã trong lớp Cow , lớp UnpaidIntern và bất kỳ lớp nào khác có thể cần hành vi Halter . Bằng cách giữ mã Halter trong một lớp Halter chuyên biệt, riêng biệt , bạn có cơ hội sử dụng lại lớp Halter trong nhiều ứng dụng.

Người dùng của lớp Horse (nghĩa là mã gọi các phương thức trên một cá thể Horse) nghĩ rằng lớp Ngựa có hành vi Halter . Ví dụ, lớp Horse có thể có phương thức buộc (dây thừng) . Người dùng của lớp Horse không bao giờ phải biết rằng khi họ gọi phương thức tie () , đối tượng Horse sẽ quay lại và ủy quyền lời gọi cho lớp Halter của nó bằng cách gọi myHalter.tie (dây thừng). Kịch bản vừa được mô tả có thể trông như thế này:

```
public class Horse extends Animal {
    private Halter myHalter = new Halter();
    public void tie(LeadRope rope) {
        myHalter.tie(rope); // Delegate tie behavior to the
                            // Halter object
    }
}
public class Halter {
    public void tie(LeadRope aRope) {
        // Do the actual tie work here
    }
}
```

Trong OO, chúng tôi không muốn người gọi lo lắng về lớp hoặc đối tượng nào thực sự đang thực hiện công việc thực sự. Để điều đó xảy ra, lớp Horse ẩn các chi tiết triển khai khỏi người dùng Horse . Người dùng Horse yêu cầu đối tượng Horse làm những việc (trong trường hợp này là tự buộc) và Horse sẽ làm điều đó hoặc như ví dụ này, yêu cầu một thứ gì đó khác (chẳng hạn như một phương thức của lớp Animal kế thừa) làm điều đó. Tuy nhiên, đối với người gọi, có vẻ như đối tượng Horse luôn tự lo cho mình. Người dùng Horse thậm chí không cần biết rằng có một thứ như lớp Halter .

TỪ LỚP HỌC

Thiết kế hướng đối tượng

Mối quan hệ IS-A và HAS-A và sự đóng gói chỉ là phần nổi của tảng băng khi nói đến thiết kế OO. Nhiều sách và luận văn tốt nghiệp có

tảng băng khi nói đến thiết kế OO. Nhiều sách và luận văn tốt nghiệp đã được dành riêng cho chủ đề này. Lý do cho việc nhấn mạnh vào thiết kế phù hợp rất đơn giản: tiền. Chi phí để cung cấp một ứng dụng phần mềm được ước tính là đắt hơn gấp mười lần đối với các chương trình được thiết kế kém.

Ngay cả những nhà thiết kế OO giỏi nhất (thường được gọi là "kiến trúc sư") cũng mắc sai lầm. Rất khó để hình dung mối quan hệ giữa hàng trăm, thậm chí hàng nghìn lớp. Khi những sai lầm được phát hiện trong giai đoạn thực hiện (viết mã) của một dự án, số lượng mã phải được viết lại đôi khi có nghĩa là các nhóm lập trình phải bắt đầu lại từ đầu.

Ngành công nghiệp phần mềm đã phát triển để hỗ trợ các nhà thiết kế. Các ngôn ngữ mô hình hóa đối tượng trực quan, chẳng hạn như Ngôn ngữ mô hình hóa hợp nhất (UML), cho phép các nhà thiết kế thiết kế và dễ dàng sửa đổi các lớp mà không cần phải viết mã trước vì các thành phần OO được biểu diễn bằng đồ họa. Điều này cho phép các nhà thiết kế tạo ra một bản đồ của các mối quan hệ lớp và giúp họ nhận ra các lỗi trước khi bắt đầu viết mã. Một sự đổi mới khác trong thiết kế OO là các mẫu thiết kế. Các nhà thiết kế nhận thấy rằng nhiều thiết kế OO áp dụng nhất quán từ dự án này sang dự án khác và việc áp dụng các thiết kế giống nhau sẽ rất hữu ích vì nó làm giảm khả năng đưa ra các lỗi thiết kế mới. Các nhà thiết kế OO sau đó bắt đầu chia sẻ những thiết kế này với nhau. Bây giờ có rất nhiều danh mục về các mẫu thiết kế này cả trên Internet và ở dạng sách.

Mặc dù việc vượt qua kỳ thi chứng chỉ Java không yêu cầu bạn phải hiểu kỹ về thiết kế OO này, hy vọng thông tin cơ bản này sẽ giúp bạn hiểu rõ hơn lý do tại sao người viết bài kiểm tra lại chọn đưa tính đóng gói và các mối quan hệ IS-A và HAS-A vào bài thi.

—Jonathan Meeks,

Lập trình viên Java được chứng nhận của Sun

MỤC TIÊU XÁC NHÂN

Đa hình (Mục tiêu OCA 7.2)

7.2 Phát triển mã thể hiện việc sử dụng tính đa hình; bao gồm ghi đè và kiểu đối tượng so với kiểu tham chiếu (sic).

Hãy nhớ rằng bất kỳ đối tượng Java nào có thể vượt qua nhiều hơn một bài kiểm tra IS-A đều có thể được coi là đa hình. Khác với các đối tượng của kiểu Đối tượng, tất cả các đối tượng Java đều đa hình ở chỗ chúng vượt qua bài kiểm tra IS-A cho kiểu riêng của chúng và cho lớp

Sự vật.

Hãy nhớ rằng cách duy nhất để truy cập một đối tượng là thông qua một tham chiếu Biến đổi. Có một số điều chính bạn nên biết về tài liệu tham khảo:

- Một biến tham chiếu có thể chỉ có một kiểu và một khi được khai báo, kiểu đó không bao giờ có thể thay đổi được (mặc dù đối tượng mà nó tham chiếu có thể thay đổi).
- Tham chiếu là một biến, vì vậy nó có thể được gán lại cho các đối tượng khác (trừ khi tham chiếu được khai báo là cuối cùng).
- Kiểu của biến tham chiếu xác định các phương thức có thể được gọi trên đối tượng mà biến đó đang tham chiếu.
- Một biến tham chiếu có thể tham chiếu đến bất kỳ đối tượng nào cùng kiểu với tham chiếu đã khai báo, hoặc – đây là tham chiếu lớn – nó có thể tham chiếu đến bất kỳ kiểu con nào của kiểu đã khai báo!
- Một biến tham chiếu có thể được khai báo như một kiểu lớp hoặc một kiểu giao diện. Nếu biến được khai báo là một kiểu giao diện, nó có thể tham chiếu đến bất kỳ đối tượng nào của bất kỳ lớp nào triển khai giao diện.

Trước đó, chúng tôi đã tạo một lớp GameShape được mở rộng bởi hai lớp khác, Người chơi và mảnh ghép. Bây giờ, hãy tưởng tượng bạn muốn tạo hoạt ảnh cho một số hình dạng trên gameboard. Nhưng không phải tất cả các hình dạng đều có thể hoạt hình, vậy bạn sẽ làm gì với kẻ thừa lớp?

Chúng ta có thể tạo một lớp bằng phương thức `animate()` và chỉ có một số Các lớp con GameShape kế thừa từ lớp đó? Nếu chúng ta có thể, thì chúng ta có thể có PlayerPiece, ví dụ, mở rộng cả lớp GameShape và lớp Animatable, trong khi TilePiece sẽ chỉ mở rộng GameShape. Nhưng không, điều này sẽ không hoạt động! Java chỉ hỗ trợ kế thừa một lớp duy nhất! Điều đó có nghĩa là một lớp chỉ có thể có một lớp cha ngay lập tức. Nói cách khác, nếu PlayerPiece là một lớp, không có cách nào để nói điều gì đó như thế này:

```
class PlayerPeces mở rộng GameShape, Animatable { // KHÔNG! // mã khác
}
```

Một lớp không thể mở rộng nhiều hơn một lớp: nghĩa là một lớp cha cho mỗi lớp. Tuy nhiên, một lớp có thể có nhiều tổ tiên, vì lớp B có thể mở rộng lớp A và lớp C có thể mở rộng lớp B, v.v. Vì vậy, bất kỳ lớp nhất định nào cũng có thể có nhiều lớp trong cây kế thừa của nó, nhưng điều đó không giống như việc nói rằng một lớp mở rộng trực tiếp hai lớp.



Một số ngôn ngữ (chẳng hạn như C++) cho phép một lớp mở rộng nhiều hơn một lớp khác. Khả năng này được gọi là “đa kế thừa”. Lý do mà những người tạo ra Java đã chọn không cho phép kế thừa nhiều lớp là nó có thể trở nên khá lộn xộn. Tóm lại, vấn đề là nếu một lớp mở rộng hai lớp khác và cả hai lớp cha đều có phương thức doStuff (), thì lớp con sẽ kế thừa phiên bản nào của doStuff ()? Vấn đề này có thể dẫn đến một kịch bản được gọi là “Kim cương chết chóc của cái chết”, vì hình dạng của biểu đồ lớp có thể được tạo trong một thiết kế đa kế thừa. Hình thoi được hình thành khi các lớp B và C đều mở rộng A, và cả B và C kế thừa một phương thức từ A. Nếu lớp D mở rộng cả B và C, và cả B và C đã ghi đè phương thức trong A, thì lớp D có, trong lý thuyết, kế thừa hai cách triển khai khác nhau của cùng một phương pháp. Được vẽ dưới dạng một sơ đồ lớp, hình dạng của bốn lớp trông giống như một hình thoi.

Vì vậy, nếu điều đó không hiệu quả, bạn có thể làm gì khác? Bạn có thể chỉ cần đặt mã animate () vào GameShape, sau đó vô hiệu hóa phương thức này trong các lớp không thể hoạt ảnh. Nhưng đó là một lựa chọn thiết kế tồi vì nhiều lý do - nó dễ bị lỗi hơn; nó làm cho lớp GameShape kém gắn kết hơn; và điều đó có nghĩa là API GameShape “quảng cáo” rằng tất cả các hình dạng đều có thể được làm động trong khi trên thực tế, điều đó không đúng vì chỉ một số lớp con của GameShape mới có thể chạy phương thức animate () thành công.



Để nhắc lại, kể từ Java 8, các giao diện có thể có các phương thức cụ thể (được gọi là các phương thức mặc định). Điều này cho phép tạo ra một dạng nhiều lần xen kẽ, mà chúng ta sẽ thảo luận ở phần sau của chương.

Vậy bạn có thể làm gì khác ? Bạn đã biết câu trả lời – hãy tạo giao diện Hoạt hình và chỉ có các lớp con của GameShape có thể hoạt hình triển khai giao diện đó. Đây là giao diện:

```
public interface Animatable {
    public void animate();
}
```

Và đây là lớp PlayerPiece đã sửa đổi triển khai giao diện:

```
class PlayerPiece extends GameShape implements Animatable {
    public void movePiece() {
        System.out.println("moving game piece");
    }
    public void animate() {
        System.out.println("animating...");
    }
    // more code
}
```

Vì vậy, bây giờ chúng tôi có một PlayerPeces vượt qua bài kiểm tra IS-A cho cả lớp GameShape và giao diện Animatable . Điều đó có nghĩa là PlayerPiece có thể được xử lý đa hình như một trong bốn thứ tại bất kỳ thời điểm nào, tùy thuộc vào loại được khai báo của biến tham chiếu:

- Một đối tượng (vì bất kỳ đối tượng nào kế thừa từ Đối tượng)
- Một GameShape (vì PlayerPiece mở rộng GameShape)
- Một người chơi (vì đó là những gì nó thực sự là)
- Một Animatable (vì PlayerPiece triển khai Animatable)

Sau đây là tất cả các khai báo hợp pháp. Nhìn kĩ:

```
PlayerPiece player = new PlayerPiece();
Object o = player;
GameShape shape = player;
Animatable mover = player;
```

Chỉ có một đối tượng ở đây - một phiên bản của loại PlayerPiece - nhưng có bốn loại biến tham chiếu khác nhau, tất cả đều đè cập đến một đối tượng đó trên heap . Câu đố pop: Biến tham chiếu nào ở trước có thể gọi phương thức displayShape () ? Gợi ý: Chỉ có thể sử dụng hai trong bốn khai báo để gọi phương thức displayShape () .

Hãy nhớ rằng các lệnh gọi phương thức được trình biên dịch cho phép chỉ dựa trên kiểu đã khai báo của tham chiếu, bất kể kiểu đối tượng là gì. Vì vậy, hãy xem xét lại bốn loại tham chiếu – Object, GameShape, PlayerPiece và

Hoạt hình – kiểu nào trong bốn kiểu này biết về phương thức displayShape () ?

Bạn đoán nó – cả lớp GameShape và lớp PlayerPiece đều được biết (bởi trình biên dịch) để có phương thức displayShape () , vì vậy một trong hai kiểu tham chiếu đó có thể được sử dụng để gọi displayShape () . Hãy nhớ rằng đối với trình biên dịch, PlayerP mảnh IS-A GameShape, vì vậy trình biên dịch nói, "Tôi thấy rằng loại được khai báo là PlayerPiece, và vì PlayerPiece mở rộng GameShape, điều đó có nghĩa là PlayerPiece kế thừa phương thức displayShape () . Do đó, PlayerPiece có thể được sử dụng để gọi phương thức displayShape () . "

Phương thức nào có thể được gọi khi đối tượng PlayerPiece đang được tham chiếu bằng cách sử dụng tham chiếu được khai báo là kiểu Animatable? Chỉ có phương thức animate () . Tất nhiên, điều thú vị ở đây là bất kỳ lớp nào từ bất kỳ cây kế thừa nào cũng có thể triển khai Animatable, điều đó có nghĩa là nếu bạn có một phương thức với đối số được khai báo là kiểu Animatable, bạn có thể truyền vào các đối tượng PlayerP mảnh , đối tượng SpinningLogo và bất kỳ thứ gì khác một thể hiện của một lớp triển khai Animatable. Và bạn có thể sử dụng tham số đó (thuộc loại Hoạt hình) để gọi phương thức animate () , nhưng không gọi phương thức displayShape () (mà nó có thể không có) hoặc bất kỳ thứ gì khác ngoài những gì đã biết đối với trình biên dịch dựa trên loại tham chiếu . Tuy nhiên, trình biên dịch luôn biết rằng bạn có thể gọi các phương thức của lớp Object trên bất kỳ đối tượng nào, vì vậy chúng an toàn để gọi bất kể tham chiếu - lớp hoặc giao diện - được sử dụng để tham chiếu đến đối tượng.

Chúng tôi đã bỏ qua một phần lớn của tất cả điều này, đó là mặc dù trình biên dịch chỉ biết về kiểu tham chiếu đã khai báo, Máy ảo Java (JVM) trong thời gian chạy biết đối tượng thực sự là gì. Và điều đó có nghĩa là ngay cả khi phương thức displayShape () của đối tượng PlayerPiece được gọi bằng cách sử dụng biến tham chiếu GameShape , nếu PlayerPiece ghi đè phương thức displayShape () , JVM sẽ gọi phiên bản PlayerPiece ! JVM nhìn vào đối tượng thực ở đầu kia của tham chiếu, "thấy" rằng nó đã ghi đè phương thức của kiểu biến tham chiếu đã khai báo và gọi phương thức của lớp thực tế của đối tượng. Nhưng có một điều khác cần ghi nhớ:

Các lời gọi phương thức đa hình chỉ áp dụng cho các phương thức cá thể. Bạn luôn có thể tham chiếu đến một đối tượng có kiểu biến tham chiếu tổng quát hơn (lớp cha hoặc giao diện), nhưng trong thời gian chạy, CHỈ những thứ được chọn động dựa trên đối tượng thực tế (chứ không phải kiểu tham chiếu) là các phương thức thể hiện. Không phải là phương thức tĩnh . Không phải biến. Chỉ các phương thức phiên bản bị ghi đè mới được gọi động dựa trên kiểu của đối tượng thực.

Bởi vì định nghĩa này phụ thuộc vào sự hiểu biết rõ ràng về ghi đè và sự khác biệt giữa phương thức tĩnh và phương thức cá thể, chúng tôi sẽ đề cập đến những điều đó ở phần sau của chương.

MỤC TIÊU XÁC NHẬN

Ghi đè / Quá tải (Mục tiêu 6.1 và 7.2 của OCA)

6.1 Tạo phương thức với đối số và giá trị trả về; bao gồm các phương thức được nạp chồng.

7.2 Phát triển mã thể hiện việc sử dụng tính đa hình; bao gồm ghi đè và kiểu đối tượng so với kiểu tham chiếu (sic).

Đề thi sẽ sử dụng các phương pháp ghi đè và quá tải trên rất nhiều câu hỏi. Hai khái niệm này thường bị nhầm lẫn (có lẽ vì chúng có tên giống nhau?), Nhưng mỗi khái niệm có bộ quy tắc độc đáo và phức tạp của riêng nó. Điều quan trọng là phải thực sự rõ ràng về việc "over" sử dụng các quy tắc nào!

Các phương thức bị ghi đè

Bất kỳ lúc nào một kiểu kế thừa một phương thức từ một kiểu siêu cấp, bạn có cơ hội ghi đè phương thức (trừ khi, như bạn đã học trước đó, phương thức được đánh dấu là cuối cùng). Lợi ích chính của việc ghi đè là khả năng xác định hành vi cụ thể cho một loại phụ cụ thể. Ví dụ sau minh họa một lớp con Ngựa của Động vật ghi đè phiên bản Động vật của phương thức eat () :

```
public class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay, oats, "
                           + "and horse treats");
    }
}
```

Đối với các phương thức trừu tượng mà bạn kế thừa từ một siêu kiểu, bạn không có lựa chọn nào khác: Bạn phải triển khai phương thức trong kiểu con trừ khi kiểu con đó cũng là trừu tượng. Các phương thức trừu tượng phải được thực thi bởi lớp con cụ thể đầu tiên, nhưng điều này giống như nói rằng lớp con cụ thể ghi đè các phương thức trừu tượng của

(các) siêu kiểu. Vì vậy, bạn có thể nghĩ về các phương thức trừu tượng như là các phương thức mà bạn buộc phải ghi đè – cuối cùng.

Người tạo lớp Động vật có thể đã quyết định rằng vì mục đích đa hình, tất cả các kiểu con Động vật phải có phương thức eat () được định nghĩa theo một cách duy nhất. Nói một cách đa hình, khi một tham chiếu Động vật không tham chiếu đến một cá thể Động vật mà đến một cá thể lớp con Động vật , người gọi sẽ có thể gọi hàm eat () trên tham chiếu Động vật , nhưng đối tượng thời gian chạy thực tế (ví dụ, một cá thể Ngựa) sẽ chạy nó riêng phương thức eat () cụ thể . Đánh dấu trừu tượng phương thức eat () là cách lập trình viên Animal nói với tất cả các nhà phát triển lớp con, "Sẽ không có ý nghĩa gì đối với kiểu con mới của bạn khi sử dụng phương thức eat () chung chung , vì vậy bạn phải nghĩ ra phương thức ăn của riêng mình () thực hiện phương pháp! " Một ví dụ (nonabstract) về việc sử dụng tính đa hình trông như sau:

```
public class TestAnimals {
    public static void main (String [] args) {
        Animal a = new Animal();
        Animal b = new Horse(); // Animal ref, but a Horse object
        a.eat(); // Runs the Animal version of eat()
        b.eat(); // Runs the Horse version of eat()
    }
}
class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay, oats, "
                           + "and horse treats");
    }
    public void buck() { }
}
```

Trong đoạn mã trước, lớp kiểm tra sử dụng tham chiếu Animal để gọi một phương thức trên đối tượng Horse . Hãy nhớ rằng, trình biên dịch sẽ chỉ cho phép các phương thức trong lớp Animal được gọi khi sử dụng một tham chiếu đến một Animal. Điều sau sẽ không hợp pháp nếu mã trước đó:

```
Animal c = new Horse();
c.buck(); // Can't invoke buck();
           // Animal class doesn't have that method
```

Để nhắc lại, trình biên dịch chỉ xem xét kiểu tham chiếu, không phải trường hợp loại hình. Tính đa hình cho phép bạn sử dụng một tham chiếu siêu kiểu trừu tượng hơn (bao gồm một giao diện) cho một trong các kiểu con của nó (bao gồm cả những người triển khai giao diện).

Phương thức ghi đè không được có công cụ sửa đổi quyền truy cập hạn chế hơn phương thức đang được ghi đè (ví dụ: bạn không thể ghi đè phương thức được đánh dấu là công khai và đặt nó được bảo vệ). Hãy suy nghĩ về điều đó: Nếu lớp Animal quăng cáo một phương thức eat () công khai và ai đó có một tham chiếu Animal (nói cách khác, một tham chiếu được khai báo là loại Animal), thì ai đó sẽ cho rằng việc gọi eat () trên tham chiếu Animal là an toàn bất kể của trường hợp thực tế mà tham chiếu Động vật đang đè cập đến. Nếu một kiểu con được phép đột nhập và thay đổi công cụ sửa đổi truy cập trên phương thức ghi đè, thì đột ngột trong thời gian chạy – khi JVM gọi phiên bản của đối tượng thực (Horse) của phương thức chứ không phải phiên bản của kiểu tham chiếu (Động vật) – chương trình sẽ chết một cái chết kinh hoàng. (Chưa kể đến nỗi đau đớn về tình cảm cho kẻ bị phản bội bởi kiểu phụ lưu manh.)

Hãy sửa đổi ví dụ đa hình mà chúng ta đã thấy trước đó trong phần này:

```
public class TestAnimals {
    public static void main (String [] args) {
        Animal a = new Animal();
        Animal b = new Horse(); // Animal ref, but a Horse object
        a.eat(); // Runs the Animal version of eat()
        b.eat(); // Runs the Horse version of eat()
    }
}
class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
class Horse extends Animal {
    private void eat() { // whoa! - it's private!
        System.out.println("Horse eating hay, oats, "
                           + "and horse treats");
    }
}
```

Nếu mã này được biên dịch (mã này không được biên dịch), thì mã sau sẽ không thành công trong thời gian chạy:

```
Animal b = new Horse(); // Animal ref, but a Horse
// object, so far so good
b.eat(); // Meltdown at runtime!
```

Biến b có kiểu là Animal, có phương thức là public eat () . Nhưng hãy nhớ rằng trong thời gian chạy, Java sử dụng lệnh gọi phương thức ảo để chọn động phiên bản thực của phương thức sẽ chạy, dựa trên phiên bản thực tế.

Tham chiếu Động vật luôn có thể tham chiếu đến cá thể Ngựa , vì Ngựa LÀ (n) Động vật. Điều làm cho tham chiếu siêu kiểu đó đến một cá thể kiểu con có thể thực hiện được là kiểu con được đảm bảo có thể thực hiện mọi thứ mà siêu kiểu có thể làm.

Cho dù cá thể Horse ghi đè các phương thức kế thừa của Động vật hay đơn giản là kế thừa chúng, bất kỳ ai có tham chiếu Động vật đến cá thể Ngựa đều có thể gọi tất cả các phương thức Động vật có thể truy cập được. Vì lý do đó, một phương thức ghi đè phải thực hiện hợp đồng của lớp cha.

Lưu ý: Trong [Chương 5](#) , chúng ta sẽ tìm hiểu chi tiết việc xử lý ngoại lệ. Khi bạn đã đã nghiên cứu [Chương 5](#), bạn sẽ đánh giá cao danh sách các quy tắc ghi đè đơn giản này.

Các quy tắc ghi đè một phương thức như sau:

- Danh sách đối số phải khớp chính xác với danh sách của phương thức được ghi đè. Nếu chúng không khớp, bạn có thể kết thúc với một phương thức quá tải mà bạn không có ý định.
- Kiểu trả về phải giống, hoặc một kiểu con của kiểu trả về được khai báo trong phương thức ghi đè ban đầu trong lớp cha. (Thông tin thêm về điều này trong một vài trang khi chúng ta thảo luận về lợi tức hiệp phương sai.)
- Cấp độ truy cập không thể hạn chế hơn cấp độ của phương thức được ghi đè.
- Cấp độ truy cập CÓ THỂ ít hạn chế hơn so với cấp độ của phương thức được ghi đè.
- Phương thức phiên bản chỉ có thể được ghi đè nếu chúng được kế thừa bởi kiểu con. Một kiểu con trong cùng một gói với kiểu siêu kiểu của cá thể có thể ghi đè lên bất kỳ phương thức siêu kiểu nào không được đánh dấu là riêng tư hoặc cuối cùng.
Một kiểu con trong một gói khác chỉ có thể ghi đè những phương thức không phải là phương thức cuối cùng được đánh dấu là công khai hoặc được bảo vệ (vì các phương thức được bảo vệ được kế thừa bởi kiểu con).
- Phương thức ghi đè CÓ THỂ ném bất kỳ ngoại lệ không được kiểm tra (thời gian chạy) nào, bất kể phương thức ghi đè có tuyên bố ngoại lệ hay không.
(Xem thêm trong [Chương 5](#).)
- Phương thức ghi đè KHÔNG được ném các ngoại lệ đã kiểm tra mới hoặc rộng hơn các ngoại lệ được khai báo bởi phương thức ghi đè. Ví dụ: một phương thức khai báo một FileNotFoundException không thể bị ghi đè bởi một phương thức khai báo một SQLException, Exception hoặc bất kỳ phương thức nào khác

ngoại lệ nonruntime trừ khi đó là một lớp con của `FileNotFoundException`.

- Phương thức ghi đè có thể tạo ra các ngoại lệ hẹp hơn hoặc ít hơn. Chỉ vì một phương thức ghi đè "chấp nhận rủi ro" không có nghĩa là ngoại lệ của kiểu con ghi đè cũng chịu rủi ro tương tự. Điểm mấu chốt: một phương thức ghi đè không phải khai báo bắt kỳ ngoại lệ nào mà nó sẽ không bao giờ ném ra, bất kể phương thức ghi đè khai báo những gì.
- Bạn không thể ghi đè một phương pháp được đánh dấu là cuối cùng.
- Bạn không thể ghi đè một phương thức được đánh dấu là tĩnh. Chúng ta sẽ xem xét một ví dụ trong một vài trang khi chúng ta thảo luận chi tiết hơn về các phương thức tĩnh.
- Nếu một phương thức không thể được kế thừa, bạn không thể ghi đè nó. Hãy nhớ rằng ghi đè ngụ ý rằng bạn đang thực hiện lại một phương pháp mà bạn đã kế thừa!
Ví dụ: mã sau không hợp pháp và ngay cả khi bạn đã thêm một phương thức `eat()` vào `Horse`, nó sẽ không phải là một bản ghi đè của phương thức `eat()` của `Animal`:

```
public class TestAnimals {
    public static void main (String [] args) {
        Horse h = new Horse();
        h.eat(); // Not legal because Horse didn't inherit eat()
    }
}
class Animal {
    private void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
class Horse extends Animal { }
```

Gọi một phiên bản siêu kiểu của một phương thức bị ghi đè Thông thường, bạn sẽ muốn tận dụng một số mã trong phiên bản siêu kiểu của một phương thức, nhưng vẫn ghi đè nó để cung cấp một số hành vi cụ thể bổ sung. Nó giống như nói, "Chạy phiên bản supertype của phương thức, sau đó quay lại đây và kết thúc với mã phương thức bổ sung subtype của tôi."

(Lưu ý rằng không có yêu cầu rằng phiên bản supertype phải chạy trước mã subtype.) Rất dễ thực hiện trong mã bằng cách sử dụng từ khóa super như sau:

```

public class Animal {
    public void eat() { }
    public void printYourself() {
        // Useful printing code goes here
    }
}
class Horse extends Animal {
    public void printYourself() {
        // Take advantage of Animal code, then add some more
        super.printYourself(); // Invoke the superclass
        // (Animal) code
        // Then do Horse-specific
        // print work here
    }
}

```

Theo cách tương tự, bạn có thể truy cập phương thức ghi đè của giao diện bằng cú pháp:

Giao diệnX.super.doStuff ();

Lưu ý: Việc sử dụng super để gọi một phương thức ghi đè chỉ áp dụng cho các phương thức cá thể. (Hãy nhớ rằng không thể ghi đè các phương thức tĩnh.) Và bạn chỉ có thể sử dụng super để truy cập một phương thức trong siêu kiểu của một kiểu, không phải siêu kiểu của siêu kiểu – nghĩa là bạn không thể nói super.super.doStuff () và bạn không thể nói: InterfaceX.super.super.doStuff ().



Nếu một phương thức bị ghi đè nhưng bạn sử dụng tham chiếu đa hình (siêu kiểu) để tham chiếu đến đối tượng kiểu con với phương thức ghi đè, trình biên dịch sẽ giả định rằng bạn đang gọi phiên bản siêu kiểu của phương thức. Nếu phiên bản siêu kiểu khai báo một ngoại lệ đã kiểm tra, nhưng phương thức ghi đè kiểu con thì không, trình biên dịch vẫn nghĩ rằng bạn đang gọi một phương thức khai báo một ngoại lệ (xem thêm trong [Chương 5](#)). Hãy xem một ví dụ:

```

class Animal {
    public void eat() throws Exception {
        // throws an Exception
    }
}
class Dog2 extends Animal {
    public void eat() { /* no Exceptions */ }
    public static void main(String [] args) {
        Animal a = new Dog2();
        Dog2 d = new Dog2();
        d.eat();           // ok
        a.eat();          // compiler error -
                          // unreported exception
    }
}

```

Mã này sẽ không biên dịch vì ngoại lệ được khai báo trên phương thức Động vật eat () . Điều này xảy ra mặc dù trong thời gian chạy, phương thức eat () được sử dụng sẽ là phiên bản Dog , không khai báo ngoại lệ.

Ví dụ về ghi đè phương pháp bất hợp pháp

Hãy xem qua việc ghi đè phương thức eat () của Animal:

```

lớp công cộng Động vật {
    public void eat () {}
}

```

[Bảng 2-2](#) liệt kê các ví dụ về ghi đè bất hợp pháp của phương thức Animal eat () , đã đưa ra phiên bản trước của lớp Animal .

BẢNG 2-2	về Ghi đè Bất hợp pháp
----------	------------------------

Illegal Override Code	Problem with the Code
private void eat() { }	Access modifier more restrictive
public void eat() throws IOException { }	Declares a checked exception not defined by superclass version
public void eat(String food) { }	A legal overload, not an override, because the argument list changed
public String eat() { }	Not an override because of the return type, and not an overload either because there's no change in the argument list

Phương pháp quá tải

Các phương thức bị quá tải cho phép bạn sử dụng lại cùng một tên phương thức trong một lớp, nhưng với các đối số khác nhau (và, tùy chọn, một kiểu trả về khác). Việc quá tải một phương thức thường có nghĩa là bạn đang đẽp hơn một chút đối với những người gọi phương thức của bạn bởi vì mã của bạn có gánh nặng đối phó với các loại đối số khác nhau thay vì buộc người gọi thực hiện chuyển đổi trước khi gọi phương thức của bạn.

Các quy tắc không quá phức tạp:

- Các phương thức bị quá tải PHẢI thay đổi danh sách đối số.
- Các phương thức bị quá tải CÓ THỂ thay đổi kiểu trả về.
- Các phương thức bị quá tải CÓ THỂ thay đổi công cụ sửa đổi quyền truy cập.
- Các phương thức bị quá tải CÓ THỂ khai báo các ngoại lệ được kiểm tra mới hoặc rộng hơn.
- Một phương thức có thể được nạp chồng trong cùng một kiểu hoặc trong một kiểu con. Nói cách khác, nếu lớp A định nghĩa một phương thức doStuff (int i) , thì lớp con B có thể định nghĩa một phương thức doStuff (String s) mà không cần ghi đè phiên bản lớp cha lấy một int. Vì vậy, hai phương thức có cùng tên nhưng khác kiểu vẫn có thể được coi là quá tải nếu kiểu con kế thừa một phiên bản của phương thức và sau đó khai báo một phiên bản được nạp chồng khác trong định nghĩa kiểu của nó.



Các nhà phát triển Java ít kinh nghiệm thường nhầm lẫn về sự khác biệt tinh tế giữa các phương thức được nạp chồng và ghi đè. Hãy cẩn thận để nhận ra khi một phương thức bị quá tải thay vì bị ghi đè. Bạn có thể thấy một phương thức dường như vi phạm quy tắc ghi đè, nhưng đó thực sự là một phương thức quá tải hợp pháp, như sau:

```
public class Foo {
    public void doStuff(int y, String s) { }
    public void moreThings(int x) { }
}
class Bar extends Foo {
    public void doStuff(int y, long s) throws IOException { }
}
```

Thật hấp dẫn khi coi IOException là vấn đề vì phương thức doStuff () bị ghi đè không khai báo ngoại lệ và IOException được trình biên dịch kiểm tra. Nhưng phương thức doStuff () không bị ghi đè! Subclass Bar nạp chồng phương thức doStuff () bằng cách thay đổi danh sách đối số, vì vậy IOException vẫn ổn.

Quá tải hợp pháp

Hãy xem xét một phương thức chúng ta muốn nạp chồng:

```
public void changeSize(int size, String name, float pattern) { }
```

The following methods are legal overloads of the changeSize () method:

```
public void changeSize(int size, String name) { }
private int changeSize(int size, float pattern) { }
public void changeSize(float pattern, String name)
        throws IOException { }
```

Gọi các phương thức quá tải Trong [Chương](#)

[6](#) , chúng ta sẽ xem xét cách thức boxing và var-args ảnh hưởng đến quá tải. (Tuy nhiên, bạn vẫn phải chú ý đến những gì được đề cập ở đây.)

Khi một phương thức được gọi, nhiều phương thức cùng tên có thể tồn tại cho kiểu đối tượng mà bạn đang gọi một phương thức. Ví dụ, lớp Horse có thể có ba phương thức có cùng tên nhưng có danh sách đối số khác nhau, điều đó có nghĩa là phương thức này bị quá tải.

Việc quyết định phương thức so khớp nào sẽ gọi dựa trên các đối số. Nếu bạn gọi phương thức có đối số Chuỗi , thì phiên bản nạp chồng nhận Chuỗi được gọi. Nếu bạn gọi một phương thức có cùng tên nhưng chuyển nó vào một float, thì phiên bản quá tải có một float sẽ chạy. Nếu bạn gọi phương thức cùng tên nhưng chuyển cho nó một đối tượng Foo và không có phiên bản quá tải nào lấy một Foo, thì trình biên dịch sẽ phản nản rằng nó không thể tìm thấy một đối tượng phù hợp. Sau đây là các ví dụ về việc gọi các phương thức nạp chồng:

```
class Adder {
    public int addThem(int x, int y) {
        return x + y;
    }

    // Overload the addThem method to add doubles instead of ints
    public double addThem(double x, double y) {
        return x + y;
    }
}

// From another class, invoke the addThem() method
public class TestAdder {
    public static void main (String [] args) {
        Adder a = new Adder();
        int b = 27;
        int c = 3;
        int result = a.addThem(b,c);           // Which addThem is invoked?
        double doubleResult = a.addThem(22.5,9.3); // Which addThem?
    }
}
```

Trong mã TestAdder này , lệnh gọi đầu tiên tới a.addThem (b, c) chuyển hai int cho phương thức, do đó, phiên bản đầu tiên của addThem () - phiên bản nạp chồng có hai đối số int - được gọi. Lệnh gọi thứ hai tới a.addThem (22.5, 9.3) chuyển hai đối số kép cho phương thức, do đó, phiên bản thứ hai của addThem () - phiên bản nạp chồng có hai đối số kép - được gọi.

Việc gọi các phương thức nạp chồng lấy tham chiếu đối tượng thay vì các phương thức nguyên thủy sẽ thú vị hơn một chút. Giả sử bạn có một phương thức quá tải sao cho một phiên bản lấy Động vật và một phiên bản lấy Ngựa (lớp con của Động vật). Nếu bạn chuyển một đối tượng Horse trong lệnh gọi phương thức, bạn sẽ gọi quá tải

phiên bản có một con ngựa. Hoặc như vậy, thoát nhìn có vẻ như:

```
class Animal { }
class Horse extends Animal { }
class UseAnimals {
    public void doStuff(Animal a) {
        System.out.println("In the Animal version");
    }

    public void doStuff(Horse h) {
        System.out.println("In the Horse version");
    }
    public static void main (String [] args) {
        UseAnimals ua = new UseAnimals();
        Animal animalObj = new Animal();
        Horse horseObj = new Horse();
        ua.doStuff(animalObj);
        ua.doStuff(horseObj);
    }
}
```

Đầu ra là những gì bạn mong đợi:

```
Trong phiên bản động vật
Trong phiên bản Ngựa
```

Nhưng nếu bạn sử dụng tham chiếu Động vật cho đối tượng Ngựa thì sao?

```
Animal AnimalRefToHorse = new Horse (); ua.doStuff
    (animalRefToHorse);
```

Phiên bản quá tải nào được gọi? Bạn có thể muốn trả lời, "Cái láy một Con ngựa vì nó là một đối tượng Con ngựa trong thời gian chạy đang được chuyển cho phương thức." Nhưng đó không phải là cách nó hoạt động. Đoạn mã trước đó sẽ thực sự in ra:

trong phiên bản Động vật

Mặc dù đối tượng thực tế trong thời gian chạy là Ngựa chứ không phải là Động vật, việc lựa chọn phương thức nạp chồng nào để gọi (nói cách khác, chữ ký của phương thức) KHÔNG được quyết định động trong thời gian chạy.

Chỉ cần nhớ rằng kiểu tham chiếu (không phải kiểu đối tượng) xác định phương thức nạp chồng nào được gọi!

Tóm lại, phiên bản ghi đè của phương thức nào để gọi (nói cách khác, từ lớp nào trong cây kế thừa) được quyết định trong thời gian chạy dựa trên kiểu đối tượng , nhưng phiên bản được nạp chồng của phương thức sẽ gọi dựa trên kiểu tham chiếu của đối số được truyền vào lúc biên dịch .

Nếu bạn gọi một phương thức chuyển nó một tham chiếu Động vật đến một đối tượng Ngựa , trình biên dịch chỉ biết về Động vật, vì vậy nó sẽ chọn phiên bản quá tải của phương thức lấy Động vật. Nó không quan trọng rằng, trong thời gian chạy, một con ngựa thực sự đang được vượt qua.



Main () có thể bị quá tải không?

```
class DuoMain {
    public static void main(String[] args) {
        main(1);
    }
    static void main(int i) {
        System.out.println("overloaded main");
    }
}
```

Chắc chắn rồi! Nhưng main () duy nhất có siêu năng lực JVM là cái có chữ ký mà bạn đã thấy khoảng 100 lần trong cuốn sách này.

Tính đa hình trong các phương thức bị quá tải và bị ghi đè Tính đa hình hoạt động như thế nào với các phương thức bị quá tải? Từ những gì chúng ta vừa xem xét, có vẻ như tính đa hình không quan trọng khi một phương thức bị quá tải. Nếu bạn truyền tham chiếu Động vật , phương thức nạp chồng lấy Động vật sẽ được gọi, ngay cả khi đối tượng thực được truyền là Ngựa. Tuy nhiên, khi Ngựa giả dạng Động vật tham gia vào phương thức, đối tượng Ngựa vẫn là Ngựa mặc dù đã được chuyển vào một phương thức mong đợi một Động vật. Vì vậy, đúng là tính đa hình không xác định phiên bản quá tải nào được gọi; tuy nhiên, tính đa hình phát huy tác dụng khi quyết định về phiên bản ghi đè của phương thức nào được gọi. Nhưng đôi khi một phương thức vừa bị quá tải vừa bị ghi đè. Hãy tưởng tượng rằng các lớp Động vật và Ngựa trông như thế này:

```
public class Animal {  
    public void eat() {  
        System.out.println("Generic Animal Eating Generically");  
    }  
}  
public class Horse extends Animal {  
    public void eat() {  
        System.out.println("Horse eating hay ");  
    }  
    public void eat(String s) {  
        System.out.println("Horse eating " + s);  
    }  
}
```

Lưu ý rằng lớp Horse có cả quá tải và ghi đè phương thức eat().
[Bảng 2-3](#) cho thấy phiên bản nào của ba phương thức eat() sẽ chạy tùy thuộc vào cách chúng được gọi.

BẢNG 2-3 dụ về Ghi đè hợp pháp và bất hợp pháp

Method Invocation Code	Result
Animal a = new Animal(); a.eat();	Generic Animal Eating Generically
Horse h = new Horse(); h.eat();	Horse eating hay
Animal ah = new Horse(); ah.eat();	Horse eating hay Polymorphism works—the actual object type (Horse), not the reference type (Animal), is used to determine which eat () is called.
Horse he = new Horse(); he.eat("Apples");	Horse eating Apples The overloaded eat (String s) method is invoked.
Animal a2 = new Animal(); a2.eat("treats");	Compiler error! Compiler sees that the Animal class doesn't have an eat () method that takes a String.
Animal ah2 = new Horse(); ah2.eat("Carrots");	Compiler error! Compiler still looks only at the reference and sees that Animal doesn't have an eat () method that takes a String. Compiler doesn't care that the actual object might be a Horse at runtime.



Đừng để bị lừa bởi một phương thức bị quá tải nhưng không bị ghi đè bởi một lớp con. Hoàn toàn hợp pháp khi làm những việc sau:

```
public class Foo {
    void doStuff() { }
}
class Bar extends Foo {
    void doStuff(String s) { }
}
```

Lớp Bar có hai phương thức doStuff () : phiên bản no-arg mà nó kế thừa từ Foo (và không ghi đè) và doStuff (chuỗi s) được định nghĩa trong lớp Bar . Mã có tham chiếu đến Foo chỉ có thể gọi

phiên bản no-arg, nhưng mã có tham chiếu đến một Bar có thể gọi một trong hai phiên bản đã được nạp chồng.

[Bảng 2-4](#) tóm tắt sự khác biệt giữa các phương thức được nạp chồng và ghi đè.

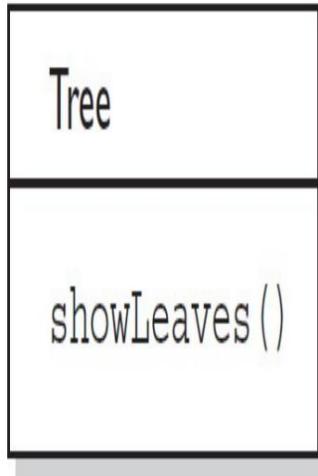
BẢNG 2-4

c biệt giữa các phương pháp được ghi đè và được ghi đè

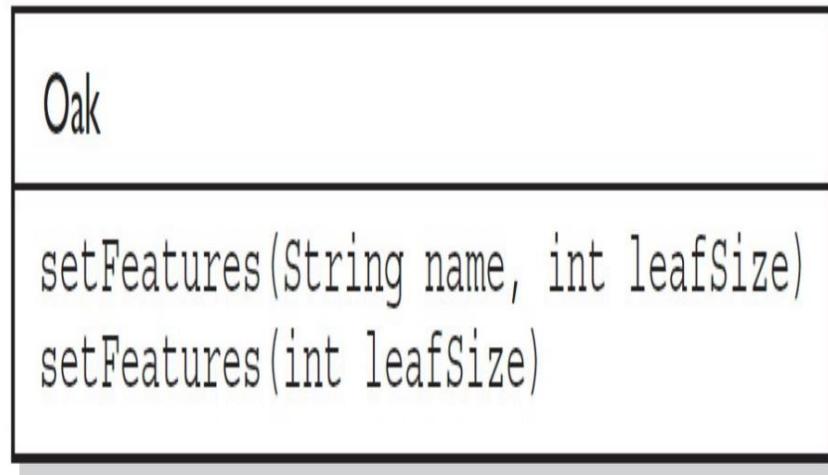
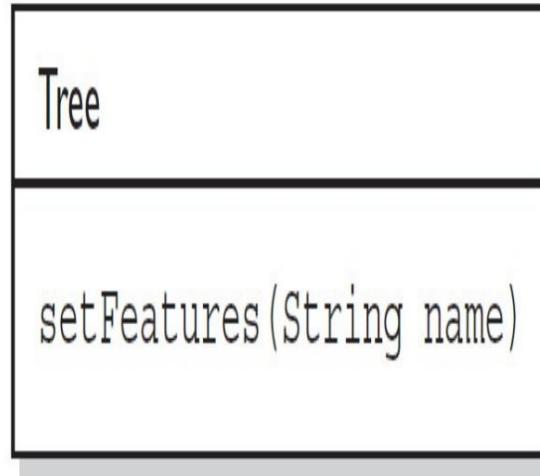
	Overloaded Method	Overridden Method
Argument(s)	Must change.	Must not change.
Return type	Can change.	Can't change except for covariant returns. (Covered later this chapter.)
Exceptions	Can change.	Can reduce or eliminate. Must not throw new or broader checked exceptions.
Access	Can change.	Must not make more restrictive (can be less restrictive).
Invocation	<i>Reference</i> type determines which overloaded version (based on declared argument types) is selected. Happens at <i>compile</i> time. The actual <i>method</i> that's invoked is still a virtual method invocation that happens at runtime, but the compiler will already know the <i>signature</i> of the method to be invoked. So at runtime, the argument match will already have been nailed down, just not the <i>class</i> in which the method lives.	<i>Object</i> type (in other words, <i>the type of the actual instance on the heap</i>) determines which method is selected. Happens at <i>runtime</i> .

Chúng tôi sẽ đề cập đến quá tải hàm tạo ở phần sau của chương, nơi chúng tôi cũng sẽ bao gồm các chủ đề liên quan đến hàm tạo khác có trong kỳ thi. [Hình 2-4](#) minh họa cách các phương thức được nạp chồng và ghi đè xuất hiện trong các mối quan hệ lớp.

Overriding



Overloading



HÌNH 2-4

Mô hình lớp và phương thức được ghi đè và ghi đè trong các mối quan hệ lớp

MỤC TIÊU XÁC NHẬN

Truyền (Mục tiêu OCA 2.2 và 7.3)

2.2 Phân biệt giữa biến tham chiếu đối tượng và biến nguyên thủy.

7.3 Xác định thời điểm cần thiết phải đúc.

Bạn đã thấy cách sử dụng tài liệu tham khảo chung vừa khả thi vừa phô biến kiểu biến để tham chiếu đến các kiểu đối tượng cụ thể hơn. Đó là trung tâm của tính đa hình. Ví dụ: dòng mã này bây giờ phải là bản chất thứ hai:

```
Thú vật = new Dog();
```

Nhưng điều gì sẽ xảy ra khi bạn muốn sử dụng biến tham chiếu động vật đó để gọi một phương thức mà chỉ lớp Dog mới có? Bạn biết nó đề cập đến một con chó và bạn muốn làm một điều cụ thể cho con chó? Trong đoạn mã sau, chúng ta có một mảng Động vật và bắt đầu khi nào chúng ta tìm thấy Chó trong mảng, chúng ta muốn làm một điều đặc biệt về Chó. Bây giờ chúng ta hãy đồng ý rằng tất cả mã này đều ổn, ngoại trừ việc chúng ta không chắc chắn về dòng mã gọi phương thức playDead .

```
class Animal {
    void makeNoise() {System.out.println("generic noise"); }
}
class Dog extends Animal {
    void makeNoise() {System.out.println("bark"); }
    void playDead() { System.out.println("roll over"); }
}

class CastTest2 {
    public static void main(String [] args) {
        Animal [] a = {new Animal(), new Dog(), new Animal() };
        for(Animal animal : a) {
            animal.makeNoise();
            if(animal instanceof Dog) {
                animal.playDead();           // try to do a Dog behavior?
            }
        }
    }
}
```

Khi chúng tôi cố gắng biên dịch mã này, trình biên dịch sẽ nói như sau:

không thể tìm thấy biểu tượng

Trình biên dịch nói, "Này, lớp Animal không có phương thức playDead () ." Hãy sửa đổi khỏi mã if :

```

if(animal instanceof Dog) {
    Dog d = (Dog) animal;           // casting the ref. var.
    d.playDead();
}

```

Khối mã mới và được cải tiến có chứa một lớp ép kiểu, trong trường hợp này đôi khi được gọi là khói lệnh downcast, vì chúng ta đang truyền cây thừa kế xuống một lớp cụ thể hơn. Bây giờ trình biên dịch là hạnh phúc. Trước khi có gắng gọi playDead, chúng ta ép biến biến động vật thành kiểu Dog. Những gì chúng tôi đang nói với trình biên dịch là, "Chúng tôi biết nó thực sự đề cập đến một đối tượng Dog , vì vậy bạn có thể tạo một biến tham chiếu Dog mới để tham chiếu đến đối tượng đó." Trong trường hợp này, chúng tôi an toàn, vì trước khi thử diễn viên, chúng tôi sẽ thực hiện kiểm tra phiên bản để đảm bảo.

Điều quan trọng cần biết là trình biên dịch buộc phải tin tưởng chúng tôi khi chúng tôi thực hiện ảm đạm, ngay cả khi chúng ta gặp khó khăn:

```

class Animal { }
class Dog extends Animal { }
class DogTest {
    public static void main(String [] args) {
        Animal animal = new Animal();
        Dog d = (Dog) animal;           // compiles but fails later
    }
}

```

Nó có thể làm bạn điên tiết! Mã này biên dịch! Nhưng khi chúng tôi cố gắng chạy nó, chúng tôi sẽ nhận được một ngoại lệ, giống như sau:

```
java.lang.ClassCastException
```

Tại sao chúng ta không thể tin tưởng trình biên dịch để giúp chúng ta ở đây? Nó không thể nhìn thấy con vật đó thuộc loại Động vật? Tất cả những gì mà trình biên dịch có thể làm là xác minh rằng hai loại nằm trong cùng một cây kế thừa, để tùy thuộc vào bất kỳ mã nào có thể có trước khi downcast, có thể con vật đó thuộc loại Dog. Trình biên dịch phải cho phép những thứ có thể hoạt động trong thời gian chạy. Tuy nhiên, nếu trình biên dịch biết chắc chắn rằng quá trình truyền không thể hoạt động, thì quá trình biên dịch sẽ thất bại.

Khối mã thay thế sau sẽ KHÔNG biên dịch:

```

Thú vật = new Animal ();
Dog d = (Con chó) động vật;
String s = (Chuỗi) động vật; // động vật không thể BAO GIỜ là một chuỗi

```

Trong trường hợp này, bạn sẽ gặp lỗi như sau:

các loại không thể nghĩ bàn

Không giống như dự báo xuống, dự báo lên (tạo cây kế thừa thành một loại chung) hoạt động ngầm (nghĩa là bạn không phải nhập kiểu) bởi vì khi bạn upcast, bạn đã ngầm hạn chế số lượng phương thức mà bạn có thể gọi, trái ngược với downcasting, điều này ngụ ý rằng sau này, bạn có thể muốn gọi một phương thức cụ thể hơn . Đây là một ví dụ:

```
class Animal { }
class Dog extends Animal { }

class DogTest {
    public static void main(String [] args) {
        Dog d = new Dog();
        Animal a1 = d;           // upcast ok with no explicit cast
        Animal a2 = (Animal) d; // upcast ok with an explicit cast
    }
}
```

Cả hai bản upcast trước đó sẽ biên dịch và chạy mà không có ngoại lệ vì Dog IS-A (n) Animal, có nghĩa là bất cứ điều gì một Động vật có thể làm, một Con chó có thể làm. Tuy nhiên, Chó có thể làm được nhiều việc hơn, nhưng vấn đề là bất kỳ ai có tham chiếu Động vật đều có thể gọi các phương thức Động vật trên một cá thể Chó một cách an toàn . Các phương thức Animal có thể đã bị ghi đè trong lớp Dog , nhưng tất cả những gì chúng ta quan tâm bây giờ là Chó luôn có thể làm ít nhất mọi thứ mà Động vật có thể làm. Trình biên dịch và JVM cũng biết điều đó, do đó, upcast ngầm luôn hợp pháp để gán một đối tượng của kiểu con cho một tham chiếu của một trong các lớp siêu kiểu (hoặc các giao diện) của nó. Nếu Dog thực hiện Pet và Pet định nghĩa beFriendly () , thì Dog có thể được truyền ngầm thành Pet, nhưng phương thức Dog duy nhất mà bạn có thể gọi sau đó là beFriendly () , mà Dog buộc phải thực hiện vì Dog triển khai giao diện Pet .

Một điều nữa. nếu Dog triển khai Pet, thì, nếu Beagle mở rộng Dog nhưng Beagle không tuyên bố rằng nó triển khai Pet, Beagle vẫn là Pet! Beagle là Pet đơn giản vì nó mở rộng tuổi thọ của Dog, và Dog đã chăm sóc các bộ phận của Pet cho chính nó và cho tất cả các con của nó. Lớp Beagle luôn có thể ghi đè bất kỳ phương thức nào mà nó kế thừa từ Dog, bao gồm các phương thức mà Dog đã triển khai để thực hiện hợp đồng giao diện của nó.

Và chỉ một điều nữa ... nếu Beagle tuyên bố rằng nó triển khai Pet, chỉ cần để những người khác nhìn vào API lớp Beagle có thể dễ dàng nhận thấy Beagle IS-A Pet mà không cần phải xem các lớp cha của Beagle, Beagle vẫn không cần triển khai phương thức beFriendly () nếu lớp Dog (lớp cha của Beagle) đã sử dụng quan tâm đến điều đó. Nói cách khác, nếu Beagle IS-A Dog và Dog IS-A Pet,

sau đó Beagle IS-A Pet và đã đáp ứng các nghĩa vụ Pet của nó để triển khai phương thức `beFriendly()` vì nó kế thừa phương thức `beFriendly()`. Trình biên dịch đủ thông minh để nói, "Tôi biết Beagle đã là một con chó, nhưng bạn có thể làm cho nó rõ ràng hơn bằng cách thêm một dàn diễn viên."

Vì vậy, đừng để bị lừa bởi mã hiển thị lớp cụ thể tuyên bố rằng nó triển khai một giao diện nhưng không triển khai các phương thức của giao diện.

Trước khi bạn có thể biết liệu mã có hợp pháp hay không, bạn phải biết các siêu kiểu của lớp triển khai này đã khai báo những gì. Nếu bất kỳ siêu kiểu nào trong cây kế thừa của nó đã cung cấp các triển khai phương thức cụ thể (nghĩa là `nonabstract`), thì bất kể siêu kiểu đó có tuyên bố rằng nó triển khai giao diện hay không, lớp con không có nghĩa vụ thực hiện lại (ghi đè) các phương thức đó.



Người tạo bài kiểm tra sẽ cho bạn biết rằng họ buộc phải nhét hàng tần mã vào những khoảng trống nhỏ "vì công cụ kiểm tra". Mặc dù điều đó đúng một phần, họ CŨNG thích làm xáo trộn. Đoạn mã sau

```
Animal a = new Dog();  
Dog d = (Dog) a;  
d.doDogStuff();
```

có thể được thay thế bằng một chút thú vị dễ đọc này:

```
Animal a = new Dog();  
((Dog) a).doDogStuff();
```

Trong trường hợp này, trình biên dịch cần tất cả các dấu ngoặc đơn đó; nếu không, nó nghĩ rằng nó đã được trao một tuyên bố không đầy đủ.

MỤC TIÊU XÁC NHẬN

Triển khai một giao diện (Mục tiêu 7.5 của OCA)

7.5 Sử dụng các lớp và giao diện trừu tượng.

Khi bạn triển khai một giao diện, bạn đồng ý tuân thủ hợp đồng được xác định trong giao diện. Điều đó có nghĩa là bạn đồng ý cung cấp các triển khai hợp pháp cho mọi phương thức trừu tượng được xác định trong giao diện và bất kỳ ai biết các phương thức giao diện trông như thế nào (không phải cách chúng được triển khai, mà là cách chúng có thể được gọi và chúng trả về) hãy yên tâm rằng họ có thể gọi các phương thức đó trên một phiên bản của lớp triển khai của bạn.

Ví dụ: nếu bạn tạo một lớp triển khai giao diện Runnable (vì vậy mã của bạn có thể được thực thi bởi một luồng cụ thể), bạn phải cung cấp phương thức public void run () . Nếu không, luồng nghèo có thể được yêu cầu thực thi mã đối tượng Runnable của bạn và – ngạc nhiên, ngạc nhiên – sau đó luồng phát hiện ra đối tượng không có phương thức run () ! (Tại thời điểm đó, sợi chỉ sẽ nổ tung và JVM sẽ sụp đổ trong một vụ nổ ngoạn mục nhưng khủng khiếp.) Rất may, Java ngăn chặn sự hỗn loạn này xảy ra bằng cách chạy kiểm tra trình biên dịch trên bất kỳ lớp nào yêu cầu triển khai một giao diện. Nếu lớp nói rằng nó đang triển khai một giao diện, thì tốt hơn hết là nên có một triển khai cho từng phương thức trừu tượng trong giao diện (với một vài ngoại lệ mà chúng ta sẽ xem xét trong giây lát).

Giả sử một giao diện Bounceable với hai phương thức, bounce () và setBounceFactor (), lớp sau sẽ biên dịch:

```
public class Ball implements Bounceable { // Keyword
                                         // 'implements'
    public void bounce() { }
    public void setBounceFactor(int bf) { }
}
```

Được rồi, chúng tôi biết bạn đang nghĩ gì: "Đây phải là lớp triển khai tồi tệ nhất trong lịch sử các lớp triển khai". Tuy nhiên, nó biên dịch. Và nó chạy. Hợp đồng giao diện đảm bảo rằng một lớp sẽ có phương thức (nói cách khác, những người khác có thể gọi phương thức là đối tượng để kiểm soát truy cập), nhưng nó không bao giờ đảm bảo việc triển khai tốt – hoặc thậm chí bất kỳ mã triển khai thực tế nào trong phần thân của phương thức. (Mặc dù vậy, hãy nhớ rằng nếu giao diện tuyên bố rằng một phương thức KHÔNG phải là vô hiệu, thì mã triển khai của lớp của bạn phải bao gồm một câu lệnh trả về.) Trình biên dịch sẽ không bao giờ nói, "Um, xin lỗi, nhưng bạn thực sự có ý định đặt không có gì giữa những dấu ngoặc nhọn? XIN CHÀO. Dù sao đây cũng là một phương pháp, như vậy không nên làm cái gì? "

Các lớp triển khai phải tuân thủ các quy tắc tương tự để thực hiện phương thức như một lớp mở rộng một lớp trừu tượng . Để trở thành một người hợp pháp

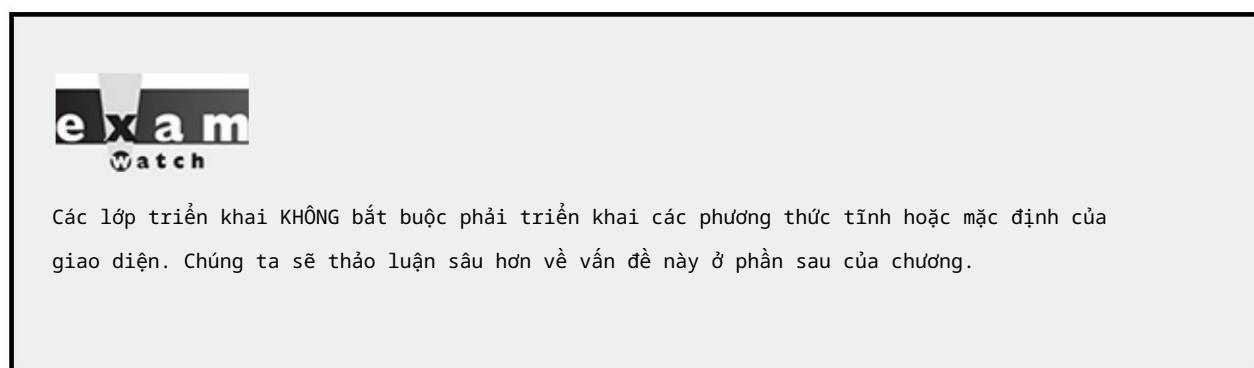
lớp thực thi, một lớp triển khai nonabstract phải làm như sau:

- Cung cấp các triển khai cụ thể (nonabstract) cho tất cả các phương thức trừu tượng từ giao diện đã khai báo.
- Thực hiện theo tất cả các quy tắc về ghi đè pháp lý, chẳng hạn như sau:
 - Khai báo không có ngoại lệ đã kiểm tra trên các phương thức triển khai khác với những ngoại lệ được khai báo bởi phương thức giao diện hoặc các lớp con của những ngoại lệ được khai báo bởi phương thức giao diện.
 - Duy trì chữ ký của phương thức giao diện và duy trì cùng một kiểu trả về (hoặc một kiểu con). (Nhưng nó không phải khai báo các ngoại lệ được khai báo trong khai báo phương thức giao diện.)

Nhưng xin chờ chút nữa! Bản thân một lớp thực thi có thể là trừu tượng! Vì ví dụ, điều sau là hợp pháp đối với một Bóng lớp triển khai Có thể trả lại:

`lớp trừu tượng Ball thực hiện Bounceable {}`

Nhận thấy điều gì còn thiếu? Chúng tôi không bao giờ cung cấp các phương pháp thực hiện. Và điều đó không sao. Nếu lớp thực thi là trừu tượng, nó có thể chỉ cần chuyển buck vào lớp con cụ thể đầu tiên của nó. Ví dụ: nếu lớp BeachBall mở rộng Ball và BeachBall không trừu tượng, thì BeachBall phải cung cấp một triển khai cho tất cả các phương thức trừu tượng từ Bounceable:



```

class BeachBall extends Ball {
    // Even though we don't say it in the class declaration above,
    // BeachBall implements Bounceable, since BeachBall's abstract
    // superclass (Ball) implements Bounceable

    public void bounce() {
        // interesting BeachBall-specific bounce code

    }
    public void setBounceFactor(int bf) {
        // clever BeachBall-specific code for setting
        // a bounce factor

    }

    // if class Ball defined any abstract methods,
    // they'll have to be
    // implemented here as well.
}

```

Tìm kiếm các lớp yêu cầu triển khai một giao diện nhưng không cung cấp các triển khai phương thức chính xác. Trừ khi lớp thực thi là trừu tượng, lớp thực hiện phải cung cấp các triển khai cho tất cả các phương thức trừu tượng được định nghĩa trong giao diện.

Bạn cần biết thêm hai quy tắc nữa, và sau đó chúng ta có thể đưa chủ đề này vào chế độ ngủ (hoặc đưa bạn vào giấc ngủ; chúng tôi luôn nhầm lẫn hai điều đó):

1. Một lớp có thể triển khai nhiều hơn một giao diện. Hoàn toàn hợp pháp khi nói, ví dụ như sau:

lớp công khai Ball triển khai Có thể trả lại, Có thể nối tiếp, Có thể chạy
được {...}

Bạn chỉ có thể mở rộng một lớp, nhưng bạn có thể triển khai nhiều giao diện (đối với Java 8, có nghĩa là một dạng đa kế thừa, mà chúng ta sẽ thảo luận ngay sau đây). Nói cách khác, lớp con xác định bạn là ai và bạn là gì, trong khi việc triển khai xác định vai trò bạn có thể đóng hoặc chiếc mũ bạn có thể đội, mặc dù bạn có thể khác với một số lớp khác triển khai cùng một giao diện như thế nào (nhưng từ một cây kế thừa khác). Ví dụ, một Person mở rộng HumanBeing (mặc dù đối với một số người, điều đó còn gây tranh cãi). Nhưng một Người cũng có thể triển khai Lập trình viên, Người trượt ván, Nhân viên, Phụ huynh hoặc NgườiCrazyEnoughToTakeThisExam.

2. Bản thân một giao diện có thể mở rộng một giao diện khác. Đoạn mã sau hoàn toàn hợp pháp:

```
giao diện công cộng Có thể trả lại mở rộng Có thể di chuyển {} // ok!
```

Điều đó nghĩa là gì? Lớp triển khai cụ thể (nonabstract) đầu tiên của Bounceable phải triển khai tất cả các phương thức trừu tượng của Bounceable, cộng với tất cả các phương thức trừu tượng của Moveable! Giao diện con, như chúng ta gọi, chỉ đơn giản là thêm nhiều yêu cầu hơn vào hợp đồng của giao diện siêu cấp. Bạn sẽ thấy khái niệm này được áp dụng trong nhiều lĩnh vực của Java, đặc biệt là Java EE, nơi bạn thường phải xây dựng giao diện của riêng mình để mở rộng một trong các giao diện Java EE.

Tuy nhiên, hãy chờ đợi, bởi vì đây là nơi mà nó trở nên kỳ lạ. Một giao diện có thể mở rộng nhiều hơn một giao diện! Hãy suy nghĩ về điều đó trong một thời điểm. Bạn biết rằng khi chúng ta đang nói về các lớp học, những điều sau đây là bất hợp pháp:

```
public class Lập trình viên mở rộng Employee, Geek {} // Bất hợp pháp!
```

Như chúng tôi đã đề cập trước đó, một lớp không được phép mở rộng nhiều lớp trong Java. Tuy nhiên, một giao diện miễn phí để mở rộng nhiều giao diện:

```
interface Bounceable extends Moveable, Spherical {    // ok!
    void bounce();
    void setBounceFactor(int bf);
}
interface Moveable {
    void moveIt();
}
interface Spherical {
    void doSphericalThing();
}
```

Trong ví dụ tiếp theo, Ball được yêu cầu triển khai Bounceable, cộng với tất cả các phương thức trừu tượng từ các giao diện mà Bounceable mở rộng (bao gồm bất kỳ giao diện nào mà các giao diện đó mở rộng, v.v., cho đến khi bạn đạt đến đầu ngã rẽ – hoặc là cuối cây rơm?). Vì vậy, Ball sẽ cần phải trông giống như sau:

```

class Ball implements Bounceable {

    public void bounce() { }           // Implement Bounceable's methods
    public void setBounceFactor(int bf) { }

    public void moveIt() { }           // Implement Moveable's method

    public void doSphericalThing() { }  // Implement Spherical
}

```

Nếu lớp Ball không thực hiện được bất kỳ phương thức trừu tượng nào từ Bounceable, Moveable hoặc Spherical, trình biên dịch sẽ nhảy lên và xuống dữ dội, mặt đỏ rực, cho đến khi nó thực hiện được. Trừ khi, lớp Ball được đánh dấu là trừu tượng. Trong trường hợp đó, Ball có thể chọn triển khai bất kỳ, tất cả hoặc không có phương thức trừu tượng nào từ bất kỳ giao diện nào, do đó để phần triển khai còn lại cho một lớp con cụ thể của Ball, như sau:

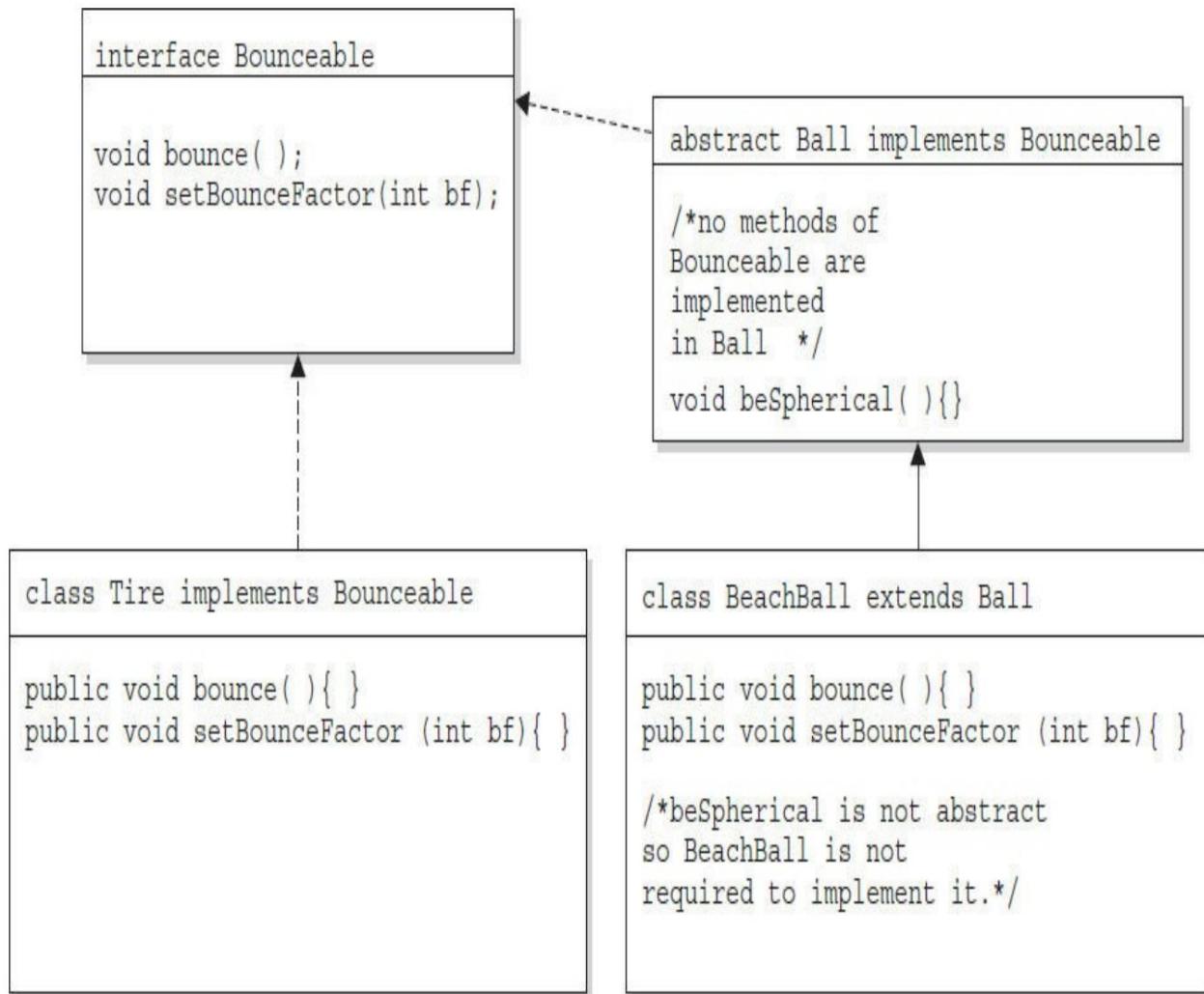
```

abstract class Ball implements Bounceable {
    public void bounce() { ... }      // Define bounce behavior
    public void setBounceFactor(int bf) { ... }
    // Don't implement the rest; leave it for a subclass
}
class SoccerBall extends Ball {    // class SoccerBall must
                                    // implement the interface
                                    // methods that Ball didn't

    public void moveIt() { ... }
    public void doSphericalThing() { ... }
    // SoccerBall can choose to override the Bounceable methods
    // implemented by Ball
    public void bounce() { ... }
}

```

[Hình 2-5](#) so sánh các ví dụ cụ thể và trừu tượng về mở rộng và triển khai cho cả lớp và giao diện.



Because BeachBall is the first concrete class to implement Bounceable, it must provide implementations for all methods of Bounceable, except those defined in the abstract class Ball. Because Ball did not provide implementations of Bounceable methods, BeachBall was required to implement all of them.

HÌNH 2-5 Mô hình các ví dụ cụ thể và trừu tượng về mở rộng và thực hiện

Java 8 – Hiện có Nhiều Kế thừa!

Bạn có thể đã nhận ra rằng vì các giao diện giờ đây có thể có các phương thức cụ thể và các lớp có thể triển khai nhiều giao diện, nên bóng ma của sự đa thừa kế và Viên kim cương chết chóc có thể tạo nên cái đầu xấu xí của nó!

đa thừa kế và Viên kim cương chét chóc có thể nuôi dưỡng cái đầu xấu xí của nó! Chà, bạn đúng một phần. Một lớp CÓ THỂ triển khai các giao diện với các chữ ký phương thức trùng lặp, cụ thể! Nhưng tin tốt là trình biên dịch đã hỗ trợ bạn, và nếu bạn NÊN triển khai cả hai giao diện, bạn sẽ phải cung cấp một phương thức ghi đè trong lớp của mình. Hãy xem đoạn mã sau:



Tìm kiếm việc sử dụng bất hợp pháp các phần mở rộng và phần thực hiện. Sau đây là các ví dụ về khai báo giao diện và lớp hợp pháp và bất hợp pháp:

```

class Foo { }                                // OK
class Bar implements Foo { }                 // No! Can't implement a class
interface Baz { }                           // OK
interface Fi { }                            // OK
interface Fee implements Baz { }           // No! an interface can't
                                            // implement an interface
interface Zee implements Foo { }            // No! an interface can't
                                            // implement a class
interface Zoo extends Foo { }               // No! an interface can't
                                            // extend a class
interface Boo extends Fi { }                // OK. An interface can extend
                                            // an interface
class Toon extends Foo, Button { }          // No! a class can't extend
                                            // multiple classes
class Zoom implements Fi, Baz { }           // OK. A class can implement
                                            // multiple interfaces
interface Vroom extends Fi, Baz { }         // OK. An interface can extend
                                            // multiple interfaces
class Yow extends Foo implements Fi { }      // OK. A class can do both
                                            // (extends must be 1st)
class Yow extends Foo implements Fi, Baz { } // OK. A class can do all three
                                            // (extends must be 1st)

```

Hãy ghi những thứ này vào và để ý xem có sự lạm dụng trong các câu hỏi bạn nhận được trong bài kiểm tra hay không. Bất kể câu hỏi dường như đang kiểm tra điều gì, vấn đề thực sự có thể là khai báo lớp hoặc giao diện. Trước khi bạn bị cuốn vào, chẳng hạn như lần theo một luồng phức tạp, hãy kiểm tra xem liệu mã có được biên dịch hay không. (Chỉ mèo đó thôi cũng có thể đáng để bạn đặt chúng tôi vào ý muốn của bạn!) (Bạn sẽ bị ấn tượng bởi nỗ lực mà các nhà phát triển kỳ thi đã bỏ ra để đánh lạc hướng bạn khỏi vấn đề thực sự.) ?)

```

interface I1 {
    default int doStuff() { return 1; }
}
interface I2 {
    default int doStuff() { return 2; }
}

public class MultiInt implements I1, I2 { // needs to override
    public static void main(String[] args) {
        new MultiInt().go();
    }
    void go() {
        System.out.println(doStuff());
    }
    // public int doStuff() {
    //     return 3;
    // }
}

```

Như là viết tắt của mã, nó sẽ KHÔNG SỐ SÁNH vì không rõ phiên bản doStuff () nào nên được sử dụng. Để biên dịch mã, bạn cần ghi đè doStuff () trong lớp. Bỏ chú thích phương thức doStuff () của lớp sẽ cho phép mã biên dịch và khi chạy sẽ tạo ra kết quả sau:

3

MỤC TIÊU XÁC NHẬN

Các hình thức hoàn trả hợp pháp (Mục tiêu 2.2 và 6.1 của OCA)

2.2 Phân biệt giữa biến tham chiếu đối tượng và biến nguyên thủy.

6.1 Tạo phương thức với đối số và giá trị trả về; bao gồm các phương thức được nạp chồng.

Phần này bao gồm hai khía cạnh của các loại trả lại: những gì bạn có thể khai báo dưới dạng kiểu trả về và những gì bạn thực sự có thể trả về dưới dạng giá trị. Những gì bạn có thể và không thể khai báo khá đơn giản, nhưng tất cả phụ thuộc vào việc bạn đang ghi đè một phương thức kế thừa hay chỉ đơn giản là khai báo một phương thức mới (bao gồm các phương thức được nạp chồng). Chúng tôi sẽ chỉ xem xét nhanh sự khác biệt

bao gồm các phương thức nạp chồng). Chúng ta sẽ chỉ xem xét nhanh sự khác biệt giữa các quy tắc kiểu trả về cho các phương thức nạp chồng và ghi đè, bởi vì chúng ta đã đề cập đến vấn đề đó trong chương này. Tuy nhiên, chúng tôi sẽ đề cập đến một chút cơ sở mới khi chúng tôi xem xét các kiểu trả về đa hình và các quy tắc cho những gì được và không hợp pháp để thực sự trả về.

Khai báo loại trả lại

Phần này xem xét những gì bạn được phép khai báo dưới dạng kiểu trả về, điều này phụ thuộc chủ yếu vào việc bạn đang ghi đè, nạp chồng hay khai báo một phương thức mới.

Các kiểu trả về trên các phương thức bị quá tải Hãy

nhớ rằng quá tải phương thức không nhiều hơn việc sử dụng lại tên. Phương thức nạp chồng là một phương thức hoàn toàn khác với bất kỳ phương thức cùng tên nào khác. Vì vậy, nếu bạn kế thừa một phương thức nhưng nạp chồng nó trong một kiểu con, bạn không phải chịu các hạn chế của việc ghi đè, có nghĩa là bạn có thể khai báo bất kỳ kiểu trả về nào mà bạn thích. Những gì bạn không thể làm là chỉ thay đổi kiểu trả về. Để nạp chồng cho một phương thức, hãy nhớ rằng, bạn phải thay đổi danh sách đối số. Đoạn mã sau cho thấy một phương thức được nạp chồng:

```
public class Foo{
    void go() { }
}
public class Bar extends Foo {
    String go(int x) {
        return null;
    }
}
```

Lưu ý rằng phiên bản Bar của phương thức sử dụng kiểu trả về khác. Điều đó hoàn toàn tốt. Miễn là bạn đã thay đổi danh sách đối số, bạn đang nạp chồng phương thức, vì vậy kiểu trả về không cần phải khớp với kiểu của phiên bản supertype. Điều bạn KHÔNG được phép làm là:

```

public class Foo{
    void go() { }
}
public class Bar extends Foo {
    String go() { // Not legal! Can't change only the return type
        return null;
    }
}

```

Ghi đè và trả về các kiểu và trả về cùng phương sai Khi một kiểu con muốn thay đổi việc triển khai phương thức của một phương thức kế thừa (ghi đè), kiểu con phải xác định một phương thức khớp chính xác với phiên bản được kế thừa. Hoặc, kể từ Java 5, bạn được phép thay đổi kiểu trả về trong phương thức ghi đè miễn là kiểu trả về mới là kiểu con của kiểu trả về đã khai báo của phương thức ghi đè (lớp cha).

Hãy xem xét lợi tức hiệp biến trong hành động:

```

class Alpha {
    Alpha doStuff(char c) {
        return new Alpha();
    }
}
class Beta extends Alpha {
    Beta doStuff(char c) { // legal override since Java 1.5
        return new Beta();
    }
}

```

Kể từ Java 5, mã này biên dịch. Nếu bạn cố gắng biên dịch mã này bằng trình biên dịch 1.4 hoặc với cờ nguồn như sau,

```
javac -source 1.4 Beta.java
```

bạn sẽ gặp lỗi trình biên dịch như thế này:

cố gắng sử dụng loại trả lại không tương thích

Các quy tắc khác áp dụng cho việc ghi đè, bao gồm các quy tắc dành cho công cụ sửa đổi quyền truy cập và các ngoại lệ đã khai báo, nhưng các quy tắc đó không liên quan đến thảo luận kiểu trả về.



Đối với bài kiểm tra, hãy chắc chắn rằng bạn biết rằng các phương thức nạp chồng có thể thay đổi kiểu trả về, nhưng các phương thức ghi đè chỉ có thể làm như vậy trong giới hạn của các phương thức trả về hiệp phương sai. Chỉ những kiến thức đó thôi cũng sẽ giúp bạn vượt qua hàng loạt các đề thi.

Trả lại giá trị

Bạn chỉ phải nhớ sáu quy tắc để trả về một giá trị:

1. Bạn có thể trả về null trong một phương thức có kiểu trả về tham chiếu đối tượng.

```
public Button doStuff() {
    return null;
}
```

2. Mảng là kiểu trả về hoàn toàn hợp pháp.

```
public String[] go() {
    return new String[] {"Fred", "Barney", "Wilma"};
```

3. Trong một phương thức có kiểu trả về nguyên thủy, bạn có thể trả về bất kỳ giá trị hoặc biến nào có thể được chuyển đổi hoàn toàn thành kiểu trả về đã khai báo.

```
public int foo() {
    char c = 'c';
    return c; // char is compatible with int
}
```

4. Trong một phương thức có kiểu trả về nguyên thủy, bạn có thể trả về bất kỳ giá trị hoặc biến nào có thể được ép kiểu rõ ràng thành kiểu trả về đã khai báo.

```
public int foo() {
    float f = 32.5f;
    return (int) f;
}
```

5. Bạn không được trả về bất cứ thứ gì từ một phương thức có kiểu trả về void.

```
public void bar() {
    return "this is it"; // Not legal!!
}
```

6. Trong phương thức có kiểu trả về tham chiếu đối tượng, bạn có thể trả về bất kỳ kiểu đối tượng nào có thể được ép kiểu ngầm thành kiểu trả về đã khai báo.

```
public Animal getAnimal() {
    return new Horse(); // Assume Horse extends Animal
}

public Object getObject() {
    int[] nums = {1,2,3};
    return nums;          // Return an int array, which is still an object
}

public interface Chewable { }
public class Gum implements Chewable { }

public class TestChewable {
    // Method with an interface return type
    public Chewable getChewable() {
        return new Gum(); // Return interface implementer
    }
}
```



Theo dõi các phương thức khai báo kiểu trả về lớp trừu tượng hoặc giao diện và biết rằng bất kỳ đối tượng nào vượt qua kiểm tra IS-A (nói cách khác, sẽ kiểm tra true bằng cách sử dụng toán tử instanceof) đều có thể được trả về từ phương thức đó. Ví dụ:

```

public abstract class Animal { }
public class Bear extends Animal { }
public class Test {
    public Animal go() {
        return new Bear(); // OK, Bear "is-a" Animal
    }
}

```

Mã này sẽ biên dịch và giá trị trả về là một kiểu con.

MỤC TIÊU XÁC NHẬN

Trình xây dựng và Trình tạo (Mục tiêu OCA 6.3 và 7.4)

6.3 Tạo và nạp chồng các hàm tạo; bao gồm tác động đến các trình tạo mặc định (sic)

7.4 Sử dụng super và this để truy cập các đối tượng và hàm tạo.

Các đối tượng được xây dựng. Bạn KHÔNG THỂ tạo một đối tượng mới mà không gọi một phương thức khởi tạo. Trên thực tế, bạn không thể tạo một đối tượng mới mà không gọi không chỉ hàm tạo của kiểu lớp thực tế của đối tượng mà còn cả hàm tạo của mỗi lớp cha của nó! Bộ tạo là mã chạy bất cứ khi nào bạn sử dụng từ khóa mới. (Được rồi, chính xác hơn một chút, cũng có thể có các khôi khởi tạo chạy khi bạn nói mới và chúng ta sẽ đề cập đến các khôi init và các đối tác khởi tạo tĩnh của chúng sau khi chúng ta thảo luận về các hàm tạo.) về đây – chúng ta sẽ xem xét cách các hàm tạo được mã hóa, ai mã hóa chúng và cách chúng hoạt động trong thời gian chạy. Vì vậy, lấy hardhat của bạn và một cái búa, và chúng ta hãy xây dựng một số đối tượng.

Kiến thức cơ bản về xây dựng

Mọi lớp, kể cả các lớp trừu tượng, PHẢI có một hàm tạo. Ghi điều đó vào bộ não của bạn. Nhưng chỉ vì một lớp phải có một hàm tạo không có nghĩa là lập trình viên phải gõ nó. Một hàm tạo trông như thế này:

```
lớp học Foo {
```

```
    Foo () {} // Hàm tạo cho lớp Foo
}
```

Chú ý những gì còn thiếu? Không có loại trả lại! Hai điểm chính cần nhớ về các hàm tạo là chúng không có kiểu trả về và tên của chúng phải khớp chính xác với tên lớp. Thông thường, các hàm tạo được sử dụng để khởi tạo trạng thái biến cá thể, như sau:

```
class Foo {
    int size;
    String name;
    Foo(String name, int size) {
        this.name = name;
        this.size = size;
    }
}
```

Trong ví dụ mã trước, lớp Foo không có no-arg người xây dựng. Điều đó có nghĩa là phần sau sẽ không biên dịch được:

`Foo f = new Foo (); // Sẽ không biên dịch, không có hàm tạo phù hợp`
nhưng phần sau sẽ biên dịch:

```
Foo f = new Foo("Fred", 43); // No problem. Arguments match
// the Foo constructor.
```

Vì vậy, việc một lớp có một hàm tạo không đối số là rất phổ biến (và mong muốn), bất kể có bao nhiêu hàm tạo được nạp chồng khác trong lớp (vâng, các hàm tạo có thể được nạp chồng). Không phải lúc nào bạn cũng có thể làm cho điều đó hoạt động cho các lớp học của mình; đôi khi bạn có một lớp mà việc tạo một thẻ hiện mà không cung cấp thông tin cho hàm tạo là vô nghĩa. Ví dụ, một đối tượng `java.awt.Color` không thể được tạo bằng cách gọi một hàm tạo no-arg, bởi vì điều đó giống như nói với JVM, "Hãy tạo cho tôi một đối tượng Màu mới và tôi thực sự không quan tâm đến điều gì màu sắc nó là. bạn chọn." Bạn có thực sự muốn JVM đưa ra quyết định về phong cách của bạn không?

Constructor Chaining

Chúng tôi biết rằng các hàm tạo được gọi trong thời gian chạy khi bạn nói mới trên một số loại lớp như sau:

```
Horse h = new Horse();
```

Nhưng điều gì thực sự xảy ra khi bạn nói `new Horse ()`? (Giả sử Ngựa mở rộng Động vật và Động vật mở rộng Đối tượng.)

1. Hàm tạo Horse được gọi. Mọi hàm tạo đều gọi phương thức khởi tạo của lớp cha của nó với một cuộc gọi (ngầm định) đến `super ()`, trừ khi phương thức khởi tạo gọi một phương thức khởi tạo được nạp chồng của cùng một lớp (nhiều hơn nữa trong một phút).
2. Hàm tạo Động vật được gọi (Động vật là lớp cha của Ngựa).
3. Hàm tạo đối tượng được gọi (Đối tượng là lớp cha cuối cùng của tất cả các lớp, vì vậy lớp Animal mở rộng Đối tượng ngay cả khi bạn không thực sự nhập “expand Object” vào khai báo lớp Animal ; đó là ẩn.) Tại thời điểm này, chúng ta trên đầu ngăn xếp.
4. Nếu class Object có bất kỳ biến thể hiện nào, thì chúng sẽ được cung cấp các giá trị rõ ràng của chúng. Theo giá trị rõ ràng , chúng tôi có nghĩa là các giá trị được chỉ định tại thời điểm các biến được khai báo, chẳng hạn như `int x = 27`, trong đó 27 là giá trị rõ ràng (trái ngược với giá trị mặc định) của biến cá thể.
5. Phương thức khởi tạo đối tượng hoàn thành.
6. Các biến instance của Animal được cung cấp các giá trị rõ ràng của chúng (nếu có).
7. Hàm tạo Animal hoàn tất.
8. Các biến cá thể Horse được cung cấp các giá trị rõ ràng của chúng (nếu có).
9. Bộ xây dựng Horse hoàn thành.

[Hình 2-6](#) cho thấy cách các hàm tạo hoạt động trên ngăn xếp cuộc gọi.



HÌNH 2-6 Các trúc trên ngăn xếp cuộc gọi

Quy tắc dành cho người xây dựng

Danh sách sau đây tóm tắt các quy tắc bạn cần biết cho kỳ thi (và để hiểu phần còn lại của phần này). Bạn PHẢI ghi nhớ những điều này, vì vậy hãy nhớ nghiên cứu chúng nhiều hơn một lần.

- Trình tạo có thể sử dụng bất kỳ công cụ sửa đổi quyền truy cập nào, bao gồm cả private. (Một phương thức khởi tạo riêng có nghĩa là chỉ mã bên trong chính lớp đó mới có thể khởi tạo một đối tượng thuộc kiểu đó, vì vậy nếu lớp khởi tạo riêng muốn cho phép một thẻ hiện của lớp được sử dụng, thì lớp đó phải cung cấp một phương thức tĩnh hoặc biến cho phép truy cập vào một cá thể được tạo từ bên trong lớp.)
- Tên phương thức khởi tạo phải khớp với tên của lớp.
- Trình tạo không được có kiểu trả về.
- Thật hợp pháp (nhưng thật ngu ngốc) khi có một phương thức trùng tên với lớp, nhưng điều đó không làm cho nó trở thành một phương thức khởi tạo. Nếu bạn thấy kiểu trả về, đó là một phương thức chứ không phải là một phương thức khởi tạo. Trên thực tế, bạn có thể có cả một phương thức và một phương thức khởi tạo có cùng tên – tên của lớp – trong cùng một lớp và đó không phải là vấn đề đối với Java. Hãy cẩn thận để không nhầm một phương thức với một phương thức khởi tạo – hãy đảm bảo tìm kiếm kiểu trả về.
- Nếu bạn không nhập một hàm tạo vào mã lớp của mình, một hàm tạo mặc định sẽ được trình biên dịch tạo tự động.
- Hàm tạo mặc định LUÔN LUÔN là một hàm tạo không đối số.
- Nếu bạn muốn một hàm tạo no-arg và bạn đã nhập bất kỳ (các) hàm tạo nào khác vào mã lớp của mình, trình biên dịch sẽ không cung cấp hàm tạo no-arg (hoặc bất kỳ hàm tạo nào khác) cho bạn. Nói cách khác, nếu bạn đã nhập một hàm tạo với các đối số, bạn sẽ không có một hàm tạo no-arg trừ khi bạn tự nhập nó vào chính mình!
- Mọi hàm tạo đều có, như là câu lệnh đầu tiên của nó, hoặc là một lệnh gọi đến một hàm tạo được nạp chồng (this ()) hoặc một lệnh gọi đến hàm tạo lớp cha (super()), mặc dù hãy nhớ rằng lời gọi này có thể được chèn bởi trình biên dịch.
- Nếu bạn nhập một hàm tạo (trái ngược với việc dựa vào hàm tạo mặc định do trình biên dịch tạo ra) và bạn không nhập lệnh gọi tới super () hoặc lệnh gọi this (), trình biên dịch sẽ chèn một lệnh gọi no-arg tới super () đối với bạn là câu lệnh đầu tiên trong hàm tạo.
- Một lệnh gọi tới super () có thể là một lệnh gọi no-arg hoặc có thể bao gồm các đối số được truyền cho hàm tạo siêu.

- Một phương thức khởi tạo no-arg không nhất thiết phải là phương thức khởi tạo mặc định (nghĩa là do trình biên dịch cung cấp), mặc dù phương thức khởi tạo mặc định luôn là một phương thức khởi tạo no-arg. Hàm tạo mặc định là hàm mà trình biên dịch cung cấp! Mặc dù hàm tạo mặc định luôn là hàm tạo không đối số, bạn có thể tự do đưa vào hàm tạo không đối số của riêng mình.
- Bạn không thể thực hiện cuộc gọi đến một phương thức cá thể hoặc truy cập một biến cá thể cho đến sau khi hàm tạo siêu chạy.
- Chỉ các biến và phương thức tĩnh mới có thể được truy cập như một phần của lệnh gọi tới super () hoặc this (). (Ví dụ: super (Animal.NAME) là OK, vì NAME được khai báo là một biến tĩnh.)
- Các lớp trừu tượng có các hàm tạo và các hàm tạo đó luôn được gọi khi một lớp con cụ thể được khởi tạo.
- Các giao diện không có hàm tạo. Các giao diện không phải là một phần của cây kế thừa của một đối tượng.
- Cách duy nhất một phương thức khởi tạo có thể được gọi là từ bên trong một phương thức khởi tạo khác. Nói cách khác, bạn không thể viết mã thực sự gọi một hàm tạo như sau:

```
class Horse {
    Horse() { } // constructor
    void doStuff() {
        Horse(); // calling the constructor - illegal!
    }
}
```

Xác định xem một hàm tạo mặc định sẽ là Tạo

Ví dụ sau cho thấy một lớp Horse có hai hàm tạo:

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

Trình biên dịch có đặt một hàm tạo mặc định cho lớp này không? Không!

Làm thế nào về biến thể sau đây của lớp?

```
hạng Ngựa {  
    Ngựa (Tên chuỗi) {}  
}
```

Bây giờ trình biên dịch sẽ chèn một hàm tạo mặc định? Không!

Còn lớp này thì sao?

```
hạng Ngựa {}
```

Bây giờ chúng ta nói chuyện. Trình biên dịch sẽ tạo ra một phương thức khởi tạo mặc định

cho lớp này vì lớp này không có bất kỳ phương thức khởi tạo nào được xác định.

Được rồi, còn lớp này?

```
hạng Ngựa {void  
    Horse () {}  
}
```

Có vẻ như trình biên dịch sẽ không tạo một phương thức khởi tạo, vì nó đã nằm trong lớp Horse . Hoặc là nó? Hãy nhìn lại lớp Horse trước đó .

Có gì sai với hàm tạo Horse () ? Nó hoàn toàn không phải là một hàm tạo! Nó chỉ đơn giản là một phương thức có cùng tên với lớp. Hãy nhớ rằng, kiểu trả lại là một món quà chết mà chúng ta đang xem xét một phương pháp, không phải người xây dựng.

Làm thế nào để bạn biết chắc chắn liệu một hàm tạo mặc định sẽ được tạo ra hay không?

Bởi vì bạn đã không viết bất kỳ hàm tạo nào trong lớp của mình.

Làm thế nào để bạn biết hàm tạo mặc định sẽ trông như thế nào?

Tại vì...

- Hàm tạo mặc định có cùng một công cụ sửa đổi quyền truy cập như lớp.
- Hàm tạo mặc định không có đối số.
- Hàm tạo mặc định bao gồm một lệnh gọi no-arg tới hàm tạo siêu (super ()) .

[Bảng 2-5](#) cho thấy những gì trình biên dịch sẽ (hoặc không) tạo ra cho lớp của bạn.

BẢNG 2-5 Mô hình ứng dụng do trình biên dịch tạo

Class Code (What You Type)	Compiler-Generated Constructor Code (in Bold)
class Foo {}	class Foo { Foo() { super() ; } }
class Foo { Foo() {} }	class Foo { Foo() { super() ; } }
public class Foo {}	public class Foo { public Foo() { super() ; } }
class Foo { Foo(String s) {} }	class Foo { Foo(String s) { super() ; } }
class Foo { Foo(String s) { super() ; } }	<i>Nothing; compiler doesn't need to insert anything.</i>
class Foo { void Foo() {} }	class Foo { void Foo() {} Foo() { super() ; } } (void Foo() is a method, not a constructor.)

Điều gì xảy ra nếu hàm tạo siêu có đối số? Các hàm tạo có thể có các đối số giống như các phương thức có thể, và nếu bạn cố gắng gọi một phương thức nhận, chẳng hạn, một int, nhưng bạn không truyền bất cứ thứ gì cho phương thức, trình biên dịch sẽ khiếu nại như sau:

```
class Bar {
    void takeInt(int x) { }
}

class UseBar {
    public static void main (String [] args) {
        Bar b = new Bar();
        b.takeInt(); // Try to invoke a no-arg takeInt() method
    }
}
```

Trình biên dịch sẽ phản nàn rằng bạn không thể gọi takeInt () mà không chuyển một số nguyên. Tất nhiên, trình biên dịch thích câu đó, vì vậy thông báo mà nó phát ra trên một số phiên bản của JVM (số dặm của bạn có thể thay đổi) ít rõ ràng hơn:

```
UseBar.java:7: takeInt (int) trong Bar không thể áp dụng cho () b.takeInt ();
^
```

Như ng bạn hiểu ý rồi đây. Điểm mấu chốt là phải có sự phù hợp với phương pháp. Và đối sánh, chúng tôi có nghĩa là các loại đối số phải có thể chấp nhận các giá trị hoặc biến bạn đang chuyển và theo thứ tự bạn đang chuyển chúng. Điều này đưa chúng ta trở lại các hàm tạo (và ở đây bạn đã nghĩ rằng chúng ta sẽ không bao giờ đạt được điều đó), hoạt động giống hệt như vậy.

Vì vậy, nếu hàm tạo siêu của bạn (nghĩa là, hàm tạo của superclass / parent) có các đối số, bạn phải nhập lệnh gọi tới super (), cung cấp các đối số thích hợp. Điểm quan trọng: nếu lớp cha của bạn không có hàm tạo no-arg, bạn phải nhập một hàm tạo trong lớp của mình (lớp con) vì bạn cần một nơi để đặt lệnh gọi tới super () với các đối số thích hợp.

Sau đây là một ví dụ về vấn đề:

```

class Animal {
    Animal(String name) { }
}

class Horse extends Animal {
    Horse() {
        super(); // Problem!
    }
}

```

Và một lần nữa trình biên dịch đối xử với chúng ta bằng sự sáng suốt tuyệt vời:

```

Horse.java:7: cannot resolve symbol
symbol : constructor Animal()
location: class Animal
super(); // Problem!
^

```

Nếu bạn may mắn (và đó là trăng tròn), trình biên dịch của bạn có thể rõ ràng hơn một chút.

Nhưng một lần nữa, vấn đề là không có sự phù hợp với những gì chúng ta đang cố gọi với `super()` - một hàm tạo `Animal` không có đối số.

Một cách khác để đặt vấn đề này là nếu lớp cha của bạn không có phưƠng thức khởi tạo, thì trong lớp con của bạn, bạn sẽ không thể sử dụng phưƠng thức khởi tạo mặc định do trình biên dịch cung cấp. Nó đơn giản mà. Bởi vì trình biên dịch chỉ có thể đặt một lệnh gọi tới no-arg `super()`, bạn thậm chí sẽ không thể biên dịch một cái gì đó như thế này:

```

class Clothing {
    Clothing(String s) { }
}
class TShirt extends Clothing { }

```

Việc cố gắng biên dịch mã này mang lại cho chúng tôi chính xác lỗi mà chúng tôi gặp phải khi đặt một hàm tạo trong lớp con với một lệnh gọi đến phiên bản no-arg của `super()`:

```

Clothing.java:4: cannot resolve symbol
symbol : constructor Clothing()
location: class Clothing
class TShirt extends Clothing { }
^

```

Trên thực tế, mã `Clothing` và `TShirt` trứ ơc đó hoàn toàn giống với mã sau, trong đó chúng tôi đã cung cấp một hàm tạo cho `TShirt` giống với hàm tạo mặc định được cung cấp bởi trình biên dịch:

```

class Clothing {
    Clothing(String s) { }
}
class TShirt extends Clothing {
    // Constructor identical to compiler-supplied
    // default constructor
    TShirt() {
        super(); // Won't work!
    } // tries to invoke a no-arg Clothing constructor
    // but there isn't one
}

```

Một điểm cuối cùng về toàn bộ hàm tạo mặc định (và nó có thể rất hiển nhiên, như ng chúng ta phải nói điều đó nếu không chúng ta sẽ cảm thấy tội lỗi trong nhiều năm), các hàm tạo không bao giờ được ợc kế thừa. Chúng không phải là phư ơng pháp. Không thể ghi đè chúng (vì chúng không phải là phư ơng thức và chỉ có thể ghi đè các phư ơng thức phiên bản). Vì vậy, loại (các) phư ơng thức khởi tạo mà lớp cha của bạn không có cách nào xác định loại phư ơng thức khởi tạo mặc định mà bạn sẽ nhận đư ợc. Một số ngư ời nhầm tư ờng rằng hàm tạo mặc định bằng cách nào đó khớp với hàm tạo siêu, hoặc bởi các đối số mà hàm tạo mặc định sẽ có (hãy nhớ rằng, hàm tạo mặc định luôn là không-đối số) hoặc bởi các đối số đư ợc sử dụng trong lệnh gọi super do trình biên dịch cung cấp ().

Vì vậy, mặc dù không thể ghi đè các hàm tạo, như ng bạn đã thấy rằng chúng có thể bị quá tải, và thư ờng là.

Trình tạo quá tải

Quá tải một hàm tạo có nghĩa là nhập vào nhiều phiên bản của hàm tạo, mỗi phiên bản có một danh sách đối số khác nhau, như các ví dụ sau:

```

class Foo {
    Foo() { }
    Foo(String s) { }
}

```

Lớp Foo trư ớc đó có hai hàm tạo đư ợc nạp chồng: một hàm nhận chuỗi và một chuỗi không có đối số. Bởi vì không có mã nào trong phiên bản no-arg, nó thực sự giống với hàm tạo mặc định mà trình biên dịch cung cấp- như ng hãy nhớ, vì đã có một hàm tạo trong lớp này (lớp lấy chuỗi), trình biên dịch sẽ không cung cấp một hàm mặc định ngư ời xây dựng. Nếu bạn muốn một hàm tạo no-arg làm quá tải phiên bản with-args mà bạn đã có, bạn sẽ phải tự mình nhập nó, giống như trong ví dụ Foo .

Việc ghi đè một phư ơng thức khởi tạo thư ờng đư ợc sử dụng để cung cấp các cách thay thế cho các máy khách để khởi tạo các đối tư ợng thuộc lớp của bạn. Ví dụ: nếu một khách hàng biết tên động vật, họ có thể chuyển tên đó cho một phư ơng thức khởi tạo Động vật có một chuỗi. Như ng nếu họ không biết tên, máy khách có thể gọi hàm tạo no-arg và hàm tạo đó có thể cung cấp một tên mặc định. Đây là những gì nó trông giống như :

```

1. public class Animal {
2.     String name;
3.     Animal(String name) {
4.         this.name = name;
5.     }
6.
7.     Animal() {
8.         this(makeRandomName());
9.     }
10.
11.    static String makeRandomName() {
12.        int x = (int) (Math.random() * 5);
13.        String name = new String[] {"Fluffy", "Fido",
14.                                    "Rover", "Spike",
15.                                    "Gigi"} [x];
16.
17.        return name;
18.    }
19.
20.    public static void main (String [] args) {
21.        Animal a = new Animal();
22.        System.out.println(a.name);
23.        Animal b = new Animal("Zeus");
24.        System.out.println(b.name);
25.    }
26. }
```

Chạy mã bốn lần sẽ tạo ra kết quả này:

```
% java Animal
Gigi
Zeus
```

```
% java Animal
```

```
Fluffy
```

```
Zeus
```

```
% java Animal
```

```
Rover
```

```
Zeus
```

```
% java Animal
```

```
Fluffy
```

```
Zeus
```

Có rất nhiều điều xảy ra trong đoạn mã trư ớc. [Hình 2-7](#) cho thấy ngăn xếp cuộc gọi cho các lệnh gọi phư ơng thức khởi tạo khi một phư ơng thức khởi tạo bị quá tải.

4. Object()

3. Animal(String s) calls super()

2. Animal() calls this(randomlyChosenNameString)

1. main() calls new Animal()

HÌNH 2-7 Các lệnh gọi bị quá tải trên ngăn xếp cuộc gọi

Hãy xem ngăn xếp cuộc gọi, và sau đó chúng ta hãy xem xét mã ngay từ trên cùng.

- Dòng 2 Khai báo tên biến cá thể Chuỗi .
- Dòng 3-5 Khởi tạo lệnh nhận một Chuỗi và gán nó cho tên biến cá thể.
- Dòng 7 Đây là nơi nó trở nên thú vị. Giả sử mọi động vật đều cần một cái tên, như ng máy khách (mã gọi) có thể không phải lúc nào cũng biết tên đó phải là gì, vì vậy lớp Động vật sẽ chỉ định một tên ngẫu nhiên. Phư ơng thức khởi tạo no-arg tạo ra một tên bằng cách gọi phư ơng thức makeRandomName () .
- Dòng 8 Phư ơng thức khởi tạo no-arg gọi phư ơng thức khởi tạo đư ợc nạp chồng của chính nó nhận một Chuỗi, trên thực tế, cách gọi nó giống như cách nó sẽ đư ợc gọi nếu mă máy khách đang thực hiện một hành động mới để khởi tạo một đối tượng, chuyển cho nó một Chuỗi cho tên. Lời kêu gọi quá tải sử dụng từ khóa this, như ng sử dụng nó như

mặc dù nó là một phuơng thức có tên `this()`. Vì vậy, dòng 8 chỉ đơn giản là gọi hàm tạo trên dòng 3, chuyển cho nó một Chuỗi được chọn ngẫu nhiên chứ không phải là tên đã chọn do mã khách hàng.

■ Dòng 11 Lưu ý rằng phuơng thức `makeRandomName()` được đánh dấu là tinh!

Đó là bởi vì bạn không thể gọi một phuơng thức thể hiện (nói cách khác là không tinh) (hoặc truy cập vào một biến thể hiện) cho đến khi phuơng thức siêu khởi tạo đã chạy. Và vì hàm tạo siêu sẽ được gọi từ hàm tạo ở dòng 3, thay vì từ hàm trên dòng 7, dòng 8 chỉ có thể sử dụng một phuơng thức tinh để tạo tên. Nếu chúng ta muốn tất cả các động vật không được ngưng gọi đặt tên cụ thể có cùng một tên mặc định, hãy nói, "Fred", thì dòng 8 có thể đọc là này ("Fred"); chứ không phải gọi một phuơng thức trả về một chuỗi với tên được chọn ngẫu nhiên.

■ Dòng 12 Điều này không liên quan gì đến các hàm tạo, như ng vì tất cả chúng ta đều ở đây để tìm hiểu, nó tạo ra một số nguyên ngẫu nhiên từ 0 đến 4.

■ Dòng 13 Cú pháp kỳ lạ, chúng tôi biết. Chúng tôi đang tạo một đối tượng Chuỗi mới (chỉ là một cá thể Chuỗi duy nhất), như ng chúng tôi muốn chuỗi được chọn ngẫu nhiên từ một danh sách. Ngoại trừ chúng tôi không có danh sách, vì vậy chúng tôi cần phải làm cho nó. Vì vậy, trong một dòng mã đó, chúng tôi

1. Khai báo tên biến `String`.
2. Tạo một mảng Chuỗi (tên danh – chúng tôi không chỉ định chính mảng thành một biến).
3. Lấy chuỗi tại chỉ số `[x]` (`x` là số ngẫu nhiên được tạo trên dòng 12) của mảng `String` mới được tạo.
4. Gán chuỗi được truy xuất từ mảng cho cá thể đã khai báo tên biến. Chúng tôi có thể làm cho nó dễ đọc hơn nhiều nếu chúng tôi chỉ viết

```
String[] nameList = {"Fluffy", "Fido", "Rover", "Spike",
"Gigi"};
```

```
String name = nameList[x];
```

Như ng niềm vui trong đó ở đâu? Việc ném vào các cú pháp bất thường (đặc biệt là đối với mã hoàn toàn không liên quan đến câu hỏi thực) là tinh thần của kỳ thi.

Đừng giật mình! (Được rồi, hãy giật mình, như ng sau đó chỉ cần tự nói với chính mình, "Chà!" Và tiếp tục với nó.)

■ Dòng 18 Chúng tôi đang gọi phiên bản không đối số của hàm tạo (gây ra

tên ngẫu nhiên từ danh sách được chuyển cho phương thức khởi tạo khác).

- Dòng 20 Chúng tôi đang gọi phương thức khởi tạo nạp chồng có một chuỗi đại diện cho tên.

Điểm quan trọng để nhận được từ ví dụ mã này là ở dòng 8. Thay vì gọi super(), chúng ta đang gọi this() và this() luôn có nghĩa là một lời gọi đến một hàm tạo khác trong cùng một lớp. Được rồi, tốt thôi, nhưng điều gì sẽ xảy ra sau cuộc gọi tới cái này()? Sớm muộn gì thì hàm tạo super() cũng được gọi, phải không? Đúng vậy.

Một cuộc gọi tới this() chỉ có nghĩa là bạn đang trì hoãn điều không thể tránh khỏi. Một số hàm tạo, ở đâu đó, phải thực hiện lệnh gọi tới super().

Quy tắc chính: Dòng đầu tiên trong một hàm tạo phải là một lệnh gọi tới super() hoặc một lệnh gọi this().

Không có ngoại lệ. Nếu bạn không có lệnh gọi nào trong hàm tạo của mình, trình biên dịch sẽ chèn lệnh gọi no-arg vào super(). Nói cách khác, nếu hàm tạo A() có lệnh gọi this(), trình biên dịch biết rằng hàm tạo A() sẽ không phải là hàm gọi super().

Quy tắc trứ ớc có nghĩa là một hàm tạo không bao giờ có thể có cả lời gọi tới super() và lệnh gọi this(). Bởi vì mỗi lệnh gọi đó phải là câu lệnh đầu tiên trong một hàm tạo, bạn không thể sử dụng hợp pháp cả hai trong cùng một hàm tạo. Điều đó cũng có nghĩa là trình biên dịch sẽ không đặt lệnh gọi tới super() trong bất kỳ hàm tạo nào có lệnh gọi this().

Câu hỏi suy nghĩ: Bạn nghĩ điều gì sẽ xảy ra nếu bạn cố gắng biên dịch đoạn mã sau?

```
class A {
    A() {
        this("foo");
    }
    A(String s) {
        this();
    }
}
```

Trình biên dịch của bạn có thể không thực sự gấp sự cố (nó thay đổi tùy thuộc vào trình biên dịch của bạn, như hầu hết sẽ không giải quyết được sự cố). Nó cho rằng bạn biết mình đang làm gì. Bạn có thể phát hiện ra lỗi hỏng không? Cho rằng một hàm tạo siêu phải luôn được gọi, thì lệnh gọi tới super() sẽ đi đâu? Hãy nhớ rằng, trình biên dịch sẽ không đưa vào một hàm tạo mặc định nếu bạn đã có một hoặc nhiều hàm tạo trong lớp của mình. Và khi trình biên dịch không đặt một hàm tạo mặc định, nó vẫn chèn một lệnh gọi tới super() trong bất kỳ hàm tạo nào không có lệnh gọi đến

hàm tạo siêu - trừ khi, nghĩa là, hàm tạo đã có lệnh gọi `this ()`. Vậy trong đoạn mã trư ớc, `super ()` có thể đi đâu? Hai hàm tạo duy nhất trong lớp đều có lệnh gọi `this ()` và trên thực tế, bạn sẽ nhận được chính xác những gì bạn nhận được nếu bạn nhập mã phư ơng thức sau:

```
public void go ()
    {doStuff ();
}

public void doStuff ()
    {go ();
}
```

Bây giờ bạn có thể thấy vấn đề? Tất nhiên bạn có thể. Ngăn xếp phát nổ! Nó được cao hơn và cao hơn và cao hơn cho đến khi nó chỉ mở ra và mã phư ơng thức tràn ra ngoài, chảy ra khỏi JVM ngay trên sàn. Hai hàm tạo được nạp chồng cùng gọi `this ()` là hai hàm tạo gọi nhau – lặp đi lặp lại, dẫn đến điều này:

```
% java A
Exception trong luồng "main" java.lang.StackOverflowError
```

Lợi ích của việc có các hàm tạo quá tải là bạn cung cấp các cách linh hoạt để khởi tạo các đối tượng từ lớp của bạn. Lợi ích của việc có một hàm tạo gọi một hàm tạo khác được nạp chồng là để tránh trùng lặp mã. Trong ví dụ `Animal`, không có bất kỳ mã nào ngoài việc đặt tên, nhưng hãy tưởng tượng nếu sau dòng 4 vẫn còn nhiều việc phải làm trong hàm tạo. Bằng cách đặt tất cả các công việc của hàm tạo khác chỉ trong một hàm tạo và sau đó yêu cầu các hàm tạo khác gọi nó, bạn không cần phải viết và duy trì nhiều phiên bản của mã hàm tạo quan trọng khác đó. Về cơ bản, mỗi hàm tạo được nạp chồng không-phải-thực-khác sẽ gọi một hàm tạo bị quá tải khác, truyền cho nó bất kỳ dữ liệu nào nó cần (dữ liệu mà mã máy khách không cung cấp).

Các trình xây dựng và khởi tạo thậm chí còn trở nên thú vị hơn (chỉ khi bạn nghĩ rằng nó an toàn) khi bạn đến các lớp bên trong, nhưng chúng tôi biết bạn có thể chỉ có rất nhiều niềm vui trong một chương và ngoài ra, bạn không cần phải đối phó với các lớp bên trong cho đến khi bạn vượt qua kỳ thi OCP.

MỤC TIÊU XÁC NHẬN

Khởi khởi tạo (Mục tiêu OCA 1.2 và 6.3-ish)

1.2 Xác định cấu trúc của một lớp Java

6.3 Tạo và nạp chồng các hàm tạo; bao gồm tác động đến các trình tạo mặc định

Chúng ta đã nói về hai nơi trong một lớp nơi bạn có thể đặt mã thực hiện các hoạt động: phư ơng thức và hàm tạo. Các khôi khởi tạo là vị trí thứ ba trong một chương trình Java, nơi các hoạt động có thể được thực hiện. Các khôi khởi tạo tinh chạy khi lớp được tải lần đầu tiên và các khôi khởi tạo thể hiện chạy bất cứ khi nào một thẻ hiện được tạo (một chút tương tự như một hàm tạo). Hãy xem một ví dụ:

```
class SmallInit {
    static int x;
    int y;

    static { x = 7; }           // static init block
    { y = 8; }                  // instance init block
}
```

Như bạn có thể thấy, cú pháp cho các khôi khởi tạo khá ngắn gọn. Họ không có tên, họ không thể tranh luận, và họ không trả lại bất cứ điều gì. Một khôi khởi tạo tinh chạy một lần khi lớp được tải lần đầu tiên. Một khôi khởi tạo thể hiện sẽ chạy một lần mỗi khi một thẻ hiện mới được tạo. Hãy nhớ khi chúng ta nói về thứ tự mà mã khởi tạo được thực thi? Mã khôi init phiên bản chạy ngay sau lệnh gọi tới super () trong một hàm tạo – nói cách khác, sau khi tất cả các hàm tạo siêu đã chạy.

Bạn có thể có nhiều khôi khởi tạo trong một lớp. Điều quan trọng cần lưu ý là không giống như các phư ơng thức hoặc hàm tạo, thứ tự các khôi khởi tạo xuất hiện trong một lớp rất quan trọng. Khi đến lúc các khôi khởi tạo chạy, nếu một lớp có nhiều hơn một, chúng sẽ chạy theo thứ tự xuất hiện trong tệp lớp – nói cách khác là từ trên xuống. Dựa trên các quy tắc chúng ta vừa thảo luận, bạn có thể xác định đầu ra của chương trình sau không?

```

class Init {
    Init(int x) { System.out.println("1-arg const"); }
    Init() { System.out.println("no-arg const"); }
    static { System.out.println("1st static init"); }
    { System.out.println("1st instance init"); }
    { System.out.println("2nd instance init"); }
    static { System.out.println("2nd static init"); }

    public static void main(String [] args) {
        new Init();
        new Init(7);
    }
}

```

Để tìm ra điều này, hãy nhớ các quy tắc sau:

- các khối init thực thi theo thứ tự mà chúng xuất hiện.
- Các khối init tĩnh chạy một lần, khi lớp được tải lần đầu tiên.
- Các khối init phiên bản chạy mỗi khi một cá thể lớp được tạo.
- Các khối init thẻ hiện chạy sau lời gọi của hàm tạo tới super () .

Với những quy tắc đó, kết quả sau sẽ có ý nghĩa:

```

1st static init
2nd static init
1st instance init
2nd instance init
no-arg const
1st instance init
2nd instance init
1-arg const

```

Như bạn có thể thấy, các khối init instance mỗi lần chạy hai lần. Khối init phiên bản thư ờng được sử dụng như một nơi để đặt mã mà tất cả các hàm tạo trong một lớp nên chia sẻ. Bằng cách đó, mã không cần phải được sao chép giữa các hàm tạo.

Cuối cùng, nếu bạn mắc lỗi trong khối init tĩnh của mình , JVM có thể ném lỗi `ExceptionInInitializerError`. Hãy xem một ví dụ:

```

class InitError {
    static int [] x = new int[4];
    static { x[4] = 5; }           // bad array index!
    public static void main(String [] args) { }
}

```

Nó tạo ra một cái gì đó như thế này:

```
Exception in thread "main" java.lang.ExceptionInInitializerError  
Caused by: java.lang.ArrayIndexOutOfBoundsException: 4  
        at InitError.<clinit>(InitError.java:3)
```



Theo quy ước, các khối init thường xuất hiện gần đầu tệp lớp, ở đâu đó xung quanh các hàm tạo. Tuy nhiên, đây là kỳ thi OCA mà chúng ta đang nói đến. Đừng ngạc nhiên nếu bạn tìm thấy một khối init nằm giữa một vài phương pháp, tìm kiếm tất cả mọi thứ giống như một lỗi trình biên dịch đang chờ xảy ra!

MỤC TIÊU XÁC NHẬN

Tin học (Mục tiêu 6.2 của OCA)

6.2 Áp dụng từ khóa static cho các phương thức và trư ờng.

Các biến và phương thức tĩnh

Công cụ sửa đổi tĩnh có tác động sâu sắc đến hành vi của một phương thức hoặc biến mà chúng ta đang coi nó như một khái niệm hoàn toàn tách biệt với các công cụ sửa đổi khác. Để hiểu cách thức hoạt động của một thành viên tĩnh, trước tiên chúng ta sẽ xem xét lý do sử dụng một thành viên tĩnh. Hãy tưởng tượng bạn có một lớp tiện ích hoặc giao diện với một phương thức luôn chạy theo cùng một cách; chức năng duy nhất của nó là trả về một số ngẫu nhiên. Không quan trọng trư ờng hợp nào của lớp thực hiện phương thức – nó sẽ luôn hoạt động giống hệt nhau. Nói cách khác, hành vi của phương thức không phụ thuộc vào trạng thái (giá trị biến thể hiện) của một đối tượng. Vậy tại sao bạn lại cần một đối tượng trong khi phương thức sẽ không bao giờ là đối tượng cụ thể? Tại sao không chỉ yêu cầu loại chính nó chạy phương thức?

Hãy tưởng tượng một tình huống khác: Giả sử bạn muốn giữ số lượng đang chạy của tất cả các phiên bản được khởi tạo từ một lớp cụ thể. Bạn thực sự giữ biến đó ở đâu? Sẽ không hiệu quả nếu giữ nó như một biến thể hiện trong lớp có

các tru ờng hợp bạn đang theo dõi, vì số lư ợng sẽ chỉ đư ợc khởi tạo trở lại giá trị mặc định với mỗi phiên bản mới. Câu trả lời cho cả kịch bản tiện ích-method always-running-the-same và kịch bản keep-a-running-total-instance là sử dụng công cụ sửa đổi tĩnh . Các biến và phư ơng thức đư ợc đánh dấu là tĩnh thuộc về kiểu chữ không thuộc về bất kỳ tru ờng hợp cụ thể nào. Trên thực tế, đối với các lớp, bạn có thể sử dụng một phư ơng thức hoặc biến static mà không cần có bất kỳ tru ờng hợp nào của lớp đó. Bạn chỉ cần có sẵn kiểu để có thể gọi một phư ơng thức tĩnh hoặc truy cập một biến tĩnh . Các biến static cũng có thể đư ợc truy cập mà không cần có một thê hiện của một lớp. Như ng nếu có các tru ờng hợp, một biến tĩnh của một lớp sẽ đư ợc chia sẻ bởi tất cả các tru ờng hợp của lớp đó; chỉ có một bản sao.

Đoạn mã sau khai báo và sử dụng một biến bộ đếm tĩnh :

```
class Frog {
    static int frogCount = 0; // Declare and initialize
                             // static variable
    public Frog() {
        frogCount += 1;           // Modify the value in the constructor
    }
}

public static void main (String [] args) {
    new Frog();
    new Frog();
    new Frog();
    System.out.println("Frog count is now " + frogCount);
}
```

Trong đoạn mã tru ớc, biến static échCount đư ợc đặt thành 0 khi lớp Frog đư ợc tải lần đầu tiên bởi JVM, tru ớc khi bất kỳ tru ờng hợp Frog nào đư ợc tạo! (Nhân tiện, bạn không thực sự cần phải khởi tạo một biến static về 0; các biến static nhận cùng giá trị mặc định mà các biến thê hiện nhận đư ợc.) Bất cứ khi nào một cá thể Frog đư ợc tạo, hàm tạo Frog sẽ chạy và tăng biến static échCount . Khi mã này thực thi, ba phiên bản Frog đư ợc tạo trong main () và kết quả là

Số lư ợng éch bây giờ là 3

Bây giờ, hãy tư ờng tư ợng điều gì sẽ xảy ra nếu échCount là một biến thê hiện (nói cách khác là không tĩnh):

```

class Frog {
    int frogCount = 0; // Declare and initialize
                       // instance variable
    public Frog() {
        frogCount += 1; // Modify the value in the constructor
    }
    public static void main (String [] args) {
        new Frog();
        new Frog();
        new Frog();
        System.out.println("Frog count is now " + frogCount);
    }
}

```

Khi mã này thực thi, nó vẫn sẽ tạo ra ba phiên bản Frog trong main (), nhưng kết quả là lỗi trình biên dịch! Chúng tôi không thể lấy mã này để biên dịch, hãy để một mình chạy.

```

Frog.java:11: nonstatic variable frogCount cannot be referenced
from a static context
    System.out.println("Frog count is " + frogCount);
                                         ^
1 error

```

JVM không biết tài khoản ếch của đôi tư ợng Frog nào bạn đang cố gắng truy cập. Vấn đề là main () tự nó là một phư ơng thức tĩnh và do đó không chạy với bất kỳ trư ờng hợp cụ thể nào của lớp; thay vào đó nó đang chạy trên chính lớp đó. Một phư ơng thức tĩnh không thể truy cập vào một biến không tĩnh (phiên bản) vì không có phiên bản nào! Điều đó không có nghĩa là không có trư ờng hợp nào của lớp còn sống trên heap, mà đúng hơn là ngay cả khi có, phư ơng thức tĩnh không biết gì về chúng. Điều tư ợng tự cũng áp dụng cho các phư ơng thức thể hiện; một phư ơng thức static không thể gọi trực tiếp một phư ơng thức nonstatic. Hãy nghĩ static = class, nonstatic = instance. Làm cho phư ơng thức đư ợc gọi bởi JVM (main ()) thành một phư ơng thức tĩnh có nghĩa là JVM không phải tạo một thể hiện của lớp của bạn chỉ để bắt đầu chạy mã.



Một trong những sai lầm mà các lập trình viên Java mới thường mắc phải là cố gắng truy cập vào một biến thể hiện (có nghĩa là không ổn định

biến) từ phư ơng thức static main () (không biết gì về bất kỳ trư ờng hợp nào, vì vậy nó không thể truy cập vào biến). Đoạn mã sau là một ví dụ về truy cập bất hợp pháp một biến nonstatic từ một phư ơng thức static :

```
class Foo {
    int x = 3;
    public static void main (String [] args) {
        System.out.println("x is " + x);
    }
}
```

Hãy hiểu rằng mã này sẽ không bao giờ biên dịch, bởi vì bạn không thể truy cập một biến không tĩnh (phiên bản) từ một phư ơng thức tĩnh . Chỉ cần nghĩ đến trình biên dịch nói, "Này, tôi không biết bạn đang cố in biến x của đối tượng Foo nào!" Hãy nhớ rằng, đó là lớp chạy phư ơng thức main () , không phải là một thẻ hiện của lớp.

Tất nhiên, phần khó cho kỳ thi là câu hỏi sẽ không rõ ràng như mã trư ớc đó. Vấn đề mà bạn đang đư ợc kiểm tra- truy cập một biến không tĩnh từ một phư ơng thức tĩnh – sẽ bị chôn vùi trong mã có vẻ như đang thử nghiệm thứ gì đó khác. Ví dụ: mã trư ớc sẽ có nhiều khả năng xuất hiện dưới dạng

```
class Foo {
    int x = 3;
    float y = 4.3f;
    public static void main (String [] args) {
        for (int z = x; z < ++x; z--, y = y + z)
            // complicated looping and branching code
    }
}
```

Vì vậy, trong khi bạn đang cố gắng tuân theo logic, vấn đề thực sự là x và y không thể đư ợc sử dụng trong hàm main () vì x và y là các biến thẻ hiện, không phải là biến tĩnh! Điều tư ơng tự cũng áp dụng cho việc truy cập các phư ơng thức nonstatic từ một phư ơng thức tĩnh . Quy tắc là, một phư ơng thức tĩnh của một lớp không thể truy cập vào một phư ơng thức hoặc biến (thẻ hiện) nonstatic của chính lớp đó.

Truy cập các phư ơng thức và biến tĩnh

Vì bạn không cần phải có một thẻ hiện để gọi một phư ơng thức tĩnh hoặc truy cập một biến tĩnh, làm cách nào để bạn gọi hoặc sử dụng một thành viên tĩnh ? Cú pháp là gì? Chúng tôi biết rằng với một phư ơng thức phiên bản cũ thông thường, bạn sử dụng toán tử dấu chấm trên một tham chiếu đến một phiên bản:

```
class Frog {
    int frogSize = 0;
    public int getFrogSize() {
        return frogSize;
    }
    public Frog(int s) {
        frogSize = s;
    }
    public static void main (String [] args) {
        Frog f = new Frog(25);
        System.out.println(f.getFrogSize()); // Access instance
                                            // method using f
    }
}
```

Trong đoạn mã trư ớc, chúng ta khởi tạo một Frog, gán nó cho biến tham chiếu f, và sau đó sử dụng tham chiếu f đó để gọi một phư ơng thức trên cá thể Frog mà chúng ta vừa tạo. Nói cách khác, phư ơng thức getFrogSize () đang đư ợc gọi trên một đối tượng Frog cụ thể trên heap.

Như ng cách tiếp cận này (sử dụng tham chiếu đến một đối tượng) không thích hợp để truy cập một phư ơng thức tĩnh , vì có thể không có bất kỳ trư ờng hợp nào của lớp! Vì vậy, cách chúng tôi truy cập một phư ơng thức tĩnh (hoặc biến tĩnh) là sử dụng toán tử dấu chấm trên tên kiểu, trái ngược với việc sử dụng nó trên một tham chiếu đến một cá thể, như sau:

```

class Frog {
    private static int frogCount = 0; // static variable
    static int getCount() {           // static getter method
        return frogCount;
    }
    public Frog() {
        frogCount += 1;             // Modify the value in the constructor
    }
}

class TestFrog {
    public static void main (String [] args) {
        new Frog();
        new Frog();
        new Frog();
        System.out.println("from static "+Frog.getCount()); // static context
        new Frog();

        new TestFrog().go();
        Frog f = new Frog();
        System.out.println("use ref var "+f.getCount()); // use reference var
    }
    void go() {
        System.out.println("from inst "+Frog.getCount()); // instance context
    }
}

```

tạo ra đầu ra:

từ static 3 từ
 instance 4 sử dụng
 ref var 5

Như ng chỉ để làm cho nó thực sự khó hiểu, ngôn ngữ Java cũng cho phép bạn sử dụng một biến tham chiếu đối tượng để truy cập một thành viên tĩnh. Bạn có bắt đư ợc dòng cuối cùng của hàm main () không? Nó bao gồm lời kêu gọi này:

f.getCount (); // Truy cập tĩnh bằng một biến cá thể

Trong đoạn mã trước đó, chúng tôi khởi tạo một Frog, gán đối tượng Frog mới cho biến tham chiếu f, và sau đó sử dụng tham chiếu f để gọi một phương thức tĩnh ! Như ng mặc dù chúng tôi đang sử dụng một cá thể Frog cụ thể để truy cập vào phương thức tĩnh , các quy tắc vẫn không thay đổi. Đây chỉ là một thủ thuật cú pháp để cho phép bạn sử dụng một biến tham chiếu đối tượng (như ng không phải đối tượng mà nó tham chiếu đến) để truy cập một phương thức hoặc biến tĩnh, như ng thành viên tĩnh vẫn không biết về cá thể cụ thể đư ợc sử dụng để gọi thành viên tĩnh . Trong ví dụ Frog , trình biên dịch

biết rằng biến tham chiếu f thuộc kiểu Frog, và do đó, phư ơng thức tĩnh của lớp Frog đư ợc chạy mà không có nhận thức hoặc mối quan tâm nào đối với cá thể Frog ở đầu kia của tham chiếu f . Nói cách khác, trình biên dịch chỉ quan tâm rằng biến tham chiếu f đư ợc khai báo là kiểu Frog.

Việc gọi các phư ơng thức tĩnh từ các giao diện gần giống như việc gọi các phư ơng thức tĩnh từ các lớp, ngoại trừ "thủ thuật cú pháp biến thể hiện" vừa thảo luận chỉ hoạt động cho các phư ơng thức tĩnh trong các lớp. Đoạn mã sau minh họa cách các phư ơng thức tĩnh trong giao diện có thể và không thể đư ợc gọi:

```
interface FrogBoilable {
    static int getCtoF(int cTemp) {                                // interface static method
        return (cTemp * 9 / 5) + 32;
    }
    default String hop() { return "hopping"; }      // interface default method
}

public class DontBoilFrogs implements FrogBoilable {
    public static void main(String[] args) {
        new DontBoilFrogs().go();
    }

    void go() {
        System.out.println(hop());                                // 1 - ok for default m
        // System.out.println(getCtoF(100));                      // 2 - cannot find symbol

        System.out.println(FrogBoilable.getCtoF(100));          // 3 - ok for static m
        DontBoilFrogs dbf = new DontBoilFrogs();
        // System.out.println(dbf.getCtoF(100));                  // 4 - cannot find symbol
    }
}
```

Hãy xem lại đoạn mã:

- Dòng 1 là lời gọi hợp pháp của phư ơng thức mặc định của giao diện .
- Dòng 2 là một nỗ lực bắt hợp pháp để gọi phư ơng thức tĩnh của một giao diện .
- Dòng 3 là cách hợp pháp để gọi phư ơng thức tĩnh của giao diện .
- Dòng 4 là một nỗ lực bắt hợp pháp khác để gọi phư ơng thức tĩnh của giao diện .

[Hình 2-8](#) minh họa ảnh hư ờng của công cụ sửa đổi tĩnh đối với các phư ơng thức và biến.

```
class Foo  
  
int size = 42;  
static void doMore( ) {  
    int x = size;  
}
```

static method cannot
access an instance
(nonstatic) variable

```
class Bar  
  
void go(){}
static void doMore( ) {  
    go();  
}
```

static method cannot
access a nonstatic
method

```
class Baz  
  
static int count;  
static void woo( ) {}  
static void doMore( ) {  
    woo( );  
    int x = count;  
}
```

static method
can access a static
method or variable

HÌNH 2-8 Ví dụ về lỗi trong cách sử dụng của static đối với các phư ơng thức và biến

Cuối cùng, hãy nhớ rằng không thể ghi đè các phư ơng thức tĩnh ! Điều này không có nghĩa là chúng không thể được xác định lại trong một lớp con, như việc xác định lại và ghi đè không phải là điều giống nhau. Hãy xem một ví dụ về phư ơng thức tĩnh được xác định lại (nhớ, không bị ghi đè) :

```

class Animal {
    static void doStuff() {
        System.out.print("a ");
    }
}
class Dog extends Animal {
    static void doStuff() { // it's a redefinition,
                           // not an override
        System.out.print("d ");
    }
    public static void main(String [] args) {
        Animal [] a = {new Animal(), new Dog(), new Animal()};
        for(int x = 0; x < a.length; x++) {
            a[x].doStuff(); // invoke the static method
        }
        Dog.doStuff(); // invoke using the class name
    }
}

```

Chạy mã này tạo ra kết quả này:

aaad

Hãy nhớ rằng, cú pháp `a [x] .doStuff ()` chỉ là một phím tắt (thủ thuật cú pháp) - trình biên dịch sẽ thay thế một cái gì đó như `Animal.doStuff ()` để thay thế.

Cũng lưu ý rằng bạn có thể gọi một phương thức tĩnh bằng cách sử dụng tên lớp.

Lưu ý rằng chúng tôi đã không sử dụng vòng lặp `for` nâng cao ở đây (được đề cập trong [Chương 5](#)), mặc dù chúng tôi có thể có. Kỳ vọng sẽ thấy sự kết hợp của cả hai kiểu mã hóa Java 1.4 và Java 5-8 và thực hành trong kỳ thi.

TÓM TẮT CHỨNG NHẬN

Chúng ta bắt đầu chương này bằng cách thảo luận về tầm quan trọng của việc đóng gói trong thiết kế OO tốt, và sau đó chúng ta nói về cách thực hiện đóng gói tốt: với các biến cá thể riêng, bộ nhận và bộ định tuyến công cộng.

Tiếp theo, chúng tôi đã đề cập đến tầm quan trọng của tính kế thừa, để bạn có thể nắm được ghi đè, nạp chồng, đa hình, đúc tham chiếu, kiểu trả về và các nhà xây dựng.

Chúng tôi đã đề cập đến IS-A và HAS-A. IS-A được thực hiện bằng cách sử dụng kế thừa và HAS-A được thực hiện bằng cách sử dụng các biến cá thể tham chiếu đến các đối tượng khác.

Tiếp theo là tính đa hình. Mặc dù kiểu của biến tham chiếu không thể là

Tiếp theo là tính đa hình. Mặc dù không thể thay đổi kiểu của một biến tham chiếu, như ng nó có thể được sử dụng để tham chiếu đến một đối tượng có kiểu là một kiểu con của chính nó. Chúng tôi đã học cách xác định những phương pháp nào có thể lập hóa đơn cho một biến tham chiếu nhất định.

Chúng tôi đã xem xét sự khác biệt giữa các phương thức được ghi đè và quá tải, biết rằng một phương thức được ghi đè xảy ra khi một kiểu con kế thừa một phương thức từ một kiểu siêu và sau đó thực hiện lại phương thức để thêm các hành vi chuyên biệt hơn. Chúng tôi đã biết rằng, trong thời gian chạy, JVM sẽ gọi phiên bản kiểu con trên một phiên bản của kiểu con và phiên bản siêu kiểu trên một phiên bản của siêu kiểu. Các phương thức trừu tượng phải được "ghi đè" (về mặt kỹ thuật, các phương thức trừu tượng phải được thực hiện, trái ngược với ghi đè, vì thực sự không có gì để ghi đè).

Chúng tôi thấy rằng các phương thức ghi đè phải khai báo cùng một danh sách đối số và kiểu trả về hoặc họ có thể trả về kiểu con của kiểu trả về đã khai báo của phương thức ghi đè của siêu kiểu) và công cụ sửa đổi truy cập không thể hạn chế hơn. Phương thức ghi đè cũng không thể ném bất kỳ ngoại lệ mới hoặc đã kiểm tra rộng hơn nào không được khai báo trong phương thức ghi đè. Bạn cũng biết rằng phương thức ghi đè có thể được gọi bằng cú pháp super.doSomething () ;.

Các phương thức bị quá tải cho phép bạn sử dụng lại cùng tên phương thức trong một lớp, như với các đối số khác nhau (và, tùy chọn, một kiểu trả về khác). Trong khi các phương thức ghi đè không được thay đổi danh sách đối số, thì các phương thức ghi đè phải thay đổi. Nhưng không giống như các phương thức ghi đè, các phương thức nạp chồng có thể tự do thay đổi kiểu trả về, công cụ sửa đổi quyền truy cập và các ngoại lệ đã khai báo theo bất kỳ cách nào chúng muốn.

Chúng tôi đã học được cơ chế truyền (chủ yếu là dự báo xuống) các biến tham chiếu và khi nào cần thiết phải làm như vậy.

Tiếp theo là các giao diện triển khai. Một giao diện mô tả một hợp đồng mà lớp thực hiện phải tuân theo. Các quy tắc để triển khai một giao diện tương tự như các quy tắc để mở rộng một lớp trừu tượng . Kể từ Java 8, các giao diện có thể có các phương thức cụ thể, được gắn nhãn mặc định. Ngoài ra, hãy nhớ rằng một lớp có thể triển khai nhiều hơn một giao diện và các giao diện đó có thể mở rộng giao diện khác.

Chúng tôi cũng đã xem xét các kiểu trả về của phương thức và thấy rằng bạn có thể khai báo bất kỳ kiểu trả về nào mà bạn thích (giả sử bạn có quyền truy cập vào một lớp cho kiểu trả về tham chiếu đối tượng), trừ khi bạn đang ghi đè một phương thức. Bỏ qua một trả về hiệp phương sai, một phương thức ghi đè phải có cùng kiểu trả về với phương thức ghi đè của lớp cha. Chúng ta thấy rằng, mặc dù các phương thức ghi đè không được thay đổi kiểu trả về, nhưng các phương thức ghi đè có thể (miễn là chúng cũng thay đổi danh sách đối số).

Cuối cùng, bạn đã học được rằng việc trả về bất kỳ giá trị hoặc biến nào có thể là hợp pháp

đư ợc chuyển đổi ngầm định thành kiểu trả về đã khai báo. Vì vậy, ví dụ, một short có thể được trả về khi kiểu trả về đư ợc khai báo là int. Và (giả sử Horse mở rộng động vật), một tham chiếu Horse có thể đư ợc trả về khi kiểu trả về đư ợc khai báo là Động vật.

Chúng tôi đã đề cập chi tiết đến các hàm tạo, biết rằng nếu bạn không cung cấp một hàm tạo cho lớp của mình, trình biên dịch sẽ chèn một hàm tạo. Hàm tạo do trình biên dịch tạo ra đư ợc gọi là hàm tạo mặc định và nó luôn là một hàm tạo không đối số với lời gọi no-arg tới super (). Hàm tạo mặc định sẽ không bao giờ đư ợc tạo nếu ngay cả một hàm tạo duy nhất tồn tại trong lớp của bạn (bất kể các đối số của hàm tạo đó là gì); vì vậy nếu bạn cần nhiều hơn một hàm tạo trong lớp của mình và bạn muốn một hàm tạo no-arg, bạn sẽ phải tự viết nó. Chúng tôi cũng thấy rằng các hàm tạo không đư ợc kế thừa và bạn có thể bị nhầm lẫn bởi một phư ơng thức có cùng tên với lớp (là hợp pháp). Kiểu trả về là tặng phẩm mà một phư ơng thức không phải là phư ơng thức tạo bởi vì các phư ơng thức khởi tạo không có kiểu trả về.

Chúng ta đã thấy cách tất cả các hàm tạo trong cây kế thừa của một đối tư ợng sẽ luôn đư ợc gọi khi đối tư ợng đư ợc khởi tạo bằng cách sử dụng new. Chúng tôi cũng thấy rằng các hàm tạo có thể đư ợc nạp chồng, có nghĩa là xác định các hàm tạo với các danh sách đối số khác nhau. Một hàm tạo có thể gọi một hàm tạo khác cùng lớp bằng cách sử dụng từ khóa this (), như thể hàm tạo là một phư ơng thức có tên this (). Chúng tôi thấy rằng mọi hàm tạo phải có this () hoặc super () làm câu lệnh đầu tiên (mặc dù trình biên dịch có thể chèn nó cho bạn).

Sau hàm tạo, chúng ta đã thảo luận về hai loại khởi khởi tạo và làm thế nào và khi mã của họ chạy.

Chúng tôi đã xem xét các phư ơng thức và biến tĩnh . các thành viên tĩnh đư ợc gắn với lớp hoặc giao diện, không phải là một cá thể, vì vậy chỉ có một bản sao của bất kỳ thành viên tĩnh nào . Một sai lầm phổ biến là cố gắng tham chiếu đến một biến thể hiện từ một phư ơng thức tĩnh . Sử dụng tên lớp hoặc giao diện tư ợng ứng với toán tử dấu chấm để truy cập các thành viên tĩnh .

Và, một lần nữa, bạn đã biết rằng kỳ thi bao gồm các câu hỏi phức tạp đư ợc thiết kế phần lớn để kiểm tra khả năng nhận biết mức độ khó của các câu hỏi.

✓ KHOAN HAI PHÚT

Dưới đây là một số điểm chính từ mỗi mục tiêu chứng nhận trong chương này.

Đóng gói, IS-A, HAS-A * (Mục tiêu OCA 6.5)

- Đóng gói giúp ản việc triển khai đằng sau một giao diện (hoặc API).
- Mã đóng gói có hai tính năng: Các biến cá thể
 - đư ợc giữ ở chế độ bảo vệ (thư ờng là với công cụ sửa đổi riêng).
 - Phư ơng thức getter và setter cung cấp quyền truy cập vào các biến cá thể.
- IS-A đè cập đến sự kế thừa hoặc triển khai.
- IS-A đư ợc thể hiện với từ khóa mở rộng hoặc thực hiện.
- IS-A, "kế thừa từ" và "là một kiểu con của" đều là các biểu thức tư ơng đư ơng.
- HAS-A có nghĩa là một thể hiện của một lớp "có một tham chiếu" đến một thể hiện của một lớp khác hoặc một thể hiện khác của cùng một lớp. * HAS-A KHÔNG có trong kỳ thi, như ng thật tốt khi biết.

Kế thừa (Mục tiêu 7.1 của OCA)

- Kế thừa cho phép một kiểu là một kiểu con của kiểu siêu và do đó kế thừa các biến và phư ơng thức công khai và đư ợc bảo vệ của kiểu siêu đó.
- Kế thừa là một khái niệm chính làm nền tảng cho IS-A, tính đa hình, ghi đè, nạp chồng và ép kiểu.
- Tất cả các lớp (ngoại trừ lớp Đôi tư ợng) là các lớp con của kiểu Đôi tư ợng, và do đó chúng kế thừa các phư ơng thức của Đôi tư ợng .

Đa hình (Mục tiêu OCA 7.2)

- Đa hình có nghĩa là "nhiều hình thức".
- Một biến tham chiếu luôn là một kiểu duy nhất, không thể thay đổi, như ng nó có thể tham chiếu đến một đối tư ợng kiểu con.
- Một đối tư ợng duy nhất có thể đư ợc tham chiếu bởi các biến tham chiếu thuộc nhiều kiểu khác nhau - miến là chúng cùng kiểu hoặc là siêu kiểu của đối tư ợng.
- Kiểu của biến tham chiếu (không phải kiểu của đối tư ợng) xác định phư ơng thức nào có thể đư ợc gọi!
- Các lệnh gọi phư ơng thức đa hình chỉ áp dụng cho các phư ơng thức cá thể bị ghi đè .

Ghi đè và quá tải (Mục tiêu 6.1 và 7.2 của OCA)

- Các phư ơng thức có thể bị ghi đè hoặc quá tải; các hàm tạo có thể đư ợc nạp chòng như ng không đư ợc ghi đè.
- Đối với phư ơng thức nó ghi đè, phư ơng thức ghi đè
 - Phải có cùng môt danh sách đối số
 - Phải có cùng môt kiểu trả về hoặc môt lớp con (đư ợc gọi là trả về cùng phư ơng)
- Không đư ợc có công cụ sửa đổi quyền truy cập hạn chế hơn
- Có thể có công cụ sửa đổi quyền truy cập ít hạn chế hơn
- Không đư ợc ném các ngoại lệ mới hoặc đã kiểm tra rộng hơn
- Có thể ném ít hơn hoặc thu hẹp các ngoại lệ đã kiểm tra hoặc bất kỳ ngoại lệ nào chưa đư ợc kiểm tra
- các phư ơng thức cuối cùng không thể đư ợc ghi đè.
- Chỉ các phư ơng thức kế thừa mới có thể bị ghi đè và hấy nhớ rằng các phư ơng thức riêng tư không đư ợc kế thừa.
- Một lớp con sử dụng super.overriddenMethodName () để gọi phiên bản lớp cha của môt phư ơng thức đư ợc ghi đè.
- Một lớp con sử dụng MyInterface.super.overriddenMethodName () để gọi phiên bản siêu giao diện trên môt phư ơng thức đư ợc ghi đè.
- Nạp chòng có nghĩa là sử dụng lại môt tên phư ơng thức như ng với các đối số khác nhau.
- Phư ơng thức bị quá tải
 - Phải có các danh sách đối số khác nhau
 - Có thể có các kiểu trả về khác nhau, nếu danh sách đối số cũng khác nhau
 - Có thể có các công cụ sửa đổi quyền truy cập khác nhau
 - Có thể ném các ngoại lệ khác nhau
- Các phư ơng thức từ môt siêu kiểu có thể đư ợc nạp chòng trong môt kiểu con.
- Tính đa hình áp dụng cho ghi đè, không áp dụng cho quá tải.
- Kiểu đối tư ơng (không phải kiểu của biến tham chiếu) xác định phư ơng thức ghi đè nào đư ợc sử dụng trong thời gian chạy.
- Kiểu tham chiếu xác định phư ơng thức nạp chòng nào sẽ đư ợc sử dụng tại thời điểm biến dịch.

Đúc biến tham chiếu (Mục tiêu 7.3 của OCA)

- Có hai loại truyền biến tham chiếu: dự báo xuống và dự báo lên.
- Downcasting Nếu bạn có một biến tham chiếu tham chiếu đến một đối tượng kiểu con, bạn có thể gán nó cho một biến tham chiếu của kiểu con. Bạn phải thực hiện một phép ép kiểu rõ ràng để thực hiện việc này và kết quả là bạn có thể truy cập các thành viên của kiểu con bằng biến tham chiếu mới này.
- Upcasting Bạn có thể gán một biến tham chiếu cho một biến tham chiếu supertype một cách rõ ràng hoặc ngầm định. Đây là một hoạt động vốn dĩ an toàn vì việc gán hạn chế khả năng truy cập của biến mới.

Triển khai một giao diện (Mục tiêu 7.5 của OCA)

- Khi bạn triển khai một giao diện, bạn đang hoàn thành hợp đồng của nó.
- Bạn triển khai một giao diện bằng cách triển khai đúng và cụ thể tất cả các phương thức trừu tượng được xác định bởi giao diện.
- Một lớp duy nhất có thể triển khai nhiều giao diện.

Các loại lợi nhuận (Mục tiêu OCA 7.2 và 7.5)

- Các phương thức bị quá tải có thể thay đổi kiểu trả về; các phương thức ghi đè không thể, ngoại trừ trường hợp trả về phương sai.
- Các kiểu trả về tham chiếu đối tượng có thể chấp nhận null làm giá trị trả về.
- Mảng là một kiểu trả về hợp pháp, cả hai để khai báo và trả về dưới dạng một giá trị.
- Đối với các phương thức có kiểu trả về nguyên thủy, bất kỳ giá trị nào có thể được chuyển đổi ngầm thành kiểu trả về đều có thể được trả về.
- Không có gì có thể được trả lại từ một khoảng trống, nhưng bạn không thể trả lại gì. Bạn chỉ được phép nói return trong bất kỳ phương thức nào có kiểu trả về void để loại bỏ một phương thức sớm. Nhưng bạn không thể trả về không có gì từ một phương thức có kiểu trả về không void .
- Các phương thức có kiểu trả về tham chiếu đối tượng có thể trả về kiểu con.
- Các phương thức có kiểu trả về giao diện có thể trả về bất kỳ trình triển khai nào.

Trình xây dựng và Trình tạo (Mục tiêu OCA 6.3 và 7.4)

- Một hàm tạo luôn đư ợc gọi khi một đối tượng mới đư ợc tạo.
- Mỗi lớp cha trong cây kế thừa của một đối tượng sẽ có một hàm tạo đư ợc gọi.
- Mọi lớp, ngay cả một lớp trứu tư ơng, đều có ít nhất một hàm tạo.
- Các trình xây dựng phải có cùng tên với lớp.
- Các hàm tạo không có kiểu trả về. Nếu bạn thấy mã có kiểu trả về, thì đó là một phư ơng thức có cùng tên với lớp; nó không phải là một hàm tạo.
- Việc thực thi hàm tạo điển hình xảy ra như sau:
 - Hàm tạo gọi phư ơng thức khởi tạo lớp cha của nó, phư ơng thức này gọi phư ơng thức khởi tạo lớp cha của nó, và cứ tiếp tục như vậy cho đến phư ơng thức khởi tạo đối tượng .
 - Phư ơng thức khởi tạo đối tượng thực thi và sau đó quay trở lại phư ơng thức khởi tạo đang gọi, nó chạy đến khi hoàn thành và sau đó quay trở lại phư ơng thức khởi tạo đang gọi của nó, và cứ tiếp tục quay trở lại quá trình hoàn thành phư ơng thức khởi tạo của cá thể thực tế đang đư ợc tạo.
- Trình tạo có thể sử dụng bất kỳ công cụ sửa đổi quyền truy cập nào (thậm chí là riêng tư!).
- Trình biên dịch sẽ tạo một hàm tạo mặc định nếu bạn không tạo bất kỳ hàm tạo nào trong lớp của mình.
- Hàm tạo mặc định là một hàm tạo không đối số với một lệnh gọi no-arg tới super () .
- Câu lệnh đầu tiên của mọi phư ơng thức khởi tạo phải là một lệnh gọi tới this () (một hàm tạo đư ợc nạp chồng) hoặc tới super () .
- Trình biên dịch sẽ thêm lệnh gọi vào super () trừ khi bạn đã thực hiện lệnh gọi this () hoặc super () .
- Các thành viên phiên bản chỉ có thể truy cập đư ợc sau khi hàm tạo siêu chạy.
- Các lớp trứu tư ơng có các hàm tạo đư ợc gọi khi một lớp con cụ thể đư ợc khởi tạo.
- Các giao diện không có hàm tạo.
- Nếu lớp cha của bạn không có hàm tạo no-arg, bạn phải tạo một hàm tạo và chèn một lệnh gọi tới super () với các đối số phù hợp với các đối số của hàm tạo lớp cha.
- Các trình xây dựng không bao giờ đư ợc kế thừa; do đó chúng không thể bị ghi đè.

- Một phư ơng thức khởi tạo chỉ có thể đư ợc gọi trực tiếp bởi một phư ơng thức khởi tạo khác (sử dụng lời gọi tới super () hoặc this ()).
- Về các vấn đề với lệnh gọi this (): Chúng có
 - thể chỉ xuất hiện dưới dạng câu lệnh đầu tiên trong một hàm tạo.
 - Danh sách đối số xác định phư ơng thức khởi tạo nạp chòng nào đư ợc gọi.
 - Các hàm tạo có thể gọi các hàm tạo, v.v., như ng sớm hay muộn một trong số chúng sẽ gọi super () hoặc ngăn xếp sẽ phát nổ.
 - Các lệnh gọi this () và super () không thể nằm trong cùng một hàm tạo. Bạn có thể có cái này hoặc cái kia, nhưng không bao giờ có cả hai.

Khởi khởi tạo (Mục tiêu OCA 1.2 và 6.3-ish)

- Sử dụng các khởi init tĩnh – mã static { / * tại đây * / } –đối với mã bạn muốn chạy một lần, khi lớp đư ợc tải lần đầu tiên. Nhiều khởi chạy từ trên xuống.
- Sử dụng các khởi init bình thư ờng– { / * mã tại đây } –đối với mã bạn muốn chạy cho mọi phiên bản mới, ngay sau khi tắt cả các siêu cấu trúc đã chạy.
Một lần nữa, nhiều khởi chạy từ đầu lớp xuống.

Tin học (Mục tiêu 6.2 của OCA)

- Sử dụng các phư ơng thức tĩnh để thực hiện các hành vi không bị ảnh hư ờng bởi trạng thái của bất kỳ trư ờng hợp nào.
- Sử dụng các biến tĩnh để giữ dữ liệu dành riêng cho lớp chứ không phải trư ờng hợp cụ thể – sẽ chỉ có một bản sao của một biến tĩnh .
- Tắt cả các thành viên tĩnh thuộc về lớp, không thuộc bất kỳ trư ờng hợp nào.
- Một phư ơng thức tĩnh không thể truy cập trực tiếp vào một biến cá thể.
- Sử dụng toán tử dot để truy cập các thành viên tĩnh , như ng hãy nhớ rằng việc sử dụng một biến tham chiếu với toán tử dot thực sự là một thủ thuật cú pháp và trình biên dịch sẽ thay thế tên lớp cho biến tham chiếu; vì ví dụ:

`d.doStuff ();`

trở thành

Dog.doStuff();

- Để gọi phương thức tĩnh của giao diện, hãy sử dụng cú pháp MyInterface.doStuff().
- không thể ghi đè các phương thức tĩnh, như chúng có thể được định nghĩa lại.

TỰ KIỂM TRA

1. Cho:

```
giao diện trừu tượng chung Frobinate {public void  
twiddle (String s); }
```

Đó là một lớp học chính xác? (Chọn tất cả các áp dụng.)

- A. public abstract class Frob implements Frobinate {
 public abstract void twiddle(String s) { }
}
- B. public abstract class Frob implements Frobinate { }
- C. public class Frob extends Frobinate {
 public void twiddle(Integer i) { }
}
- D. public class Frob implements Frobinate {
 public void twiddle(Integer i) { }
}
- E. public class Frob implements Frobinate {
 public void twiddle(String i) { }
 public void twiddle(Integer s) { }
}

2. Đưa ra:

```
class Top {  
    public Top(String s) { System.out.print("B"); }  
}  
public class Bottom2 extends Top {  
    public Bottom2(String s) { System.out.print("D"); }  
    public static void main(String [] args) {  
        new Bottom2("C");  
        System.out.println(" ");  
    }  
}
```

Kết quả là gì?

Kết quả là gì?

- A. BD
- B. DB
- C. BDC
- D. DBC
- E. Biên dịch không thành công

3. Đưa ra:

```
class Clidder {  
    private final void flipper() { System.out.println("Clidder"); }  
}  
public class Clidlet extends Clidder {  
    public final void flipper() { System.out.println("Clidlet"); }  
    public static void main(String [] args) {  
        new Clidlet().flipper();  
    }  
}
```

Kết quả là gì?

- A. Clidlet
- B. Clidder
- C. Clidder
 Clidlet
- D. Clidlet
 Clidder
- E. Biên dịch không thành

công Lưu ý đặc biệt: Câu hỏi tiếp theo mô phỏng một cách thô sơ kiểu câu hỏi
đư ợc gọi là "kéo và thả". Trong suốt kỳ thi SCJP 6, các câu hỏi kéo và thả đã đư ợc
đưa vào kỳ thi. Kể từ mùa xuân năm 2014, Oracle KHÔNG đưa bất kỳ câu hỏi kéo và thả nào
vào các kỳ thi Java của mình, như ng đê phòng trư ờng hợp chính sách của Oracle thay đổi,
chúng tôi đã để lại một số câu hỏi trong cuốn sách.

4. Sử dụng các đoạn bên dưới, hoàn thành đoạn mã sau để nó biên dịch.

Lưu ý rằng bạn có thể không phải điền vào tất cả các vị trí.

Mã số:

```

class AgedP {
    public AgedP(int x) {
    }
}
public class Kinder extends AgedP {
    public Kinder(int x) {
    }
}

```

Các đoạn: Sử dụng các đoạn sau không hoặc nhiều lần:

AgedP	super	this	
()	{	}
;			

5. Đề ra:

```

class Bird {
    { System.out.print("b1 "); }
    public Bird() { System.out.print("b2 "); }
}
class Raptor extends Bird {
    static { System.out.print("r1 "); }
    public Raptor() { System.out.print("r2 "); }
    { System.out.print("r3 "); }
    static { System.out.print("r4 "); }
}
class Hawk extends Raptor {
    public static void main(String[] args) {
        System.out.print("pre ");
        new Hawk();
        System.out.println("hawk ");
    }
}

```

Kết quả là gì?

- A. pre b1 b2 r3 r2 hawk B.
- pre b2 b1 r2 r3 hawk C. pre
- b2 b1 r2 r3 hawk r1 r4 D. r1 r4
- pre b1 b2 r3 r2 hawk

E. r1 r4 pre b2 b1 r2 r3 hawk F.
pre r1 r4 b1 b2 r3 r2 hawk G. pre
r1 r4 b2 b1 r2 r3 hawk H. Không
thể đoán tru ớc thứ tự đầu ra I. Biên dịch
không thành công Lưu ý: Bạn có thể sẽ
không bao giờ xem nhiều lựa chọn này trong kỳ thi thật!

6. Cho những điều sau:

```
1. class X { void do1() { } }
```

```
2. class Y extends X { void do2() { } }
```

```
3.
```

```
4. class Chrome {
```

```
5.     public static void main(String [] args) {
```

```
6.         X x1 = new X();
```

```
7.         X x2 = new Y();
```

```
8.         Y y1 = new Y();
```

```
9.         // insert code here
```

```
10.    } }
```

Phần nào sau đây, được chèn ở dòng 9, sẽ biên dịch? (Chọn tất cả các áp dụng.)

- A. x2.do2 ();
- B. (Y) x2.do2 ();
- C. ((Y) x2) .do2 ();
- D. Không câu lệnh nào ở trên sẽ biên dịch

7. Cho:

```
public class Locomotive {  
    Locomotive() { main("hi"); }  
  
    public static void main(String[] args) {  
        System.out.print("2 ");  
    }  
    public static void main(String args) {  
        System.out.print("3 " + args);  
    }  
}
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

- A. 2 sẽ được bao gồm trong đầu ra

B. 3 sẽ được đưa vào đầu ra C. hi
sẽ được đưa vào đầu ra D. Biên dịch
không thành công E. Một ngoại lệ được
đưa ra trong thời gian chạy

8. Cho:

```
3. class Dog {  
4.     public void bark() { System.out.print("woof "); }  
5. }  
6. class Hound extends Dog {  
7.     public void sniff() { System.out.print("sniff "); }  
8.     public void bark() { System.out.print("howl "); }  
9. }  
10. public class DogShow {  
11.     public static void main(String[] args) { new DogShow().go(); }  
12.     void go() {  
13.         new Hound().bark();  
14.         ((Dog) new Hound()).bark();  
15.         ((Dog) new Hound()).sniff();  
16.     }  
17. }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

- A. tru hú đánh hơi
- B. hú gâu gâu
- C. howl howl theo sau bởi một ngoại lệ
- D. howl woof theo sau bởi một ngoại lệ E.

Biên dịch không thành công với lỗi ở dòng 14

F. Biên dịch không thành công với lỗi ở dòng 15

9. Cho:

```
3. public class Redwood extends Tree {  
4.     public static void main(String[] args) {  
5.         new Redwood().go();  
6.     }  
7.     void go() {  
8.         go2(new Tree(), new Redwood());  
9.         go2((Redwood) new Tree(), new Redwood());  
10.    }  
11.    void go2(Tree t1, Redwood r1) {  
12.        Redwood r2 = (Redwood)t1;  
13.        Tree t2 = (Tree)r1;  
14.    }  
15. }  
16. class Tree { }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

- A. Một ngoại lệ được đưa ra trong thời gian chạy
- B. Mã biên dịch và chạy không có đầu ra
- C. Biên dịch không thành công với lỗi ở dòng 8 D.
- Biên dịch không thành công với lỗi ở dòng 9 E.
- Biên dịch không thành công với lỗi ở dòng 12 F.
- Biên dịch không thành công với lỗi ở dòng 13

10. Cho:

```
3. public class Tenor extends Singer {  
4.     public static String sing() { return "fa"; }  
5.     public static void main(String[] args) {  
6.         Tenor t = new Tenor();  
7.         Singer s = new Tenor();  
8.         System.out.println(t.sing() + " " + s.sing());  
9.     }  
10. }  
11. class Singer { public static String sing() { return "la"; } }
```

Kết quả là gì?

- A. fa fa
- B. fa la
- C. la la
- D. Biên dịch không thành công
- E. Một ngoại lệ được đưa ra trong thời gian chạy

11. Cho:

```
3. class Alpha {  
4.     static String s = " ";  
5.     protected Alpha() { s += "alpha "; }  
6. }  
7. class SubAlpha extends Alpha {  
8.     private SubAlpha() { s += "sub "; }  
9. }  
10. public class SubSubAlpha extends Alpha {  
11.     private SubSubAlpha() { s += "subsub "; }  
12.     public static void main(String[] args) {  
13.         new SubSubAlpha();  
14.         System.out.println(s);  
15.     }  
16. }
```

Kết quả là gì?

- A. subub
- B. sub sub
- C. alpha subsub D.
- alpha sub subsub E. Biên
- dịch không thành công F.

Một ngoại lệ đưa ra trong thời gian chạy

12. Cho:

```
3. class Alpha {  
4.     static String s = " ";  
5.     protected Alpha() { s += "alpha "; }  
6. }  
7. class SubAlpha extends Alpha {  
8.     private SubAlpha() { s += "sub "; }  
9. }  
10. public class SubSubAlpha extends Alpha {  
11.     private SubSubAlpha() { s += "subsub "; }  
12.     public static void main(String[] args) {  
13.         new SubSubAlpha();  
14.         System.out.println(s);  
15.     }  
16. }
```

Kết quả là gì?

- A. h hn x

- B. hn xh
- C. bh hn x
- D. b hn xh
- E. bn xh hn x
- F. b bn xh hn x
- G. bn xbh hn x

H. Biên dịch không thành công

13. Cho:

```
3. class Mammal {  
4.     String name = "furry ";  
5.     String makeNoise() { return "generic noise"; }  
6. }  
7. class Zebra extends Mammal {  
8.     String name = "stripes ";  
9.     String makeNoise() { return "bray"; }  
10. }  
11. public class ZooKeeper {  
12.     public static void main(String[] args) { new ZooKeeper().go(); }  
13.     void go() {  
14.         Mammal m = new Zebra();  
15.         System.out.println(m.name + m.makeNoise());  
16.     }  
17. }
```

Kết quả là gì?

- A. tiếng ồn giống như lông vũ
- B. tiếng ồn sọc bray
- C. tiếng ồn chung có lông
- D. tiếng ồn chung của các đư ờng sọc
- E. Quá trình biên dịch không thành công

Một ngoại lệ đư ợc đư a ra trong thời gian chạy

14. Cho:

```
1. interface FrogBoilable {  
2.     static int getCtoF(int cTemp) {  
3.         return (cTemp * 9 / 5) + 32;  
4.     }  
5.     default String hop() { return "hopping"; }
```

```

6. }
7. public class DontBoilFrogs implements FrogBoilable {
8.     public static void main(String[] args) {
9.         new DontBoilFrogs().go();
10.    }
11.    void go() {
12.        System.out.print(hop());
13.        System.out.println(getCtoF(100));
14.        System.out.println(FrogBoilable.getCtoF(100));
15.        DontBoilFrogs dbf = new DontBoilFrogs();
16.        System.out.println(dbf.getCtoF(100));
17.    }
18. }

```

Kết quả là gì? (Chọn tất cả các áp dụng.)

A. nhảy 212 B.

Biên dịch không thành công do lỗi ở dòng 2 C.

Biên dịch không thành công do lỗi ở dòng 5 D.

Biên dịch không thành công do lỗi ở dòng 12 E.

Biên dịch không thành công do lỗi ở dòng 13 F.

Biên dịch không thành công do lỗi trên dòng 14

G. Biên dịch không thành công do lỗi trên dòng 16

15. Cho:

```

interface I1 {
    default int doStuff() { return 1; }
}
interface I2 {
    default int doStuff() { return 2; }
}
public class MultiInt implements I1, I2 {
    public static void main(String[] args) {
        new MultiInt().go();
    }
    void go() {
        System.out.println(doStuff());
    }
    int doStuff() {
        return 3;
    }
}

```

Kết quả là gì?

- A. 1
- B. 2
- C. 3
- D. Đầu ra không thể đoán trước
- E. Biên dịch không thành công
- F. Một ngoại lệ được đưa ra trong thời gian chạy

16. Cho:

```
interface MyInterface {  
    default int doStuff() {  
        return 42;  
    }  
}  
public class IfaceTest implements MyInterface {  
    public static void main(String[] args) {  
        new IfaceTest().go();  
    }  
    void go() {  
        // INSERT CODE HERE  
    }  
    public int doStuff() {  
        return 43;  
    }  
}
```

(Các) dòng mã nào, được chèn độc lập tại // CHÈN MÃ TẠI ĐÂY, sẽ cho phép mã để biên dịch? (Chọn tất cả các áp dụng.)

- A. System.out.println ("lớp:" + doStuff ());
- B. System.out.println ("iface:" + super.doStuff ());
- C. System.out.println ("iface:" +
MyInterface.super.doStuff ());
- D. System.out.println ("iface:" + MyInterface.doStuff ());
- E. System.out.println ("iface:" +
super.MyInterface.doStuff ());
- F. Không có dòng nào, A - E sẽ cho phép mã biên dịch

CÂU TRẢ LỜI TỰ KIỂM TRA

1. E đúng. B đúng vì một lớp trừu tượng không cần triển khai bất kỳ hoặc tất cả các phương thức của giao diện. E đúng vì lớp thực thi phương thức giao diện và bổ sung quá tải phương thức twiddle () .
- A, C và D không chính xác. A không chính xác vì các phương thức trừu tượng không có phần thân. C không chính xác vì các lớp triển khai các giao diện; họ không mở rộng chúng. D không chính xác vì nạp chồng một phương thức sẽ không thực hiện nó. (Mục tiêu của OCA 7.1 và 7.5)
2. E đúng. Lời gọi hàm super () ngụ ý trong hàm tạo của Bottom2 không thể được thỏa mãn vì không có hàm tạo no-arg trong Top. Trình biên dịch chỉ tạo ra một phương thức khởi tạo đối số, mặc định nếu lớp không có phương thức khởi tạo nào được định nghĩa rõ ràng.
- A, B, C và D là không chính xác dựa trên các điều trên. (Mục tiêu 6.3 của OCA)
3. A đúng. Mặc dù không thể ghi đè phương thức cuối cùng, như trong trường hợp, phương thức là riêng tư và do đó, ẩn. Hiệu quả là một trình lật phương thức mới, có thể truy cập được, được tạo ra. Do đó, không có đa hình nào xảy ra trong ví dụ này, phương thức được gọi đơn giản là phương thức của lớp con và không xảy ra lỗi.
- B, C, D và E không chính xác dựa trên phần trước. (Mục tiêu 7.2 của OCA)

Lưu ý đặc biệt: Câu hỏi tiếp theo này mô phỏng một cách thô sơ kiểu câu hỏi được gọi là "kéo và thả". Trong suốt kỳ thi SCJP 6, các câu hỏi kéo và thả đã được đưa vào kỳ thi. Kể từ mùa xuân năm 2014, Oracle KHÔNG đưa bất kỳ câu hỏi kéo và thả nào vào các kỳ thi Java của mình, như ng để phòng trường hợp chính sách của Oracle thay đổi, chúng tôi đã để lại một số câu hỏi trong cuốn sách.

4. Đây là câu trả lời:

```
class AgedP {
    AgedP() {}
    public AgedP(int x) {
    }
}
public class Kinder extends AgedP {
    public Kinder(int x) {
        super();
    }
}
```

Vì không có ô có thể thả xuống cho biến x và các dấu ngoặc đơn (trong

Phư ơng thức khởi tạo Kinder) đã đư ợc đặt sẵn và trống, không có cách nào để tạo một lời gọi đến phư ơng thức khởi tạo lớp cha nhận đối số. Do đó, khả năng còn lại duy nhất là tạo lời gọi đến hàm tạo lớp cha no-arg. Điều này đư ợc thực hiện dư ới dạng super () ;. Không đư ợc để trống dòng vì dấu ngoặc đã đư ợc đặt sẵn. Hơn nữa, vì hàm tạo lớp cha đư ợc gọi là phiên bản không có đối số, nên hàm tạo này phải đư ợc tạo. Nó sẽ không đư ợc tạo bởi trình biên dịch vì một phư ơng thức khởi tạo khác đã có mặt. (Mục tiêu 6.3 và 7.4 của OCA) Lưu ý: Như bạn có thể thấy, nhiều câu hỏi kiểm tra Mục tiêu 7.1 của OCA, chúng tôi sẽ ngừng đề cập đến mục tiêu 7.1.

5. D đúng. Các khối init tĩnh đư ợc thực thi tại thời điểm tải lớp; các khối init instance chạy ngay sau lệnh gọi tới super () trong một hàm tạo. Khi nhiều khối init của một loại duy nhất xuất hiện trong một lớp, chúng sẽ chạy theo thứ tự, từ trên xuống.
 A, B, C, E, F, G, H và tôi không chính xác dựa trên những điều trên. Lưu ý: Có thể bạn sẽ không bao giờ thấy nhiều lựa chọn này trong kỳ thi thật! (Mục tiêu 6.3 của OCA)
6. C đúng. Trước khi bạn có thể gọi phư ơng thức do2 của Y , bạn phải ép x2 thành kiểu Y.
 A, B và D là không chính xác dựa trên phần trước. B trông giống như một phép ép kiểu thích hợp, như ng không có tập hợp dấu ngoặc thứ hai, trình biên dịch nghĩ rằng đó là một câu lệnh chư a hoàn chỉnh. (Mục tiêu 7.3 của OCA)
7. A đúng. Việc quá tải main () là hợp pháp. Vì không có tru ờng hợp nào của Đầu máy đư ợc tạo, hàm tạo không chạy và phiên bản quá tải của main () không chạy.
 B, C, D và E không chính xác dựa trên phần trước. (Mục tiêu 1.3 và 6.3 của OCA)
8. F đúng. Class Dog không có phư ơng pháp đánh hơi .
 A, B, C, D và E không chính xác dựa trên thông tin trên. (OCA Mục tiêu 7.2 và 7.3)
9. A đúng. Một ClassCastException sẽ đư ợc ném ra khi mã cốc gắng truyền một Cây xuống Redwood.
 B, C, D, E và F không chính xác dựa trên thông tin trên. (Mục tiêu 7.3 của OCA)
10. B đúng. Mã đúng, như ng đa hình không áp dụng cho các phư ơng thức tĩnh .

A, C, D và E không chính xác dựa trên thông tin trên. (OCA
Mục tiêu 6.2 và 7.2)

11. C đúng. Hãy coi chừng, vì SubSubAlpha mở rộng Alpha! Vì mã không cố gắng tạo SubAlpha, nên hàm tạo riêng trong SubAlpha vẫn ổn.

A, B, D, E và F không chính xác dựa trên thông tin trên. (OCA
Mục tiêu 6.3 và 7.2)

12. C đúng. Hãy nhớ rằng các hàm tạo gọi lớp cha của chúng
các hàm tạo, thực thi đầu tiên và các hàm tạo có thể đư ợc nạp chồng.
 A, B, D, E, F, G và H là không chính xác dựa trên thông tin trên.
(Mục tiêu 6,3 và 7,4 của OCA)

13. A đúng. Tính đa hình chỉ dành cho các phu ơng thức thể hiện, không phải đối tư ợng
biến.

B, C, D, E và F không chính xác dựa trên thông tin trên. (Mục tiêu 6.3
của OCA)

14. E và G đúng. Cả hai dòng mã này đều không sử dụng đúng
cú pháp để gọi phu ơng thức tĩnh của giao diện.

A, B, C, D và F không chính xác dựa trên thông tin trên. (OCP
Mục tiêu 6.2 và 7.5)

15. E đúng. Đây là một loại câu hỏi mèo; phu ơng pháp thực hiện phải đư ợc đánh
dấu công khai. Nếu đúng như vậy, tất cả các mã khác đều hợp pháp và kết
quả đầu ra sẽ là 3. Nếu bạn hiểu tất cả các quy tắc đa kế thừa và chỉ bỏ sót
công cụ sửa đổi truy cập, hãy tự ghi nhận một nửa.

A, B, C, D và F không chính xác dựa trên thông tin trên. (Mục tiêu 7.5
của OCP)

16. A và C đúng. A sử dụng cú pháp chính xác để gọi phu ơng thức của lớp và C sử
dụng cú pháp chính xác để gọi phu ơng thức mặc định đã đư ợc nạp chồng của
giao diện .

B, D, E và F không chính xác. (Mục tiêu 7.5 của OCP)



3

Bài tập

CHỨNG NHẬN

MỤC TIÊU

- Sử dụng các thành viên trong lớp
- Hiểu truyền nguyên thủy • Hiểu phạm vi biến • Phân biệt giữa biến nguyên thủy và biến tham chiếu
- Xác định ảnh hưởng của việc chuyển các biến vào các phư ơng thức • Hiểu Vòng đời đối tượng và Thu gom rác

✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

Stack and Heap – Đánh giá nhanh

Đối với hầu hết mọi người, hiểu những kiến thức cơ bản về ngăn xếp và đống sẽ giúp hiểu các chủ đề như truyền đối số, đa hình, chủ đề, ngoại lệ và thu gom rác dễ dàng hơn nhiều. Trong phần này, chúng ta sẽ đi vào tổng quan, như ng chúng ta sẽ mở rộng các chủ đề này nhiều lần nữa trong suốt cuốn sách.

Đối với hầu hết các phần, các phần khác nhau (phư ơng thức, biến và đối tượng) của Java chư ơng trình nằm ở một trong hai nơi trong bộ nhớ: ngăn xếp hoặc đống. Hiện tại, chúng tôi chỉ quan tâm đến ba loại thứ – biến cá thể, biến cục bộ và đối tượng:

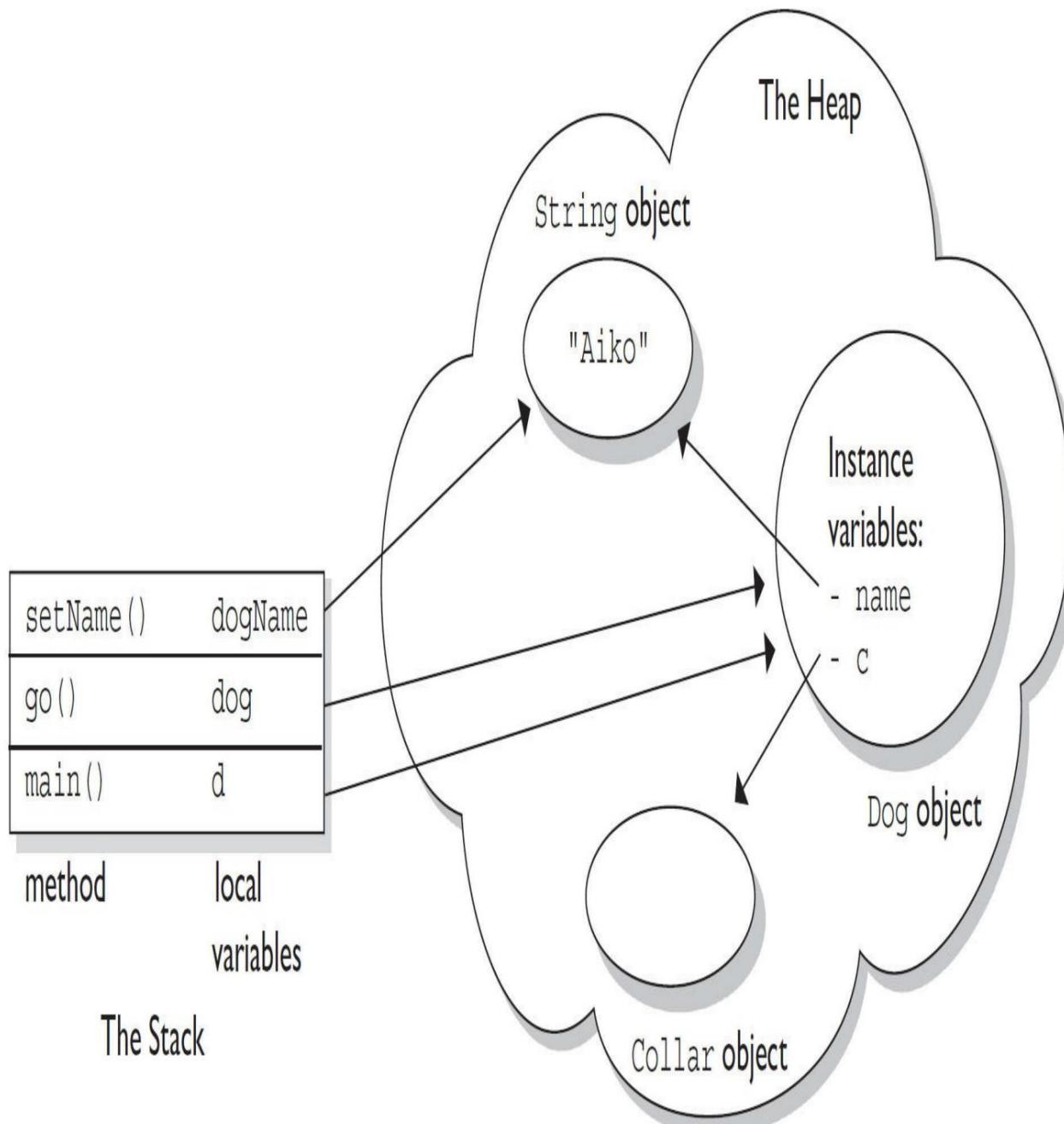
- Các biến cá thể và các đối tượng sống trên heap.
- Các biến cục bộ sống trên ngăn xếp.

Chúng ta hãy xem một chư ơng trình Java và cách các phần khác nhau của nó được tạo ra và ánh xạ vào ngăn xếp và đống:

ánh xạ vào ngăn xếp và đống:

```
1. class Collar { }
2.
3. class Dog {
4.     Collar c;                      // instance variable
5.     String name;                  // instance variable
6.
7.     public static void main(String [] args) {
8.
9.         Dog d;                      // local variable: d
10.        d = new Dog();
11.        d.go(d);
12.    }
13.    void go(Dog dog) {           // local variable: dog
14.        c = new Collar();
15.        dog.setName("Aiko");
16.    }
17.    void setName(String dogName) { // local var: dogName
18.        name = dogName;
19.        // do more stuff
20.    }
21. }
```

[Hình 3-1](#) cho thấy trạng thái của ngăn xếp và đống khi chương trình đạt đến dòng 19. Sau đây là một số điểm chính:



HÌNH 3-1 Mô hình về ngăn xếp và đống

- Dòng 7 – main () được đặt trên ngăn xếp.
- Dòng 9 – Biến tham chiếu d được tạo trên ngăn xếp, như ngôn ngữ C có đối tượng Dog.
- Dòng 10 – Một đối tượng Dog mới được tạo trên heap và được gán cho biến tham chiếu d.

- Dòng 11 – Một bản sao của biến tham chiếu d được chuyển cho phương thức go () .
- Dòng 13 – Phương thức go () được đặt trên ngăn xếp, với tham số dog là một biến cục bộ.
- Dòng 14 – Một đối tượng Collar mới được tạo trên heap và được gán cho biến thẻ hiện của Dog .
- Dòng 17 – setName () được thêm vào ngăn xếp, với tham số dogName là biến cục bộ của nó.
- Dòng 18 – Biến cá thể tên bây giờ cũng tham chiếu đến đối tượng String .
- Lưu ý rằng hai biến cục bộ khác nhau tham chiếu đến cùng một đối tượng Dog .
- Lưu ý rằng một biến cục bộ và một biến phiên bản đều tham chiếu đến cùng Chuỗi Aiko.
- Sau khi Dòng 19 hoàn thành, setName () hoàn thành và được xóa khỏi ngăn xếp. Tại thời điểm này, biến local dogName cũng biến mất, mặc dù đối tượng String mà nó tham chiếu vẫn nằm trên heap.

MỤC TIÊU XÁC NHẬN

Chữ viết, Bài tập và Biến số (Mục tiêu 2.1, 2.2 và 2.3 của OCA)

2.1 Khai báo và khởi tạo biến (bao gồm ép kiểu dữ liệu nguyên thủy).

2.2 Phân biệt giữa biến tham chiếu đối tượng và biến nguyên thủy.

2.3 Biết cách đọc hoặc ghi vào các trử ờng đối tượng.

Giá trị văn bản cho tất cả các loại nguyên thủy

Một ký tự nguyên thủy chỉ đơn thuần là một biểu diễn mã nguồn của các kiểu dữ liệu nguyên thủy – nói cách khác, một số nguyên, số dấu phẩy động, boolean hoặc ký tự mà bạn nhập vào trong khi viết mã. Sau đây là các ví dụ về các chữ nguyên thủy:

```
'b'          // char literal
42           // int literal
false        // boolean literal
2546789.343 // double literal
```

Chữ số nguyên Có bốn

cách để biểu diễn số nguyên trong ngôn ngữ Java: thập phân (cơ số 10), bát phân (cơ số 8), hệ thập lục phân (cơ số 16), và đối với Java 7, hệ nhị phân (cơ số 2). Hầu hết các câu hỏi thi với các chữ số nguyên sử dụng các biểu diễn thập phân, như ng một số ít sử dụng hệ bát phân, thập lục phân hoặc nhị phân rất đáng để nghiên cứu. Mặc dù tỷ lệ cự ợc mà bạn sẽ thực sự sử dụng hệ bát phân trong thế giới thực là rất nhỏ, như ng chúng đư ợc đư a vào kỳ thi chỉ để giải trí. Trước khi xem xét bốn cách để biểu diễn số nguyên, tru ớc tiên chúng ta hãy thảo luận về một tính năng mới đư ợc thêm vào Java 7: các chữ có dấu gạch dư ới.

Chữ số có dấu gạch dư ới Kể từ Java 7, các chữ số có thể đư ợc khai báo bằng cách sử dụng các ký tự dấu gạch dư ới (_), bè ngoài là để cải thiện khả năng đọc. Hãy so sánh một khai báo tru ớc Java 7 với một khai báo Java 7 dễ đọc hơn:

```
int pre7 = 1000000;      // pre Java 7 - we hope it's a million
int with7 = 1_000_000;    // much clearer!
```

Quy tắc chính mà bạn phải theo dõi là bạn KHÔNG THỂ sử dụng ký tự gạch dư ới ở đầu hoặc cuối của từ. Gợi ý tiềm năng ở đây là bạn có thể tự do sử dụng dấu gạch dư ới ở những nơi "kỳ lạ":

```
int i1 = _1_000_000;    // illegal, can't begin with an "_"
int i2 = 10_0000_0;     // legal, but confusing
```

Lưu ý cuối cùng, hãy nhớ rằng bạn có thể sử dụng ký tự gạch dư ới cho bất kỳ loại số nào (bao gồm cả số kép và số nổi), như ng đối với số kép và số nổi, bạn KHÔNG THỂ thêm ký tự gạch dư ới trực tiếp bên cạnh dấu thập phân hoặc bên cạnh dấu X hoặc B ở dạng số hex hoặc số nhị phân (sắp ra mắt).

Decimal Literals Số nguyên thập phân không cần giải thích; bạn đã sử dụng chúng từ lớp một trở lên. Rất có thể bạn không giữ số séc của mình trong hex. (Nếu bạn làm vậy, có một nhóm Geeks Anonymous [GA] sẵn sàng trợ giúp.) Trong ngôn ngữ Java, chúng đư ợc trình bày nguyên dạng, không có bất kỳ tiền tố nào, như sau:

```
int length = 343;
```

Chữ nhị phân Cũng mới đổi với Java 7 là việc bổ sung các chữ nhị phân. Các ký tự nhị phân chỉ có thể sử dụng các chữ số 0 và 1. Các ký tự nhị phân phải bắt đầu bằng 0B hoặc 0b, như đư ợc hiển thị:

```
int b1 = 0B101010; // set b1 to binary 101010 (decimal 42)
int b2 = 0b00011; // set b2 to binary 11 (decimal 3)
```

Chữ số bát phân Các số nguyên bát phân chỉ sử dụng các chữ số từ 0 đến 7. Trong Java, bạn biểu diễn một số nguyên ở dạng bát phân bằng cách đặt một số 0 trước số, như sau:

```
class Octal {
    public static void main(String [] args) {
        int six = 06; // Equal to decimal 6
        int seven = 07; // Equal to decimal 7
        int eight = 010; // Equal to decimal 8
        int nine = 011; // Equal to decimal 9
        System.out.println("Octal 010 = " + eight);
    }
}
```

Bạn có thể có tối đa 21 chữ số trong một số bát phân, không bao gồm số 0 đứng đầu. Nếu chúng ta chạy chương trình trước đó, nó sẽ hiển thị như sau:

```
Hệ bát phân 010 = 8
```

Chữ số thập lục phân Các số thập lục phân (viết tắt là hex) được xây dựng bằng cách sử dụng 16 ký hiệu riêng biệt. Bởi vì chúng tôi chưa bao giờ phát minh ra các ký hiệu có một chữ số cho các số từ 10 đến 15, nên chúng tôi sử dụng các ký tự chữ cái để biểu thị các chữ số này. Đếm từ 0 đến 15 trong hex trông như thế này:

```
0 1 2 3 4 5 6 7 8 9 abcdef
```

Java sẽ chấp nhận chữ hoa hoặc chữ thường cho các chữ số phụ (một trong các vài chỗ Java không phân biệt chữ hoa chữ thường!). Bạn được phép tối đa 16 chữ số trong một số thập lục phân, không bao gồm tiền tố 0x (hoặc 0X) hoặc phần mở rộng hậu tố tùy chọn L, sẽ được giải thích một chút ở phần sau của chương. Tất cả các phép gán hệ thập lục phân sau đây đều hợp pháp:

```
class HexTest {
    public static void main (String [] args) {
        int x = 0X0001;
        int y = 0x7fffffff;
        int z = 0xDEadCafe;
        System.out.println("x = " + x + " y = " + y + " z = " + z);
    }
}
```

Chạy HexTest tạo ra kết quả sau:

```
x = 1 y = 2147483647 z = -559035650
```

Đừng để bị nhầm lẫn bởi những thay đổi trong trường hợp cho một chữ số thập lục phân hoặc x đứng trước nó. 0XCAFE và 0xcafe đều hợp pháp và có giá trị như nhau.

Tất cả bốn chữ số nguyên (nhị phân, bát phân, thập phân và thập lục phân) được định nghĩa là int theo mặc định, nhưng chúng cũng có thể được chỉ định miễn là bằng cách đặt hậu tố L hoặc l sau số:

```
long jo = 110599L; dài  
như vậy = 0xFFFFl; // Lưu ý chữ thư ờng 'l'
```

Chữ số dấu phẩy động Các số

dấu phẩy động được định nghĩa là một số, một ký hiệu thập phân và các số khác đại diện cho phân số. Trong ví dụ sau, số 11301874.9881024 là giá trị chữ:

```
gấp đôi d = 11301874.9881024;
```

Các ký tự dấu phẩy động được định nghĩa là kép (64 bit) theo mặc định, vì vậy nếu bạn muốn gán một ký tự dấu phẩy động cho một biến kiểu float (32 bit), bạn phải gắn hậu tố F hoặc f vào số. Nếu bạn không làm điều này, trình biên dịch sẽ phản nàn về việc có thể mất độ chính xác, bởi vì bạn đang cố gắng lắp một số vào một "vùng chứa" (có khả năng) kém chính xác hơn. Hậu tố F cung cấp cho bạn một cách để nói với trình biên dịch, "Này, tôi biết mình đang làm gì và tôi sẽ chấp nhận rủi ro, cảm ơn bạn rất nhiều."

```
float f = 23.467890;           // Compiler error, possible loss  
                                // of precision  
float g = 49837849.029847F; // OK; has the suffix "F"
```

Bạn cũng có thể tùy chọn đính kèm D hoặc d vào các ký tự kép, nhưng không cần thiết vì đây là hành vi mặc định.

```
double d = 110599.995011D; // Optional, not required  
double g = 987.897;        // No 'D' suffix, but OK because the  
                            // literal is a double by default
```

Tìm các ký tự số bao gồm dấu phẩy; đây là một ví dụ:

```
int x = 25,343;             // Won't compile because of the comma
```

Boolean Literals

Boolean Literals

Các ký tự boolean là biểu diễn mã nguồn cho các giá trị boolean. Giá trị boolean chỉ có thể được xác định là true hoặc false. Mặc dù trong C (và một số ngôn ngữ khác), người ta thường sử dụng số để biểu thị đúng hoặc sai, điều này sẽ không hoạt động trong Java. Một lần nữa, hãy lặp lại sau tôi: "Java không phải là C."

```
boolean t = true; // Legal
boolean f = 0;   // Compiler error!
```

Hãy chú ý đến các câu hỏi sử dụng số mà yêu cầu boolean. Bạn có thể thấy một kiểm tra if sử dụng một số, như sau:

```
int x = 1; if (x) {} // Lỗi trình biên dịch!
```

Ký tự chữ

Một ký tự char được biểu thị bằng một ký tự duy nhất trong dấu ngoặc kép:

```
char a = 'a';
char b = '@';
```

Bạn cũng có thể nhập giá trị Unicode của ký tự, sử dụng Unicode ký hiệu tiền tố giá trị bằng \ u, như sau:

```
char letterN = '\ u004E'; // Chữ cái 'N'
```

Hãy nhớ rằng, các ký tự chỉ là các số nguyên không dấu 16 bit bên dưới. Điều đó có nghĩa là bạn có thể gán một số theo nghĩa đen, giả sử nó sẽ phù hợp với phạm vi 16 bit không dấu (0 đến 65535). Ví dụ: tất cả những điều sau đây đều hợp pháp:

```
char a = 0x892;           // hexadecimal literal
char b = 982;             // int literal
char c = (char)70000;      // The cast is required; 70000 is
                           // out of char range
char d = (char) -98;       // Ridiculous, but legal
```

Và những điều sau đây không hợp pháp và tạo ra lỗi trình biên dịch:

```
char e = -29;           // Possible loss of precision; needs a cast
char f = 70000;          // Possible loss of precision; needs a cast
```

Bạn cũng có thể sử dụng mã thoát (dấu gạch chéo ngược) nếu bạn muốn đại diện cho một ký tự không thể nhập dưới dạng một chữ, bao gồm các ký tự cho dòng cấp dữ liệu, dòng mới, tab ngang, xóa lùi và dấu ngoặc kép:

```
char c = '\"'; // A double quote
char d = '\n'; // A newline
char tab = '\t'; // A tab
```

Các giá trị theo nghĩa đen cho chuỗi

Một ký tự chuỗi là một biểu diễn mã nguồn của một giá trị của một đối tượng Chuỗi . Sau đây là một ví dụ về hai cách để biểu diễn một chuỗi ký tự:

```
String s = "Bill Joy";
System.out.println("Bill" + " Joy");
```

Mặc dù các chuỗi không phải là nguyên thủy, nhưng chúng được bao gồm trong phần này vì chúng có thể được biểu diễn dưới dạng các ký tự – nói cách khác, chúng có thể được nhập trực tiếp vào mã. Kiểu không trực quan duy nhất khác có biểu diễn theo nghĩa đen là một mảng, chúng ta sẽ xem xét ở phần sau của chương.

Chủ đề t = ??? // giá trị chữ nào có thể có ở đây?

Người điều hành nhiệm vụ

Việc gán giá trị cho một biến có vẻ đủ đơn giản; bạn chỉ cần gán nội dung ở bên phải của dấu = cho biến ở bên trái. Vâng, chắc chắn, nhưng đừng mong đợi được thử nghiệm trên cái gì đó như thế này:

```
x = 6;
```

Không, bạn sẽ không được kiểm tra các bài tập không có trí tuệ (thuật ngữ kỹ thuật). Bạn Tuy nhiên, sẽ được kiểm tra trên các bài tập phức tạp hơn liên đến các biểu thức phức tạp và ép kiểu. Chúng ta sẽ xem xét cả phép gán biến nguyên thủy và tham chiếu. Như ng trước kia bắt đầu, hãy sao lưu và xem xét bên trong một biến. Biến là gì? Biến và giá trị của nó có liên quan như thế nào?

Các biến chỉ là những người nắm giữ bit với một kiểu được chỉ định. Bạn có thể có một ngăn giữ int , một ngăn giữ đôi , một ngăn giữ Nút , và thậm chí một ngăn giữ Chuỗi [] . Bên trong ngăn chứa đó là một loạt các bit đại diện cho giá trị. Đối với nguyên thủy, các bit đại diện cho một giá trị số (mặc dù chúng ta không biết mẫu bit đó trông như thế nào đối với boolean, may mắn là chúng ta không quan tâm). Ví dụ, một byte có giá trị là 6 có nghĩa là mẫu bit trong biến (bộ giữ byte) là 00000110 , đại diện cho 8 bit.

Vì vậy, giá trị của một biến nguyên thủy là rõ ràng, như bên trong một giá trị đối tư ợng là gì? Nếu bạn nói,

ngữ ời giữ? Nếu bạn nói,

Nút b = new Nút ();

có gì bên trong Giá đỡ nút b? Nó có phải là đối tượng Nút không? Không! Một biến tham chiếu đến một đối tượng chỉ là - một biến tham chiếu. Một bộ giữ bit biến tham chiếu chứa các bit đại diện cho cách đi đến đối tượng. Chúng tôi không biết định dạng là gì. Cách mà các tham chiếu đối tượng được lưu trữ là máy ảo cụ thể (nó là một con trỏ đến một cái gì đó, chúng tôi chỉ không biết cái gì đó thực sự là gì). Tất cả những gì chúng ta có thể nói chắc chắn là giá trị của biến không phải là đối tượng, mà là một giá trị đại diện cho một đối tượng cụ thể trên heap. Hoặc null. Nếu biến tham chiếu chưa được gán giá trị hoặc đã được gán giá trị null một cách rõ ràng, thì biến sẽ giữ các bit biểu diễn – bạn đoán nó – null. Bạn có thể đọc

Nút b = null;

là "Biến nút b không tham chiếu đến bất kỳ đối tượng nào."

Vì vậy, bây giờ chúng ta biết một biến chỉ là một hộp nhỏ o 'bit, chúng ta có thể tiếp tục công việc thay đổi các bit đó. Đầu tiên chúng ta sẽ xem xét việc gán giá trị cho các biến nguyên thủy và sau đó kết thúc với việc gán cho các biến tham chiếu.

Phép gán nguyên thủy Dấu bằng (=)

được sử dụng để gán giá trị cho một biến và nó được đặt tên khéo léo là toán tử gán. Thực tế có 12 toán tử gán, nhưng chỉ có 5 toán tử gán được sử dụng phổ biến nhất trong đề thi và chúng được đề cập trong [Chú ý](#) 4.

Bạn có thể gán một biến nguyên thủy bằng cách sử dụng một ký tự hoặc kết quả của một biểu thức.

Hãy xem những điều sau:

```
int x = 7;      // literal assignment
int y = x + 2; // assignment with an expression
                // (including a literal)
int z = x * y; // assignment with an expression
```

Điểm quan trọng nhất cần nhớ là một số nguyên theo nghĩa đen (chẳng hạn như 7) luôn ngầm định là một số nguyên. Nghĩ lại [Chú ý](#) 1, bạn sẽ nhớ lại rằng giá trị int là một giá trị 32 bit. Không có gì to tát nếu bạn đang gán một giá trị cho một biến int hoặc một biến dài, nhưng nếu bạn đang gán cho một biến byte thì sao? Rốt cuộc, một byte có kích thước

người giữ không thể chứa nhiều bit như người giữ có kích thước int. Đây là nơi nó trở nên kỳ lạ. Sau đây là hợp pháp,

```
byte b = 27;
```

nhưng chỉ vì trình biên dịch tự động thu hẹp giá trị chữ thành một byte.

Nói cách khác, trình biên dịch đưa vào dàn diễn viên. Mã trù ớc giống với mã sau:

```
byte b = (byte) 27; // Truyền ký tự int thành byte một cách rõ ràng
```

Có vẻ như trình biên dịch cung cấp cho bạn thời gian nghỉ ngơi và cho phép bạn thực hiện một phím tắt với các phép gán cho các biến số nguyên nhỏ hơn một số nguyên. (Mọi thứ chúng ta đang nói về byte đều áp dụng như nhau cho char và short, cả hai đều nhỏ hơn int.) Nhân tiện, chúng ta vẫn chưa thực sự ở phần kỳ lạ.

Chúng ta biết rằng một số nguyên theo nghĩa đen luôn là một số nguyên, nhưng quan trọng hơn, kết quả của một biểu thức liên quan đến bất kỳ thứ gì có kích thước int hoặc nhỏ hơn luôn là một số nguyên. Nói cách khác, cộng hai byte với nhau và bạn sẽ nhận được một số nguyên - ngay cả khi hai byte đó rất nhỏ. Nhân một số nguyên và một số ngắn và bạn sẽ nhận được một số nguyên. Chia một đoạn ngắn cho một byte và bạn sẽ nhận được một số nguyên. Được rồi, bây giờ chúng ta đang ở phần kỳ lạ.
Kiểm tra cái này:

```
byte a = 3;      // No problem, 3 fits in a byte
byte b = 8;      // No problem, 8 fits in a byte
byte c = a + b; // Should be no problem, sum of the two bytes
                 // fits in a byte
```

Dòng cuối cùng sẽ không biên dịch! Bạn sẽ gặp lỗi giống như lỗi của bạn trong nội thư ứng nhận được:

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
    byte c = a + b;
               ^
```

Chúng tôi đã cố gắng gán tổng hai byte cho một biến byte, kết quả là (11) chắc chắn đủ nhỏ để vừa với một byte, nhưng trình biên dịch không quan tâm. Nó biết quy tắc về các biểu thức int hoặc nhỏ hơn luôn dẫn đến một int. Nó sẽ được biên dịch nếu chúng tôi thực hiện phân loại rõ ràng :

```
byte c = (byte) (a + b);
```



Chúng tôi đang đấu tranh để tìm ra một cách hay để giảng dạy chủ đề này và bạn của chúng tôi, người đồng quản trị JavaRanch * và người đánh giá kỹ thuật lặp lại Marc Peabody, đã đưa ra những điều sau đây. Chúng tôi nghĩ rằng anh ấy đã làm một công việc tuyệt vời: Hoàn toàn hợp pháp khi khai báo nhiều biến cùng loại với một dòng duy nhất bằng cách đặt dấu phẩy giữa mỗi biến:

```
int a, b, c;
```

Bạn cũng có tùy chọn để khởi tạo bất kỳ số lượng biến nào ngay tại chỗ:

```
int j, k = 1, l, m = 3;
```

Và các biến này được đánh giá theo thứ tự mà bạn đọc, từ trái sang phải. Nó giống như thể bạn khai báo từng cái trên một dòng riêng biệt:

```
int j;
int k =
1; int l;
int m = 3;
```

Như ng thứ tự là quan trọng. Điều này là hợp pháp:

```
int j, k=1, l, m=k+3; // legal: k is initialized before m uses it
```

Như ng đây không phải là:

```
int j, k=m+3, l, m=1; // illegal: m is not declared and initialized
                      // before k uses it
int x, y=x+1, z;    // illegal: x is not initialized before y uses it
```

* Tôi biết, tôi biết, như ng nó sẽ luôn là JavaRanch trong trái tim chúng ta.

Truyền nguyên thủy cho phép

bạn chuyển đổi các giá trị nguyên thủy từ kiểu này sang kiểu khác. Chúng tôi đã đề cập đến tính năng ép kiểu nguyên thủy trong phần truớc, như ng bây giờ chúng ta sẽ xem xét sâu hơn. (Đúc đồi tư ợng đã được đề cập trong [Chú ý](#) 2.)

Truyền có thể ẩn hoặc rõ ràng. Truyền ngầm có nghĩa là bạn không phải viết mã cho truyền; việc chuyển đổi diễn ra tự động. Thông thường, một phép ép kiểu ngầm xảy ra khi bạn đang thực hiện chuyển đổi mở rộng – nói cách khác, đặt một thứ nhỏ hơn (ví dụ, một byte) vào một vùng chứa lớn hơn (chẳng hạn như một int). Hãy nhớ những lỗi trình biên dịch "có thể mất độ chính xác" mà chúng ta đã thấy trong phần bài tập? Điều đó đã xảy ra khi chúng tôi cố gắng đặt một thứ lớn hơn (ví dụ, dài) vào một vật chứa nhỏ hơn (chẳng hạn như đồ ngắn). Việc chuyển đổi giá trị lớn thành vùng chứa nhỏ được gọi là thu hẹp và yêu cầu diễn viên rõ ràng, trong đó bạn cho trình biên dịch biết rằng bạn nhận thức được mối nguy hiểm và chấp nhận hoàn toàn trách nhiệm.

Trước tiên, chúng ta sẽ xem xét một dàn diễn viên ngầm:

```
int a = 100;
dài b = a; Dài // Kết hợp ngầm định, một giá trị int luôn phù hợp với một
```

Một dàn diễn viên rõ ràng trông như thế này:

```
float a = 100,001f; int b
= (int) a; // ép kiểu rõ ràng, int có thể mất một số thông tin của float!
```

Giá trị số nguyên có thể được gán cho một biến kép mà không cần ép kiểu rõ ràng, bởi vì bất kỳ giá trị số nguyên nào cũng có thể nằm trong bộ đôi 64 bit. Dòng sau đây chứng minh điều này:

```
đôi d = 100L; // Diễn viên ẩn
```

Trong câu lệnh trước, một double được khởi tạo với một giá trị dài (như được biểu thị bằng chữ L sau giá trị số). Không cần đúc trong thường hợp này vì một bộ đôi có thể chứa mọi thông tin mà một bộ dài có thể lưu trữ. Tuy nhiên, nếu chúng tôi muốn gán một giá trị kép cho một kiểu số nguyên, chúng tôi đang cố gắng thu hẹp chuyển đổi và trình biên dịch biết điều đó:

```
class Casting {
    public static void main(String [] args) {
        int x = 3957.229; // illegal
    }
}
```

Nếu chúng tôi cố gắng biên dịch mã trước đó, chúng tôi gặp lỗi như sau:

```
%javac Casting.java
Casting.java:3: Incompatible type for declaration. Explicit cast
needed to convert double to int.
    int x = 3957.229; // illegal
1 error
```

Trong đoạn mã trư ớc, một giá trị dấu phẩy động đang đư ợc gán cho một biến số nguyên. Bởi vì một số nguyên không có khả năng lưu trữ các vị trí thập phân, một lỗi xảy ra. Để thực hiện công việc này, chúng tôi sẽ chuyển số dấu phẩy động thành số int:

```
class Casting {
    public static void main(String [] args) {
        int x = (int)3957.229; // legal cast
        System.out.println("int x = " + x);
    }
}
```

Khi bạn truyền một số dấu phẩy động sang kiểu số nguyên, giá trị sẽ mất tất cả các chữ số sau số thập phân. Đoạn mã trư ớc đó sẽ tạo ra kết quả sau:

```
int x = 3957
```

Chúng ta cũng có thể ép kiểu số lớn hơn, chẳng hạn như dài, thành kiểu số nhỏ hơn, chẳng hạn như byte. Nhìn vào phần sau:

```
class Casting {
    public static void main(String [] args) {
        long l = 56L;
        byte b = (byte)l;
        System.out.println("The byte is " + b);
    }
}
```

Mã trư ớc đó sẽ biên dịch và chạy tốt. Như ng điều gì sẽ xảy ra nếu giá trị dài lớn hơn 127 (số lớn nhất mà một byte có thể lưu trữ)? Hãy sửa đổi mã:

```

class Casting {
    public static void main(String [] args) {
        long l = 130L;
        byte b = (byte)l;
        System.out.println("The byte is " + b);
    }
}

```

Mã biên dịch tốt và khi chúng tôi chạy nó, chúng tôi nhận được những điều sau:

```
% java Truyền
Byte là -126
```

Chúng tôi không gặp lỗi thời gian chạy, ngay cả khi giá trị bị thu hẹp lớn cho loại. Các bit ở bên trái của 8 thấp hơn chỉ ... biến mất. Nếu bit ngoài cùng bên trái (bit đầu) trong byte (hoặc bất kỳ số nguyên nguyên thủy nào) bây giờ là 1, thì nguyên thủy sẽ có giá trị âm.

BÀI TẬP 3-1

Đúc nguyên bản Tạo một kiểu

số thực của bất kỳ giá trị nào và gán giá trị đó cho một giá trị ngắn bằng cách sử dụng ép kiểu.

1. Khai báo một biến float: float f = 234.56F;
 2. Gán phao cho short: short s = (short) f;
-

Gán số dấu phẩy động Các số dấu phẩy động có hành

vì gán hơi khác so với kiểu số nguyên. Đầu tiên, bạn phải biết rằng mọi ký tự dấu phẩy động hoàn toàn là một ký tự kép (64 bit), không phải là số thực . Vì vậy, ví dụ, nghĩa đen 32.3 được coi là một kép. Nếu bạn cố gắng gán double cho float, trình biên dịch sẽ biết rằng bạn không có đủ chỗ trong container float 32 bit để giữ độ chính xác của double 64 bit và nó sẽ cho bạn biết. Đoạn mã sau có vẻ tốt, nhưng nó sẽ không biên dịch:

```
float f = 32,3;
```

Bạn có thể thấy rằng 32.3 sẽ vừa vặn với một biến có kích thước float, như ng

trình biên dịch sẽ không cho phép nó. Để gán một ký tự dấu phẩy động cho một biến float , bạn phải ép kiểu giá trị hoặc thêm f vào cuối ký tự.

Các bài tập sau sẽ biên dịch:

```
float f = (float) 32.3;
float g = 32.3f;
float h = 32.3F;
```

Gán một chữ quá lớn cho biến Chúng ta cũng sẽ gặp lỗi trình biên dịch nếu chúng ta cố gắng gán một giá trị chữ mà trình biên dịch biết là quá lớn để vừa với biến.

```
byte a = 128; // byte chỉ có thể chứa tối đa 127
```

Mã trù ớc cung cấp cho chúng tôi một lỗi như sau:

```
TestBytes.java:5: possible loss of precision
found   : int
required: byte
byte a = 128;
```

Chúng tôi có thể sửa chữa nó bằng một dàn diễn viên:

```
byte a = (byte) 128;
```

Như ng sau đó kết quả là gì? Khi bạn thu hẹp một nguyên thủy, Java chỉ đơn giản là cắt bớt các bit bậc cao hơn không phù hợp. Nói cách khác, nó mất tất cả các bit ở bên trái của các bit mà bạn đang thu hẹp.

Hãy xem những gì xảy ra trong đoạn mã trù ớc. Ở đó, 128 là mẫu bit 10000000. Cần 8 bit đầy đủ để đại diện cho 128. Nhưng vì 128 theo nghĩa đen là một int, chúng ta thực sự nhận đư ợc 32 bit, với 128 sống ở ngoài cùng bên phải (thứ tự thấp hơn) 8 bit. Vì vậy, một 128 theo nghĩa đen thực sự là

```
000000000000000000000000000000010000000
```

Hãy từ của chúng tôi cho nó; có 32 bit ở đó.

Để thu hẹp 32 bit đại diện cho 128, Java chỉ cần loại bỏ 24 bit ngoài cùng bên trái (bậc cao hơn). Những gì còn lại chỉ là 10000000. Nhưng hãy nhớ rằng một byte đư ợc ký, với bit ngoài cùng bên trái đại diện cho dấu (và không phải là một phần của giá trị của biến). Vì vậy, chúng tôi kết thúc với một số âm (số 1 đư ợc sử dụng để đại diện cho 128 bây giờ đại diện cho bit dấu âm). Hãy nhớ rằng, để tìm ra giá trị của một số âm bằng cách sử dụng ký hiệu phần bù của 2, bạn lật tất cả

giá trị của một số âm bằng cách sử dụng ký hiệu bổ sung của 2, bạn lật tất cả các bit và sau đó thêm 1. Lật 8 bit cho chúng ta 01111111 và thêm 1 vào số đó cho chúng ta 10000000 hoặc quay lại 128! Và khi chúng ta áp dụng bit dấu, chúng ta kết thúc bằng -128.

Bạn phải sử dụng một phép ép kiểu rõ ràng để gán 128 cho một byte và phép gán để lại cho bạn giá trị -128. Diễn viên không gì khác hơn là cách bạn nói với trình biên dịch, "Hãy tin tôi. Tôi là một người chuyên nghiệp. Tôi hoàn toàn chịu trách nhiệm về bất cứ điều gì kỳ lạ xảy ra khi những bit hàng đầu đó bị cắt bỏ. "

Điều đó đưa chúng ta đến các toán tử gán ghép. Điều này sẽ biên dịch:

```
byte b = 3;
b += 7;           // No problem - adds 7 to b (result is 10)
```

and it is equivalent to this:

```
byte b = 3;
b = (byte) (b + 7); // Won't compile without the
                     // cast, since b + 7 results in an int
```

Toán tử gán ghép += cho phép bạn thêm vào giá trị của b mà không cần đưa vào một dàn diễn viên rõ ràng. Trên thực tế, +=, -=, *=, và /= đều sẽ ẩn dàn diễn viên.

Gán một biến nguyên thủy cho một biến nguyên thủy khác Khi bạn gán một biến nguyên thủy này cho một biến nguyên thủy khác, nội dung của biến bên phải sẽ được sao chép. Ví dụ:

```
int a = 6;
int b = a;
```

Đoạn mã này có thể được đọc là, "Gán mẫu bit cho số 6 cho biến int a. Sau đó sao chép mẫu bit trong a và đặt bản sao vào biến b. "

Vì vậy, cả hai biến bây giờ giữ một mẫu bit cho 6, như ng hai biến không có mối quan hệ nào khác. Chúng tôi đã sử dụng biến a chỉ để sao chép nội dung của nó. Tại thời điểm này, a và b có nội dung giống hệt nhau (nói cách khác là các giá trị giống nhau), nhưng nếu chúng ta thay đổi nội dung của a hoặc b thì biến còn lại sẽ không bị ảnh hưởng.

Hãy xem ví dụ sau:

```

class ValueTest {
    public static void main (String [] args) {
        int a = 10; // Assign a value to a
        System.out.println("a = " + a);
        int b = a;
        b = 30;
        System.out.println("a = " + a + " after change to b");
    }
}

```

Đầu ra từ chương trình này là

```

% java ValueTest a =
10 a = 10 sau khi
thay đổi thành b

```

Lưu ý rằng giá trị của a vẫn ở mức 10. Điểm mấu chốt cần nhớ là ngay cả sau khi bạn gán a cho b, a và b vẫn không tham chiếu đến cùng một vị trí trong bộ nhớ. Các biến a và b không chia sẻ một giá trị duy nhất; chúng có các bản sao giống hệt nhau.

Tham chiếu các phép gán biến

Bạn có thể gán một đối tượng mới được tạo cho một biến tham chiếu đối tượng như sau:

```
Nút b = new Nút();
```

Dòng trư ớc thực hiện ba điều quan trọng:

- Tạo một biến tham chiếu có tên b, kiểu Nút
- Tạo một đối tượng Nút mới trên heap
- Gán đối tượng Nút mới tạo cho biến tham chiếu b

Bạn cũng có thể gán null cho một biến tham chiếu đối tượng, điều này đơn giản có nghĩa là biến không tham chiếu đến bất kỳ đối tượng nào:

```
Nút c = null;
```

Dòng trư ớc đó tạo không gian cho biến tham chiếu Nút (giá trị giữ bit cho giá trị tham chiếu), nhưng nó không tạo đối tượng Nút thực sự.

Như chúng ta đã thảo luận trong chương trư ớc, bạn cũng có thể sử dụng một biến tham chiếu để tham chiếu đến bất kỳ đối tượng nào là lớp con của kiểu biến tham chiếu đã khai báo, như sau:

```

public class Foo {
    public void doFooStuff() { }
}
public class Bar extends Foo {
    public void doBarStuff() { }
}
class Test {
    public static void main (String [] args) {
        Foo reallyABar = new Bar(); // Legal because Bar is a
                                    // subclass of Foo
        Bar reallyAFoo = new Foo(); // Illegal! Foo is not a
                                    // subclass of Bar
    }
}

```

Quy tắc là bạn có thể gán một lớp con của kiểu đã khai báo như ng không phải là lớp cha của kiểu đã khai báo. Hãy nhớ rằng, một đối tượng Bar được đảm bảo có thể làm bất cứ điều gì mà Foo có thể làm, vì vậy bất kỳ ai có tham chiếu Foo đều có thể gọi các phương thức Foo ngay cả khi đối tượng thực sự là một Bar.

Trong đoạn mã trư ớc, chúng ta thấy rằng Foo có một phương thức doFooStuff () mà ai đó có tham chiếu Foo có thể có gắng gọi. Nếu đối tượng được tham chiếu bởi biến Foo thực sự là Foo thì không vấn đề gì. Nhưng cũng không có vấn đề gì nếu đối tượng là một Bar vì Bar kế thừa phương thức doFooStuff () . Tuy nhiên, bạn không thể làm cho nó hoạt động ngược lại. Nếu ai đó có tham chiếu Bar , họ sẽ gọi doBarStuff () , nhưng nếu đối tượng là Foo, nó sẽ không biết cách phản hồi.



Kỳ thi OCA 8 bao gồm các lớp trình bao bọc. Chúng ta có thể đã thảo luận về các lớp trình bao bọc trong chương này, nhưng chúng tôi cảm thấy sẽ hợp lý hơn khi thảo luận về chúng trong ngữ cảnh của ArrayLists (mà chúng ta sẽ đề cập trong [Chương 6](#)). Vì vậy, cho đến khi bạn đến [Chương 6](#), tất cả những gì bạn cần biết về các trình bao bọc sau:

Một đối tượng wrapper là một đối tượng giữ giá trị của một nguyên thủy. Mỗi loại nguyên thủy đều có một lớp bao bọc liên quan: Boolean, Byte, Character, Double, Float, Integer, Long và Short. Đoạn mã sau tạo hai đối tượng trình bao bọc và sau đó in giá trị của chúng:

```
Long x = new Long(42);      // create an instance of Long with a value of 42
Short s = new Short("57");  // create an instance of Short with a value of 57
System.out.println(x + " " + s);
```

tạo ra kết quả sau:

42 57

Chúng ta sẽ tìm hiểu sâu hơn nhiều về các yếu tố bao bọc trong [Chương 5](#).

MỤC TIÊU XÁC NHẬN

Phạm vi (Mục tiêu 1.1 của OCA)

1.1 Xác định phạm vi của các biến.

Phạm vi biến đổi

Khi bạn đã khai báo và khởi tạo một biến, một câu hỏi tự nhiên là "Biến này sẽ tồn tại trong bao lâu?" Đây là một câu hỏi liên quan đến phạm vi của các biến.

Và phạm vi không chỉ là một điều quan trọng để hiểu nói chung, nó còn đóng một vai trò quan trọng trong kỳ thi. Hãy bắt đầu bằng cách xem tệp lớp:

```
class Layout {                                // class
    static int s = 343;                      // static variable
    int x;                                     // instance variable
    { x = 7; int x2 = 5; }                    // initialization block
    Layout() { x += 8; int x3 = 6; }          // constructor

    void doStuff() {                          // method
        int y = 0;                            // local variable
        for(int z = 0; z < 4; z++) {         // 'for' code block
            y += z + x;
        }
    }
}
```

Như với các biến trong tất cả các chương trình Java, các biến trong chương trình này (s, x, x2, x3, y và z) đều có phạm vi:

- s là một biến tĩnh.
- x là một biến thể hiện.
- y là một biến cục bộ (đôi khi được gọi là một biến "phương thức cục bộ").
- z là một biến khôi.
- x2 là một biến khôi init , một biến cục bộ. x3 là một biến xây dựng, một phương vị của biến cục bộ.

Với mục đích thảo luận về phạm vi của các biến, chúng ta có thể nói rằng có bốn phạm vi cơ bản:

1. Biến tĩnh có phạm vi dài nhất; chúng được tạo khi lớp được tải và chúng tồn tại miễn là lớp vẫn được tải trong Máy ảo Java (JVM).
2. Biến cá thể là biến tồn tại lâu nhất tiếp theo; chúng được tạo khi một cá thể mới được tạo và chúng tồn tại cho đến khi cá thể đó bị xóa.
3. Biến cục bộ là tiếp theo; họ tồn tại miễn là phương pháp của họ vẫn còn trên cây rơm. Tuy nhiên, như chúng ta sẽ sớm thấy, các biến cục bộ có thể tồn tại và vẫn "nằm ngoài phạm vi".
4. Các biến khôi chỉ tồn tại miễn là khôi mà đang thực thi.

Lỗi xác định phạm vi có nhiều kích cỡ và hình dạng. Một sai lầm phổ biến xảy ra khi một biến bị che khuất và hai phạm vi trùng nhau. Chúng tôi sẽ xem xét chi tiết về việc tạo bóng trong một vài trang. Lý do phổ biến nhất cho lỗi xác định phạm vi là cố gắng truy cập một biến không nằm trong phạm vi. Hãy xem ba ví dụ phổ biến của loại lỗi này:

- Có gắng truy cập một biến cá thể từ ngữ cảnh tĩnh (thường là từ main()):

```
class ScopeErrors {
    int x = 5;
    public static void main(String[] args) {
        x++; // won't compile, x is an 'instance' variable
    }
}
```

- Có gắng truy cập một biến cục bộ của phương thức đã gọi bạn.

Khi một phương thức, nói go (), gọi phương thức khác, nói go2 (), go2 ()

sẽ không có quyền truy cập vào các biến cục bộ của go(). Trong khi go2() đang thực thi, các biến cục bộ của go() vẫn còn tồn tại, nhưng chúng nằm ngoài phạm vi. Khi go2() hoàn thành, nó bị xóa khỏi ngăn xếp và go() tiếp tục thực thi. Tại thời điểm này, tất cả các biến được khai báo trước đó của go() đều trở lại trong phạm vi. Ví dụ:

```
class ScopeErrors {
    public static void main(String [] args) {
        ScopeErrors s = new ScopeErrors();
        s.go();
    }
    void go() {
        int y = 5;
        go2();
        y++;           // once go2() completes, y is back in scope
    }
    void go2() {
        y++;           // won't compile, y is local to go()
    }
}
```

- Có gắng sử dụng một biến khỏi sau khi khởi mã đã hoàn thành. Việc khai báo và sử dụng một biến trong một khởi mã là rất phổ biến, nhưng hãy cẩn thận đừng cố gắng sử dụng biến sau khi khởi đã hoàn thành:

```
void go3() {
    for(int z = 0; z < 5; z++) {
        boolean test = false;
        if(z == 3) {
            test = true;
            break;
        }
    }
    System.out.print(test);   // 'test' is an ex-variable,
                            // it has ceased to be...
}
```

Trong hai ví dụ cuối cùng, trình biên dịch sẽ nói như sau:

không thể tìm thấy biểu tư ợng

Đây là cách nói của trình biên dịch, "Biến đó mà bạn vừa thử sử dụng? Chà, nó có thể có giá trị trong quá khứ xa xôi (như một dòng mã trước đây), nhưng đây là thời của Internet, em yêu, tôi không nhớ gì về một biến số như vậy. "

Thời gian Internet, em yêu, tôi không có trí nhớ của một biến như vậy. "



Chú ý thêm đến các lỗi xác định phạm vi khỏi mã. Bạn có thể thấy chúng trong các vòng chuyển đổi, try-catch, for, do và while , mà chúng tôi sẽ đề cập trong các chương sau.

MỤC TIÊU XÁC NHẬN

Khởi tạo biến (Mục tiêu OCA 2.1, 4.1 và 4.2)

2.1 Khai báo và khởi tạo biến (bao gồm ép kiểu dữ liệu nguyên thủy).

4.1 Khai báo, khởi tạo, khởi tạo và sử dụng mảng một chiều 4.2 Khai báo, khởi tạo, khởi tạo và sử dụng mảng đa chiều (sic)

Sử dụng một phần tử biến hoặc mảng

Đó là chư a đư ợc khởi tạo và chư a đư ợc chỉ định

Java cung cấp cho chúng ta tùy chọn khởi tạo một biến đã khai báo hoặc để nó chư a đư ợc khởi tạo. Khi chúng ta cố gắng sử dụng biến chư a đư ợc khởi tạo, chúng ta có thể nhận đư ợc các hành vi khác nhau tùy thuộc vào loại biến hoặc mảng mà chúng ta đang xử lý (nguyên thủy hoặc đối tượng). Hành vi cũng phụ thuộc vào cấp độ (phạm vi) mà chúng ta đang khai báo biến của mình. Một biến thể hiện đư ợc khai báo bên trong lớp như ng bên ngoài bất kỳ phư ơng thức hoặc hàm tạo nào, trong khi một biến cục bộ đư ợc khai báo trong một phư ơng thức (hoặc trong danh sách đối số của phư ơng thức).

Các biến cục bộ đôi khi đư ợc gọi là ngăn xếp, tạm thời, tự động hoặc phư ơng thức như ng các quy tắc cho các biến này đều giống nhau bắt kề bạn gọi chúng là gì. Mặc dù bạn có thể để một biến cục bộ chư a đư ợc khởi tạo, trình biên dịch sẽ phàn nàn nếu bạn cố gắng sử dụng một biến cục bộ trước khi khởi tạo nó với một giá trị, như chúng ta sẽ thấy.

Các biến đối tư ợng kiểu nguyên thủy và đối tư ợng

Nguyên bản và kiểu đối tư ợng Biến cá thể Biến đối tư ợng (còn được gọi là biến thành viên) là những biến được định nghĩa ở cấp độ lớp. Điều đó có nghĩa là khai báo biến không được thực hiện trong một phư ơng thức, hàm tạo hoặc bất kỳ khởi tạo nào khác. Các biến phiên bản được khởi tạo thành một giá trị mặc định mỗi khi một phiên bản mới được tạo, mặc dù chúng có thể được cung cấp một giá trị rõ ràng sau khi các siêu cấu trúc của đối tư ợng đã hoàn thành. [Bảng 3-1](#) liệt kê các giá trị mặc định cho kiểu nguyên thủy và kiểu đối tư ợng.

BẢNG 3-1 Giá trị mặc định cho kiểu nguyên thủy và kiểu tham chiếu

Variable Type	Default Value
Object reference	null (not referencing any object)
byte, short, int, long	0
float, double	0.0
boolean	false
char	'\u0000'

Các biến phiên bản nguyên thủy

Trong ví dụ sau, số nguyên năm được định nghĩa là một thành viên của lớp vì nó nằm trong dấu ngoặc nhọn ban đầu của lớp và không nằm trong dấu ngoặc nhọn của một phư ơng thức:

```
public class BirthDate {
    int year; // Instance variable
    public static void main(String [] args) {
        BirthDate bd = new BirthDate();
        bd.showYear();
    }
    public void showYear() {
        System.out.println("The year is " + year);
    }
}
```

Khi chương trình được bắt đầu, nó cung cấp cho biến năm một giá trị bằng 0, giá trị mặc định cho các biến thể hiện số nguyên thủy.



Bạn nên khởi tạo tất cả các biến của mình, ngay cả khi bạn đang gán chúng với giá trị mặc định. Mã của bạn sẽ dễ đọc hơn; những lập trình viên phải duy trì mã của bạn (sau khi bạn trúng số và chuyển đến Tahiti) sẽ rất biết ơn.

Các biến phiên bản tham chiếu đôi tư ợng Khi

so sánh với các biến nguyên thủy chưa được khởi tạo, các tham chiếu đôi tư ợng không được khởi tạo là một câu chuyện hoàn toàn khác. Hãy xem đoạn mã sau:

```
public class Book {
    private String title;           // instance reference variable
    public String getTitle() {
        return title;
    }
    public static void main(String [] args) {
        Book b = new Book();
        System.out.println("The title is " + b.getTitle());
    }
}
```

Mã này sẽ biên dịch tốt. Khi chúng tôi chạy nó, đầu ra là

Tiêu đề trống

Biến tiêu đề chưa được khởi tạo rõ ràng bằng phép gán Chuỗi , vì vậy giá trị biến phiên bản là null. Hãy nhớ rằng null không giống như một Chuỗi rỗng (""). Giá trị null có nghĩa là biến tham chiếu không tham chiếu đến bất kỳ đối tư ợng nào trên heap. Việc sửa đổi mã Sách sau đây sẽ gặp sự cố:

```

public class Book {
    private String title;           // instance reference variable
    public String getTitle() {
        return title;
    }
    public static void main(String [] args) {
        Book b = new Book();
        String s = b.getTitle();      // Compiles and runs
        String t = s.toLowerCase();   // Runtime Exception!
    }
}

```

Khi chúng tôi cố gắng chạy lớp Sách , JVM sẽ tạo ra một cái gì đó như sau:

Ngoại lệ trong luồng "main" java.lang.NullPointerException
tại Book.main (Book.java:9)

Chúng tôi gặp lỗi này vì tiêu đề biến tham chiếu không trả (tham chiếu) đến một đối tượng.
Chúng ta có thể kiểm tra xem một đối tượng đã được khởi tạo hay chưa bằng cách sử dụng từ
khóa null, như đoạn mã sửa đổi sau cho thấy:

```

public class Book {
    private String title;           // instance reference variable
    public String getTitle() {
        return title;
    }
    public static void main(String [] args) {
        Book b = new Book();
        String s = b.getTitle();      // Compiles and runs
        if (s != null) {
            String t = s.toLowerCase();
        }
    }
}

```

Đoạn mã trên đó sẽ kiểm tra để đảm bảo rằng đối tượng được tham chiếu bởi biến s không
phải là giá trị rỗng trước khi cố gắng sử dụng nó. Để phòng các tình huống trong bài kiểm
tra mà bạn có thể phải truy ngược lại mã để tìm xem liệu một tham chiếu đối tượng có giá
trị null hay không. Ví dụ: trong đoạn mã trên, bạn nhìn vào khai báo biến cá thể cho tiêu
đề, thấy rằng không có khởi tạo rõ ràng, nhận ra rằng biến tiêu đề sẽ được cung cấp giá trị
mặc định là null, và sau đó nhận ra rằng biến s cũng sẽ có giá trị của null.

Hãy nhớ rằng, giá trị của s là một bản sao của giá trị của tiêu đề (như được trả về bởi

phương thức getTitle ()), vì vậy nếu tiêu đề là một tham chiếu rỗng , s cũng sẽ như vậy.

Biến thể hiện mảng Trong [Chương 5](#) ,

chúng ta sẽ xem xét rất chi tiết về việc khai báo, xây dựng và khởi tạo mảng và mảng đa chiều. Bây giờ, chúng ta sẽ chỉ xem xét quy tắc cho các giá trị mặc định của phần tử mảng.

Một mảng là một đối tượng; do đó, một biến cá thể mảng được khai báo như ng không được khởi tạo rõ ràng sẽ có giá trị là null, giống như bất kỳ biến cá thể tham chiếu đối tượng nào khác. Như ng, nếu mảng được khởi tạo, điều gì sẽ xảy ra với các phần tử có trong mảng? Tất cả các phần tử mảng đều được cung cấp các giá trị mặc định của chúng – các giá trị mặc định giống nhau mà các phần tử của kiểu đó nhận được khi chúng là các biến thể hiện. Điểm mấu chốt: Các phần tử của mảng luôn, luôn luôn, luôn được cung cấp các giá trị mặc định, bất kể vị trí của chính mảng đó được khởi tạo.

Nếu chúng ta khởi tạo một mảng, các phần tử tham chiếu đối tượng sẽ bằng null nếu chúng không được khởi tạo riêng lẻ với các giá trị. Nếu các nguyên thủy được chứa trong một mảng, chúng sẽ được cung cấp các giá trị mặc định tương ứng. Ví dụ: trong đoạn mã sau, năm mảng sẽ chứa 100 số nguyên mà tất cả đều bằng 0 (không) theo mặc định:

```
public class BirthDays {  
    static int [] year = new int[100];  
    public static void main(String [] args) {  
        for(int i=0;i<100;i++)  
            System.out.println("year[" + i + "] = " + year[i]);  
    }  
}
```

Khi mã trư ớc đó chạy, kết quả cho biết rằng tất cả 100 số nguyên trong mảng đều có giá trị là 0.

Nguyên bản và đối tượng cục bộ (ngăn xếp, tự động)

Các biến cục bộ được định nghĩa trong một phương thức và chúng bao gồm các tham số của phương thức.



Tự động chỉ là một thuật ngữ khác cho biến cục bộ. Nó không có nghĩa là biến tự động được tự động gán một giá trị! Trong thực tế, điều ngược lại là đúng. Một biến tự động phải được gán một giá trị trong mã, nếu không trình biên dịch sẽ phàn nàn.

Nguyên thủy địa phư ơng

Trong trình mô phỏng du hành thời gian sau đây, số nguyên năm được định nghĩa là một biến tự động vì nó nằm trong dấu ngoặc nhọn của một phư ơng thức:

```
public class TimeTravel {
    public static void main(String [] args) {
        int year = 2050;
        System.out.println("The year is " + year);
    }
}
```

Các biến cục bộ, bao gồm các biến nguyên thủy, luôn luôn phải được khởi tạo trước khi bạn cố gắng sử dụng chúng (mặc dù không nhất thiết phải trên cùng một dòng mã). Java không cung cấp cho các biến cục bộ một giá trị mặc định; bạn phải khởi tạo chúng một cách rõ ràng với một giá trị, như trong ví dụ trước. Nếu bạn cố gắng sử dụng nguyên thủy chưa được khởi tạo trong mã của mình, bạn sẽ gặp lỗi trình biên dịch:

```
public class TimeTravel {
    public static void main(String [] args) {
        int year; // Local variable (declared but not initialized)
        System.out.println("The year is " + year); // Compiler error
    }
}
```

Biên dịch tạo ra kết quả như thế này:

```
%javac TimeTravel.java
TimeTravel.java:4: Variable year may not have been initialized.
    System.out.println("The year is " + year);
1 error
```

Để sửa mã của chúng tôi, chúng tôi phải cung cấp cho năm số nguyên một giá trị. Trong này đã cập nhật Ví dụ, chúng tôi khai báo nó trên một dòng riêng biệt, hoàn toàn hợp lệ:

```

public class TimeTravel {
    public static void main(String [] args) {
        int year;           // Declared but not initialized
        int day;            // Declared but not initialized
        System.out.println("You step into the portal.");
        year = 2050;        // Initialize (assign an explicit value)
        System.out.println("Welcome to the year " + year);
    }
}

```

Lưu ý trong ví dụ trư ớc, chúng tôi đã khai báo một số nguyên đư ợc gọi là ngày không bao giờ đư ợc khởi tạo, như ng mã biên dịch và chạy tốt. Về mặt pháp lý, bạn có thể khai báo một biến cục bộ mà không cần khởi tạo nó miễn là bạn không sử dụng biến đó – như ng, hãy đổi mặt với nó, nếu bạn đã khai báo nó, bạn có thể có lý do (mặc dù chúng ta đã nghe nói về việc các lập trình viên khai báo các biến cục bộ ngẫu nhiên chỉ cho thê thao, để xem liệu họ có thể tìm ra cách thức và lý do tại sao chúng đư ợc sử dụng).



Không phải lúc nào trình biên dịch cũng cho biết liệu một biến cục bộ đã đư ợc khởi tạo trư ớc khi sử dụng hay chư a. Ví dụ: nếu bạn khởi tạo trong một khối điều kiện logic (nói cách khác, một khối mã có thể không chạy, chẳng hạn như khối if hoặc vòng lặp for mà không có giá trị đúng hoặc sai trong thử nghiệm), trình biên dịch biết rằng quá trình khởi tạo có thể không xảy ra và có thể tạo ra lỗi. Đoạn mã sau sẽ đảo lộn trình biên dịch:

```

public class TestLocal {
    public static void main(String [] args) {
        int x;
        if (args[0] != null) { // assume you know this is true
            x = 7;           // compiler can't tell that this
                               // statement will run
        }
        int y = x;          // the compiler will choke here
    }
}

```

Trình biên dịch sẽ tạo ra một lỗi như sau:

TestLocal.java:9: biến x có thể chư a đư ợc khởi tạo

Do vấn đề của trình biên dịch-không-thể-nói-cho-nhất định, đôi khi bạn sẽ cần khởi tạo biến của mình bên ngoài khối điều kiện, chỉ để thực hiện

trình biên dịch vui vẻ. Bạn biết lý do tại sao điều đó lại quan trọng nếu bạn đã nhìn thấy nhãn dán đậm, "Khi trình biên dịch không hài lòng, không ai hài lòng cả."

Tham chiếu đối tượng cục bộ Các

tham chiếu đối tượng cũng hoạt động khác nhau khi được khai báo trong một phư ơng thức chứ không phải là biến cá thể. Với các tham chiếu đối tượng biến đối tượng, bạn có thể thoát khỏi việc để một tham chiếu đối tượng chưa được khởi tạo, miễn là mã kiểm tra để đảm bảo rằng tham chiếu không rỗng trước khi sử dụng nó. Hãy nhớ rằng, đối với trình biên dịch, null là một giá trị. Bạn không thể sử dụng toán tử dấu chấm trên một tham chiếu null, vì không có đối tượng nào ở đầu kia của nó, nhưng một tham chiếu null không giống như một tham chiếu chưa được khởi tạo. Các tham chiếu được khai báo cục bộ không thể thoát khỏi việc kiểm tra null trước khi sử dụng, trừ khi bạn khởi tạo biến cục bộ thành null một cách rõ ràng.

Trình biên dịch sẽ phản nàn về đoạn mã sau:

```
import java.util.Date;
public class TimeTravel {
    public static void main(String [] args) {
        Date date;
        if (date == null)
            System.out.println("date is null");
    }
}
```

Việc biên dịch mã dẫn đến một lỗi tự động tự như sau:

```
%javac TimeTravel.java
TimeTravel.java:5: Variable date may not have been initialized.
    if (date == null)
1 error
```

Tham chiếu biến bản lô luôn được cung cấp giá trị mặc định là null, cho đến khi chúng được khởi tạo một cách rõ ràng sang một thứ khác. Như ng các tham chiếu cục bộ không được cung cấp một giá trị mặc định; nói cách khác, chúng không rỗng. Nếu bạn không khởi tạo biến tham chiếu cục bộ, thì theo mặc định, giá trị của nó là – đó là toàn bộ điểm: nó không có bất kỳ giá trị nào! Vì vậy, chúng tôi sẽ làm cho điều này đơn giản: Chỉ cần đặt điều đáng yêu thành null một cách rõ ràng cho đến khi bạn sẵn sàng khởi tạo nó thành một thứ khác. Biến cục bộ sau sẽ biên dịch đúng cách:

```
Date date = null; // Explicitly set the local reference
                  // variable to null
```

Mảng cục bộ

Cũng giống như bất kỳ tham chiếu đối tư ợng nào khác, tham chiếu mảng đư ợc khai báo trong một phư ơng thức phải đư ợc gán một giá trị trư ớc khi sử dụng. Điều đó chỉ có nghĩa là bạn phải khai báo và xây dựng mảng. Tuy nhiên, bạn không cần phải khởi tạo rõ ràng các phần tử của một mảng. Chúng tôi đã nói trư ớc đây, nhưng điều đó đủ quan trọng để lặp lại: Các phần tử của mảng đư ợc cung cấp các giá trị mặc định của chúng (0, false, null, '\u0000', v.v.) bất kể mảng đư ợc khai báo là một phiên bản hay biến cục bộ. Tuy nhiên, bản thân đối tư ợng mảng sẽ không đư ợc khởi tạo nếu nó đư ợc khai báo cục bộ. Nói cách khác, bạn phải khởi tạo rõ ràng một tham chiếu mảng nếu nó đư ợc khai báo và sử dụng trong một phư ơng thức, như ng tại thời điểm bạn xây dựng một đối tư ợng mảng, tất cả các phần tử của nó đều đư ợc gán giá trị mặc định của chúng.

Gán một biến tham chiếu cho một biến khác Với các biến nguyên

thủy, việc gán một biến này cho một biến khác có nghĩa là nội dung (mẫu bit) của một biến này đư ợc sao chép sang một biến khác. Các biến tham chiếu đối tư ợng hoạt động theo cùng một cách. Nội dung của một biến tham chiếu là một mẫu bit, vì vậy nếu bạn gán biến tham chiếu a1 cho biến tham chiếu b1, thì mẫu bit trong a1 sẽ đư ợc sao chép và bản sao mới đư ợc đặt vào b1. (Một số ngư ời đã tạo ra một trò chơi xoay quanh việc đếm bao nhiêu lần chúng ta sử dụng từ sao chép trong chư ơng này. khái niệm sao chép này là một vấn đề lớn!) Nếu chúng ta gán một phiên bản hiện có của một đối tư ợng cho một biến tham chiếu mới, thì hai biến tham chiếu sẽ giữ cùng một mẫu bit – một mẫu bit đè cập đến một đối tư ợng cụ thể trên heap.

Nhìn vào đoạn mã sau:

```
import java.awt.Dimension;
class ReferenceTest {
    public static void main (String [] args) {
        Dimension a1 = new Dimension(5,10);
        System.out.println("a1.height = " + a1.height);
        Dimension b1 = a1;
        b1.height = 30;
        System.out.println("a1.height = " + a1.height +
                           " after change to b1");
    }
}
```

Trong ví dụ trư ớc, một đối tư ợng Thứ nguyên a1 đư ợc khai báo và khởi tạo với chiều rộng là 5 và chiều cao là 10. Tiếp theo, Thứ nguyên b1 đư ợc khai báo và gán giá trị của a1. Tại thời điểm này, cả hai biến (a1 và b1) đều giữ các giá trị giống hệt nhau vì nội dung của a1 đã đư ợc sao chép vào b1. Vẫn chỉ có một đối tư ợng Thứ nguyên – đối tư ợng mà cả a1 và b1 đều tham chiếu đến. Cuối cùng, chiều cao

thuộc tính được thay đổi bằng cách sử dụng tham chiếu b1 . Vậy giờ hãy suy nghĩ trong một phút: điều này cũng sẽ thay đổi thuộc tính chiều cao của a1 ? Hãy xem đầu ra sẽ là gì:

```
%java ReferenceTest
a1.height = 10
a1.height = 30 after change to b1
```

Từ kết quả này, chúng ta có thể kết luận rằng cả hai biến đều tham chiếu đến cùng một trự ờng hợp của đối tượng Thứ nguyên . Khi chúng tôi thực hiện thay đổi đối với b1 , thuộc tính chiều cao cũng được thay đổi cho a1 .

Một ngoại lệ đối với cách gán tham chiếu đối tượng là Chuỗi . Trong Java, các đối tượng Chuỗi được xử lý đặc biệt. Đối với một điều, các đối tượng String là bất biến; bạn không thể thay đổi giá trị của một đối tượng Chuỗi (nhiều hơn về khái niệm này trong [Chương 6](#)) . Như ng có vẻ như bạn có thể. Kiểm tra đoạn mã sau:

```
class StringTest {
    public static void main(String [] args) {
        String x = "Java"; // Assign a value to x
        String y = x;      // Now y and x refer to the same
                           // String object

        System.out.println("y string = " + y);
        x = x + " Bean"; // Now modify the object using
                          // the x reference
        System.out.println("y string = " + y);
    }
}
```

Vì Chuỗi là đối tượng, bạn có thể nghĩ Chuỗi y sẽ chứa các ký tự Java Bean sau khi biến x được thay đổi. Hãy xem đầu ra là gì:

```
%java StringTest
y string = Java
y string = Java
```

Như bạn có thể thấy, mặc dù y là một biến tham chiếu đến cùng một đối tượng mà x đc cập đến, khi chúng ta thay đổi x, nó không thay đổi y! Đối với bất kỳ loại đối tượng nào khác, trong đó hai tham chiếu tham chiếu đến cùng một đối tượng, nếu một trong hai tham chiếu được sử dụng để sửa đổi đối tượng, cả hai tham chiếu sẽ thấy sự thay đổi vì vẫn chỉ có một đối tượng duy nhất. Như ng bất kỳ khi nào chúng tôi thực hiện bất kỳ thay đổi nào đối với một Chuỗi, máy ảo sẽ cập nhật biến tham chiếu để tham chiếu đến một đối tượng khác. Khác nhau

đôi tư ợng có thể là một đôi tư ợng mới, hoặc có thể không, nhưng nó chắc chắn sẽ là một đôi tư ợng khác. Lý do chúng ta không thể nói chắc chắn liệu một đôi tư ợng mới có được tạo hay không là do nhóm hàng chuỗi , mà chúng ta sẽ đề cập trong [Chương 6](#).

Bạn cần hiểu điều gì sẽ xảy ra khi bạn sử dụng biến tham chiếu Chuỗi để sửa đổi một chuỗi:

- Một chuỗi mới được tạo (hoặc một Chuỗi phù hợp được tìm thấy trong nhóm Chuỗi), giữ nguyên đôi tư ợng Chuỗi ban đầu .
- Sau đó, tham chiếu được sử dụng để sửa đổi Chuỗi (hoặc đúng hơn, tạo một Chuỗi mới bằng cách sửa đổi bản sao của bản gốc) sau đó được gán đổi tư ợng Chuỗi hoàn toàn mới.

Vì vậy, khi bạn nói,

```
1. String s = "Fred";
2. String t = s;      // Now t and s refer to the same
                      // String object
3. t.toUpperCase();  // Invoke a String method that changes
                      // the String
```

bạn chưa thay đổi đôi tư ợng String ban đầu được tạo trên dòng 1. Khi dòng 2 hoàn thành, cả t và s đều tham chiếu đến cùng một đối tượng String . Nhưng khi dòng 3 chạy, thay vì sửa đổi đối tư ợng được tham chiếu bởi t và s (là đối tư ợng Chuỗi duy nhất cho đến thời điểm này), một đối tư ợng Chuỗi hoàn toàn mới được tạo ra. Và sau đó nó bị bỏ rơi. Vì Chuỗi mới không được gán cho một biến Chuỗi , nên Chuỗi mới được tạo (chứa chuỗi "FRED") là bánh mì nư ơng. Vì vậy, mặc dù hai đối tư ợng Chuỗi đã được tạo trong mã trự ớc, chỉ một đối tư ợng thực sự được tham chiếu và cả t và s đều tham chiếu đến nó. Hành vi của chuỗi là cực kỳ quan trọng trong kỳ thi, vì vậy chúng tôi sẽ trình bày chi tiết hơn về nó trong [Chương 6](#).

MỤC TIÊU XÁC NHẬN

Chuyển các biến vào các phu ơng thức (Mục tiêu 6.6 của OCA)

6.8 Xác định ảnh hưởng đối với các tham chiếu đối tư ợng và các giá trị nguyên thủy khi chúng được truyền vào các phu ơng thức thay đổi giá trị.

Các phu ơng thức có thể được khai báo để nhận các tham chiếu nguyên thủy và / hoặc đối tư ợng. Bạn

Các phu ơng thức có thể đư ợc khai báo để nhận các tham chiếu nguyên thủy và / hoặc đối tư ợng. Bạn cần biết làm thế nào (hoặc nếu) biến của ngư ời gọi có thể bị ảnh hứ ơng bởi phu ơng thức đư ợc gọi. Sự khác biệt giữa tham chiếu đối tư ợng và các biến nguyên thủy, khi đư ợc truyền vào các phu ơng thức, là rất lớn và quan trọng. Để hiểu phần này, bạn cần nắm rõ thông tin đư ợc đề cập trong phần "Chữ viết, Bài tập và Biến số" trong phần đầu của chư ơng này.

Chuyển các biến tham chiếu đối tư ợng

Khi bạn truyền một biến đối tư ợng vào một phu ơng thức, bạn phải lưu ý rằng bạn đang truyền tham chiếu đối tư ợng, không phải chính đối tư ợng thực tế. Hãy nhớ rằng một biến tham chiếu giữ các bit đại diện (cho VM bên dưới) một cách để truy cập một đối tư ợng cụ thể trong bộ nhớ (trên heap). Quan trọng hơn, bạn phải nhớ rằng bạn thậm chí không chuyển biến tham chiếu thực tế, mà là một bản sao của biến tham chiếu. Bản sao của một biến có nghĩa là bạn nhận đư ợc một bản sao của các bit trong biến đó, vì vậy khi bạn chuyển một biến tham chiếu, bạn đang chuyển một bản sao của các bit đại diện cho cách đi đến một đối tư ợng cụ thể. Nói cách khác, cả ngư ời gọi và phu ơng thức đư ợc gọi bây giờ sẽ có các bản sao giống hệt nhau của tham chiếu; do đó, cả hai sẽ tham chiếu đến cùng một đối tư ợng chính xác (không phải là bản sao) trên heap.

Đối với ví dụ này, chúng tôi sẽ sử dụng lớp Thủ nguyên từ gói `java.awt`:

```

1. import java.awt.Dimension;
2. class ReferenceTest {
3.     public static void main (String [] args) {
4.         Dimension d = new Dimension(5,10);
5.         ReferenceTest rt = new ReferenceTest();
6.         System.out.println("Before, d.height: " + d.height);
7.         rt.modify(d);
8.         System.out.println("After, d.height: " + d.height);
9.     }
10.    void modify(Dimension dim) {
11.        dim.height = dim.height + 1;
12.        System.out.println("dim = " + dim.height);
13.    }
}

```

Khi chúng ta chạy lớp này, chúng ta có thể thấy phu ơng thức `mod ()` thực sự có thể sửa đổi đối tư ợng Kích thư ớc ban đầu (và duy nhất) đư ợc tạo trên dòng 4.

```

C:\Java Projects\Reference>java ReferenceTest
Before, d.height: 10
dim = 11
After, d.height: 11

```

Lưu ý khi đổi tư ợng Thứ nguyên trên dòng 4 được chuyển đến phu ơng thức mod () , bất kỳ thay đổi nào đối với đối tư ợng xảy ra bên trong phu ơng thức đang được thực hiện cho đối tư ợng có tham chiêu được truyền. Trong ví dụ trư ớc, các biến tham chiêu d và dim đều trở đến cùng một đối tư ợng.

Java có sử dụng ngữ nghĩa truyền qua giá trị không?

Nếu Java chuyển các đối tư ợng bằng cách chuyển biến tham chiêu thay thế, điều đó có nghĩa là Java sử dụng tham chiêu truyền cho các đối tư ợng? Không chính xác, mặc dù bạn sẽ thường nghe và đọc điều đó. Java thực sự là giá trị truyền cho tất cả các biến chạy trong một máy ảo duy nhất. Giá trị truyền qua có nghĩa là giá trị chuyển theo biến. Và điều đó có nghĩa là pass-by-copy-of-the-variable! (Lại có bản sao từ đó !)

Không có gì khác biệt nếu bạn đang chuyển các biến nguyên thủy hoặc tham chiêu; bạn luôn chuyển một bản sao của các bit trong biến. Vì vậy, đối với một biến nguyên thủy, bạn đang truyền một bản sao của các bit đại diện cho giá trị. Ví dụ: nếu bạn truyền một biến int có giá trị là 3, bạn đang truyền một bản sao của các bit đại diện cho 3. Sau đó, phu ơng thức được gọi sẽ nhận bản sao của chính nó về giá trị để thực hiện những gì nó thích.

Và nếu bạn đang truyền một biến tham chiêu đối tư ợng, bạn đang truyền một bản sao của các bit đại diện cho tham chiêu tới một đối tư ợng. Phu ơng thức được gọi sau đó sẽ nhận bản sao của biến tham chiêu của chính nó để làm với nó những gì nó thích. Nhưng vì hai biến tham chiêu giống hệt nhau tham chiêu đến cùng một đối tư ợng chính xác, nếu phu ơng thức được gọi sửa đổi đối tư ợng (ví dụ: bằng cách gọi các phu ơng thức setter), người gọi sẽ thấy rằng đối tư ợng mà biến ban đầu của người gọi tham chiêu đến cũng đã bị thay đổi. Trong phần tiếp theo, chúng ta sẽ xem bức tranh thay đổi như thế nào khi chúng ta đang nói về nguyên thủy.

Điểm mấu chốt về giá trị truyền: Phu ơng thức được gọi không thể thay đổi biến của người gọi, mặc dù đối với các biến tham chiêu đối tư ợng, phu ơng thức được gọi có thể thay đổi đối tư ợng mà biến được tham chiêu. Sự khác biệt giữa thay đổi biến và thay đổi đối tư ợng là gì? Đối với tham chiêu đối tư ợng, điều đó có nghĩa là phu ơng thức được gọi không thể gán lại biến tham chiêu ban đầu của người gọi và làm cho nó tham chiêu đến một đối tư ợng khác hoặc null. Ví dụ, trong đoạn mã sau,

```

void bar() {
    Foo f = new Foo();
    doStuff(f);
}
void doStuff(Foo g) {
    g.setName("Boo");
    g = new Foo();
}

```

gán lại g không gán lại f! Ở cuối phư ơng thức bar () , hai đối tư ợng Foo đã đư ợc tạo: một đối tư ợng đư ợc tham chiếu bởi biến cục bộ f và một đư ợc tham chiếu bởi biến cục bộ (đối số) g. Bởi vì phư ơng thức doStuff () có một bản sao của biến tham chiếu, nó có một cách đẻ truy cập đối tư ợng Foo ban đầu , ví dụ như đẻ gọi phư ơng thức setName () . Nhưng phư ơng thức doStuff () không có cách nào đẻ truy cập biến tham chiếu f . Vì vậy, doStuff () có thể thay đổi các giá trị trong đối tư ợng mà f tham chiếu đến, nhưng doStuff () không thể thay đổi nội dung thực tế (mẫu bit) của f. Nói cách khác, doStuff () có thể thay đổi trạng thái của đối tư ợng mà f tham chiếu đến, nhưng nó không thể làm cho f tham chiếu đến một đối tư ợng khác!

Chuyển các biến số nguyên thủy

Hãy xem điều gì sẽ xảy ra khi một biến nguyên thủy đư ợc truyền cho một phư ơng thức:

```

class ReferenceTest {
    public static void main (String [] args) {
        int a = 1;
        ReferenceTest rt = new ReferenceTest();
        System.out.println("Before modify() a = " + a);
        rt.modify(a);
        System.out.println("After modify() a = " + a);
    }
    void modify(int number) {
        number = number + 1;
        System.out.println("number = " + number);
    }
}

```

Trong chư ơng trình đơn giản này, biến a đư ợc chuyển đến một phư ơng thức có tên là mod () , làm tăng biến số lên 1. Kết quả đầu ra trông giống như sau:

```

Before modify() a = 1
number = 2
After modify() a = 1

```

Lưu ý rằng a không thay đổi sau khi nó được chuyển cho phuơng thức. Hãy nhớ rằng, nó là một bản sao của một đã được chuyển cho phuơng thức. Khi một biến nguyên thủy được truyền cho một phuơng thức, nó sẽ được truyền theo giá trị, có nghĩa là truyền từng bit trong biến.

TỪ LỚP HỌC

Thế giới bóng tối của các biến Ngay khi bạn

nghĩ rằng bạn đã tìm ra tất cả, bạn sẽ thấy một đoạn mã với các biến không hoạt động theo cách bạn nghĩ. Bạn có thể đã vấp phải mã với một biến bị che khuất. Bạn có thể phủ bóng một biến theo một số cách. Chúng tôi sẽ xem xét một cách có thể khiến bạn khó hiểu: ẩn một biến tinh bằng cách phủ bóng nó với một biến cục bộ.

Shadowing liên quan đến việc sử dụng lại một tên biến đã được khai báo ở một nơi khác. Tác dụng của việc tạo bóng là ẩn biến đã khai báo trước đó theo cách mà nó có thể trông như thể bạn đang sử dụng biến ẩn, nhưng thực ra bạn đang sử dụng biến ẩn. Bạn có thể tìm thấy lý do để có tình che giấu một biến, nhưng thông thường nó xảy ra một cách tình cờ và gây ra các lỗi khó tìm. Trong bài kiểm tra, bạn có thể mong đợi thấy các câu hỏi mà bóng tối đóng một vai trò nào đó.

Bạn có thể tạo bóng một biến bằng cách khai báo một biến cục bộ giống nhau tên, trực tiếp hoặc là một phần của đối số:

```
class Foo {
    static int size = 7;
    static void changeIt(int size) {
        size = size + 200;
        System.out.println("size in changeIt is " + size);
    }
    public static void main (String [] args) {
        Foo f = new Foo();
        System.out.println("size = " + size);
        changeIt(size);
        System.out.println("size after changeIt is " + size);
    }
}
```

Đoạn mã trứ ớc xuất hiện để thay đổi biến kích thư ớc tĩnh trong phư ơng thức changeIt () , như ng vì changeIt () có một tham số có tên là kích thư ớc, biến kích thư ớc cục bộ đư ợc sửa đổi trong khi biến kích thư ớc tĩnh không bị ảnh hư ởng.

Đang chạy Foo lớp in ra cái này:

```
%java Foo
size = 7
size in changeIt is 207
size after changeIt is 7
```

Mọi thứ trở nên thú vị hơn khi biến bị che khuất là một đối tư ợng tham chiêu, thay vì nguyên thủy:

```
class Bar {
    int barNum = 28;
}

class Foo {
    Bar myBar = new Bar();
    void changeIt(Bar myBar) {
        myBar.barNum = 99;
        System.out.println("myBar.barNum in changeIt is " + myBar.barNum);
        myBar = new Bar();
        myBar.barNum = 420;
        System.out.println("myBar.barNum in changeIt is now " + myBar.barNum);
    }
    public static void main (String [] args) {
        Foo f = new Foo();
        System.out.println("f.myBar.barNum is " + f.myBar.barNum);
        f.changeIt(f.myBar);
        System.out.println("f.myBar.barNum after changeIt is "
                           + f.myBar.barNum);
    }
}
```

Mã trứ ớc in ra sau:

```
f.myBar.barNum is 28
myBar.barNum in changeIt is 99
myBar.barNum in changeIt is now 420
f.myBar.barNum after changeIt is 99
```

Bạn có thể thấy rằng biến đó bóng (tham số cục bộ myBar trong changeIt ()) vẫn có thể ảnh hư ởng đến biến cá thể myBar , vì tham số myBar nhận tham chiêu đến cùng một đối tư ợng Bar . Nhưng khi myBar cục bộ đư ợc chỉ định lại một đối tư ợng Bar mới , sau đó chúng tôi sửa đổi bằng cách thay đổi

giá trị barNum , biến phiên bản myBar ban đầu của Foo không bị ảnh hưởng.

MỤC TIÊU XÁC NHẬN

Thu gom rác (Mục tiêu 2.4 của OCA)

2.4 Giải thích vòng đời của một đối tượng (tạo, "tham chiếu bằng cách gán lại" và thu gom rác)

Cụm từ thu gom rác thường như đến và đi từ mục tiêu của kỳ thi. Đối với kỳ thi OCA 8, nó đã trở lại, và chúng tôi rất vui. Thu gom rác là một ý tưởng nổi tiếng và là một cụm từ phổ biến trong khoa học máy tính.

Tổng quan về Quản lý Bộ nhớ và Rác Bộ sưu tập

Đây là phần bạn đã chờ đợi! Cuối cùng đã đến lúc khám phá thế giới tuyệt vời của quản lý bộ nhớ và thu thập rác.

Quản lý bộ nhớ là một yếu tố quan trọng trong nhiều loại ứng dụng.

Hãy xem xét một chương trình đọc một lượng lớn dữ liệu, chẳng hạn như từ một nơi khác trên mạng, rồi ghi dữ liệu đó vào cơ sở dữ liệu trên ổ cứng. Một thiết kế điển hình sẽ là đọc dữ liệu vào một số loại tập hợp trong bộ nhớ, thực hiện một số thao tác trên dữ liệu và sau đó ghi dữ liệu vào cơ sở dữ liệu.

Sau khi dữ liệu được ghi vào cơ sở dữ liệu, bộ sưu tập lưu trữ dữ liệu tạm thời phải được làm trống dữ liệu cũ hoặc xóa và tạo lại trước khi xử lý đợt tiếp theo. Thao tác này có thể được thực hiện hàng nghìn lần và trong các ngôn ngữ như C hoặc C ++ không cung cấp tính năng thu gom rác tự động, một lỗi hỏng nhỏ trong logic làm trống hoặc xóa cấu trúc dữ liệu thu thập theo cách thủ công có thể cho phép lấy lại một lượng nhỏ bộ nhớ không đúng cách hoặc mất đi. Mỗi mài. Những tổn thất nhỏ này được gọi là rò rỉ bộ nhớ, và qua hàng nghìn lần lặp lại, chúng có thể khiến bộ nhớ không thể truy cập đủ khiến các chương trình cuối cùng sẽ gặp sự cố. Tạo mã thực hiện quản lý bộ nhớ thủ công một cách sạch sẽ và triệt để là một nhiệm vụ không hề nhỏ và phức tạp, và trong khi các ước tính khác nhau, có thể cho rằng quản lý bộ nhớ thủ công có thể tăng gấp đôi nỗ lực phát triển cho một chương trình phức tạp.

Bộ thu gom rác của Java cung cấp một giải pháp tự động để quản lý bộ nhớ.

Trong hầu hết các trường hợp, nó giải phóng bạn khỏi phải thêm bất kỳ bộ nhớ nào

ban quản lý. Trong hầu hết các trường hợp, nó giải phóng bạn khỏi việc phải thêm bất kỳ logic quản lý bộ nhớ nào vào ứng dụng của mình. Như ợc điểm của tính năng thu gom rác tự động là bạn không thể kiểm soát hoàn toàn khi nào nó chạy và khi nào nó không chạy.

Tổng quan về Trình thu gom rác của Java

Hãy xem chúng tôi muốn nói gì khi nói về việc thu gom rác ở vùng đất Java. Từ mức 30.000 foot, thu gom rác là cụm từ đư ợc sử dụng để mô tả việc quản lý bộ nhớ tự động trong Java. Bất cứ khi nào một chương trình phần mềm thực thi (trong Java, C, C++, Lisp, Ruby, v.v.), nó sử dụng bộ nhớ theo một số cách khác nhau. Chúng ta sẽ không đi sâu vào Khoa học Máy tính 101 ở đây, như ng nó là điển hình cho bộ nhớ đư ợc sử dụng để tạo một ngăn xếp, một đồng, trong trường hợp vùng hằng số và vùng phư ơng thức của Java. Heap là một phần của bộ nhớ nơi các đối tượng Java sống, và nó là một phần duy nhất của bộ nhớ tham gia vào quá trình thu gom rác theo bất kỳ cách nào.

Một đồng là một đồng là một đồng. Đối với kỳ thi, điều quan trọng là bạn phải biết rằng bạn có thể gọi nó là đồng, bạn có thể gọi nó là đồng rác thu gom, hoặc bạn có thể gọi nó là Johnson, nhưng có một và chỉ một đồng.

Vì vậy, tất cả việc thu gom rác đều xoay quanh việc đảm bảo heap có nhiều không gian trống nhất có thể. Đối với mục đích của bài kiểm tra, điều này tổng hợp lại là xóa bất kỳ đối tượng nào mà chương trình Java đang chạy không còn có thể truy cập đư ợc nữa. Chúng ta sẽ nói thêm về "có thể tiếp cận" nghĩa là gì sau một phút nữa, như ng hãy đi sâu vào vấn đề này. Khi trình thu gom rác chạy, mục đích của nó là tìm và xóa các đối tượng không thể tiếp cận. Nếu bạn nghĩ rằng một chương trình Java đang trong một chu kỳ liên tục tạo ra các đối tượng mà nó cần (chiếm không gian trên heap) và sau đó loại bỏ chúng khi chúng không còn cần thiết, tạo các đối tượng mới, loại bỏ chúng, v.v. mảnh ghép còn thiếu là người thu gom rác. Khi nó chạy, nó sẽ tìm kiếm những đối tượng bị loại bỏ và xóa chúng khỏi bộ nhớ, để chương trình sử dụng bộ nhớ và giải phóng nó có thể tiếp tục.

Ah, vòng tròn tuyệt vời của cuộc sống.

Khi nào thì bộ thu gom rác chạy?

Người thu gom rác dưới sự kiểm soát của JVM; JVM quyết định thời điểm chạy bộ thu gom rác. Từ bên trong chương trình Java của mình, bạn có thể yêu cầu JVM chạy bộ thu gom rác; như ng không có đảm bảo nào, trong bất kỳ trường hợp nào, rằng JVM sẽ tuân thủ. Để lại cho các thiết bị của chính nó, JVM thường sẽ chạy bộ thu gom rác khi nó nhận thấy rằng bộ nhớ sắp hết.

Kinh nghiệm chỉ ra rằng khi chương trình Java của bạn đưa ra yêu cầu thu gom rác, JVM thường sẽ cấp yêu cầu của bạn theo thứ tự ngắn, như ng có

thu thập, JVM thư ờng sẽ cung cấp yêu cầu của bạn trong thời gian ngắn, như ng không có đảm bảo. Chỉ khi bạn nghĩ rằng bạn có thể tin tưởng vào nó, JVM sẽ quyết định bỏ qua yêu cầu của bạn.

Bộ thu gom rác hoạt động như thế nào?

Bạn chỉ không thể chắc chắn. Bạn có thể nghe nói rằng bộ thu gom rác sử dụng một thuật toán đánh dấu và quét và đối với bất kỳ triển khai Java nhất định nào có thể đúng, như ng đặc tả Java không đảm bảo bất kỳ triển khai cụ thể nào. Bạn có thể nghe nói rằng bộ thu gom rác sử dụng phư ơng pháp đếm tham chiếu; một lần nữa, có thể có, có thể không. Khái niệm quan trọng để bạn hiểu cho kỳ thi là: Khi nào một đối tư ợng đủ điều kiện để thu gom rác?

Tóm lại, mọi chương trình Java đều có từ một đến nhiều luồng. Mỗi luồng có một ngăn xếp thực thi nhỏ của riêng nó. Thông thường, bạn (người lập trình) tạo ra ít nhất một luồng chạy trong chương trình Java, luồng có phư ơng thức main () ở cuối ngăn xếp. Tuy nhiên, có nhiều lý do thực sự tuyệt vời để khởi chạy các chuỗi bổ sung từ chuỗi ban đầu của bạn (Bạn sẽ gặp phải vấn đề này nếu chuẩn bị cho kỳ thi OCP 8). Ngoài việc có một ngăn xếp thực thi nhỏ của riêng nó, mỗi luồng có vòng đời riêng của nó. Hiện tại, tất cả những gì bạn cần biết là các chủ đề có thể còn sống hoặc đã chết.

Với thông tin cơ bản này, bây giờ chúng ta có thể nói rõ ràng tuyệt vời và giải quyết rằng một đối tư ợng đủ điều kiện để thu gom rác khi không có luồng trực tiếp nào có thể truy cập vào nó. (Lưu ý: Do sự khác biệt của nhóm hàng số Chuỗi, bài kiểm tra tập trung các câu hỏi thu thập rác vào các đối tư ợng không phải Chuỗi và do đó, các cuộc thảo luận về thu gom rác của chúng tôi cũng chỉ áp dụng cho các đối tư ợng không phải Chuỗi.)

Dựa trên định nghĩa đó, bộ thu gom rác thực hiện một số hoạt động kỳ diệu, chưa được biết đến; và khi nó phát hiện ra một đối tư ợng mà bất kỳ luồng trực tiếp nào không thể tiếp cận được, nó sẽ coi đối tư ợng đó đủ điều kiện để xóa và thậm chí nó có thể xóa nó vào một thời điểm nào đó. (Bạn đoán xem: nó cũng có thể không bao giờ xóa nó.) Khi chúng ta nói về việc tiếp cận một đối tư ợng, chúng ta thực sự đang nói về việc có một biến tham chiếu có thể truy cập đến đối tư ợng đó được đề cập. Nếu chương trình Java của chúng tôi có một biến tham chiếu tham chiếu đến một đối tư ợng và biến tham chiếu đó có sẵn cho một luồng trực tiếp, thì đối tư ợng đó được coi là có thể truy cập được. Chúng ta sẽ nói thêm về cách các đối tư ợng có thể trở nên không thể truy cập được trong phần sau.

Ứng dụng Java có thể hết bộ nhớ không? Đúng. Thu gom rác hệ thống cố gắng xóa các đối tư ợng khỏi bộ nhớ khi chúng không được sử dụng. Tuy nhiên, nếu bạn duy trì quá nhiều đối tư ợng trực tiếp (đối tư ợng được tham chiếu từ các đối tư ợng trực tiếp khác), hệ thống có thể hết bộ nhớ. Việc thu gom rác không thể đảm bảo rằng có đủ bộ nhớ, chỉ có điều bộ nhớ có sẵn sẽ được quản lý hiệu quả nhất có thể.

đư ợc quản lý hiệu quả nhất có thể.

Viết mã rõ ràng làm cho các đối tượng đủ điều kiện để thu thập

Trong phần trước, bạn đã học các lý thuyết đằng sau việc thu gom rác của Java. Trong phần này, chúng tôi chỉ ra cách làm cho các đối tượng đủ điều kiện để thu gom rác bằng cách sử dụng mã thực tế. Chúng tôi cũng thảo luận về cách cố gắng buộc thu thập rác nếu cần thiết và cách bạn có thể thực hiện dọn dẹp bổ sung trên các đối tượng trước khi chúng bị xóa khỏi bộ nhớ.

Vô hiệu hóa một tham chiếu Như

chúng ta đã thảo luận trước đó, một đối tượng trở nên đủ điều kiện để thu gom rác khi không còn tham chiếu nào có thể truy cập đư ợc nữa. Rõ ràng, nếu không có tham chiếu nào có thể truy cập đư ợc, thì điều gì xảy ra với đối tượng cũng không thành vấn đề. Đối với mục đích của chúng tôi, nó chỉ lơ lửng trong không gian, không đư ợc sử dụng, không thể truy cập và không còn cần thiết nữa.

Cách đầu tiên để xóa tham chiếu đến một đối tượng là đặt tham chiếu biến tham chiếu đến đối tượng thành null. Kiểm tra đoạn mã sau:

```

1. public class GarbageTruck {
2.     public static void main(String [] args) {
3.         StringBuffer sb = new StringBuffer("hello");
4.         System.out.println(sb);
5.         // The StringBuffer object is not eligible for collection
6.         sb = null;
7.         // Now the StringBuffer object is eligible for collection
8.     }
9. }
```

Đối tượng StringBuffer với giá trị hello đư ợc gán cho biến tham chiếu sb ở dòng thứ ba. Để làm cho đối tượng đủ điều kiện (để thu gom rác), chúng tôi đặt biến tham chiếu sb thành null, biến tham chiếu này loại bỏ tham chiếu duy nhất tồn tại đối với đối tượng StringBuffer . Khi dòng 6 đã chạy, đối tượng StringBuffer xin chào nhỏ vui vẻ của chúng ta sẽ bị hủy diệt, đủ điều kiện để thu gom rác.

Gán lại một biến tham chiếu Chúng ta cũng

có thể tách một biến tham chiếu khỏi một đối tượng bằng cách đặt biến tham chiếu để tham chiếu đến một đối tượng khác. Kiểm tra đoạn mã sau:

```

class GarbageTruck {
    public static void main(String [] args) {
        StringBuffer s1 = new StringBuffer("hello");
        StringBuffer s2 = new StringBuffer("goodbye");
        System.out.println(s1);
        // At this point the StringBuffer "hello" is not eligible
        s1 = s2; // Redirects s1 to refer to the "goodbye" object
        // Now the StringBuffer "hello" is eligible for collection
    }
}

```

Các đối tượng được tạo trong một phư ơng thức cũng cần được xem xét. Khi một phư ơng thức được gọi, bất kỳ biến cục bộ nào được tạo chỉ tồn tại trong thời gian của phư ơng thức. Khi phư ơng thức đã trả về, các đối tượng được tạo trong phư ơng thức đủ điều kiện để thu gom rác. Tuy nhiên, có một ngoại lệ rõ ràng. Nếu một đối tượng được trả về từ phư ơng thức, thì tham chiếu của nó có thể được gán cho một biến tham chiếu trong phư ơng thức đã gọi nó; do đó, nó sẽ không đủ điều kiện để thu thập. Kiểm tra đoạn mã sau:

```

import java.util.Date;
public class GarbageFactory {
    public static void main(String [] args) {
        Date d = getDate();
        doComplicatedStuff();
        System.out.println("d = " + d);
    }

    public static Date getDate() {
        Date d2 = new Date();
        StringBuffer now = new StringBuffer(d2.toString());
        System.out.println(now);
        return d2;
    }
}

```

Trong ví dụ trư ớc, chúng tôi đã tạo một phư ơng thức có tên getDate () trả về một đối tượng Ngày . Phư ơng thức này tạo ra hai đối tượng: Date và StringBuffer chứa thông tin ngày. Vì phư ơng thức trả về một tham chiếu đến đối tượng Date và tham chiếu này được gán cho một biến cục bộ, nó sẽ không đủ điều kiện để thu thập ngay cả sau khi phư ơng thức getDate () đã hoàn thành. Tuy nhiên , đối tượng StringBuffer sẽ đủ điều kiện, mặc dù chúng tôi đã không đặt biến now thành null một cách rõ ràng.

Cô lập một tham chiếu

Cô lập một tham chiếu Có

một cách khác mà các đối tượng có thể đủ điều kiện để thu gom rác, ngay cả khi chúng vẫn có các tham chiếu hợp lệ! Chúng tôi gọi kịch bản này là “những hòn đảo cô lập”.

Một ví dụ đơn giản là một lớp có một biến thể hiện là một tham chiếu biến thành một thê hiện khác của cùng một lớp. Hãy giờ hãy thử ờng tư ợng rằng hai trư ờng hợp như vậy tồn tại và chúng tham chiếu đến nhau. Nếu tắt cả các tham chiếu khác đến hai đối tượng này bị xóa, thì mặc dù mỗi đối tượng vẫn có một tham chiếu hợp lệ, sẽ không có cách nào cho bất kỳ luồng trực tiếp nào truy cập vào một trong hai đối tượng. Khi bộ thu gom rác chạy, nó thư ờng có thể phát hiện ra bất kỳ đảo vật thể nào như vậy và loại bỏ chúng. Như bạn có thể ờng tư ợng, những hòn đảo như vậy có thể trở nên khá lớn, về mặt lý thuyết chứa hàng trăm vật thể. Kiểm tra đoạn mã sau:

```
public class Island {
    Island i;
    public static void main(String [] args) {
        Island i2 = new Island();
        Island i3 = new Island();
        Island i4 = new Island();

        i2.i = i3;    // i2 refers to i3
        i3.i = i4;    // i3 refers to i4
        i4.i = i2;    // i4 refers to i2

        i2 = null;
        i3 = null;
        i4 = null;

        // do complicated, memory intensive stuff
    }
}
```

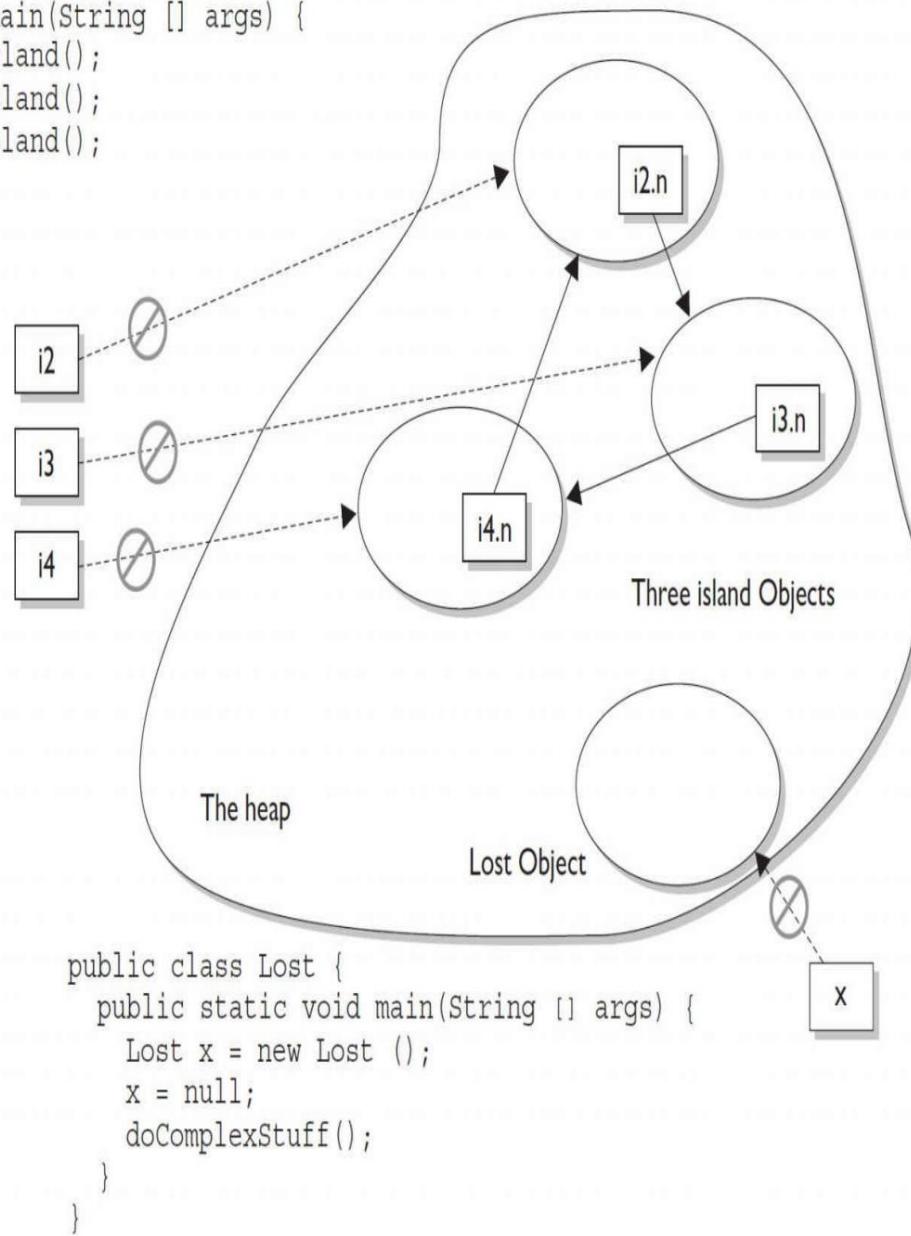
Khi mã đạt đến // làm phức tạp, ba đối tượng Island (trư ớc đây được gọi là i2, i3 và i4) có các biến thể hiện để chúng tham chiếu đến nhau, như ng liên kết của chúng với thế giới bên ngoài (i2, i3 và i4) đã bị vô hiệu hóa. Ba đối tượng này đủ điều kiện để thu gom rác.

Điều này bao gồm mọi thứ bạn cần biết về việc làm cho các đối tượng đủ điều kiện để thu gom rác. Nghiên cứu [hình 3-2](#) để củng cố các khái niệm về các đối tượng không có tham chiếu và đảo cô lập.

```

public class Island {
    Island n;
    public static void main(String [] args) {
        Island i2 = new Island();
        Island i3 = new Island();
        Island i4 = new Island();
        i2.n = i3;
        i3.n = i4;
        i4.n = i2;
        i2 = null;
        i3 = null;
        i4 = null;
        doComplexStuff();
    }
}

```



HÌNH 3-2 Các đối tượng đảo đủ điều kiện để thu gom rác

Buộc thu gom rác Điều đầu tiên

mà chúng ta cần đề cập ở đây là, trái với tiêu đề của phần này, không thể bắt buộc thu gom rác. Tuy nhiên, Java cung cấp một số phương thức cho phép bạn yêu cầu JVM thực hiện việc thu gom rác.

Lưu ý: Trình thu gom rác Java đã phát triển đến trạng thái nâng cao

bạn không nên gọi `System.gc()` trong mã của mình – hãy để nó cho JVM.

Trên thực tế, chỉ có thể đề xuất với JVM rằng nó thực hiện thu gom rác. Tuy nhiên, không có gì đảm bảo rằng JVM sẽ thực sự loại bỏ tất cả các đối tượng không sử dụng khỏi bộ nhớ (ngay cả khi quá trình thu gom rác được chạy). Điều cần thiết là bạn phải hiểu khái niệm này cho kỳ thi.

Các quy trình thu gom rác mà Java cung cấp là thành viên của lớp Runtime . Lớp Runtime là một lớp đặc biệt có một đối tượng duy nhất (một Singleton) cho mỗi chương trình chính. Đối tượng Runtime cung cấp cơ chế giao tiếp trực tiếp với máy ảo. Để lấy cá thể Runtime , bạn có thể sử dụng phương thức `Runtime.getRuntime()`, trả về Singleton. Khi bạn có Singleton, bạn có thể gọi bộ thu gom rác bằng phương thức `gc()` . Ngoài ra, bạn có thể gọi cùng một phương thức trên lớp Hệ thống , lớp này có các phương thức tinh có thể thực hiện công việc lấy Singleton cho bạn. Cách đơn giản nhất để yêu cầu thu gom rác (hãy nhớ– chỉ là một yêu cầu) là

```
Hệ thống.gc();
```

Về mặt lý thuyết, sau khi gọi `System.gc()`, bạn sẽ có nhiều bộ nhớ trống càng tốt. Chúng tôi nói "về mặt lý thuyết" bởi vì thói quen này không phải lúc nào cũng hoạt động theo cách đó. Đầu tiên, JVM của bạn có thể đã không thực hiện quy trình này; đặc tả ngôn ngữ cho phép quy trình này không làm gì cả. Thứ hai, một luồng khác có thể lấy nhiều bộ nhớ ngay sau khi bạn chạy trình thu gom rác.

Điều này không có nghĩa là `System.gc()` là một phương thức vô dụng – nó tốt hơn nhiều so với Không có gì. Bạn chỉ không thể dựa vào `System.gc()` để giải phóng đủ bộ nhớ để không phải lo lắng về việc hết bộ nhớ. Kỳ thi Chứng nhận quan tâm đến hành vi được đảm bảo, không phải là hành vi có thể xảy ra.

Bây giờ bạn đã phần nào làm quen với cách hoạt động của nó, hãy cùng làm một thử nghiệm nhỏ để xem tác dụng của việc thu gom rác. Chương trình sau đây cho chúng ta biết tổng bộ nhớ JVM có sẵn cho nó và bao nhiêu bộ nhớ trống mà nó có. Sau đó, nó tạo ra 10.000 đối tượng Date . Sau đó, nó cho chúng ta biết dung lượng bộ nhớ còn lại và sau đó gọi bộ thu gom rác (nếu nó quyết định chạy, sẽ tạm dừng chương trình cho đến khi tắt cả các đối tượng không sử dụng được loại bỏ). Kết quả bộ nhớ trống cuối cùng sẽ cho biết liệu nó đã chạy hay chưa. Hãy cùng xem chương trình:

```

1. import java.util.Date;
2. public class CheckGC {
3.     public static void main(String [] args) {
4.         Runtime rt = Runtime.getRuntime();
5.         System.out.println("Total JVM memory: "
6.                             + rt.totalMemory());
7.         System.out.println("Before Memory = "
8.                             + rt.freeMemory());
9.         Date d = null;
10.        for(int i = 0;i<10000;i++) {
11.            d = new Date();
12.            d = null;
13.        }
14.        System.out.println("After Memory = "
15.                            + rt.freeMemory());
16.        rt.gc(); // an alternate to System.gc()
17.        System.out.println("After GC Memory = "
18.                            + rt.freeMemory());
19.    }
20. }
```

Now, let's run the program and check the results:

```

Total JVM memory: 1048568
Before Memory = 703008
After Memory = 458048
After GC Memory = 818272
```

Như bạn có thể thấy, JVM thực sự đã quyết định thu gom rác (nghĩa là xóa) các đối tượng đủ điều kiện. Trong ví dụ trước, chúng tôi đã đề xuất rằng JVM thực hiện thu gom rác với 458.048 byte bộ nhớ còn lại và nó đáp ứng yêu cầu của chúng tôi. Chương trình này chỉ có một luồng ngủ ời dùng đang chạy, vì vậy không có gì khác xảy ra khi chúng tôi gọi `rt.gc()`. Hãy nhớ rằng hành vi khi `gc()` được gọi có thể khác nhau đối với các JVM khác nhau; do đó, không có gì đảm bảo rằng các đối tượng không sử dụng sẽ bị xóa khỏi bộ nhớ. Về điều duy nhất bạn có thể đảm bảo là nếu bạn sắp hết bộ nhớ, trình thu gom rác sẽ chạy trước khi nó ném ra một `OutOfMemoryException`.

BÀI TẬP 3-2

Thử nghiệm thu gom rác

Hãy thử thay đổi chương trình `CheckGC` bằng cách đặt dòng 13 và 14 bên trong một vòng lặp.

Bạn có thể thấy rằng không phải tất cả bộ nhớ đều được giải phóng trên bất kỳ lần chạy GC nào.

Bạn có thể thấy rằng không phải tất cả bộ nhớ đều được giải phóng trên bất kỳ lần chạy GC nào.

Dọn dẹp Trú ớc khi Thu gom rác – Phư ơng thức finalize () Java cung cấp một cơ chế cho phép bạn chạy một số mã ngay trước khi đối tượng của bạn bị bộ thu gom rác xóa. Mã này nằm trong một phư ơng thức có tên là finalize () mà tắt cả các lớp kế thừa từ Lớp Đối tượng. Nhìn bề ngoài, điều này nghe có vẻ là một ý tưởng tuyệt vời; có thể đối tượng của bạn đã mở ra một số tài nguyên và bạn muốn đóng chúng trước khi đối tượng của bạn bị xóa. Vẫn đe là, như bạn có thể đã thu thập bây giờ, bạn không bao giờ có thể tin tưởng vào trình thu gom rác để xóa một đối tượng. Vì vậy, bất kỳ mã nào bạn đưa vào phư ơng thức finalize () bị ghi đè của lớp bạn đều không được đảm bảo chạy. Bởi vì phư ơng thức finalize () cho bất kỳ đối tượng nhất định nào có thể chạy, nhưng bạn không thể tin tưởng vào nó, đừng đặt bất kỳ mã thiết yếu nào vào phư ơng thức finalize () của bạn. Trên thực tế, chúng tôi khuyên bạn, nói chung, bạn không nên ghi đè finalize () .

Tricky Little finalize () Gotchas

Có một số khái niệm liên quan đến finalize () mà bạn cần nhớ:

- Đối với bất kỳ đối tượng nhất định nào, finalize () sẽ chỉ được gọi một lần (tối đa) bởi bộ thu gom rác.
- Việc gọi finalize () thực sự có thể dẫn đến việc lưu một đối tượng khỏi bị xóa.

Hãy xem xét những tuyên bố này sâu hơn chút. Trước hết, hãy nhớ rằng bất kỳ mã bạn có thể đưa vào một phư ơng thức bình thường mà bạn có thể đưa vào finalize (). Ví dụ: trong phư ơng thức finalize (), bạn có thể viết mã chuyển một tham chiếu đến đối tượng được đề cập trở lại đối tượng khác, không đủ điều kiện để ẩn đối tượng một cách hiệu quả để thu gom rác. Nếu tại một thời điểm nào đó, cùng một đối tượng này lại đủ điều kiện để thu gom rác, thì bộ thu gom rác vẫn có thể xử lý và xóa đối tượng đó. Tuy nhiên, bộ thu gom rác sẽ nhớ rằng đối với đối tượng này, finalize () đã chạy và nó sẽ không chạy lại finalize () .

TÓM TẮT CHÚNG NHẬN

Chư ơng này bao gồm một loạt các chủ đề. Đừng lo lắng nếu bạn phải xem lại một số chủ đề này khi bạn vào các chư ơng sau. Chư ơng này bao gồm rất nhiều nội dung cơ bản sẽ phát huy tác dụng sau này.

những thứ cơ bản sẽ phát huy tác dụng sau này.

Chúng tôi bắt đầu chư ơng bằng cách xem xét ngăn xếp và đống; hãy nhớ rằng các biến cục bộ sống trên ngăn xếp và các biến cá thể sống cùng với các đối tượng của chúng trên đống.

Chúng tôi đã xem xét các ký tự pháp lý cho các nguyên thủy và chuỗi, sau đó chúng tôi thảo luận về những điều cơ bản của việc gán giá trị cho các nguyên thủy và các biến tham chiếu cũng như các quy tắc để truyền nguyên thủy.

Tiếp theo, chúng tôi thảo luận về khái niệm phạm vi hoặc "Biến này sẽ tồn tại trong bao lâu?" Hãy nhớ bốn phạm vi cơ bản để giảm tuổi thọ: static, instance, local và block.

Chúng tôi đã đề cập đến ý nghĩa của việc sử dụng các biến chưa được khởi tạo và tầm quan trọng của thực tế là các biến cục bộ PHẢI được gán giá trị một cách rõ ràng.

Chúng tôi đã nói về một số khía cạnh phức tạp của việc gán một biến tham chiếu này cho một biến tham chiếu khác và một số điểm tốt hơn của việc chuyển các biến vào các phu ứng thức, bao gồm cả thảo luận về "shadowing".

Cuối cùng, chúng tôi đi sâu vào thu rác, tính năng quản lý bộ nhớ tự động của Java. Chúng tôi đã học được rằng đống là nơi các đối tượng sinh sống và là nơi diễn ra tất cả các hoạt động thu gom rác mát mẻ. Chúng tôi biết rằng cuối cùng, JVM sẽ thực hiện thu gom rác bất cứ khi nào nó muốn. Bạn (lập trình viên) có thể yêu cầu chạy thu gom rác, nhưng bạn không thể ép buộc. Chúng tôi đã nói về việc thu gom rác chỉ áp dụng cho các đối tượng đủ điều kiện và điều kiện đủ điều kiện có nghĩa là "không thể truy cập được từ bất kỳ chuỗi trực tiếp nào". Cuối cùng, chúng ta đã thảo luận về phu ứng thức finalize () hiếm khi hữu ích và những gì bạn sẽ phải biết về nó cho kỳ thi. Nói chung, đây là một chư ơng hấp dẫn.

✓ KHOAN HAI PHÚT

Dưới đây là một số điểm chính từ chư ơng này.

Stack và Heap Các biến

- cục bộ (biến phu ứng thức) sống trên ngăn xếp.
- Các đối tượng và các biến thể hiện của chúng sống trên heap.

Chữ viết và đúc nguyên thủy (Mục tiêu 2.1 của OCA)

- Các ký tự số nguyên có thể là nhị phân, thập phân, bát phân (chẳng hạn như 013) hoặc thập lục phân

(chẳng hạn như 0x3d).

- Chữ viết cho dài kết thúc bằng L hoặc l. (Để dễ đọc, chúng tôi đề xuất "L".)
- Các ký tự nối kết thúc bằng F hoặc f và các ký tự kép kết thúc bằng một chữ số hoặc D hoặc d.
- Các ký tự boolean là true và false.
- Chữ viết cho ký tự là một ký tự duy nhất bên trong dấu ngoặc kép: 'd'.

Phạm vi (Mục tiêu 1.1 của OCA)

- Phạm vi đề cập đến thời gian tồn tại của một biến.
- Có bốn phạm vi cơ bản:
 - Các biến static về cơ bản tồn tại miễn là lớp của chúng tồn tại.
 - Các biến cá thể tồn tại miễn là đối tượng của chúng còn sống.
 - Các biến cục bộ tồn tại miễn là phu ứng thức của chúng nằm trên ngăn xếp; tuy nhiên, nếu phu ứng thức của họ gọi phu ứng thức khác, chúng tạm thời không khả dụng.
- Các biến khôi (ví dụ: trong for hoặc if) tồn tại cho đến khi khôi hoàn thành.

Nhiệm vụ cơ bản (Mục tiêu của OCA 2.1, 2.2 và 2.3)

- Các số nguyên theo nghĩa đen hoàn toàn là int.
- Biểu thức số nguyên luôn dẫn đến kết quả có kích thước int, không bao giờ nhỏ hơn.
- Số dấu phẩy động được tăng gấp đôi cách ngầm định (64 bit).
- Việc thu hẹp một nguyên thủy sẽ cắt ngắn các bit bậc cao.
- Phép gán ghép (chẳng hạn như +=) thực hiện một phép truyền tự động.
- Một biến tham chiếu giữ các bit được sử dụng để tham chiếu đến một đối tượng.
- Các biến tham chiếu có thể tham chiếu đến các lớp con của kiểu đã khai báo như ng không tham chiếu đến các lớp cha.
- Khi bạn tạo một đối tượng mới, chẳng hạn như Nút b = new Button ();, JVM thực hiện ba việc:
 - Tạo một biến tham chiếu có tên b, kiểu Nút.
 -
 - Tạo một đối tượng Nút mới.

- Gán đối tượng Button cho biến tham chiếu b.

Sử dụng một phần tử biến hoặc mảng chứa được khởi tạo và chưa được gán (Mục tiêu của OCA 4.1 và 4.2)

- Khi một mảng đối tượng được khởi tạo, các đối tượng trong mảng không được khởi tạo tự động, nhưng tất cả các tham chiếu đều nhận giá trị mặc định là null.
- Khi một mảng nguyên thủy được khởi tạo, các phần tử nhận giá trị mặc định.
- Các biến cá thể luôn được khởi tạo với một giá trị mặc định.
- Các biến cục bộ / tự động / phư ơng thức không bao giờ được cung cấp một giá trị mặc định. Nếu bạn cố gắng sử dụng một cái trư ớc khi khởi chạy nó, bạn sẽ gặp lỗi trình biên dịch.

Chuyển các biến vào các phư ơng thức (Mục tiêu 6.6 của OCA)

- Các phư ơng thức có thể lấy các tham chiếu nguyên thủy và / hoặc đối tượng làm đối số.
- Các đối số của phư ơng thức luôn là bản sao.
- Đối số phư ơng thức không bao giờ là đối tượng thực tế (chúng có thể là tham chiếu đến đối tượng).
- Một đối số nguyên thủy là một bản sao không đính kèm của đối số nguyên thủy ban đầu.
- Đối số tham chiếu là một bản sao khác của tham chiếu đến đối tượng gốc.
- Shadowing xảy ra khi hai biến có phạm vi khác nhau có cùng tên. Điều này dẫn đến những lỗi khó tìm và những đề thi khó trả lời.

Thu gom rác (Mục tiêu 2.4 của OCA)

- Trong Java, bộ thu gom rác (GC) cung cấp khả năng quản lý bộ nhớ tự động.
- Mục đích của GC là xóa các đối tượng không thể với tới.
- Chỉ JVM mới quyết định thời điểm chạy GC; bạn chỉ có thể để xuất nó.
- Bạn không thể biết chắc chắn thuật toán GC.
- Các đối tượng phải được coi là đủ điều kiện trư ớc khi chúng có thể được thu gom.
- Một đối tượng đủ điều kiện khi không có luồng trực tiếp nào có thể tiếp cận nó.

- Để tiếp cận một đối tượng, bạn phải có một tham chiếu trực tiếp, có thể truy cập đến đối tượng đó.
- Các ứng dụng Java có thể hết bộ nhớ.
- Quản lý của các đối tượng có thể được thu gom rác, mặc dù chúng tham chiếu đến nhau.
- Yêu cầu thu gom rác với System.gc();.
- Lớp Object có một phương thức finalize().
- Phương thức finalize() được đảm bảo chạy một lần và chỉ một lần trước khi trình thu gom rác xóa một đối tượng.
- Ngoài thu gom rác không đảm bảo; finalize() có thể không bao giờ chạy.
- Bạn có thể không đủ điều kiện-ize một đối tượng cho GC từ bên trong finalize().

TỰ KIỂM TRA

1. Cho:

```
class CardBoard {  
    Short story = 200;  
    CardBoard go(CardBoard cb) {  
        cb = null;  
        return cb;  
    }  
    public static void main(String[] args) {  
        CardBoard c1 = new CardBoard();  
        CardBoard c2 = new CardBoard();  
        CardBoard c3 = c1.go(c2);  
        c1 = null;  
        // do Stuff  
    } }
```

Khi đạt đến // do Stuff , có bao nhiêu đối tượng đủ điều kiện để thu gom rác?

- A. 0
- B. 1
- C. 2
- D. Biên dịch không thành công

E. Không thể biết F. Một ngoại

lệ đư ợc đư a ra trong thời gian chạy

2. Đư a ra:

```
public class Fishing {  
    byte b1 = 4;  
    int i1 = 123456;  
    long L1 = (long)i1;           // line A  
    short s2 = (short)i1;         // line B  
    byte b2 = (byte)i1;          // line C  
    int i2 = (int)123.456;        // line D  
    byte b3 = b1 + 7;             // line E  
}
```

Những dòng nào SẼ KHÔNG biên dịch? (Chọn tất cả các áp dụng.)

A. Dòng A

B. Dòng B

C. Dòng C

D. Dòng D

E. Dòng E

3. Đư a ra:

```
public class Literally {  
    public static void main(String[] args) {  
        int i1 = 1_000;           // line A  
        int i2 = 10_00;            // line B  
        int i3 = _10_000;          // line C  
        int i4 = 0b101010;        // line D  
        int i5 = 0B10_1010;        // line E  
        int i6 = 0x2_a;            // line F  
    }  
}
```

Những dòng nào SẼ KHÔNG biên dịch? (Chọn tất cả các áp dụng.)

A. Dòng A

B. Dòng B

C. Dòng C

D. Dòng D

E. Dòng E

F. Dòng F

4. Cho:

```
class Mixer {  
    Mixer() { }  
    Mixer(Mixer m) { m1 = m; }  
    Mixer m1;  
    public static void main(String[] args) {  
        Mixer m2 = new Mixer();  
        Mixer m3 = new Mixer(m2); m3.go();  
        Mixer m4 = m3.m1; m4.go();  
        Mixer m5 = m2.m1; m5.go();  
    }  
    void go() { System.out.print("hi "); }  
}
```

Kết quả là gì?

A. chào

B. xin chào

C. hi hi hi

D. Biên dịch không

thành công E. hi, sau là một

ngoại lệ F. hi hi, sau là một ngoại lệ

5. Đưa ra:

```
class Fizz {  
    int x = 5;  
    public static void main(String[] args) {  
        final Fizz f1 = new Fizz();  
        Fizz f2 = new Fizz();  
        Fizz f3 = FizzSwitch(f1, f2);  
        System.out.println((f1 == f3) + " " + (f1.x == f3.x));  
    }  
    static Fizz FizzSwitch(Fizz x, Fizz y) {  
        final Fizz z = x;  
        z.x = 6;  
        return z;  
    } }
```

Kết quả là gì?

- A. đúng sự thật
- B. sai đúng
- C. đúng sai
- D. sai sai
- E. Biên dịch không thành công F. Một ngoại lệ được đưa ra trong thời gian chạy

6. Cho:

```
public class Mirror {  
    int size = 7;  
    public static void main(String[] args) {  
        Mirror m1 = new Mirror();  
        Mirror m2 = m1;  
        int i1 = 10;  
        int i2 = i1;  
        go(m2, i2);  
        System.out.println(m1.size + " " + i1);  
    }  
    static void go(Mirror m, int i) {  
        m.size = 8;  
        i = 12;  
    }  
}
```

Kết quả là gì?

A. 7 10

B. 8 10

C. 7 12

D. 8 12

E. Biên dịch không thành

công F. Một ngoại lệ được đưa ra trong thời gian chạy

7. Cho:

```
public class Wind {  
    int id;  
    Wind(int i) { id = i; }  
    public static void main(String[] args) {  
        new Wind(3).go();  
        // commented line  
    }  
    void go() {  
        Wind w1 = new Wind(1);  
        Wind w2 = new Wind(2);  
        System.out.println(w1.id + " " + w2.id);  
    }  
}
```

Khi thực thi đến dòng nhận xét, câu nào đúng? (Chọn tất cả các áp dụng.)

A. Đầu ra chứa 1 B. Đầu

ra chứa 2 C. Đầu ra chứa

3 D. Không Đôi tư ợng

Wind đủ điều kiện để thu gom rác E. Một đôi tư ợng

Wind đủ điều kiện để thu gom rác F. Hai đôi tư ợng

Wind đủ điều kiện để thu gom rác G. Ba đôi tư ợng

Wind đủ điều kiện để thu gom rác

8. Cho:

```
3. public class Ouch {  
4.     static int ouch = 7;  
5.     public static void main(String[] args) {  
6.         new Ouch().go(ouch);  
7.         System.out.print(" " + ouch);  
8.     }  
9.     void go(int ouch) {  
10.        ouch++;  
11.        for(int ouch = 3; ouch < 6; ouch++)  
12.            ;  
13.        System.out.print(" " + ouch);  
14.    }  
15. }
```

Kết quả là gì?

- A. 5 7
- B. 5 8
- C. 8 7
- D. 8 8
- E. Biên dịch không thành công
- F. Một ngoại lệ được đưa ra trong thời gian chạy

9. Cho:

```
public class Happy {  
    int id;  
    Happy(int i) { id = i; }  
    public static void main(String[] args) {  
        Happy h1 = new Happy(1);  
        Happy h2 = h1.go(h1);  
        System.out.println(h2.id);  
    }  
    Happy go(Happy h) {  
        Happy h3 = h;  
        h3.id = 2;  
        h1.id = 3;  
        return h1;  
    }  
}
```

Kết quả là gì?

A. 1

B. 2

C. 3

D. Biên dịch không thành công

E. Một ngoại lệ được đưa ra trong thời gian chạy

10. Cho:

```
public class Network {  
    Network(int x, Network n) {  
        id = x;  
        p = this;  
        if(n != null) p = n;  
    }  
    int id;  
    Network p;  
    public static void main(String[] args) {  
        Network n1 = new Network(1, null);  
        n1.go(n1);  
    }  
    void go(Network n1) {  
        Network n2 = new Network(2, n1);  
        Network n3 = new Network(3, n2);  
        System.out.println(n3.p.p.id);  
    }  
}
```

Kết quả là gì?

A. 1

B. 2

C. 3

D. null

E. Biên dịch không thành công

11. Cho:

```
3. class Beta { }
4. class Alpha {
5.     static Beta b1;
6.     Beta b2;
7. }
8. public class Tester {
9.     public static void main(String[] args) {
10.         Beta b1 = new Beta();        Beta b2 = new Beta();
11.         Alpha a1 = new Alpha();    Alpha a2 = new Alpha();
12.         a1.b1 = b1;
13.         a1.b2 = b1;
14.         a2.b2 = b2;
15.         a1 = null;   b1 = null;   b2 = null;
16.         // do stuff
17.     }
18. }
```

Khi đến vạch 16 thì có bao nhiêu đối tượng đủ điều kiện thu gom rác?

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4
- F. 5

12. Cho:

```
public class Telescope {  
    static int magnify = 2;  
    public static void main(String[] args) {  
        go();  
    }  
    static void go() {  
        int magnify = 3;  
        zoomIn();  
    }  
    static void zoomIn() {  
        magnify *= 5;  
        zoomMore(magnify);  
        System.out.println(magnify);  
    }  
    static void zoomMore(int magnify) {  
        magnify *= 7;  
    }  
}
```

Kết quả là gì?

- A. 2
- B. 10
- C. 15
- D. 30
- E. 70
- F. 105

G. Biên dịch không thành công

13. Cho:

```
3. public class Dark {  
4.     int x = 3;  
5.     public static void main(String[] args) {  
6.         new Dark().go1();  
7.     }  
8.     void go1() {  
9.         int x;  
10.        go2(++x);  
11.    }  
12.    void go2(int y) {  
13.        int x = ++y;  
14.        System.out.println(x);  
15.    }  
16. }
```

Kết quả là gì?

- A. 2
- B. 3
- C. 4
- D. 5
- E. Biên dịch không thành công
- F. Một ngoại lệ được đưa ra trong thời gian chạy

CÂU TRẢ LỜI TỰ KIỂM TRA

1. Đúng. Chỉ một đối tượng CardBoard (c1) đủ điều kiện, nhưng nó có đối tượng trình bao bọc ngắn được liên kết cũng đủ điều kiện.
 A, B, D, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 2.4 của OCA)
2. Đúng; biên dịch dòng E không thành công. Khi một phép toán được thực hiện trên bất kỳ số nguyên nào nhỏ hơn số nguyên , kết quả sẽ tự động được chuyển thành một số nguyên.
 A, B, C và D đều là phôi nguyên thủy hợp pháp. (Mục tiêu 2.1 của OCA)
3. Đúng; dòng C sẽ KHÔNG biên dịch. Kể từ Java 7, dấu gạch dưới có thể được bao gồm trong các ký tự số, nhưng không phải ở đầu hoặc cuối.
 A, B, D, E và F không chính xác. A và B là các ký tự số hợp pháp. D và E là ví dụ về các ký tự nhị phân hợp lệ, mới đổi với Java 7 và F là một ký tự thập lục phân hợp lệ sử dụng dấu gạch dưới. (Mục tiêu OCA)

2.1)

4. F đúng. Biến thể hiện m1 của đối tượng m2 không bao giờ được khởi tạo, vì vậy khi m5 cố gắng sử dụng nó, một NullPointerException sẽ được ném ra.
 A, B, C, D và E là không chính xác dựa trên các điều trên. (Mục tiêu 2.1 và 2.3 của OCA)
5. A đúng. Các tham chiếu f1, z và f3 đều đề cập đến cùng một trường hợp của Fizz. Công cụ sửa đổi cuối cùng đảm bảo rằng một biến tham chiếu không thể được tham chiếu đến một đối tượng khác, nhưng cuối cùng không giữ trạng thái của đối tượng đó không thay đổi.
 B, C, D, E và F không chính xác dựa trên các điều trên. (Mục tiêu 2.2 của OCA)
6. B đúng. Trong phương thức go () , m tham chiếu đến cá thể Mirror duy nhất , nhưng int i là một biến int mới , một bản sao tách rời của i2.
 A, C, D, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 2.2 và 2.3 của OCA)
7. A B và G đều đúng. Hàm tạo đặt giá trị của id cho w1 và w2. Khi đến dòng nhận xét, không có đối tượng nào trong ba đối tượng Wind có thể được truy cập, vì vậy chúng đủ điều kiện để được thu gom rác.
 C, D, E và F là không chính xác dựa trên các điều trên. (Mục tiêu OCA 1.1, 2.3 và 2.4)
8. E đúng. Tham số được khai báo trên dòng 9 là hợp lệ (mặc dù xấu), nhưng tên biến ouch không thể được khai báo lại trên dòng 11 trong cùng phạm vi với khai báo trên dòng 9.
 A, B, C, D và F là không chính xác dựa trên các điều trên. (Mục tiêu 1.1 và 2.1 của OCA)
9. D đúng. Bên trong phương thức go () , h1 nằm ngoài phạm vi.
 A, B, C và E là không chính xác dựa trên các điều trên. (Mục tiêu 1.1 và 6.1 của OCA)
10. A đúng. Ba đối tượng Mạng được tạo. Đối tượng n2 có tham chiếu đến đối tượng n1 và đối tượng n3 có tham chiếu đến đối tượng n2 . SOP có thể được đọc là, "Sử dụng tham chiếu Mạng của đối tượng n3 (p đầu tiên), để tìm tham chiếu của đối tượng đó (n2) và sử dụng tham chiếu của đối tượng đó (p thứ hai) để tìm id của đối tượng (n1) và in id đó . "
 B, C, D và E là không chính xác dựa trên các điều trên. (Mục tiêu OCA, 2.2, 2.3 và 6.4)

11. B đúng. Cần phải rõ ràng rằng vẫn có một tham chiếu đến đối tượng được tham chiếu bởi a2 và vẫn có một tham chiếu đến đối tượng được tham chiếu bởi a2.b2. Điều có thể ít rõ ràng hơn là bạn vẫn có thể truy cập đối tượng Beta khác thông qua biến tĩnh a2.b1 – vì nó là biến tĩnh.

A, C, D, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 2.4 của OCA)

12. B đúng. Trong lớp Kính thiên văn , có ba biến khác nhau được đặt tên là phóng đại. Phiên bản của phương thức go () và phiên bản của phương thức zoomMore () không được sử dụng trong phương thức zoomIn () . Phương thức zoomIn () nhân biến lớp * 5.nKếtqgqgå (khoảng) để trogđemMoreemMoreãnh) nambiung trong zoomMore () . SOP in ra giá trị phóng đại của zoomIn () .

A, C, D, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 1.1 và 6.6 của OCA)

13. E đúng. Trong go1 () , biến cục bộ x không được khởi tạo.

A, B, C, D và F là không chính xác dựa trên các điều trên. (Mục tiêu 2.1 và 2.3 của OCA)



4

Các nhà khai thác

CHỨNG NHẬN MỤC TIÊU

- Sử dụng toán tử Java • Sử dụng dấu ngoặc đơn để ghi đè ưu tiên toán tử • Kiểm tra tính bình đẳng giữa chuỗi và các đối tượng khác bằng cách sử dụng == và bằng ()
- ✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

Nếu bạn có các biến, bạn sẽ sửa đổi chúng. (Trừ khi bạn là một trong những lập trình viên "FP" mới mẻ đó.) Bạn sẽ tăng chúng, cộng chúng lại với nhau và so sánh giữa chúng với nhau (theo khoảng chục cách khác nhau). Trong chương này, bạn sẽ học cách thực hiện tất cả những điều đó trong Java. Như một phần thường bổ sung, bạn sẽ học cách làm những thứ mà có thể bạn sẽ không bao giờ sử dụng trong thế giới thực, nhưng điều đó gần như chắc chắn sẽ có trong kỳ thi.

MỤC TIÊU XÁC NHẬN

Toán tử Java (Mục tiêu OCA 3.1, 3.2 và 3.3)

- 3.1 Sử dụng các toán tử Java; bao gồm dấu ngoặc đơn để ghi đè ưu tiên toán tử.
- 3.2 Kiểm tra sự bình đẳng giữa các Chuỗi và các đối tượng khác bằng cách sử dụng == và equals ().
- 3.3 Tạo if và if / else và các cấu trúc bậc ba

Các toán tử Java tạo ra các giá trị mới từ một hoặc nhiều toán hạng. (Chỉ vì vậy chúng tôi tất cả đều rõ ràng, hãy nhớ rằng toán hạng là những thứ ở bên phải hoặc bên trái của toán tử.) Kết quả của hầu hết các phép toán là một giá trị boolean hoặc số.

Bởi vì bây giờ bạn biết rằng Java không phải là C++, bạn sẽ không ngạc nhiên khi các toán tử Java thường không bị quá tải. Tuy nhiên, có một số toán tử đặc biệt bị quá tải:

- Toán tử + có thể được sử dụng để thêm hai nguyên thủy số với nhau hoặc để thực hiện một phép toán nối nếu một trong hai toán hạng là một Chuỗi.
- Tất cả các toán tử &, |, và ^ đều có thể được sử dụng theo cách khác nhau, mặc dù trong phiên bản kỳ thi này, khả năng xoay bit của chúng sẽ không được kiểm tra.

Tỉnh táo. Các nhà điều hành thường là phần của kỳ thi mà các ứng viên nhìn thấy điểm thấp nhất của họ. Ngoài ra, các toán tử và phép gán là một phần của nhiều câu hỏi liên quan đến các chủ đề khác – sẽ thật đáng tiếc nếu bạn đặt một câu hỏi lambdas thực sự khó nếu chỉ đưa nó vào một câu lệnh tăng trước.

Người điều hành nhiệm vụ

Chúng tôi đã trình bày hầu hết các chức năng của toán tử gán bằng (=) trong [Chương 3](#). Tóm lại:

- Khi gán một giá trị cho một giá trị nguyên thủy, kích thước rất quan trọng. Hãy chắc chắn rằng bạn biết khi nào truyền ngầm sẽ xảy ra, khi nào truyền rõ ràng là cần thiết và khi nào có thể xảy ra việc cắt ngắn.
- Hãy nhớ rằng một biến tham chiếu không phải là một đối tượng; đó là một cách để đến một đối tượng. (Chúng tôi biết tất cả các bạn là các lập trình viên C++ đang chết để chúng tôi nói, "đó là một con trỏ", nhưng chúng tôi sẽ không làm như vậy.)
- Khi gán giá trị cho một biến tham chiếu, kiểu quan trọng. Hãy nhớ các quy tắc cho siêu kiểu, kiểu con và mảng.

Tiếp theo, chúng ta sẽ trình bày thêm một vài chi tiết về các toán tử gán trong bài kiểm tra và khi chúng ta đến chương tiếp theo, chúng ta sẽ xem xét cách thức hoạt động của toán tử gán = với Strings (là bất biến).



Đừng dành thời gian chuẩn bị cho những chủ đề không còn trong kỳ thi!

Các chủ đề sau KHÔNG có trong kỳ thi kể từ Java 1.4:

- Toán tử dịch chuyển bit
- Toán tử bitwise
- Bổ sung của hai
- Công cụ chia cho không

Không phải đây không phải là những chủ đề quan trọng; chỉ là họ không tham gia kỳ thi nữa, và chúng tôi thực sự tập trung vào kỳ thi. (Lưu ý: Lý do chúng tôi đưa ra vấn đề này là vì bạn có thể gặp các câu hỏi thi thử về các chủ đề này – bạn có thể bỏ qua những câu hỏi đó!)

Các toán tử gán ghép Thực tế có 11 toán tử

gán ghép, nhưng chỉ có 4 toán tử được sử dụng phổ biến nhất ($=$, $- =$, $* =$, và $/ =$) là có trong bài kiểm tra. Các toán tử gán ghép cho phép những người đánh máy lười biếng loại bỏ một vài tổ hợp phím khỏi khối lượng công việc của họ.

Dưới đây là một số phép gán ví dụ, đầu tiên không sử dụng toán tử ghép:

```
y = y - 6;  
x = x + 2      * 5;
```

Bây giờ, với các toán tử ghép:

```
y -= 6;  
x += 2      * 5;
```

Hai bài tập cuối cùng cho kết quả tương tự như hai bài đầu tiên.

Toán tử quan hệ

Bài kiểm tra bao gồm sáu toán tử quan hệ ($<$, \leq , $>$, \geq , $=$, và \neq). Quan hệ

toán tử luôn dẫn đến giá trị boolean (true hoặc false) . Giá trị boolean này thường được sử dụng nhất trong kiểm tra if , như sau:

```
int x = 8;
if (x < 9) {
    // do something
}
```

Nhưng giá trị kết quả cũng có thể được gán trực tiếp cho một nguyên thủy boolean :

```
class CompareTest {
    public static void main(String [] args) {
        boolean b = 100 > 99;
        System.out.println("The value of b is " + b);
    }
}
```

Java có bốn toán tử quan hệ có thể được sử dụng để so sánh bất kỳ kết hợp số nguyên, số dấu phẩy động hoặc ký tự:

- > Lớn hơn
- > = Lớn hơn hoặc bằng
- < Ít hơn
- <= Nhỏ hơn hoặc bằng

Hãy xem xét một số so sánh pháp lý:

```
class GuessAnimal {
    public static void main(String[] args) {
        String animal = "unknown";
        int weight = 700;
        char sex = 'm';
        double colorWaveLength = 1.630;
        if (weight >= 500) { animal = "elephant"; }
        if (colorWaveLength > 1.621) { animal = "gray " + animal; }
        if (sex <= 'f') { animal = "female " + animal; }
        System.out.println("The animal is a " + animal);
    }
}
```

Trong đoạn mã trước, chúng tôi đang sử dụng phép so sánh giữa các ký tự. Cũng hợp pháp khi so sánh một ký tự nguyên thủy với bất kỳ số nào (mặc dù nó không phải là phong cách lập trình tuyệt vời). Chạy lớp trước sẽ xuất ra kết quả sau:

phong cách lập trình tuyệt vời). Chạy lớp trước sẽ xuất ra kết quả sau:

Con vật là một con voi xám

Chúng tôi đã đề cập rằng các ký tự có thể được sử dụng trong các toán tử so sánh. Khi so sánh một ký tự với một ký tự hoặc một ký tự với một số, Java sẽ sử dụng giá trị Unicode của ký tự làm giá trị số để so sánh.

Toán tử "Bình đẳng"

Java cũng có hai toán tử quan hệ (đôi khi được gọi là "toán tử bình đẳng") so sánh hai "thứ" giống nhau và trả về một boolean (true hoặc false) đại diện cho điều đúng về hai "thứ" bằng nhau. Các toán tử này là

- == Bằng (còn được gọi là bằng)
- != Không bằng (còn được gọi là không bằng)

Mỗi so sánh riêng lẻ có thể liên quan đến hai số (bao gồm cả ký tự), hai giá trị boolean hoặc hai biến tham chiếu đối tượng. Tuy nhiên, bạn không thể so sánh các loại không tương thích. Điều gì sẽ xảy ra nếu bạn hỏi nếu một boolean bằng một char? Hoặc nếu một Nút bằng một mảng Chuỗi? (Điều này là vô nghĩa, đó là lý do tại sao bạn không thể làm điều đó.) Có bốn loại khác nhau có thể được kiểm tra:

- Con số
- Nhân vật
- Các nguyên thủy Boolean
- Các biến tham chiếu đối tượng

Vậy == nhìn vào cái gì? Giá trị trong biến – nói cách khác, là mẫu bit.

Bình đẳng cho Nguyên thủy

Hầu hết các lập trình viên đã quen với việc so sánh các giá trị nguyên thủy. Đoạn mã sau hiển thị một số kiểm tra bình đẳng trên các biến nguyên thủy:

```

class ComparePrimitives {
    public static void main(String[] args) {
        System.out.println("char 'a' == 'a'? " + ('a' == 'a'));
        System.out.println("char 'a' == 'b'? " + ('a' == 'b'));
        System.out.println("5 != 6? " + (5 != 6));
        System.out.println("5.0 == 5L? " + (5.0 == 5L));
        System.out.println("true == false? " + (true == false));
    }
}

```

Chương trình này tạo ra kết quả sau:

```

char 'a' == 'a'? true
char 'a' == 'b'? false
5 != 6? true
5.0 == 5L? true
true == false? false

```

Như bạn có thể thấy, nếu một số dấu phẩy động được so sánh với một số nguyên và các giá trị giống nhau, toán tử == thường trả về true như mong đợi.

Bình đẳng cho các biến tham chiếu

Như bạn đã thấy trước đó, hai biến tham chiếu có thể tham chiếu đến cùng một đối tượng, như đoạn mã sau minh họa:

```

JButton a = new JButton("Exit");
JButton b = a;

```



Đừng nhầm = cho == trong một biểu thức boolean . Sau đây là hợp pháp:

```

11. boolean b = false;
12. if (b = true) { System.out.println("b is true");
13. } else { System.out.println("b is false"); }

```

Quan sát cẩn thận! Bạn có thể nghĩ rằng kết quả đầu ra là b là sai, nhưng hãy nhìn vào kiểm tra boolean ở dòng 12. Biến boolean b không được so sánh với true; nó đang được đặt thành true. Khi b được đặt thành true , println sẽ thực thi và chúng ta nhận được b là true. Kết quả của bất kỳ biểu thức gán nào là giá trị của biến theo sau phép gán. Đây

thay thế = for == chỉ hoạt động với các biến boolean vì phép kiểm tra if chỉ có thể được thực hiện trên các biểu thức boolean . Do đó, điều này không biên dịch:

```
7. int x = 1;
8. if (x = 0) { }
```

Vì x là một số nguyên (và không phải là boolean) nên kết quả của ($x = 0$) là 0 (kết quả của phép gán). Các int nguyên thủy không thể được sử dụng khi giá trị boolean được mong đợi, vì vậy mã ở dòng 8 sẽ không hoạt động trừ khi nó được thay đổi từ một phép gán (=) thành một phép thử bình đẳng (==) như sau:

```
8. if (x == 0) { }
```

Sau khi chạy đoạn mã này, cả biến a và biến b sẽ tham chiếu đến cùng một đối tượng (một JButton với nhãn Exit). Các biến tham chiếu có thể được kiểm tra để xem liệu chúng có tham chiếu đến cùng một đối tượng hay không bằng cách sử dụng toán tử == . Hãy nhớ rằng, toán tử == đang xem xét các bit trong biến, vì vậy đối với các biến tham chiếu, điều này có nghĩa là nếu các bit trong cả hai biến tham chiếu giống hệt nhau, thì cả hai đều tham chiếu đến cùng một đối tượng. Nhìn vào đoạn mã sau:

```
import javax.swing.JButton;
class CompareReference {
    public static void main(String[] args) {
        JButton a = new JButton("Exit");
        JButton b = new JButton("Exit");
        JButton c = a;
        System.out.println("Is reference a == b? " + (a == b));
        System.out.println("Is reference a == c? " + (a == c));
    }
}
```

Mã này tạo ra ba biến tham chiếu. Hai đối tượng đầu tiên, a và b, là các đối tượng JButton riêng biệt có cùng nhãn. Biến tham chiếu thứ ba, c, được khởi tạo để tham chiếu đến cùng một đối tượng mà a tham chiếu đến. Khi chương trình này chạy, kết quả sau được tạo ra:

```
Is reference a == b? false
Is reference a == c? true
```

Điều này cho chúng ta thấy rằng a và c tham chiếu đến cùng một phiên bản của JButton. Dấu ==

toán tử sẽ không kiểm tra xem hai đối tượng có "tương đương về mặt ý nghĩa" hay không, một khái niệm mà chúng ta sẽ đề cập chi tiết hơn trong [Chương 6](#), khi chúng ta xem xét phương thức equals () (trái ngược với toán tử equals mà chúng ta đang xem xét ở đây) .

Bình đẳng cho Chuỗi và java.lang.Object.equals ()

Chúng tôi chỉ sử dụng dấu == để xác định xem hai biến tham chiếu có tham chiếu đến cùng một đối tượng hay không. Bởi vì các đối tượng là trung tâm của Java, mọi lớp trong Java đều thừa hưởng một phương thức từ Đôi tượng lớp để kiểm tra xem hai đối tượng của lớp có "bằng nhau" hay không. Không có gì ngạc nhiên khi phương thức này được gọi là equals (). Trong trường hợp này của phương thức equals () , nên sử dụng cụm từ "tương đương một cách có ý nghĩa" thay cho từ "bằng nhau". Vì vậy, phương thức equals () được sử dụng để xác định xem hai đối tượng của cùng một lớp có "tương đương về mặt ý nghĩa" hay không. Đối với các lớp mà bạn tạo, bạn có tùy chọn ghi đè phương thức equals () mà lớp của bạn kế thừa từ Đôi tượng lớp và tạo định nghĩa của riêng bạn về "tương đương có ý nghĩa" cho các thể hiện của lớp của bạn.

Về cách hiểu phương thức equals () cho kỳ thi OCA, bạn cần
để hiểu hai khía cạnh của phương thức equals () :

- Equals () có nghĩa là gì trong đối tượng lớp
- Ý nghĩa của bằng () trong chuỗi lớp

Phương thức equals () trong Class Object Phương thức equals () trong Class Object hoạt động giống như cách mà toán tử == hoạt động. Nếu hai tham chiếu trả đến cùng một đối tượng, phương thức equals () sẽ trả về true. Nếu hai tham chiếu trả đến các đối tượng khác nhau, ngay cả khi chúng có cùng giá trị, phương thức sẽ trả về false.

Phương thức equals () trong chuỗi lớp Phương thức equals () trong chuỗi lớp đã được ghi đè. Khi phương thức equals () được sử dụng để so sánh hai chuỗi, nó sẽ trả về true nếu các chuỗi có cùng giá trị và nó sẽ trả về false nếu các chuỗi có giá trị khác nhau. Đối với phương thức equals () của String , các giá trị CÓ phân biệt chữ hoa chữ thường.

Chúng ta hãy xem cách phương thức equals () hoạt động trong thực tế (lưu ý rằng Lớp Budgie KHÔNG ghi đè Object.equals ()):

```

class Budgie {
    public static void main(String[] args) {
        Budgie b1 = new Budgie();
        Budgie b2 = new Budgie();
        Budgie b3 = b1;

        String s1 = "Bob";
        String s2 = "Bob";
        String s3 = "bob";                                // lower case "b"

        System.out.println(b1.equals(b2));    // false, different objects
        System.out.println(b1.equals(b3));    // true, same objects
        System.out.println(s1.equals(s2));    // true, same values
        System.out.println(s1.equals(s3));    // false, values are case sensitive
    }
}

```

tạo ra đầu ra:

```

sai
đúng
đúng
sai

```

Bình đẳng cho enums

Khi bạn đã khai báo một enum, nó không thể mở rộng được. Trong thời gian chạy, không có cách nào để tạo các hằng số enum bổ sung. Tất nhiên, bạn có thể có bao nhiêu biến mà bạn muốn tham chiếu đến một hằng số enum nhất định, vì vậy, điều quan trọng là có thể so sánh hai biến tham chiếu enum để xem liệu chúng có "bằng nhau" hay không – nghĩa là, chúng có tham chiếu đến cùng một hằng số enum ? Bạn có thể sử dụng toán tử == hoặc phương thức equals () để xác định xem hai biến có tham chiếu đến cùng một enum hay không không thay đổi:

```

class EnumEqual {
    enum Color {RED, BLUE}                      // ; is optional
    public static void main(String[] args) {
        Color c1 = Color.RED; Color c2 = Color.RED;
        if(c1 == c2) { System.out.println("=="); }
        if(c1.equals(c2)) { System.out.println("dot equals"); }
    }
}

```

(Chúng tôi biết }} là xâu; chúng tôi đang chuẩn bị cho bạn.) Điều này tạo ra kết quả:

```
==  
dot equals
```

so sánh instanceof

Toán tử instanceof chỉ được sử dụng cho các biến tham chiếu đối tượng và bạn có thể sử dụng nó để kiểm tra xem một đối tượng có thuộc một kiểu cụ thể hay không. Theo "type", chúng tôi có nghĩa là lớp hoặc kiểu giao diện – nói cách khác, liệu đối tượng được tham chiếu bởi biến ở phía bên trái của toán tử có vượt qua bài kiểm tra IS-A cho lớp hoặc kiểu giao diện ở phía bên phải hay không. ([Chương 2](#) trình bày chi tiết về các mối quan hệ IS-A.)

Ví dụ đơn giản sau,

```
public static void main(String[] args) {  
    String s = new String("foo");  
    if (s instanceof String) {  
        System.out.print("s is a String");  
    }  
}
```

in cái này:

s là một chuỗi

Ngay cả khi đối tượng đang được kiểm tra không phải là một khởi tạo thực tế của kiểu lớp ở phía bên phải của toán tử, instanceof vẫn sẽ trả về true nếu đối tượng được so sánh là phép gán tương thích với kiểu ở bên phải.

Ví dụ sau minh họa cách sử dụng phổ biến cho instanceof: testing

một đối tượng để xem liệu nó có phải là một phiên bản của một trong các kiểu con của nó hay không trước khi có gắng downcast:

```

class A { }
class B extends A {
    public static void main (String [] args) {
        A myA = new B();
        m2(myA);
    }
    public static void m2(A a) {
        if (a instanceof B)
            ((B)a).doBstuff();      // downcasting an A reference
                                    // to a B reference
    }
    public static void doBstuff() {
        System.out.println("'a' refers to a B");
    }
}

```

Mã biên dịch và tạo ra đầu ra này:

```
'a' đề cập đến một B
```

Trong các ví dụ như thế này, việc sử dụng toán tử instanceof bảo vệ chương trình từ việc cố gắng hạ cấp bất hợp pháp.

Bạn có thể kiểm tra một tham chiếu đối tượng dựa trên kiểu lớp của chính nó hoặc bất kỳ lớp cha nào của nó. Điều này có nghĩa là bất kỳ tham chiếu đối tượng nào cũng sẽ đánh giá là true nếu bạn sử dụng toán tử instanceof đối với kiểu Object, như sau:

```

B b = new B();
if (b instanceof Object) {
    System.out.print("b is definitely an Object");
}

```

Bản in này

```
b chắc chắn là một Đối tượng
```



Tìm câu hỏi instance của các câu hỏi kiểm tra xem một đối tượng có phải là một thể hiện của giao diện hay không khi lớp của đối tượng triển khai giao diện một cách gián tiếp. Việc triển khai gián tiếp xảy ra khi một trong các lớp cha của đối tượng triển khai một giao diện, nhưng lớp thực tế của cá thể thì không. Trong ví dụ này,

```
interface Foo { }
class A implements Foo { }
class B extends A { }
...
A a = new A();
B b = new B();
```

những điều sau là đúng:

```
a instanceof Foo
b instanceof A
b instanceof Foo // implemented indirectly
```

Một đối tượng được cho là thuộc một kiểu giao diện cụ thể (có nghĩa là nó sẽ vượt qua kiểm tra instanceof) nếu bất kỳ lớp cha nào của đối tượng triển khai giao diện.

Ngoài ra, việc kiểm tra xem tham chiếu rỗng có phải là một trường hợp của một lớp. Tất nhiên, điều này sẽ luôn dẫn đến sai . Ví dụ này,

```
class InstanceTest {
    public static void main(String [] args) {
        String a = null;
        boolean b = null instanceof String;
        boolean c = a instanceof String;
        System.out.println(b + " " + c);
    }
}
```

in cái này:

sai sai

instanceof Compiler Error

Bạn không thể sử dụng toán tử instanceof để kiểm tra trên hai cấu trúc phân cấp lớp khác nhau. Ví dụ, phần sau sẽ KHÔNG biên dịch:

```
class Cat { }
class Dog {
    public static void main(String [] args) {
        Dog d = new Dog();
        System.out.println(d instanceof Cat);
    }
}
```

Biên dịch không thành công – không đời nào d có thể tham chiếu đến Mèo hoặc một kiểu con của Con mèo.



Hãy nhớ rằng mảng là các đối tượng, ngay cả khi mảng là một mảng nguyên thủy. Để ý những câu hỏi trông giống như sau:

```
int [] nums = new int[3];
if (nums instanceof Object) { } // result is true
```

Một mảng luôn là một thể hiện của Đối tượng. Bất kỳ mảng nào.

[Bảng 4-1](#) tóm tắt việc sử dụng toán tử instanceof như sau:

BẢNG 4-1 Tóm tắt bảng hạng và kết quả sử dụng toán tử instanceof

First Operand (Reference Being Tested)	instanceof Operand (Type We're Comparing the Reference Against)	Result
null	Any class or interface type	false
Foo instance	Foo, Bar, Face, Object	true
Bar instance	Bar, Face, Object	true
Bar instance	Foo	false
Foo []	Foo, Bar, Face	compiler error
Foo []	Object	true
Foo [1]	Foo, Bar, Face, Object	true

```
interface Face { }
class Bar implements Face{ }
class Foo extends Bar { }
```

Toán tử số học

Chúng tôi chắc chắn bạn đã quen thuộc với các toán tử số học cơ bản:

- + bǎ sung
- - phép trừ
- * nhân
- / chia

Chúng có thể được sử dụng theo cách tiêu chuẩn:

```
int x = 5 * 3;
int y = x - 4;
System.out.println("x - 4 is " + y); // Prints 11
```

Toán tử Phân còn lại (%) (hay còn gọi là Toán tử Modulus)

Một toán tử mà bạn có thể không quen thuộc là toán tử còn lại: %. Các

toán tử phần dư chia toán hạng bên trái cho toán hạng bên phải và kết quả là phần còn lại, như đoạn mã sau minh họa:

```
class MathTest {
    public static void main (String [] args) {
        int x = 15;
        int y = x % 4;
        System.out.println("The result of 15 % 4 is the "
            + "remainder of 15 divided by 4. The remainder is " + y);
    }
}
```

Lớp MathTest đang chạy sẽ in ra nội dung sau:

Kết quả của $15 \% 4$ là số dư của 15 chia cho 4. Số dư là 3

(Hãy nhớ: Các biểu thức được đánh giá từ trái sang phải theo mặc định. Bạn có thể thay đổi trình tự hoặc mức độ ưu tiên này bằng cách thêm dấu ngoặc đơn. Ngoài ra, hãy nhớ rằng các toán tử *, / và % có mức độ ưu tiên cao hơn các toán tử + và -.)



Khi làm việc với ints, toán tử phần dư (hay còn gọi là toán tử mô đun) và toán tử chia liên quan với nhau theo một cách thú vị:

- Nhà điều hành mô-đun ném ra tất cả mọi thứ trừ phần còn lại.
- Toán tử chia ném ra phần còn lại.

Toán tử nối chuỗi

Dấu cộng cũng có thể được sử dụng để nối hai chuỗi với nhau, như chúng ta đã thấy trước đó (và chắc chắn chúng ta sẽ thấy lại):

```
String animal = "Gray " + "elephant";
```

Việc nối chuỗi trở nên thú vị khi bạn kết hợp các số với Đây. Kiểm tra những điều sau:

```
String a = "String";
int b = 3;
int c = 7;
System.out.println(a + b + c);
```

Toán tử + có hoạt động như một dấu cộng khi thêm các biến int b và c không? Hay toán tử + sẽ coi 3 và 7 là các ký tự và nối chúng riêng lẻ? Kết quả sẽ là String10 hay String37? Được rồi, bạn đã có đủ lâu để nghĩ về nó.

Các giá trị int đơn giản được coi là các ký tự và được dán vào bên phải bên của Chuỗi, cho kết quả:

Chuỗi37

Vì vậy, chúng ta có thể đọc đoạn mã

trước đó là "Bắt đầu với chuỗi giá trị và nối ký tự 3 (giá trị của b) với nó, để tạo ra một chuỗi mới String3, rồi nối ký tự 7 (giá trị của c) với ký tự đó, để tạo ra một chuỗi mới String37. Sau đó hãy in nó ra".

Tuy nhiên, nếu bạn đặt dấu ngoặc đơn xung quanh hai biến int , như sau,

```
System.out.println (a + (b + c));
```

bạn sẽ nhận được điều này:

Chuỗi10

Việc sử dụng dấu ngoặc đơn khiến (b + c) được đánh giá đầu tiên, do đó, toán tử + ngoài cùng bên phải hoạt động như một toán tử cộng, với điều kiện cả hai toán hạng đều là giá trị int . Điểm mấu chốt ở đây là trong dấu ngoặc đơn, toán hạng bên trái không phải là một Chuỗi. Nếu đúng như vậy, thì toán tử + sẽ thực hiện nối chuỗi . Mã trước đó có thể được đọc là

"Cộng các giá trị của b và c với nhau, sau đó lấy tổng và chuyển nó thành Chuỗi và nối nó với Chuỗi từ biến a."

Quy tắc cần nhớ là:

Nếu một trong hai toán hạng là một Chuỗi, thì toán tử + sẽ trở thành một toán tử nối chuỗi . Nếu cả hai toán hạng đều là số thì toán tử + là toán tử cộng.

Bạn sẽ thấy rằng đôi khi bạn có thể gặp khó khăn khi quyết định, chẳng hạn, toán tử bên trái có phải là Chuỗi hay không. Trong kỳ thi, đừng mong đợi nó luôn luôn hiển nhiên. (Thực ra, bây giờ chúng tôi nghĩ về nó, không mong đợi nó sẽ trở nên rõ ràng.) Hãy xem đoạn mã sau:

```
System.out.println (x.foo () + 7);
```

Bạn không thể biết toán tử + đang được sử dụng như thế nào cho đến khi bạn tìm ra phương thức foo () trả về! Nếu foo () trả về một Chuỗi, thì 7 được nối với Chuỗi được trả về. Nhưng nếu foo () trả về một số, thì toán tử + được sử dụng để thêm 7 vào giá trị trả về của foo ().

Cuối cùng, bạn cần biết rằng việc gộp toán tử phụ gia phức hợp (+ =) và Chuỗi lại với nhau là hợp pháp, như sau:

```
String s = "123";
s += "45";
s += 67;
System.out.println(s);
```

Vì cả hai lần toán tử + = được sử dụng và toán hạng bên trái là một Chuỗi, cả hai hoạt động đều là nối, dẫn đến

1234567



Nếu bạn không hiểu cách nối chuỗi hoạt động, đặc biệt là trong một câu lệnh in, bạn thực sự có thể trượt kỳ thi ngay cả khi bạn biết phần còn lại của câu trả lời cho các câu hỏi! Vì rất nhiều câu hỏi đặt ra "Kết quả là gì?", Bạn không chỉ cần biết kết quả của đoạn mã đang chạy mà còn biết kết quả đó được in như thế nào. Mặc dù ít nhất một vài câu hỏi sẽ trực tiếp kiểm tra kiến thức về Chuỗi của bạn, nhưng nối chuỗi hiển thị trong các câu hỏi khác trên hầu hết mọi mục tiêu. Cuộc thí nghiệm! Ví dụ: bạn có thể thấy một dòng như sau:

```
int b = 2;
System.out.println("" + b + 3);
```

Nó in cái này:

23

Nhưng nếu câu lệnh in thay đổi thành:

```
System.out.println (b + 3);
```

Kết quả được in trở thành

5

Các toán tử gia tăng và giảm dần

Java có hai toán tử sẽ tăng hoặc giảm một biến chính xác một. Các toán tử này là hai dấu cộng (++) hoặc hai dấu trừ (-):

- ++ Tăng dần (tiền tố và hậu tố)
- - Giảm dần (tiền tố và hậu tố)

Toán tử được đặt trước (tiền tố) hoặc sau (hậu tố) một biến để thay đổi giá trị của nó. Cho dù toán tử đến trước hoặc sau toán hạng có thể thay đổi kết quả của một biểu thức. Kiểm tra những điều sau:

```

1. class MathTest {
2.     static int players = 0;
3.     public static void main (String [] args) {
4.         System.out.println("players online: " + players++);
5.         System.out.println("The value of players is "
6.                             + players);
7.         System.out.println("The value of players is now "
8.                             + ++players);
    }
}
```

Lưu ý rằng trong dòng thứ tư của chương trình, toán tử tăng sau các trình phát biến. Điều đó có nghĩa là chúng tôi đang sử dụng toán tử gia tăng hậu tố, khiến người chơi được tăng lên một nhưng chỉ sau khi giá trị của người chơi được sử dụng trong biểu thức. Khi chúng tôi chạy chương trình này, nó xuất ra như sau:

```
%java MathTest
players online: 0
The value of players is 1
The value of players is now 2
```

Lưu ý rằng khi biến được ghi ra màn hình, lúc đầu nó cho biết giá trị là 0. Bởi vì chúng tôi đã sử dụng toán tử gia số postfix, gia số không xảy ra cho đến khi biến trình phát được sử dụng trong câu lệnh in . Hiểu rồi? Bài"

trong postfix có nghĩa là sau. Dòng 5 không tăng người chơi; nó chỉ xuất giá trị của nó ra màn hình, vì vậy giá trị tăng mới được hiển thị là 1. Dòng 6 áp dụng toán tử tăng tiền tố cho các trình phát , có nghĩa là giá trị tăng xảy ra trước khi giá trị của biến được sử dụng, do đó đầu ra là 2.

Mong đợi để xem các câu hỏi trộn các toán tử tăng và giảm với các toán tử khác, như trong ví dụ sau:

```
int x = 2; int y = 3;
if ((y == x++) | (x < ++y)) {
    System.out.println("x = " + x + " y = " + y);
}
```

Mã trước in ra sau:

x = 3 y = 4

Bạn có thể đọc mã như sau: "Nếu 3 bằng 2 HOẶC 3 <4"

Biểu thức đầu tiên so sánh x và y, và kết quả là sai, vì sự gia tăng trên x không xảy ra cho đến sau khi kiểm tra == được thực hiện. Tiếp theo, chúng ta tăng x, vì vậy bây giờ x là 3. Sau đó, chúng ta kiểm tra xem x có nhỏ hơn y hay không, nhưng chúng ta tăng y trước khi so sánh nó với x! Vậy phép thử logic thứ hai là (3 <4). Kết quả là true, vì vậy câu lệnh in sẽ chạy.

Như với nối chuỗi , các toán tử tăng và giảm là được sử dụng trong suốt kỳ thi, ngay cả đối với các câu hỏi không nhằm kiểm tra kiến thức của bạn về cách hoạt động của các toán tử đó. Bạn có thể thấy chúng trong các câu hỏi về vòng lặp, ngoại lệ hoặc thậm chí chuỗi. Hãy sẵn sàng.



Để ý các câu hỏi sử dụng toán tử tăng hoặc giảm trên biến cuối cùng . Vì không thể thay đổi các biến cuối cùng, các toán tử tăng và giảm không thể được sử dụng với chúng và bất kỳ nỗ lực nào để làm như vậy sẽ dẫn đến lỗi trình biên dịch. Mã sau sẽ không biên dịch:

```
final int x = 5;
int y = x++;
```

Nó tạo ra lỗi này:

```
Test.java:4: cannot assign a value to final variable x
int y = x++;
^
```

Bạn có thể mong đợi một vi phạm như thế này được chôn sâu trong một đoạn mã phức tạp. Nếu bạn phát hiện ra nó, bạn biết mã sẽ không biên dịch và bạn có thể tiếp tục mà không cần làm việc với phần còn lại của mã.

Câu hỏi này dường như đang kiểm tra bạn về một số câu đố toán tử số học phức tạp, trong khi trên thực tế, nó đang kiểm tra bạn về kiến thức của bạn về công cụ sửa đổi cuối cùng .

Điều hành có điều kiện

Toán tử điều kiện là một toán tử bậc ba (nó có ba toán hạng) và được sử dụng để đánh giá các biểu thức boolean – giống như một câu lệnh if , ngoại trừ việc thay vì thực thi một khối mã nếu kiểm tra là đúng, một toán tử điều kiện sẽ gán một giá trị cho một Biến đổi. Nói cách khác, mục tiêu của toán tử điều kiện là quyết định gán giá trị nào trong hai giá trị cho một biến. Toán tử này được xây dựng bằng cách sử dụng ? (dấu chấm hỏi) và dấu : (dấu hai chấm). Dấu ngoặc đơn là tùy chọn. Đây là cấu trúc của nó:

x = (biểu thức boolean)? giá trị để gán nếu đúng: giá trị để gán nếu sai

Hãy xem một toán tử có điều kiện trong mã:

```
class Salary {
    public static void main(String [] args) {
        int numOfPets = 3;
        String status = (numOfPets<4) ? "Pet limit not exceeded"
            : "too many pets";
        System.out.println("This pet status is " + status);
    }
}
```

Bạn có thể đọc mã trước là "Đặt numOfPets bằng 3".

Tiếp theo, chúng ta sẽ gán một Chuỗi cho biến trạng thái. Nếu numOfPets là nhỏ hơn 4, gán "Không vượt quá giới hạn vật nuôi" cho biến trạng thái ; nếu không, hãy chỉ định "quá nhiều vật nuôi" cho biến trạng thái .

Một toán tử có điều kiện bắt đầu bằng một phép toán boolean , theo sau là hai

các giá trị có thể cho biến ở bên trái của toán tử gán (=) . Giá trị đầu tiên (giá trị ở bên trái dấu hai chấm) được chỉ định nếu thử nghiệm có điều kiện (boolean) là đúng và giá trị thứ hai được chỉ định nếu thử nghiệm có điều kiện là sai. Bạn thậm chí có thể lồng các toán tử có điều kiện vào một câu lệnh:

```
class AssignmentOps {
    public static void main(String [] args) {
        int sizeOfYard = 10;
        int numOfPets = 3;
        String status = (numOfPets<4)?"Pet count OK"
            :(sizeOfYard > 8)? "Pet limit on the edge"
            :"too many pets";
        System.out.println("Pet status is " + status);
    }
}
```

Đừng mong đợi nhiều câu hỏi sử dụng toán tử điều kiện, nhưng bạn có thể nhận được một.

Toán tử logic

Mục tiêu kỳ thi chỉ định sáu toán tử “logic” (&, |, ^, !, &&, và ||). Một số tài liệu Oracle sử dụng thuật ngữ khác cho các toán tử này, nhưng đối với mục đích của chúng tôi và mục tiêu kỳ thi, sáu toán tử này là các toán tử logic.

Toán tử Bitwise (Không phải là một chủ đề kiểm tra!)

Được rồi, điều này sẽ gây nhầm lẫn. Trong số sáu toán tử logic vừa được liệt kê, ba trong số chúng (&, |, và ^) cũng có thể được sử dụng làm toán tử “bitwise”. Các toán tử bitwise đã được bao gồm trong các phiên bản trước của kỳ thi, nhưng chúng KHÔNG có trong kỳ thi Java 6, Java 7 hoặc Java 8. Chúng tôi đưa chúng lên đây chỉ để bạn có một bức tranh đầy đủ hơn về các toán tử logic.

Dưới đây là một số tuyên bố pháp lý sử dụng toán tử bitwise:

```
byte b1 = 6 & 8;
byte b2 = 7 | 9;
byte b3 = 5 ^ 4;
System.out.println(b1 + " " + b2 + " " + b3);
```

Toán tử bitwise so sánh hai biến từng chút một và trả về một biến

các bit của chúng đã được đặt dựa trên việc hai biến đang được so sánh có các bit tương ứng vừa là "trên" (&), vừa là "trên" (|) hay chính xác là "trên" (^) hay không. Nhìn tiện, khi chúng tôi chạy mã trước đó, chúng tôi nhận được

0 15 1



Đã nói tất cả những điều này về toán tử bitwise, điều quan trọng cần nhớ
BỘ ĐIỀU HÀNH BITWISE KHÔNG CÓ TRONG BÀI THI Java 6, Java 7 hoặc Java 8!

Toán tử logic ngắn mạch

Năm toán tử logic trong bài kiểm tra được sử dụng để đánh giá các câu lệnh chứa nhiều hơn một biểu thức boolean . Thường được sử dụng nhất trong năm là hai toán tử logic ngắn mạch :

- && Ngắn mạch AND
- || Ngắn mạch HOẶC

Chúng được sử dụng để liên kết các biểu thức boolean nhỏ với nhau để tạo thành các biểu thức boolean lớn hơn . && và || _ toán tử chỉ đánh giá các giá trị boolean . Để biểu thức AND (&&) là true, cả hai toán hạng phải đúng. Ví dụ:

```
if ((2 <3) && (3 <4)) {}
```

Biểu thức trước được đánh giá là true vì cả toán hạng một (2 <3) và toán hạng hai (3 <4) đều đánh giá là true.

Tính năng ngắn mạch của toán tử && được đặt tên như vậy bởi vì nó không lãng phí thời gian vào những đánh giá vô nghĩa. Một ngắn mạch && đánh giá phía bên trái của phép toán trước (toán hạng một) và nếu nó chuyển thành false, toán tử && không bận tâm nhìn vào phía bên phải của biểu thức (toán hạng hai) vì toán tử && đã biết rằng biểu thức hoàn chỉnh không thể đúng.

```

class Logical {
    public static void main(String [] args) {
        boolean b1 = false, b2 = false;
        boolean b3 = (b1 == true) && (b2 = true); // will b2 be set to true?
        System.out.println(b3 + " " + b2);
    }
}

```

Khi chúng tôi chạy mã trước, phép gán (`b2 = true`) không bao giờ chạy do toán tử ngắn mạch, vì vậy đầu ra là

```
%java Logical
false false
```

Cái `||` toán tử tương tự như toán tử `&&`, ngoại trừ việc nó đánh giá là `true` nếu EITHER của các toán hạng là `true`. Nếu toán hạng đầu tiên trong phép toán OR là `true`, kết quả sẽ là `true`, do đó, ngắn mạch `||` không lãng phí thời gian nhìn vào về phái của phương trình. Tuy nhiên, nếu toán hạng đầu tiên là sai, thì `||` ngắn mạch `||` phải đánh giá toán hạng thứ hai để xem kết quả của phép toán OR là đúng hay sai. Hãy chú ý đến ví dụ sau đây; bạn sẽ thấy khá nhiều câu hỏi như thế này trong bài kiểm tra:

```

1. class TestOR {
2.     public static void main(String[] args) {
3.         if ((isItSmall(3)) || (isItSmall(7))) {
4.             System.out.println("Result is true");
5.         }
6.         if ((isItSmall(6)) || (isItSmall(9))) {
7.             System.out.println("Result is true");
8.         }
9.     }
10.
11.    public static boolean isItSmall(int i) {
12.        if (i < 5) {
13.            System.out.println("i < 5");
14.            return true;
15.        } else {
16.            System.out.println("i >= 5");
17.            return false;
18.        }
19.    }
20. }
```

Kết quả là gì?

```
% java TestOR  
i < 5  
Result is true  
i >= 5  
i >= 5
```

Đây là những gì đã xảy ra khi phương thức main () chạy:

1. Khi chúng ta nhấn dòng 3, toán hạng đầu tiên trong || biểu thức (nói cách khác, phía bên trái của phép toán ||) được đánh giá.
2. Phương thức isItSmall (3) được gọi, in ra "i <5" và trả về true.
3. Vì toán hạng đầu tiên trong || biểu thức trên dòng 3 là true , || toán tử không bận tâm đánh giá toán hạng thứ hai. Vì vậy, chúng tôi không bao giờ thấy "i> = 5" mà lẽ ra đã được in ra nếu toán hạng thứ hai được đánh giá (điều này sẽ gọi isItSmall (7)).
4. Dòng 6 được đánh giá, bắt đầu bằng toán hạng đầu tiên trong || biểu hiện.
5. Phương thức isItSmall (6) được gọi, in ra "i> = 5" và trả về false.
6. Vì toán hạng đầu tiên trong || biểu thức trên dòng 6 là sai, dấu || toán tử không thể bỏ qua toán hạng thứ hai ; vẫn có khả năng biểu thức có thể đúng, nếu toán hạng thứ hai cho giá trị true.
7. Phương thức isItSmall (9) được gọi và in ra "i> = 5".
8. Phương thức isItSmall (9) trả về false, do đó, biểu thức trên dòng 6 là false, và do đó dòng 7 không bao giờ thực thi.



Cái || và các toán tử && chỉ hoạt động với toán hạng boolean. Bài kiểm tra có thẻ có gắng đánh lừa bạn bằng cách sử dụng các số nguyên với các toán tử sau:

nếu (5 && 6) {}

Có vẻ như chúng tôi đang cố gắng thực hiện một chút AND trên các bit đại diện cho các số nguyên 5 và 6, nhưng mã thậm chí sẽ không được biên dịch.

Toán tử logic (không phải ngắn mạch)

Có hai toán tử logic không ngắn mạch :

- & Không ngắn mạch AND
- | Không ngắn mạch HOẶC

Các toán tử này được sử dụng trong các biểu thức logic giống như `&&` và `||` các toán tử được sử dụng, nhưng vì chúng không phải là toán tử ngắn mạch, chúng đánh giá cả hai mặt của biểu thức – luôn luôn! Chúng không hiệu quả. Ví dụ: ngay cả khi toán hạng đầu tiên (bên trái) trong biểu thức `&` là sai, thì toán hạng thứ hai sẽ vẫn được đánh giá – mặc dù bây giờ không thể cho kết quả là đúng! Và `|` cũng không hiệu quả: nếu toán hạng đầu tiên là `true`, Máy ảo Java (JVM) vẫn tiếp tục chạy trước và đánh giá toán hạng thứ hai ngay cả khi nó biết biểu thức sẽ đúng .

Bạn sẽ tìm thấy rất nhiều câu hỏi trong bài kiểm tra sử dụng cả toán tử logic ngắn mạch và không ngắn mạch. Bạn sẽ phải biết chính xác toán hạng nào được đánh giá và toán hạng nào không, bởi vì kết quả sẽ khác nhau tùy thuộc vào việc toán hạng thứ hai trong biểu thức có được đánh giá hay không. Xem xét điều này,

```
int z = 5;
if (++z > 5 || ++z > 6) z++; // z = 7 after this code
```

so với điều này:

```
int z = 5;
if (++z > 5 | ++z > 6) z++; // z = 8 after this code
```

Toán tử logic ^ và !

Hai toán tử logic cuối cùng trong bài kiểm tra là

- ^ Exclusive-OR (XOR)
- ! Boolean invert

Toán tử `^` (unique-OR) chỉ đánh giá các giá trị boolean . Toán tử `^` có liên quan đến các toán tử không ngắn mạch mà chúng ta vừa xem xét, trong đó nó luôn đánh giá cả toán hạng bên trái và bên phải trong một biểu thức. Để một biểu thức OR (`^`) độc quyền là `true`, CHÍNH XÁC một toán hạng phải là `true`. Ví dụ này,

```
System.out.println ("xor" + ((2 <3) ^ (4> 3)));
```

tạo ra đầu ra này:

xor sai

Biểu thức trước được đánh giá là false vì cả hai toán hạng một ($2 < 3$) và toán hạng hai ($4 > 3$) đánh giá là true.

Cái ! (boolean invert) trả về toán tử ngược lại với hiện tại của boolean giá trị. Tuyên bố sau đây,

```
if (! (7 == 5)) {System.out.println ("không bằng"); }
```

có thể được đọc là “Nếu không phải là $7 == 5$ ” và câu lệnh tạo ra kết quả này:

không công bằng

Đây là một ví dụ khác sử dụng booleans:

```
boolean t = true;
boolean f = false;
System.out.println("! " + (t & !f) + " " + f);
```

Nó tạo ra đầu ra này:

! đúng sai

Trong ví dụ trước, lưu ý rằng kiểm tra & đã thành công (in true) và rằng giá trị của biến boolean f không thay đổi, vì vậy nó được in sai.

Ưu tiên điều hành

Kỳ thi OCA 8 đã giới thiệu lại chủ đề về quyền ưu tiên của toán tử. Như bạn có thể đã biết nhưng chắc chắn sẽ thấy được trình bày trong phần này, khi một số toán tử được sử dụng kết hợp, thứ tự mà chúng được đánh giá có thể thay đổi kết quả của biểu thức.

Nhà điều hành ưu tiên Rant Cho phép

chúng tôi nói chuyện trong một phút ở đây. Ưu tiên toán tử ghi nhớ đã có trong kỳ thi SCJP 1.2 cũ cách đây khoảng 15 năm. Bắt đầu với kỳ thi SCJP 1.4 và đối với tất cả các kỳ thi cho đến kỳ thi OCA 8, quyền ưu tiên của nhà điều hành không được áp dụng trong kỳ thi. Trong suốt 15 năm vinh quang, các ứng viên không cần phải ghi nhớ chút nào.

Đáng buồn thay, chủ đề này lại được đưa vào đề thi OCA 8. Tại sao chúng ta lại quan tâm nhiều đến vấn đề này? Hãy xem mã này:

```
System.out.println (true & false == false | true);
```

Bạn mong đợi kết quả nào? Hãy tưởng tượng một phiên bản thực tế hơn, đánh giá một số boolean:

```
System.out.println (b1 & b2 == b3 | b4);
```

Bạn đoán ý định của lập trình viên ở đây là gì? Có hai trường hợp có thể xảy ra:

Tình huống 1: $(b1 \& b2) == (b3 | b4)$ Nếu đây là ý định của lập trình viên, thì anh ta vừa tạo ra một lỗi.

Tình huống 2: $b1 \& (b2 == b3) | b4$ Nếu đây là ý định của lập trình viên, thì mã sẽ hoạt động như dự định, nhưng ông chủ và đồng nghiệp của anh ta sẽ muốn bóp cổ anh ta.

Đây là một cách dài dòng để nói rằng khi bạn đang viết mã, bạn không nên dựa vào trí nhớ của mọi người về mức độ ưu tiên của toán tử. Bạn chỉ nên sử dụng dấu ngoặc đơn như những người văn minh vẫn làm.

Sự bắt đầu thực tế của Phần ưu tiên của nhà điều hành

[Bảng 4-2](#) liệt kê các toán tử được sử dụng phổ biến nhất và mức độ ưu tiên tương đối của chúng, bắt đầu ở trên cùng với các toán tử có mức ưu tiên cao nhất và kết thúc ở dưới cùng với mức thấp nhất. (Lưu ý, không phải tất cả các toán tử của Java đều có trong bảng này!)

BẢNG 4-2 Mức ưu tiên của các toán tử phổ biến (từ Cao nhất đến Thấp nhất)

Types of Operators	Symbols	Example Uses
Unary operators	-, !, ++, --	<u>-7</u> * 4, !myBoolean
Multiplication, division, modulus	*, /, %	7 % 4
Addition, subtraction	+, -	7 + 4
Relational operators	<, >, <=, >=	Y > X
Equality operators	==, !=	Y != X
Logical operators (& beats)	&,	myBool & yourBool
Short-circuit (&& beats)	&&,	myBool yourBool
Assignment operators	=, +=, -=	X += 5;

Fritz Walraven, người điều hành JavaRanch (chúng tôi biết, chúng tôi biết, "Coderanch") đã chia sẻ mẹo này với chúng tôi. Chúng tôi thích nó và chúng tôi sẽ chuyển nó cho bạn: đối với bảng trên, bạn có thể tạo ra một từ như "UMARELSA" hoặc một câu sử dụng các chữ cái đầu tiên đó, để giúp bạn nhớ các quy tắc ưu tiên!

Có ba quy tắc chung quan trọng để xác định cách Java sẽ đánh giá các biểu thức với các toán tử:

- Khi hai toán tử có cùng mức độ ưu tiên trong cùng một biểu thức, Java sẽ đánh giá biểu thức từ trái sang phải.
- Khi các phần của một biểu thức được đặt trong dấu ngoặc đơn, các phần đó sẽ được đánh giá trước tiên.
- Khi các dấu ngoặc đơn được lồng vào nhau, các dấu ngoặc đơn trong cùng được đánh giá đầu tiên.

Một cách tốt để ghi những quy tắc ưu tiên này vào bộ não của bạn là – như mọi khi – viết một số mã thử nghiệm và chơi với nó. Chúng tôi đã thêm một ví dụ về một số mã thử nghiệm thể hiện một số quy tắc phân cấp mức độ ưu tiên được liệt kê ở đây.

Như bạn có thể thấy, chúng tôi thường so sánh các biểu thức không có dấu ngoặc đơn với các biểu thức có nhiều dấu ngoặc đơn của chúng để chứng minh các quy tắc:

```

System.out.println((-7 - 4) + " " + ((-7 - 4)));           // unary (-7), beats minus
                                                               // output: -11 -3

System.out.println((2 + 3 * 4) + " " + ((2 + 3) * 4));   // * beats +
                                                               // output: 14 20

System.out.println(7 > 5 && 2 > 3);                      // > beats &&
                                                               // output: false

System.out.print((true & false == false | true) + " "); // == beats & System.out.
print(((true & false) == (false | true)));      // output: true

```

Và để lặp lại, đâu ra là:

```

-11 -3
14 20
false
true false

```

Chúng tôi rất tiếc vì bạn cần phải ghi nhớ nội dung này, nhưng nếu bạn nắm vững những gì trong phần ngắn này, bạn sẽ có thể xử lý bất kỳ câu hỏi nào liên quan đến mức độ ưu tiên kỳ lạ mà kỳ thi ném vào bạn.

TÓM TẮT CHÚNG NHẬN

Nếu bạn đã nghiên cứu chương này một cách siêng năng, bạn sẽ nắm chắc về các toán tử Java và bạn sẽ hiểu ý nghĩa của sự bình đẳng khi kiểm tra với toán tử == . Hãy xem lại những điểm nổi bật của những gì bạn đã học được trong chương này.

Các toán tử logic (&&, ||, &, |, và ^) chỉ có thể được sử dụng để đánh giá hai biểu thức boolean . Sự khác biệt giữa && và & là toán tử && sẽ không bận tâm đến việc kiểm tra toán hạng bên phải nếu bên trái cho kết quả là false, bởi vì kết quả của biểu thức && không bao giờ có thể đúng. Sự khác biệt giữa || và | đó có phải là || toán tử sẽ không bận tâm kiểm tra toán hạng bên phải nếu bên trái đánh giá là true vì kết quả đã được biết là đúng tại thời điểm đó.

Toán tử == có thể được sử dụng để so sánh các giá trị của các giá trị nguyên thủy, nhưng nó cũng có thể được sử dụng để xác định xem hai biến tham chiếu có tham chiếu đến cùng một đối tượng hay không.

Toán tử instanceof được sử dụng để xác định xem đối tượng được tham chiếu đến bởi một biến tham chiếu vượt qua kiểm tra IS-A cho một loại được chỉ định.

Toán tử + được nạp chòng để thực hiện các tác vụ nối chuỗi và cũng có thể nối các chuỗi và các chuỗi nguyên thủy, nhưng hãy cẩn thận – việc nối có thể phức tạp.

Toán tử có điều kiện (hay còn gọi là “toán tử bậc ba”) có một bất thường, ba-

Toán tử có điều kiện (hay còn gọi là “toán tử bậc ba”) có ba cú pháp toán hạng – đừng nhầm nó với một câu lệnh khẳng định phức tạp.

Các toán tử `++` và `-` sẽ được sử dụng trong suốt kỳ thi, và bạn phải chú ý xem chúng được đặt trước hoặc sau tiền tố cho biến đang được cập nhật.

Mặc dù bạn nên sử dụng dấu ngoặc đơn trong cuộc sống thực, đối với kỳ thi, bạn nên ghi nhớ [Bảng 4-2](#) để có thể xác định cách mã không sử dụng dấu ngoặc đơn cho các biểu thức phức tạp sẽ được đánh giá như thế nào, dựa trên hệ thống phân cấp ưu tiên toán tử của Java.

Hãy chuẩn bị cho rất nhiều câu hỏi thi liên quan đến các chủ đề từ chương này. Ngay cả trong các câu hỏi kiểm tra kiến thức của bạn về một mục tiêu khác, mã sẽ thường xuyên sử dụng các toán tử, phép gán, chuyển đổi tượng và nguyên thủy, v.v.

✓ KHOAN HAI PHÚT

Dưới đây là một số điểm chính từ mỗi phần trong chương này.

Người điều hành quan hệ (Mục tiêu 3.1 và 3.2 của OCA)

- Các toán tử quan hệ luôn dẫn đến một giá trị boolean (đúng hoặc sai).
- Có sáu toán tử quan hệ: `>`, `>=`, `<`, `<=`, `==`, và `!=`. Hai cuối cùng (`==` và `!=`) Đôi khi được gọi là toán tử bình đẳng.
- Khi so sánh các ký tự, Java sử dụng giá trị Unicode của ký tự làm giá trị số.
- Toán tử bình đẳng
 - Có hai toán tử bình đẳng: `==` và `!=`.
 - Bốn loại thứ có thể được kiểm tra: số, ký tự, boolean và biến tham chiếu.
- Khi so sánh các biến tham chiếu, `==` chỉ trả về true nếu cả hai tham chiếu đều tham chiếu đến cùng một đối tượng.

`instanceof` Operator (OCA Objective 3.1) `instanceof` chỉ

- dành cho các biến tham chiếu; nó kiểm tra xem đối tượng có thuộc một kiểu cụ thể hay không.

- Toán tử instanceof chỉ có thể được sử dụng để kiểm tra các đối tượng (hoặc null) chống lại các loại lớp trong cùng một hệ thống phân cấp lớp.
- Đối với các giao diện, một đối tượng vượt qua kiểm tra instanceof nếu bất kỳ lớp cha nào của nó triển khai giao diện ở phía bên phải của toán tử instanceof .

Toán tử số học (Mục tiêu 3.1 của OCA)

- Bốn toán tử toán học chính là cộng (+), trừ (-), nhân (*) và chia (/).
- Toán tử phần còn lại (hay còn gọi là modulo) (%) trả về phần còn lại của một phép chia.
- Biểu thức được đánh giá từ trái sang phải, trừ khi bạn thêm dấu ngoặc đơn hoặc trừ khi một số toán tử trong biểu thức có mức độ ưu tiên cao hơn những toán tử khác.
- Các toán tử *, / và % có mức độ ưu tiên cao hơn + và -.

Toán tử kết nối chuỗi (Mục tiêu 3.1 của OCA)

- Nếu một trong hai toán hạng là một Chuỗi, toán tử + sẽ nối các toán hạng.
- Nếu cả hai toán hạng đều là số, toán tử + sẽ thêm các toán hạng.

Các nhà điều hành tăng / giảm (Mục tiêu 3.1 của OCA)

- Các toán tử tiền tố (ví dụ --x) chạy trước khi giá trị được sử dụng trong biểu thức.
- Các toán tử hậu tố (ví dụ: x++) chạy sau khi giá trị được sử dụng trong biểu thức.
- Trong bất kỳ biểu thức nào, cả hai toán hạng đều được đánh giá đầy đủ trước khi toán tử được áp dụng.
- Các biến được đánh dấu cuối cùng không thể tăng hoặc giảm.

Nhà điều hành bậc ba (có điều kiện) (Mục tiêu 3.3 của OCA)

- Trả về một trong hai giá trị dựa trên trạng thái của biểu thức boolean của nó .
 - Trả về giá trị sau dấu ? nếu biểu thức là đúng.
 - Trả về giá trị sau : nếu biểu thức sai.

Toán tử lôgic (Mục tiêu 3.1 của OCA)

Toán tử lôgic (Mục tiêu 3.1 của OCA)

- Bài kiểm tra bao gồm sáu toán tử "logic": &, |, ^, !, &&, và ||.
- Làm việc với hai biểu thức (ngoại trừ !) Phải giải quyết các giá trị boolean.
- Toán tử && và & chỉ trả về true nếu cả hai toán hạng đều đúng.
- Cái || và | toán tử trả về true nếu một trong hai hoặc cả hai toán hạng đều đúng.
- && và || _ toán tử được gọi là toán tử ngắn mạch.
- Toán tử && không đánh giá toán hạng bên phải nếu toán hạng bên trái là sai.
- Cái || không đánh giá toán hạng bên phải nếu toán hạng bên trái là true.
- & Và | _ toán tử luôn đánh giá cả hai toán hạng.
- Toán tử ^ (được gọi là "logic XOR") trả về true nếu đúng một toán hạng .
- Cái ! toán tử (được gọi là toán tử "inversion") trả về giá trị ngược lại của toán hạng boolean mà nó đúng trước.

Dấu ngoặc đơn và Mức độ ưu tiên của toán tử (Mục tiêu 3.1 của OCA)

- Trong cuộc sống thực, hãy sử dụng dấu ngoặc đơn để làm rõ mã của bạn và buộc Java đánh giá các biểu thức như dự định.
- Đối với bài kiểm tra, hãy ghi nhớ [Bảng 4-2](#) để xác định cách đánh giá mã không có dấu ngoặc đơn.

TỰ KIỂM TRA

1. Cho:

```
class Hexy {
    public static void main(String[] args) {
        int i = 42;
        String s = (i<40)? "life" : (i>50)? "universe" : "everything";
        System.out.println(s);
    }
}
```

Kết quả là gì?

A. null

- B. cuộc sống
- C. vũ trụ
- D. mọi thứ
- E. Biên dịch không thành công
- F. Một ngoại lệ được đưa ra trong thời gian chạy

2. Đưa ra:

```
public class Dog {  
    String name;  
    Dog(String s) { name = s; }  
    public static void main(String[] args) {  
        Dog d1 = new Dog("Boi");  
        Dog d2 = new Dog("Tyri");  
        System.out.print((d1 == d2) + " ");  
        Dog d3 = new Dog("Boi");  
        d2 = d1;  
        System.out.print((d1 == d2) + " ");  
        System.out.print((d1 == d3) + " ");  
    }  
}
```

Kết quả là gì?

- A. true true true
- B. đúng đúng sai
- C. sai đúng sai
- D. sai đúng đúng
- E. sai sai sai
- F. Một ngoại lệ sẽ được đưa ra trong thời gian chạy

3. Đưa ra:

```
class Fork {  
    public static void main(String[] args) {  
        if(args.length == 1 | args[1].equals("test")) {  
            System.out.println("test case");  
        } else {  
            System.out.println("production " + args[0]);  
        }  
    }  
}
```

Và lệnh gọi dòng lệnh:

```
java Fork live2
```

Kết quả là gì?

- A. trường hợp thử nghiệm
- B. sản xuất trực tiếp2
- C. trường hợp thử nghiệm trực tiếp2
- D. Biên dịch không thành công
- E. Một ngoại lệ được đưa ra trong thời gian chạy

4. Cho:

```
class Feline {  
    public static void main(String[] args) {  
        long x = 42L;  
        long y = 44L;  
        System.out.print(" " + 7 + 2 + " ");  
        System.out.print(foo() + x + 5 + " ");  
        System.out.println(x + y + foo());  
    }  
    static String foo() { return "foo"; }  
}
```

Kết quả là gì?

- A. 9 foo47 86foo
- B. 9 foo47 4244foo
- C. 9 foo425 86foo
- D. 9 foo425 4244foo
- E. 72 foo47 86foo

F. 72 foo47 4244foo

G. 72 foo425 86foo

H. 72 foo425 4244foo

I. Biên dịch không thành công

5. Lưu ý: Đây là một câu hỏi kéo và thả kiểu cũ khác. đề phòng.

Đặt các phân mảnh vào mã để tạo ra kết quả 33. Lưu ý rằng bạn phải sử dụng mỗi phân mảnh chính xác một lần.

CODE:

```
class Incr {  
    public static void main(String[] args) {  
        Integer x = 7;  
        int y = 2;  
  
        x      ____ ____;  
        ____  ____  ____;  
        ____  ____  ____;  
        ____  ____  ____;  
  
        System.out.println(x);  
    }  
}
```

KHOẢNG CÁCH:

Y Y Y Y

Y x x

- = *= *= *=

6. Cho:

```
public class Cowboys {  
    public static void main(String[] args) {  
        int x = 12;  
        int a = 5;  
        int b = 7;  
        System.out.println(x/a + " " + x/b);  
    }  
}
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

Kết quả là gì? (Chọn tất cả các áp dụng.)

- A. 2 1
- B. 2 2
- C. 3 1
- D. 3 2
- E. Một ngoại lệ được đưa ra trong thời gian chạy

7. Cho:

```
3. public class McGee {  
4.     public static void main(String[] args) {  
5.         Days d1 = Days.TH;  
6.         Days d2 = Days.M;  
7.         for(Days d: Days.values()) {  
8.             if(d.equals(Days.F)) break;  
9.             d2 = d;  
10.        }  
11.        System.out.println((d1 == d2)? "same old" : "newly new");  
12.    }  
13.    enum Days {M, T, W, TH, F, SA, SU};  
14. }
```

Kết quả là gì?

- A. cũ
- B. mới mới C.
- Biên dịch không thành công do nhiều lỗi D.
- Biên dịch không thành công chỉ do một lỗi ở dòng 7
- E. Biên dịch không thành công chỉ do một lỗi ở dòng 8
- F. Biên dịch không thành công chỉ do một lỗi ở dòng 11
- G. Biên dịch không thành công chỉ do lỗi trên dòng 13

8. Cho:

```
4. public class SpecialOps {  
5.     public static void main(String[] args) {  
6.         String s = "";  
7.         boolean b1 = true;  
8.         boolean b2 = false;  
9.         if((b2 = false) | (21%5) > 2) s += "x";  
10.        if(b1 || (b2 = true))           s += "Y";  
11.        if(b2 == true)                 s += "z";  
12.        System.out.println(s);  
13.    }  
14. }
```

Đó là sự thật? (Chọn tất cả các áp dụng.)

A. Biên dịch không

thành công B. x sẽ được đưa vào đầu

ra C. y sẽ được đưa vào đầu ra D. z

sẽ được đưa vào đầu ra E. Một ngoại

lệ được đưa ra trong thời gian chạy

9. Cho:

```
3. public class Spock {  
4.     public static void main(String[] args) {  
5.         int mask = 0;  
6.         int count = 0;  
7.         if( ((5<7) || (++count < 10)) | mask++ < 10 ) mask = mask + 1;  
8.         if( (6 > 8) ^ false)                         mask = mask + 10;  
9.         if( !(mask > 1) && ++count > 1)             mask = mask + 100;  
10.        System.out.println(mask + " " + count);  
11.    }  
12. }
```

Hai điều nào đúng về giá trị của mặt nạ và giá trị của số đếm ở dòng 10? (Chọn hai.)

A. mặt nạ là 0

B. mặt nạ là 1

C. mặt nạ là 2

D. mặt nạ là 10

E. mặt nạ lớn hơn 10

F. đếm là 0

G. số lượng lớn hơn 0

10. Cho:

```
3. interface Vessel { }
4. interface Toy { }
5. class Boat implements Vessel { }
6. class Speedboat extends Boat implements Toy { }
7. public class Tree {
8.     public static void main(String[] args) {
9.         String s = "0";
10.        Boat b = new Boat();
11.        Boat b2 = new Speedboat();
12.        Speedboat s2 = new Speedboat();
13.        if((b instanceof Vessel) && (b2 instanceof Toy)) s += "1";
14.        if((s2 instanceof Vessel) && (s2 instanceof Toy)) s += "2";
15.        System.out.println(s);
16.    }
17. }
```

Kết quả là gì?

- A. 0
- B. 01
- C. 02
- D. 012

E. Biên dịch không thành

công F. Một ngoại lệ được đưa ra trong thời gian chạy

11. Cho:

```
10. boolean b1 = false;
11. boolean b2;
12. int x = 2, y = 5;
13. b1 = 2-12/4 > 5+-7 && b1 != y++>5 == 7%4 > ++x | b1 == true;
14. b2 = (2-12/4 > 5+-7) && (b1 != y++>5) == (7%4 > ++x) | (b1 == true);
15. System.out.println(b1 + " " + b2);
```

Kết quả là gì? (Đây là một mèo phức tạp. Nếu bạn muốn có một gợi ý, hãy xem xét kỹ thuật số ưu tiên toán tử trong chương.)

- A. đúng sự thật
- B. sai đúng
- C. đúng sai
- D. sai sai
- E. Biên dịch không thành công
- F. Một ngoại lệ được đưa ra trong thời gian chạy

CÂU TRẢ LỜI TỰ KIỂM TRA

1. đúng. Đây là một con chim nhạn lồng trong một con chim nhạn. Cả ba biểu thức là sai.
 A, B, C, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 3.1 và 3.3 của OCA)
2. đúng. Toán tử == kiểm tra bình đẳng biến tham chiếu, không bình đẳng đối tượng.
 A, B, D, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 3.1 và 3.2 của OCA)
3. đúng. Bởi vì ngắn mạch (||) không được sử dụng, cả hai toán hạng đều được đánh giá. Vì args [1] vượt quá giới hạn của mảng args , một ArrayIndexOutOfBoundsException được ném ra.
 A, B, C và D là không chính xác dựa trên các điều trên. (Mục tiêu 3.1 và 3.2 của OCA)
4. đúng. Phép nối chạy từ trái sang phải, và nếu một trong hai toán hạng là Chuỗi, thì các toán hạng sẽ được nối với nhau. Nếu cả hai toán hạng đều là số, chúng được cộng lại với nhau.
 A, B, C, D, E, F, H, và tôi không chính xác dựa trên những điều trên. (Mục tiêu 3.1 của OCA)
5. Trả lời:

```

class Incr {
    public static void main(String[] args) {
        Integer x = 7;
        int y = 2;

        x *= x;
        y *= y;
        y *= y;
        x -= y;

        System.out.println(x);
    }
}

```

Vâng, chúng tôi biết đó là một dạng câu đố-y, nhưng bạn có thể gặp phải điều gì đó giống như nó trong kỳ thi thật nếu Oracle khôi phục loại câu hỏi này. (Mục tiêu 3.1 của OCA)

6. A đúng. Khi chia số nguyên, phần còn lại luôn được làm tròn xuống.
 B, C, D và E là không chính xác dựa trên các điều trên. (Mục tiêu 3.1 của OCA)
7. A đúng. Tất cả cú pháp này đều đúng. For-each lặp qua enum bằng cách sử dụng phương thức giá trị () để trả về một mảng. Một enum có thể được so sánh bằng cách sử dụng equals () hoặc ==. Một enum có thể được sử dụng trong kiểm tra boolean của toán tử bậc ba.
 B, C, D, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu của OCA 3.1, 3.2 và 3.3)
8. C đúng. Dòng 9 sử dụng toán tử mô-đun, trả về phần còn lại của phép chia, trong trường hợp này là 1. Ngoài ra, dòng 9 đặt b2 thành false và nó không kiểm tra giá trị của b2. Dòng 10 sẽ đặt b2 thành true; tuy nhiên, toán tử ngắn mạch giữ cho biểu thức b2 = true không được thực thi.
 A, B, D và E là không chính xác dựa trên các điều trên. (Mục tiêu 3.1 và 3.2 của OCA)
9. C và F đúng. Ở dòng 7, || không đếm xỉa đến tăng dần, nhưng | cho phép mặt nạ được tăng dần. Tại dòng 8, ^ chỉ trả về true nếu đúng một toán hạng. Tại dòng 9 mặt nạ là 2 và && giữ cho số đếm không được tăng lên.
 A, B, D, E và G là không chính xác dựa trên các điều trên. (Mục tiêu 3.1 của OCA)
10. D đúng. Đầu tiên, hãy nhớ rằng instanceof có thể tra cứu

nhiều cấp độ của một cây kế thừa. Cũng nên nhớ rằng instanceof thường được sử dụng trước khi cõi gắng downcast; vì vậy trong trường hợp này, sau dòng 15, có thể nói Speedboat s3 = (Speedboat) b2 ;.

A, B, C, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 3.1 của OCA)

11. A . Chúng tôi khá chắc chắn rằng bạn sẽ không gặp phải bất cứ điều gì kinh khủng như thế này trong kỳ thi thật. Nhưng nếu bạn hiểu đúng, hãy tự vỗ về mình! Cách giải quyết một vấn đề như thế này là đánh giá biểu hiện theo từng giai đoạn. Trong trường hợp này, bạn có thể giải quyết nó như vậy:

Giai đoạn 1: giải quyết mọi việc sử dụng toán tử một ngôi

Giai đoạn 2: giải quyết bất kỳ việc sử dụng các toán tử liên quan đến phép nhân

Giai đoạn 3: xử lý phép cộng và phép trừ

Giai đoạn 4: xử lý bất kỳ toán tử mối quan hệ nào

Giai đoạn 5: Đôi phó với các toán tử bình đẳng

Giai đoạn 6: Xử lý các toán tử logic

Giai đoạn 7: thực hiện các thao tác ngắn mạch

Giai đoạn 8: Cuối cùng, thực hiện các toán tử gán

B, C, D, E và F không chính xác dựa trên các điều trên. (Mục tiêu 3.1 của OCA)



5

Kiểm soát luồng và các trường hợp ngoại lệ

CHỨNG NHẬN MỤC TIÊU

- Sử dụng lệnh if và chuyển đổi
- Phát triển vòng lặp for, do và while • Sử dụng câu lệnh break và continue
- Sử dụng câu lệnh try, catch và cuối cùng • Nêu tác dụng của các trường hợp ngoại lệ • Nhận biết các trường hợp ngoại lệ phổ biến

✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

Cbạn tưởng tượng đang cố gắng viết mã bằng một ngôn ngữ không cung cấp cho bạn cách thực thi các câu lệnh một cách có điều kiện? Kiểm soát luồng là một phần quan trọng của hầu hết bất kỳ ngôn ngữ lập trình hữu ích nào và Java cung cấp một số cách để thực hiện điều đó. Một số câu lệnh, chẳng hạn như nếu các câu lệnh và vòng lặp for , phổ biến đối với hầu hết các ngôn ngữ. Nhưng Java cũng đưa vào một vài tính năng kiểm soát luồng mà bạn có thể chưa sử dụng trước đây – các ngoại lệ và xác nhận. (Chúng ta sẽ thảo luận về các khảng định trong chương tiếp theo.)

Câu lệnh if và câu lệnh switch là các loại điều khiển điều kiện / quyết định cho phép chương trình của bạn hoạt động khác nhau tại “ngã ba đường”, tùy thuộc vào kết quả của một bài kiểm tra logic. Java cũng cung cấp ba cấu trúc lặp khác nhau – for, while và do – vì vậy bạn có thể thực thi lặp đi lặp lại cùng một đoạn mã tùy thuộc vào một số điều kiện là đúng.

Các ngoại lệ cung cấp cho bạn một cách đơn giản, rõ ràng để tổ chức mã giải quyết các vấn đề có thể xảy ra trong thời gian chạy.

Với những công cụ này, bạn có thể xây dựng một chương trình mạnh mẽ có thể xử lý mọi tình huống hợp lý một cách linh hoạt. Mong đợi thấy một loạt các câu hỏi trong bài kiểm tra bao gồm kiểm soát luồng như một phần của

duyên dáng. Mong đợi thấy một loạt các câu hỏi trong bài kiểm tra bao gồm kiểm soát luồng như một phần của mã câu hỏi, ngay cả trên những câu hỏi không kiểm tra kiến thức của bạn về kiểm soát luồng.

MỤC TIÊU XÁC NHÂN

Sử dụng câu lệnh if và switch (Mục tiêu 3.3 và 3.4 của OCA)

3.3 Tạo if và if-else và các cấu trúc bậc ba.

3.5 Sử dụng câu lệnh switch.

Câu lệnh if và switch thường được gọi là câu lệnh quyết định. Khi bạn sử dụng các câu lệnh quyết định trong chương trình của mình, bạn đang yêu cầu chương trình đánh giá một biểu thức nhất định để xác định hướng hành động nào cần thực hiện. Chúng ta sẽ xem xét câu lệnh if trước.

Phân nhánh if-else

Định dạng cơ bản của câu lệnh if như sau:

```
if (booleanExpression) {
    System.out.println("Inside if statement");
}
```

Biểu thức trong ngoặc phải đánh giá thành (một boolean) true hoặc false. Thông thường, bạn đang thử nghiệm thứ gì đó để xem nó có đúng không và sau đó chạy một khối mã (một hoặc nhiều câu lệnh) nếu nó đúng và (tùy chọn) một khối mã khác nếu nó không đúng. Đoạn mã sau thể hiện một câu lệnh if-else hợp pháp :

```
if (x > 3) {
    System.out.println("x is greater than 3");
} else {
    System.out.println("x is not greater than 3");
}
```

Khối khác là tùy chọn, vì vậy bạn cũng có thể sử dụng như sau:

```
if (x > 3) {
    y = 2;
}
z += 8;
a = y + x;
```

Đoạn mã trước đó sẽ gán 2 cho y nếu kiểm tra thành công (có nghĩa là x thực sự lớn hơn 3), nhưng hai dòng còn lại sẽ thực thi bất kể. Ngay cả dấu ngoặc nhọn cũng là tùy chọn nếu bạn chỉ có một câu lệnh để thực thi trong phần thân của khối điều kiện. Ví dụ mã sau là hợp pháp (mặc dù không được khuyến nghị để dễ đọc):

```
if (x > 3) // bad practice, but seen on the exam
    y = 2;
z += 8;
a = y + x;
```

Hầu hết các nhà phát triển coi việc đặt các khối trong dấu ngoặc nhọn là một phương pháp hay, ngay cả khi chỉ có một câu lệnh trong khối. Hãy cẩn thận với mã như phần trước, vì bạn có thể nghĩ rằng nó sẽ đọc là

"Nếu x lớn hơn 3, thì đặt y thành 2, z thành z + 8 và a thành y + x."

Nhưng hai dòng cuối cùng sẽ thực hiện bất kể điều gì! Chúng không phải là một phần của quy trình có điều kiện. Bạn có thể thấy nó thậm chí còn gây hiểu lầm nếu mã được thuat vào như sau:

```
if (x > 3)
    y = 2;
    z += 8;
    a = y + x;
```

Bạn có thể cần phải lồng các câu lệnh if-else (mặc dù, một lần nữa, nó không được khuyến khích để dễ đọc, do đó, các thử nghiệm if lồng nhau nên được giữ ở mức tối thiểu). Bạn có thể thiết lập câu lệnh if-else để kiểm tra nhiều điều kiện. Ví dụ sau sử dụng hai điều kiện, vì vậy nếu thử nghiệm đầu tiên không thành công, chúng tôi muốn thực hiện thử nghiệm thứ hai trước khi quyết định phải làm gì:

```

if (price < 300) {
    buyProduct();
} else {
    if (price < 400) {
        getApproval();
    }
    else {
        dontBuyProduct();
    }
}

```

Điều này sẽ trả về cấu trúc if-else khác , if, else if, else. Mã trước đó có thể (và nên) được viết lại như thế này:

```

if (price < 300) {
    buyProduct();
} else if (price < 400) {
    getApproval();
} else {
    dontBuyProduct();
}

```

Có một số quy tắc để sử dụng else và khác nếu:

- Bạn có thể có số không hoặc số khác đối với một nếu đã cho, và nó phải đến sau bất kỳ ifs khác .
- Bạn có thể có từ 0 đến nhiều if khác cho một if đã cho và chúng phải đứng trước if (tùy chọn) khác.
- Khi một cái khác nếu thành công, không cái nào còn lại nếu cái khác cũng như cái khác sẽ được kiểm tra.

Ví dụ sau cho thấy mã được định dạng khung khiếp cho thực thế giới. Như bạn có thể đã đoán, rất có thể bạn sẽ gặp phải định dạng như thế này trong bài kiểm tra. Trong mọi trường hợp, mã thể hiện việc sử dụng nhiều ifs khác :

```

int x = 1;
if ( x == 3 ) { }
else if (x < 4) {System.out.println("<4"); }
else if (x < 2) {System.out.println("<2"); }
else { System.out.println("else"); }

```

Nó tạo ra đâu ra này:

<4

(Lưu ý rằng ngay cả khi cái khác thứ hai nếu là đúng, nó sẽ không bao giờ đạt được.)

Đôi khi bạn có thể gặp vấn đề trong việc tìm hiểu xem nếu người khác của bạn nên ghép nối với, như sau:

```
if (exam.done())
if (exam.getScore() < 0.61)
System.out.println("Try again.");
// Which if does this belong to?
else System.out.println("Java master!");
```

Chúng tôi cố tình bỏ thụt lề trong đoạn mã này để nó không đưa ra manh mối về câu lệnh if mà cái kia thuộc về. Bạn đã tìm ra chưa? Luật Java quy định rằng mệnh đề else thuộc về câu lệnh if trong cùng mà nó có thể thuộc về nó (nói cách khác, mệnh đề gần nhất trước if không có mệnh đề khác). Trong trường hợp của ví dụ trước, else thuộc về câu lệnh if thứ hai trong danh sách. Với cách thụt lề thích hợp, nó sẽ trông như thế này:

```
if (exam.done())
    if (exam.getScore() < 0.61)
        System.out.println("Try again.");
    // Which if does this belong to?
    else
        System.out.println("Java master!");
```

Tuân theo các quy ước mã hóa của chúng tôi bằng cách sử dụng dấu ngoặc nhọn, nó thậm chí còn dễ đọc hơn:

```
if (exam.done()) {
    if (exam.getScore() < 0.61) {
        System.out.println("Try again.");
        // Which if does this belong to?
    } else {
        System.out.println("Java master!");
    }
}
```

Đừng hy vọng rằng các đề thi đều đẹp và được thụt lề đúng cách. Một số thí sinh thậm chí còn có khẩu hiệu cho cách trình bày câu hỏi trong đề thi: Bất cứ điều gì có thể gây nhầm lẫn hơn sẽ được.

Hãy chuẩn bị cho những câu hỏi không những không thụt lề đẹp mà còn cố tình thụt lề một cách gây hiểu lầm. Hãy chú ý để biết sai hướng như sau:

như sau:

```
if (exam.done())
    if (exam.getScore() < 0.61)
        System.out.println("Try again.");
else
    System.out.println("Java master!"); // Hmmmm... now where does
                                            // it belong?
```

Tất nhiên, mã trước hoàn toàn giống với hai ví dụ trước, ngoại trừ giao diện của nó.

Biểu thức pháp lý cho Câu lệnh if Biểu

thức trong câu lệnh if phải là một biểu thức boolean . Bất kỳ biểu thức nào phân giải thành boolean đều tốt và một số biểu thức có thể phức tạp. Giả sử doStuff () trả về true,

```
int y = 5;
int x = 2;
if (((x > 3) && (y < 2)) | doStuff()) {
    System.out.println("true");
}
```

cái nào in

thật

Bạn có thể đọc đoạn mã trước đó là “Nếu cả ($x > 3$) và ($y < 2$) đều đúng hoặc nếu kết quả của doStuff () là đúng, thì hãy in đúng.” Vì vậy, về cơ bản, nếu chỉ một mình doStuff () là đúng, chúng ta sẽ vẫn nhận được đúng. Tuy nhiên, nếu doStuff () là false, thì cả ($x > 3$) và ($y < 2$) sẽ phải đúng để in ra true. Mã trước thậm chí còn phức tạp hơn nếu bạn bỏ đi một tập hợp các dấu ngoặc đơn như sau:

```
int y = 5;
int x = 2;
if ((x > 3) && (y < 2) | doStuff()) {
    System.out.println("true");
}
```

Cái này bây giờ in ... không có gì! Bởi vì mã trước đó (với một bộ dấu ngoặc đơn ít hơn) đánh giá như thể bạn đang nói, “Nếu ($x > 3$) là đúng, và ($y < 2$) hoặc kết quả của doStuff () là đúng, thì in true. Vì vậy, nếu ($x > 3$) không đúng, thì chẳng ích gì khi nhìn vào phần còn lại của biểu thức ”. Do ngắn mạch $&&$, biểu thức được đánh giá như thể có dấu ngoặc đơn xung quanh

(y <2) | doStuff (). Nói cách khác, nó được đánh giá là một biểu thức duy nhất trước && và một biểu thức sau &&.

Hãy nhớ rằng biểu thức hợp pháp duy nhất trong phép thử if là boolean. Trong một số ngôn ngữ, 0 == false và 1 == true. Không phải như vậy trong Java! Đoạn mã sau cho biết nếu các tuyên bố có vẻ hấp dẫn nhưng là bất hợp pháp, theo sau là các thay thế hợp pháp:

```
int trueInt = 1;
int falseInt = 0;
if (trueInt)           // illegal
if (trueInt == true)   // illegal
if (1)                 // illegal
if (falseInt == false) // illegal
if (trueInt == 1)       // legal
if (falseInt == 0)      // legal
```



Một sai lầm phổ biến mà các lập trình viên mắc phải (và điều đó có thể khó phát hiện) là gán một biến boolean khi bạn muốn kiểm tra một biến boolean . Hãy tìm mã như sau:

```
boolean boo = false;
if (boo = true) { }
```

Bạn có thể nghĩ một trong ba điều:

1. Mã biên dịch và chạy tốt, và nếu kiểm tra không thành công vì boo là sai.
2. Mã sẽ không biên dịch vì bạn đang sử dụng một phép gán (=) chứ không phải là một bài kiểm tra bình đẳng (==).
3. Mã biên dịch và chạy tốt, và kiểm tra if thành công vì boo được ĐẶT thành true (chứ không phải TESTED cho đúng) trong đối số if !

Chà, số 3 đúng - vô nghĩa, nhưng đúng. Cho rằng kết quả của bất kỳ phép gán nào là giá trị của biến sau phép gán, biểu thức (boo = true) có kết quả là true. Do đó, nếu kiểm tra thành công. Nhưng các biến duy nhất có thể được chỉ định (thay vì được kiểm tra

chóng lại một cái gì đó khác) là một boolean hoặc một Boolean; tất cả các phép gán khác sẽ dẫn đến một cái gì đó không phải boolean, vì vậy chúng không hợp pháp, như sau:

```
int x = 3;
if (x = 5) { } // Won't compile because x is not a boolean!
```

Bởi vì nếu các bài kiểm tra yêu cầu các biểu thức boolean , bạn cần phải thực sự vững chắc về cả các toán tử logic và cú pháp và ngữ nghĩa của bài kiểm tra .

chuyển đổi các câu lệnh

Bạn đã thấy cách các câu lệnh if và else-if có thể được sử dụng để hỗ trợ cả logic quyết định đơn giản và phức tạp. Trong nhiều trường hợp, câu lệnh switch cung cấp một cách rõ ràng hơn để xử lý logic quyết định phức tạp. Hãy so sánh câu lệnh if else sau đây với câu lệnh switch hoạt động tương đương :

```
int x = 3;
if(x == 1) {
    System.out.println("x equals 1");
}
else if(x == 2) {
    System.out.println("x equals 2");
}
else {
    System.out.println("No idea what x is");
}
```

Bây giờ chúng ta hãy xem cùng một chức năng được thể hiện trong cấu trúc chuyển đổi :

```
int x = 3;
switch (x) {
    case 1:
        System.out.println("x equals 1");
        break;
    case 2:
        System.out.println("x equals 2");
        break;
    default:
        System.out.println("No idea what x is");
}
```

Lưu ý: Lý do câu lệnh switch này mô phỏng if là do các câu lệnh break được đặt bên trong công tắc. Nói chung, câu lệnh break là tùy chọn và như bạn sẽ thấy trong một vài trang, việc bao gồm hoặc loại trừ chúng gây ra những thay đổi lớn về cách thực thi câu lệnh switch .

Biểu thức pháp lý cho công tắc và trường hợp

Dạng chung của câu lệnh switch là

```
switch (expression) {
    case constant1: code block
    case constant2: code block
    default: code block
}
```

Biểu thức của switch phải đánh giá thành char, byte, short, int, enum (đối với Java 5) và chuỗi (đối với Java 7). Điều đó có nghĩa là nếu bạn không sử dụng enum hoặc String, chỉ có thể chấp nhận các biến và giá trị có thể được tự động thăng cấp (nói cách khác là ép kiểu ngầm) thành int . Bạn sẽ không thể biên dịch nếu bạn sử dụng bất kỳ thứ gì khác, bao gồm các kiểu số còn lại là long, float và double.

Hằng số phải đánh giá cùng kiểu mà biểu thức switch có thể sử dụng, với một ràng buộc bổ sung – và lớn -: hằng số phải là hằng số thời gian biên dịch! Vì đối số trường hợp phải được giải quyết tại thời điểm biên dịch, bạn chỉ có thể sử dụng một hằng số hoặc một biến cuối cùng được khởi tạo ngay lập tức với một giá trị chữ. Nó là không đủ để được cuối cùng; nó phải là một hằng số thời gian biên dịch. Đây là một ví dụ:

```
final int a = 1;
final int b;
b = 2;
int x = 0;
switch (x) {
    case a:      // ok
    case b:      // compiler error
```

Ngoài ra, công tắc chỉ có thể kiểm tra sự bình đẳng. Điều này có nghĩa là các toán tử quan hệ khác chẳng hạn như lớn hơn được hiển thị không sử dụng được trong một trường hợp. Sau đây là một ví dụ về một biểu thức hợp lệ sử dụng một lệnh gọi phương thức trong một câu lệnh switch . Lưu ý rằng để mã này hợp pháp, phương thức được gọi trên tham chiếu đối tượng phải trả về một giá trị tương thích với một int.

```

String s = "xyz";
switch (s.length()) {
    case 1:
        System.out.println("length is one");
        break;
    case 2:
        System.out.println("length is two");
        break;
    case 3:
        System.out.println("length is three");
        break;
    default:
        System.out.println("no match");
}

```

Một quy tắc khác mà bạn có thể không mong đợi liên quan đến câu hỏi, "Điều gì xảy ra nếu Tôi bật một biến nhỏ hơn một int?" Nhìn vào công tắc sau:

```

byte g = 2;
switch(g) {
    case 23:
    case 128:
}

```

Mã này sẽ không biên dịch. Mặc dù đối số switch là hợp pháp - một byte được truyền ngầm thành int - đối số trường hợp thứ hai (128) quá lớn đối với một byte và trình biên dịch biết điều đó! Có gắng biên dịch ví dụ trước sẽ cho bạn một lỗi như sau:

```

Test.java:6: possible loss of precision
found   : int
required: byte
    case 128:
               ^

```

Việc có nhiều hơn một nhãn trường hợp sử dụng cùng một giá trị cũng là bất hợp pháp. Ví dụ: khỏi mã sau sẽ không biên dịch vì nó sử dụng hai trường hợp có cùng giá trị 80:

```

int temp = 90;
switch(temp) {
    case 80 : System.out.println("80");
    case 80 : System.out.println("80"); // won't compile!
    case 90 : System.out.println("90");
    default : System.out.println("default");
}

```

Việc tận dụng sức mạnh của quyền anh trong một biểu thức chuyển đổi là hợp pháp . Vì ví dụ, sau đây là hợp pháp:

```
switch(new Integer(4)) {
    case 4: System.out.println("boxing is OK");
}
```



Tìm bất kỳ vi phạm nào đối với quy tắc chuyển đổi và đổi số trường hợp. Ví dụ: bạn có thể tìm thấy các ví dụ bất hợp pháp như các đoạn mã sau:

```
switch(x) {
    case 0 {
        y = 7;
    }
}

switch(x) {
    0: { }
    1: { }
}
```

Trong ví dụ đầu tiên, trường hợp bỏ qua dấu hai chấm. Ví dụ thứ hai bỏ qua trường hợp từ khóa.

Giới thiệu về chuỗi "bình đẳng"

Như chúng ta đã thảo luận, hoạt động của các câu lệnh switch phụ thuộc vào biểu thức "khớp" hoặc "bằng" với một trong các trường hợp. Chúng ta đã nói về cách chúng ta biết khi nào các nguyên thủy bằng nhau, nhưng điều đó có nghĩa là gì đối với các đối tượng bằng nhau? Đây là một trong những chủ đề phức tạp đáng ngạc nhiên khác và đối với những bạn có ý định tham gia kỳ thi OCP, bạn sẽ dành rất nhiều thời gian để nghiên cứu "bình đẳng đối tượng". Đối với các ứng cử viên OCA, tất cả những gì bạn phải biết là đối với một câu lệnh switch , hai Chuỗi sẽ được coi là "bằng nhau" nếu chúng có cùng một chuỗi ký tự phân biệt chữ hoa chữ thường. Ví dụ, trong câu lệnh chuyển đổi từng phần sau , biểu thức sẽ khớp với trường hợp:

```
String s = "Monday";
switch(s) {
    case "Monday":    // matches!
```

Nhưng những điều sau KHÔNG phù hợp:

```
String s = "MONDAY";
switch(s) {
    case "Monday":    // Strings are case-sensitive, DOES NOT match
```

Break and Fall-Through trong khối công tắc Cuối cùng

chúng tôi đã sẵn sàng để thảo luận về tuyên bố ngắn và cung cấp thêm chi tiết về điều khiển luồng trong một tuyên bố chuyển đổi . Điều quan trọng nhất cần nhớ về quy trình thực thi thông qua một câu lệnh switch là:

hàng số trường hợp được đánh giá từ trên xuống và hàng số trường hợp đầu tiên phù hợp với biểu thức của công tắc là điểm nhập thực thi.

Nói cách khác, khi một hàng số được khớp, Máy ảo Java (JVM) sẽ thực thi khối mã được liên kết và TẮT CẢ các khối mã tiếp theo (trừ câu lệnh ngắn) cũng vậy!

Ví dụ sau sử dụng một chuỗi trong một trường hợp

bản tường trình:

```
class SwitchString {
    public static void main(String [] args) {
        String s = "green";
        switch(s) {
            case "red": System.out.print("red ");
            case "green": System.out.print("green ");
            case "blue": System.out.print("blue ");
            default: System.out.println("done");
        }
    }
}
```

Trong trường hợp ví dụ này “xanh”: đã khớp, vì vậy JVM đã thực thi khối mã đó và tắt cả các khối mã tiếp theo để tạo ra đầu ra:

xanh xanh hoàn thành

Một lần nữa, khi chương trình gặp sự cố ngắn từ khóa trong quá trình thực hiện câu lệnh switch , việc thực thi sẽ ngay lập tức chuyển ra khỏi khối switch đến câu lệnh tiếp theo sau lệnh switch. Nếu ngắn quãng bị bỏ qua, chương trình chỉ

tiếp tục thực hiện các khối trường hợp còn lại cho đến khi tìm thấy ngắt hoặc kết thúc câu lệnh switch . Kiểm tra đoạn mã sau:

```
int x = 1;
switch(x) {
    case 1: System.out.println("x is one");
    case 2: System.out.println("x is two");
    case 3: System.out.println("x is three");
}
System.out.println("out of the switch");
```

Mã sẽ in như sau:

```
x is one
x is two
x is three
out of the switch
```

Sự kết hợp này xảy ra vì mã không đạt được câu lệnh ngắt ; việc thi hành cứ giảm dần qua từng trường hợp cho đến khi kết thúc. Sự giảm xuống này thực sự được gọi là "rơi xuống", vì cách thực hiện rơi từ trường hợp này sang trường hợp tiếp theo. Hãy nhớ rằng, trường hợp khớp chỉ đơn giản là điểm nhập của bạn vào khối chuyển đổi ! Nói cách khác, bạn không được nghĩ về nó như là, "Tìm trường hợp phù hợp, chỉ thực thi mã đó và thoát ra." Đó không phải là cách nó hoạt động. Nếu bạn muốn hành vi "chỉ mã phù hợp", bạn sẽ chèn dấu ngắt vào từng trường hợp như sau:

```
int x = 1;
switch(x) {
    case 1: {
        System.out.println("x is one");
        break;
    }
    case 2: {
        System.out.println("x is two");
        break;
    }
    case 3: {
        System.out.println("x is two");
        break;
    }
}
System.out.println("out of the switch");
```

Chạy mã trước, bây giờ chúng tôi đã thêm các câu lệnh ngắt , sẽ in:

```
x is one
out of the switch
```

Và đó là nó. Chúng tôi đã nhập vào khối chuyển đổi ở trường hợp 1. Vì nó khớp với đối số switch () , chúng tôi nhận được câu lệnh println và sau đó nhấn break và nhảy đến cuối công tắc.

Một ví dụ thú vị về logic chuyển tiếp này được hiển thị trong đoạn mã sau:

```
int x = someNumberBetweenOneAndTen;
switch (x) {
    case 2:
    case 4:
    case 6:
    case 8:
    case 10: {
        System.out.println("x is an even number"); break;
    }
}
```

Câu lệnh switch này sẽ in ra x là số chẵn hoặc không, tùy thuộc vào việc số đó nằm trong khoảng từ một đến mươi và là số lẻ hay chẵn. Ví dụ: nếu x là 4, quá trình thực thi sẽ bắt đầu ở trường hợp 4, nhưng sau đó giảm xuống 6, 8 và 10, nơi nó in và sau đó ngắt. Nhân tiện, việc nghỉ ở trường hợp 10 là không cần thiết; dù sao thì chúng tôi cũng đã ở cuối quá trình chuyển đổi .

Lưu ý: Bởi vì quá trình dự phòng ít trực quan hơn, Oracle khuyến nghị bạn thêm một nhận xét, chẳng hạn như // fall through khi bạn sử dụng logic Fall-through.

Trường hợp mặc định

Điều gì sẽ xảy ra nếu, bằng cách sử dụng mã trước đó, bạn muốn in x là một số lẻ nếu không có trường hợp nào (các số chẵn) phù hợp? Bạn không thể đặt nó sau câu lệnh switch , hoặc thậm chí là trường hợp cuối cùng trong switch, bởi vì trong cả hai trường hợp đó, nó sẽ luôn in ra x là số lẻ. Để có được hành vi này, bạn sẽ sử dụng từ khóa mặc định . (Nhân tiện, nếu bạn tự hỏi tại sao có từ khóa mặc định mặc dù chúng tôi không sử dụng công cụ sửa đổi cho kiểm soát truy cập mặc định, thì bây giờ bạn sẽ thấy rằng từ khóa mặc định được sử dụng cho một mục đích hoàn toàn khác.) thay đổi chúng ta cần thực hiện là thêm trường hợp mặc định vào mã trước:

```
int x = someNumberBetweenOneAndTen;
switch (x) {
    case 2:
    case 4:
    case 6:
    case 8:
    case 10: { System.out.println("x is even"); break; }
    default: System.out.println("x is an odd number");
}
```



Trường hợp mặc định không nhất thiết phải xuất hiện ở cuối công tắc. Tìm kiếm nó ở những nơi lạ như sau:

```
int x = 2;
switch (x) {
    case 2: System.out.println("2");
    default: System.out.println("default");
    case 3: System.out.println("3");
    case 4: System.out.println("4");
}
```

Chạy mã trước sẽ in ra:

```
2
default
3
4
```

Và nếu chúng tôi sửa đổi nó để kết quả phù hợp duy nhất là trường hợp mặc định, như thế này,

```
int x = 7;
switch (x) {
    case 2: System.out.println("2");
    default: System.out.println("default");
    case 3: System.out.println("3");
    case 4: System.out.println("4");
}
```

sau đó chạy mã trước sẽ in ra:

```
default  
3  
4
```

Quy tắc cần nhớ là mặc định hoạt động giống như bất kỳ trường hợp dự phòng nào khác!

BÀI TẬP 5-1

Tạo một câu lệnh switch-case

Hãy thử tạo một câu lệnh switch sử dụng giá trị char làm trường hợp. Bao gồm một hành vi mặc định nếu không có giá trị char nào khớp.

- Đảm bảo rằng một biến char được khai báo trước câu lệnh switch .
- Mỗi câu lệnh trường hợp phải được theo sau bởi một dấu ngắt.
- Trường hợp mặc định có thể được đặt ở cuối, giữa hoặc trên cùng.

MỤC TIÊU XÁC NHẬN

Tạo cấu trúc vòng lặp (Mục tiêu của OCA 5.1, 5.2, 5.3, 5.4 và 5.5)

- 5.1 Tạo và sử dụng vòng lặp while.
- 5.2 Tạo và sử dụng vòng lặp for bao gồm vòng lặp for nâng cao.
- 5.3 Tạo và sử dụng vòng lặp do / while.
- 5.4 So sánh cấu trúc vòng lặp.
- 5.5 Sử dụng ngắt và tiếp tục.

Vòng lặp Java có ba loại: while, do và for (và kể từ Java 5, vòng lặp for có hai biến thể). Cả ba đều cho phép bạn lặp lại một khối mã miễn là một số điều kiện là đúng hoặc cho một số lần lặp cụ thể. Có thể bạn đã quen với các vòng lặp từ các ngôn ngữ khác, vì vậy ngay cả khi bạn là người mới làm quen với Java, đây sẽ không phải là vấn đề để học.

Sử dụng vòng lặp while

Vòng lặp while là tốt khi bạn không biết khỏi hoặc câu lệnh phải lặp lại bao nhiêu lần nhưng bạn muốn tiếp tục lặp lại miễn là một số điều kiện là đúng. Một câu lệnh while trông như thế này:

```
while (expression) {
    // do stuff
}
```

Hoặc cái này:

```
int x = 2;
while(x == 2) {
    System.out.println(x);
    ++x;
}
```

Trong trường hợp này, như trong tất cả các vòng lặp, biểu thức (kiểm tra) phải đánh giá thành boolean kết quả. Phần thân của vòng lặp while sẽ chỉ thực thi nếu biểu thức (đôi khi được gọi là "điều kiện") cho kết quả là giá trị true. Khi ở bên trong vòng lặp, phần thân của vòng lặp sẽ lặp lại cho đến khi điều kiện không còn được đáp ứng nữa vì nó đánh giá là false. Trong ví dụ trước, chương trình điều khiển sẽ nhập vào phần thân của vòng lặp vì x bằng 2. Tuy nhiên, x được tăng dần trong vòng lặp, vì vậy khi điều kiện được kiểm tra lại, nó sẽ đánh giá là false và thoát khỏi vòng lặp.

Bất kỳ biến nào được sử dụng trong biểu thức của vòng lặp while phải được khai báo trước khi biểu thức được đánh giá. Nói cách khác, bạn không thể nói điều này:

```
while (int x = 2) {} // không hợp pháp
```

Sau đó, một lần nữa, tại sao bạn sẽ? Thay vì kiểm tra biến, bạn sẽ khai báo và khởi tạo biến, vì vậy, biến sẽ luôn có cùng giá trị. Không có nhiều điều kiện thử nghiệm!

Điểm quan trọng cần nhớ về vòng lặp while là nó có thể không bao giờ chạy. Nếu biểu thức kiểm tra là false lần đầu tiên biểu thức while được kiểm tra, phần thân của vòng lặp sẽ bị bỏ qua và chương trình sẽ bắt đầu thực hiện ở câu lệnh đầu tiên sau vòng lặp while . Hãy xem ví dụ sau:

```
int x = 8;
while (x > 8) {
    System.out.println("in the loop");
    x = 10;
}
System.out.println("past the loop");
```

Chạy mã này sẽ tạo ra

qua vòng lặp

Bởi vì biểu thức ($x > 8$) đánh giá là false, không có mã nào trong vòng lặp while được thực thi.

Sử dụng do Loops

Vòng lặp do tương tự như vòng lặp while , ngoại trừ biểu thức không được đánh giá cho đến khi mã của vòng lặp do được thực thi. Do đó, mã trong vòng lặp do được đảm bảo thực thi ít nhất một lần. Phần sau cho thấy một vòng lặp do đang hoạt động:

```
do {
    System.out.println("Inside loop");
} while(false);
```

Câu lệnh System.out.println () sẽ in một lần, ngay cả khi biểu thức đánh giá là false. Hãy nhớ rằng, vòng lặp do sẽ luôn chạy mã trong phần thân của vòng lặp ít nhất một lần. Hãy nhớ lưu ý việc sử dụng dấu chấm phẩy ở cuối biểu thức while .



Như với các bài kiểm tra if, hãy tìm các vòng lặp while (và kiểm tra while trong vòng lặp do) có biểu thức không phân giải thành boolean. Hãy xem các ví dụ sau về các biểu thức while hợp pháp và bất hợp pháp :

```

int x = 1;
while (x) { }           // Won't compile; x is not a boolean
while (x = 5) { }       // Won't compile; resolves to 5
                      // (as the result of assignment)
while (x == 5) { }     // Legal, equality test
while (true) { }        // Legal

```

Sử dụng cho các vòng lặp

Đối với Java 5, vòng lặp for có cấu trúc thứ hai. Chúng tôi sẽ gọi kiểu cũ của vòng lặp for là "vòng lặp for cơ bản" và chúng tôi sẽ gọi kiểu mới của vòng lặp for là "vòng lặp for nâng cao" (đôi khi nó cũng được gọi là for-each). Tùy thuộc vào tài liệu bạn sử dụng, bạn sẽ thấy cả hai điều khoản, cùng với bổ sung. Tất cả các thuật ngữ for-in, for-each và "advanced for" đều đề cập đến cùng một Java xây dựng.

Vòng lặp for cơ bản linh hoạt hơn vòng lặp for nâng cao, nhưng vòng lặp for nâng cao được thiết kế để làm cho việc lặp qua các mảng và bộ sưu tập dễ viết mã hơn.

Vòng lặp for cơ bản Vòng

lặp for đặc biệt hữu ích cho việc điều khiển luồng khi bạn đã biết bạn cần thực hiện các câu lệnh trong khôi của vòng lặp bao nhiêu lần. Khai báo vòng lặp for có ba phần chính bên cạnh phần thân của vòng lặp:

- Khai báo và khởi tạo biến
- Biểu thức boolean (kiểm tra có điều kiện)
- Biểu thức lặp

Ba phần khai báo for được phân tách bằng dấu chấm phẩy. Hai ví dụ sau đây minh họa cho vòng lặp for. Ví dụ đầu tiên hiển thị các phần của vòng lặp for ở dạng mã giả và ví dụ thứ hai cho thấy ví dụ điển hình về vòng lặp for :

```

for /*Initialization*/ ; /*Condition*/ ; /* Iteration */) {
    /* loop body */
}

for (int i = 0; i<10; i++) {
    System.out.println("i is " + i);
}

```

Cơ bản cho Vòng lặp: Khai báo và Khởi tạo Phần đầu tiên của câu lệnh for cho phép bạn khai báo và khởi tạo không, một hoặc nhiều biến cùng kiểu bên trong dấu ngoặc đơn sau từ khóa for . Nếu bạn khai báo nhiều biến cùng kiểu, bạn sẽ cần phân tách chúng bằng dấu phẩy như sau:

```
for (int x = 10, y = 3; y> 3; y ++) {}
```

Việc khai báo và khởi tạo xảy ra trước bất kỳ thứ gì khác trong vòng lặp for . Và trong khi hai phần khác – kiểm tra boolean và biểu thức lặp – sẽ chạy với mỗi lần lặp của vòng lặp, việc khai báo và khởi tạo chỉ xảy ra một lần ngay từ đầu. Bạn cũng phải biết rằng phạm vi của các biến được khai báo trong vòng lặp for kết thúc bằng vòng lặp for ! Những điều sau đây chứng minh điều này:

```

for (int x = 1; x < 2; x++) {
    System.out.println(x); // Legal
}
System.out.println(x); // Not Legal! x is now out of scope
                      // and can't be accessed.

```

Nếu bạn cố gắng biên dịch cái này, bạn sẽ nhận được một cái gì đó như thế này:

```

Test.java:19: cannot resolve symbol
symbol : variable x
location: class Test
    System.out.println(x);
               ^

```

Basic for Loop : Conditional (boolean) Bạn có thể chỉ có một biểu thức logic, nhưng nó có thể rất phức tạp. Hãy tìm mã sử dụng các biểu thức logic như sau:

```
for (int x = 0; (((x < 10) && (y-- > 2)) | x == 3)); x++) {}
```

Mã trước là hợp pháp, nhưng mã sau thì không:

```
for (int x = 0; (x > 5), (y < 2); x++) { } // too many
                                                // expressions
```

Trình biên dịch sẽ cho bạn biết vấn đề:

```
TestLong.java:20: ';' expected
for (int x = 0; (x > 5), (y < 2); x++) { }
```

Quy tắc cần nhớ là: Bạn chỉ có thể có một biểu thức kiểm tra.

Nói cách khác, bạn không thể sử dụng nhiều bài kiểm tra được phân tách bằng dấu phẩy, thậm chí tuy nhiên, hai phần khác của câu lệnh for có thể có nhiều phần.

Cơ bản cho vòng lặp: Biểu thức lặp Sau mỗi lần

thực hiện phần thân của vòng lặp for , biểu thức lặp được thực thi. Đây là nơi bạn có thể nói những gì bạn muốn xảy ra với mỗi lần lặp lại của vòng lặp. Hãy nhớ rằng nó luôn xảy ra sau khi phần thân của vòng lặp chạy!

Nhìn vào phần sau:

```
for (int x = 0; x < 1; x++) {
    // body code that doesn't change the value of x
}
```

Vòng lặp này chỉ thực hiện một lần. Lần đầu tiên vào vòng lặp, x được đặt thành 0, sau đó x được kiểm tra để xem nó có nhỏ hơn 1 (chính là) hay không, và sau đó phần thân của vòng lặp sẽ thực thi. Sau khi phần nội dung của vòng lặp chạy, biểu thức lặp chạy, tăng x lên 1. Tiếp theo, kiểm tra điều kiện được kiểm tra và vì kết quả bây giờ là false, nên việc thực thi sẽ nhảy xuống bên dưới vòng lặp for và tiếp tục.

Hãy nhớ rằng chặn một lối ra bắt buộc, đánh giá biểu thức lặp và sau đó đánh giá biểu thức điều kiện luôn là hai điều cuối cùng xảy ra trong vòng lặp for !

Ví dụ về các lần thoát bắt buộc bao gồm ngắt, trả về, System.exit () và ngoại lệ, tất cả sẽ khiến một vòng lặp kết thúc đột ngột mà không cần chạy biểu thức lặp. Nhìn vào đoạn mã sau:

```

static boolean doStuff() {
    for (int x = 0; x < 3; x++) {
        System.out.println("in for loop");
        return true;
    }
    return true;
}

```

Chạy mã này sẽ tạo ra

trong vòng lặp

Câu lệnh chỉ in một lần vì trả về khiến việc thực thi không chỉ để lại lần lặp hiện tại của một vòng lặp mà còn toàn bộ phương thức. Vì vậy, biểu thức lặp không bao giờ chạy trong trường hợp đó. [Bảng 5-1](#) liệt kê các nguyên nhân và kết quả của việc chấm dứt vòng lặp đột ngột.

BẢNG 5-1 Các nguyên nhân chấm dứt sớm vòng lặp

Code in Loop	What Happens
break	Execution jumps immediately to the first statement after the <code>for</code> loop.
return	Execution jumps immediately back to the calling method.
<code>System.exit()</code>	All program execution stops; the VM shuts down.

Cơ bản cho vòng lặp: cho các vấn đề về

vòng lặp Không yêu cầu phần nào trong ba phần của phần khai báo! Ví dụ sau là hoàn toàn hợp pháp (mặc dù không nhất thiết phải là thông lệ tốt):

```

for( ; ; ) {
    System.out.println("Inside an endless loop");
}

```

Trong ví dụ này, tất cả các phần khai báo được bỏ đi, vì vậy vòng lặp `for` sẽ hoạt động giống như một vòng lặp vô tận.

Đối với kỳ thi, điều quan trọng là phải biết rằng nếu không có khởi tạo và các phần tăng dần, vòng lặp sẽ hoạt động giống như vòng lặp `while`. Ví dụ sau minh họa cách thực hiện điều này:

```

int i = 0;

for (;i<10;) {
    i++;
    // do some other work
}

```

Ví dụ tiếp theo minh họa một vòng lặp for với nhiều biến đang hoạt động. Đầu phẩy phân tách các biến và chúng phải cùng loại. Hãy nhớ rằng các biến được khai báo trong câu lệnh for đều là cục bộ của vòng lặp for và không thể được sử dụng bên ngoài phạm vi của vòng lặp.

```

for (int i = 0,j = 0; (i<10) && (j<10); i++, j++) {
    System.out.println("i is " + i + " j is " +j);
}

```



Phạm vi biến đóng một vai trò lớn trong kỳ thi. Bạn cần biết rằng một biến được khai báo trong vòng lặp for không thể được sử dụng ngoài vòng lặp for . Nhưng một biến chỉ được khởi tạo trong câu lệnh for (nhưng được khai báo trước đó) có thể được sử dụng ngoài vòng lặp. Ví dụ: điều sau là hợp pháp:

```

int x = 3;
for (x = 12; x < 20; x++) { }
System.out.println(x);

```

Nhưng đây không phải là:

```
for (int x = 3; x <20; x ++) {} System.out.println (x);
```

Điều cuối cùng cần lưu ý là cả ba phần của vòng lặp for đều độc lập của nhau. Ba biểu thức trong câu lệnh for không cần phải hoạt động trên cùng một biến, mặc dù chúng thường hoạt động. Nhưng ngay cả biểu thức trình lặp, mà nhiều người gọi nhầm là “biểu thức tăng”, không cần phải tăng hoặc đặt bất kỳ thứ gì; bạn có thể đưa vào hầu như bất kỳ câu lệnh mã tùy ý nào mà bạn muốn xảy ra với mỗi lần lặp lại của vòng lặp. Nhìn vào phần sau:

```

int b = 3;
for (int a = 1; b != 1; System.out.println("iterate")) {
    b = b - a;
}

```

Mã trước sẽ in

lặp đi
lặp lại



Nhiều câu hỏi trong danh sách các bài kiểm tra Java 8 "Biên dịch không thành công" và "Một ngoại lệ xảy ra trong thời gian chạy" là các câu trả lời có thể, khiến chúng khó hơn vì bạn không thể đơn giản làm việc thông qua hành vi của mã. Trước tiên, bạn phải đảm bảo mã không vi phạm bất kỳ quy tắc cơ bản nào dẫn đến lỗi trình biên dịch và sau đó tìm kiếm các ngoại lệ có thể có. Chỉ sau khi bạn đã thỏa mãn hai điều đó, bạn mới nên đi sâu vào logic và luồng mã trong câu hỏi.

Tăng cường cho Vòng lặp (cho Mảng)

Vòng lặp for nâng cao , mới của Java 5, là vòng lặp for chuyên biệt giúp đơn giản hóa việc lặp qua một mảng hoặc một tập hợp. Trong chương này, chúng ta sẽ tập trung vào việc sử dụng for nâng cao để lặp qua các mảng. Trong [Chương 6](#) , chúng ta sẽ xem lại phần nâng cao cho, khi chúng ta thảo luận về lớp bộ sưu tập ArrayList – nơi phần tăng cường cho thực sự xuất hiện.

Thay vì có ba thành phần, nâng cao cho có hai. Hãy lặp lại thông qua một mảng theo cách cơ bản (cũ) và sau đó sử dụng nâng cao cho:

```

int [] a = {1,2,3,4};
for(int x = 0; x < a.length; x++) // basic for loop
    System.out.print(a[x]);
for(int n : a) // enhanced for loop
    System.out.print(n);

```

Điều này tạo ra kết quả sau:

12341234

Chính thức hơn, hãy mô tả phần nâng cao như sau:

```
for (khai báo: biểu thức)
```

Hai phần của câu lệnh for là

- khai báo Biến khói mới khai báo có kiểu tương thích với các phần tử của mảng bạn đang truy cập. Biến này sẽ có sẵn trong khối for và giá trị của nó sẽ giống như phần tử mảng hiện tại.
- biểu thức Điều này phải đánh giá đến mảng bạn muốn lặp qua.
Đây có thể là một biến mảng hoặc một lệnh gọi phương thức trả về một mảng. Mảng có thể là bất kỳ kiểu nào: nguyên thủy, đối tượng hoặc thậm chí là mảng của mảng.

Sử dụng các định nghĩa trước, chúng ta hãy xem xét một số nâng cao hợp pháp và bất hợp pháp cho các khai báo:

```
int x;
long x2;
long [] la = {7L, 8L, 9L};
int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
String [] sNums = {"one", "two", "three"};
Animal [] animals = {new Dog(), new Cat()};

// legal 'for' declarations
for(long y : la) ;           // loop thru an array of longs
for(int[] n : twoDee) ;       // loop thru the array of arrays
for(int n2 : twoDee[2]) ;     // loop thru the 3rd sub-array
for(String s : sNums) ;       // loop thru the array of Strings
for(Object o : sNums) ;       // set an Object reference to
                             // each String
for(Animal a : animals) ;     // set an Animal reference to each
                             // element

// ILLEGAL 'for' declarations
for(x2 : la) ;               // x2 is already declared
for(int x4 : twoDee) ;        // can't stuff an array into an int
for(int x3 : la) ;             // can't stuff a long into an int
for(Dog d : animals) ;        // you might get a Cat!
```

Vòng lặp for nâng cao giả định rằng, trừ khi thoát khỏi vòng lặp sớm, bạn sẽ luôn lặp lại mọi phần tử của mảng. Các cuộc thảo luận sau về break và tiếp tục áp dụng cho cả vòng lặp for cơ bản và nâng cao .

Sử dụng break và tiếp tục

Các từ khóa break và continue được sử dụng để dừng toàn bộ vòng lặp (break) hoặc chỉ lặp lại hiện tại (tiếp tục). Thông thường, nếu bạn đang sử dụng break hoặc continue, bạn sẽ thực hiện kiểm tra if trong vòng lặp và nếu một số điều kiện trở thành true (hoặc false tùy thuộc vào chương trình), bạn muốn thoát ra ngay lập tức.

Sự khác biệt giữa chúng là bạn có tiếp tục với một lần lặp mới hay không hoặc chuyển đến câu lệnh đầu tiên bên dưới vòng lặp và tiếp tục từ đó.



Hãy nhớ rằng, các câu lệnh continue phải nằm trong một vòng lặp; nếu không, bạn sẽ gặp lỗi trình biên dịch. câu lệnh break phải được sử dụng bên trong một vòng lặp hoặc một câu lệnh chuyển đổi .

Câu lệnh break khiến chương trình ngừng thực thi phần trong cùng lặp lại và bắt đầu xử lý dòng mã tiếp theo sau khối.

Câu lệnh continue chỉ khiếun quá trình lặp hiện tại của vòng lặp trong cùng dừng lại và lần lặp tiếp theo của cùng một vòng lặp sẽ bắt đầu nếu điều kiện của vòng lặp được đáp ứng. Khi sử dụng câu lệnh continue với vòng lặp for , bạn cần xem xét những ảnh hưởng của câu lệnh continue đối với việc lặp lại vòng lặp. Kiểm tra đoạn mã sau:

```
for (int i = 0; i < 10; i++) {
    System.out.println("Inside loop");
    continue;
}
```

Câu hỏi đặt ra là đây có phải là một vòng lặp vô tận? Câu trả lời là không. Khi câu lệnh continue được nhấn, biểu thức lặp vẫn chạy! Nó chạy giống như thể lặp lại hiện tại đã kết thúc "theo cách tự nhiên." Vì vậy, trong ví dụ trước, i sẽ vẫn tăng trước khi điều kiện ($i < 10$) được kiểm tra lại.

Hầu hết thời gian, tiếp tục được sử dụng trong kiểm tra if như sau:

```

for (int i = 0; i < 10; i++) {
    System.out.println("Inside loop");
    if (foo.doStuff() == 5) {
        continue;
    }
    // more loop code, that won't be reached when the above if
    // test is true
}

```

Tuyên bố không được gắn nhãn

Cả câu lệnh break và câu lệnh continue đều có thể được bỏ nhãn hoặc gắn nhãn. Mặc dù việc sử dụng break và continue không được gắn nhãn phổ biến hơn nhiều, nhưng bài kiểm tra hy vọng bạn biết cách hoạt động của các câu lệnh break và continue được gắn nhãn . Như đã nêu trước đây, một câu lệnh break (không được gắn nhãn) sẽ thoát ra khỏi cấu trúc lặp trong cùng và tiếp tục với dòng mã tiếp theo bên ngoài khôi lặp. Ví dụ sau minh họa một câu lệnh break :

```

boolean problem = true;
while (true) {
    if (problem) {
        System.out.println("There was a problem");
        break;
    }
}
// next line of code

```

Trong ví dụ trước, câu lệnh break không được gắn nhãn. Sau đây là một ví dụ về câu lệnh continue không được gắn nhãn :

```

while (!EOF) {
    // read a field from a file
    if (wrongField) {
        continue;           // move to the next field in the file
    }
    // otherwise do other stuff with the field
}

```

Trong ví dụ này, một tệp đang được đọc một trường tại một thời điểm. Khi một lỗi là gặp phải, chương trình di chuyển đến trường tiếp theo trong tệp và sử dụng câu lệnh continue để quay lại vòng lặp (nếu nó không nằm ở cuối tệp) và tiếp tục đọc các trường khác nhau. Nếu lệnh break được sử dụng thay thế, mã sẽ ngừng đọc tệp khi lỗi xảy ra và chuyển sang lệnh tiếp theo

dòng mã sau vòng lặp. Câu lệnh continue cung cấp cho bạn một cách để nói, "Việc lặp lại cụ thể này của vòng lặp cần phải dừng lại, nhưng không phải dừng toàn bộ vòng lặp. Tôi chỉ không muốn phần còn lại của mã trong lần lặp này kết thúc, vì vậy hãy thực hiện biểu thức lặp và sau đó bắt đầu lại với bài kiểm tra và đừng lo lắng về những gì bên dưới câu lệnh continue".

Tuyên bố được gắn nhãn

Mặc dù nhiều câu lệnh trong chương trình Java có thể được gắn nhãn, nhưng thông thường nhất là sử dụng nhãn với các câu lệnh lặp như for hoặc while, kết hợp với các câu lệnh break và continue. Câu lệnh nhãn phải được đặt ngay trước câu lệnh được gắn nhãn và nó bao gồm một số nhận dạng hợp lệ kết thúc bằng dấu hai chấm (:).

Bạn cần hiểu sự khác biệt giữa ngắt có gắn nhãn và ngắt không gắn nhãn và tiếp tục. Các giống được gắn nhãn chỉ cần thiết trong các tình huống mà bạn có một vòng lặp lồng nhau và chúng cần cho biết bạn muốn ngắt khỏi vòng lặp lồng nhau nào hoặc từ vòng lặp lồng nhau nào mà bạn muốn tiếp tục với lần lặp tiếp theo. Câu lệnh break sẽ thoát ra khỏi vòng lặp có nhãn, trái ngược với vòng lặp trong cùng, nếu từ khóa break được kết hợp với một nhãn.

Dưới đây là một ví dụ về nhãn trông như thế nào:

```
foo:
    for (int x = 3; x < 20; x++) {
        while(y > 7) {
            y--;
        }
    }
```

Nhãn phải tuân thủ các quy tắc cho một tên biến hợp lệ và phải tuân theo quy ước đặt tên của Java. Cú pháp để sử dụng tên nhãn kết hợp với câu lệnh break là từ khóa break, sau đó là tên nhãn, theo sau là dấu chấm phẩy. Một ví dụ đầy đủ hơn về việc sử dụng câu lệnh break được gắn nhãn như sau:

```

boolean.isTrue = true;
outer:
for(int i=0; i<5; i++) {
    while (isTrue) {
        System.out.println("Hello");
        break outer;
    }      // end of inner while loop
    System.out.println("Outer loop."); // Won't print
}          // end of outer for loop
System.out.println("Good-Bye");

```

Chạy mã này sẽ tạo ra

```

Xin chào
Tạm biệt

```

Trong ví dụ này, từ Xin chào sẽ được in một lần. Sau đó, câu lệnh break được gắn nhãn sẽ được thực thi và luồng sẽ thoát ra khỏi vòng lặp có nhãn bên ngoài. Dòng mã tiếp theo sẽ in Good-Bye.

Hãy xem điều gì sẽ xảy ra nếu câu lệnh continue được sử dụng thay cho câu lệnh break . Ví dụ mã sau đây tương tự như ví dụ trước, ngoại trừ việc thay thế tiếp tục cho ngắt:

```

outer:
for (int i=0; i<5; i++) {
    for (int j=0; j<5; j++) {

        System.out.println("Hello");
        continue outer;
    }      // end of inner loop
    System.out.println("outer"); // Never prints
}
System.out.println("Good-Bye");

```

Chạy mã này sẽ tạo ra

```

Xin chào
Xin chào
Xin chào
Xin chào
Xin chào
Tạm biệt

```

Trong ví dụ này, Hello sẽ được in năm lần. Sau khi câu lệnh continue được thực thi, luồng tiếp tục với lần lặp tiếp theo của vòng lặp được xác định bằng

nhãn. Cuối cùng, khi điều kiện trong vòng lặp ngoài đánh giá là `false`, vòng lặp này sẽ kết thúc và `Good-Bye` sẽ được in.

BÀI TẬP 5-2

Tạo một nhãn trong khi lặp lại

Hãy thử tạo một vòng lặp while được gắn nhãn . Đặt nhãn bên ngoài và cung cấp một điều kiện để kiểm tra xem một biến tuổi có nhỏ hơn hay bằng 21. Trong vòng lặp, hãy tăng tuổi lên 1. Mỗi khi chương trình đi qua vòng lặp, hãy kiểm tra xem tuổi có phải là 16. Nếu có, in thông báo “lấy bằng lái xe của bạn” và tiếp tục đến vòng bên ngoài. Nếu không, hãy in “Một năm nữa”.

- Nhãn bên ngoài sẽ xuất hiện ngay trước khi vòng lặp while bắt đầu.
- Đảm bảo rằng tuổi được khai báo bên ngoài vòng lặp while .



Các câu lệnh `continue` và `break` được gắn nhãn phải nằm trong vòng lặp có cùng tên nhãn; nếu không, mã sẽ không biên dịch.

MỤC TIÊU XÁC NHẬN

Xử lý các trường hợp ngoại lệ (Mục tiêu của OCA 8.1, 8.2, 8.3, 8.4 và 8.5)

- 8.1 Phân biệt giữa các ngoại lệ đã kiểm tra, ngoại lệ chưa kiểm tra và lỗi.
- 8.2 Tạo một khối `try-catch` và xác định cách các ngoại lệ thay đổi luồng chương trình bình thường.
- 8.3 Mô tả những ưu điểm của việc xử lý Ngoại lệ.
- 8.4 Tạo và gọi một phương thức ném một ngoại lệ.
- 8.5 Nhận dạng các lớp ngoại lệ phổ biến (chẳng hạn như `NullPointerException`,

ArithmException, ArrayIndexOutOfBoundsException, ClassCastException) (sic)

Một câu châm ngôn cũ trong phát triển phần mềm nói rằng 80 phần trăm công việc được sử dụng 20 phần trăm thời gian. 80% đề cập đến nỗ lực cần thiết để kiểm tra và xử lý lỗi. Trong nhiều ngôn ngữ, việc viết mã chương trình để kiểm tra và xử lý lỗi thật tệ nhạt và khiến nguồn ứng dụng trở thành món mỳ Ý khó hiểu. Tuy nhiên, phát hiện và xử lý lỗi có thể là thành phần quan trọng nhất của bất kỳ ứng dụng mạnh mẽ nào. Dưới đây là một số lợi ích của các tính năng xử lý ngoại lệ của Java:

- Nó cung cấp cho các nhà phát triển một cơ chế thanh lịch để xử lý lỗi tạo ra mã xử lý lỗi có tổ chức và hiệu quả.
- Nó cho phép các nhà phát triển phát hiện lỗi một cách dễ dàng mà không cần viết mã đặc biệt để kiểm tra các giá trị trả về.
- Nó cho phép chúng tôi giữ mã xử lý ngoại lệ được tách biệt rõ ràng khỏi mã tạo ngoại lệ .
- Nó cũng cho phép chúng tôi sử dụng cùng một mã xử lý ngoại lệ để đối phó với một loạt các ngoại lệ có thể xảy ra.

Java 7 đã thêm một số khả năng xử lý ngoại lệ mới cho ngôn ngữ này. Vì mục đích của chúng tôi, Oracle chia các chủ đề xử lý ngoại lệ khác nhau thành hai phần chính:

1. Kỳ thi OCA bao gồm phiên bản Java 6 về xử lý ngoại lệ.
2. Kỳ thi OCP bổ sung các tính năng ngoại lệ mới được thêm vào trong Java 7.

Để phản ánh các mục tiêu OCA 8 của Oracle so với các mục tiêu OCP 8, chương này sẽ chỉ cung cấp cho bạn những kiến thức cơ bản về xử lý ngoại lệ – nhưng còn rất nhiều điều để xử lý kỳ thi OCA 8.

Bắt một ngoại lệ Sử dụng thử và bắt

Trước khi bắt đầu, chúng ta hãy giới thiệu một số thuật ngữ. Thuật ngữ ngoại lệ có nghĩa là "điều kiện ngoại lệ" và là một sự cố làm thay đổi dòng chương trình bình thường. Một loạt những thứ có thể dẫn đến ngoại lệ, bao gồm lỗi phần cứng, cạn kiệt tài nguyên và các lỗi cũ còn tồn tại. Khi một sự kiện ngoại lệ xảy ra trong Java, một ngoại lệ được cho là "ném". Mã chịu trách nhiệm thực hiện

một cái gì đó về ngoại lệ được gọi là "trình xử lý ngoại lệ" và nó "bắt" ngoại lệ đã ném.

Xử lý ngoại lệ hoạt động bằng cách chuyển việc thực thi một chương trình sang một trình xử lý ngoại lệ thích hợp khi một ngoại lệ xảy ra. Ví dụ: nếu bạn gọi một phương thức mở tệp nhưng tệp không thể mở được, việc thực thi phương thức đó sẽ dừng và mã bạn đã viết để giải quyết tình huống này sẽ được chạy.

Do đó, chúng ta cần một cách để cho JVM biết mã nào sẽ thực thi khi một ngoại lệ nào đó xảy ra. Để làm điều này, chúng tôi sử dụng từ khóa `try` and `catch`. Thủ được sử dụng để xác định một khôi mã trong đó có thể xảy ra ngoại lệ. Khôi mã này được gọi là "vùng được bảo vệ" (điều này thực sự có nghĩa là "mã rủi ro ở đây").

Một hoặc nhiều mệnh đề bắt khớp với một ngoại lệ cụ thể (hoặc một nhóm ngoại lệ—sẽ có thêm điều đó sau này) với một khôi mã xử lý nó. Đây là cách nó trông như thế nào trong mã giả:

```

1. try {
2.   // This is the first line of the "guarded region"
3.   // that is governed by the try keyword.
4.   // Put code here that might cause some kind of exception.
5.   // We may have many code lines here or just one.
6. }
7. catch(MyFirstException) {
8.   // Put code here that handles this exception.
9.   // This is the next line of the exception handler.
10.  // This is the last line of the exception handler.
11. }
12. catch(MySecondException) {
13.   // Put code here that handles this exception
14. }
15.
16. // Some other unguarded (normal, non-risky) code begins here

```

Trong ví dụ về mã giả này, các dòng từ 2 đến 5 tạo thành vùng được bảo vệ được điều chỉnh bởi mệnh đề `try`. Dòng 7 là một trình xử lý ngoại lệ cho một ngoại lệ của kiểu `MyFirstException`. Dòng 12 là một trình xử lý ngoại lệ cho một ngoại lệ của kiểu `MySecondException`. Lưu ý rằng các khôi bắt ngay sau khôi thủ. Đây là một yêu cầu; nếu bạn có một hoặc nhiều khôi bắt, chúng phải ngay lập tức tuân theo khôi thủ. Ngoài ra, tất cả các khôi bắt phải tuân theo nhau, không có bất kỳ câu lệnh hoặc khôi nào khác ở giữa. Ngoài ra, thứ tự mà các khôi bắt xuất hiện cũng quan trọng, như chúng ta sẽ thấy một chút sau.

Việc thực thi vùng được bảo vệ bắt đầu từ dòng 2. Nếu chương trình thực hiện hết dòng 5 mà không có ngoại lệ nào được ném ra, quá trình thực thi sẽ chuyển đến

dòng 15 và tiếp tục đi xuống. Tuy nhiên, nếu bắt kỳ lúc nào trong các dòng từ 2 đến 5 (khối try) một ngoại lệ của loại MyFirstException được ném ra, việc thực thi sẽ ngay lập tức chuyển đến dòng 7. Các dòng 8 đến 10 sau đó sẽ được thực thi để toàn bộ khối bắt chạy, và sau đó việc thực thi sẽ chuyển đến dòng 15 và tiếp tục.

Lưu ý rằng nếu một ngoại lệ xảy ra trên dòng 3 của khối try, các dòng còn lại trong khối try (4 và 5) sẽ không bao giờ được thực thi. Khi điều khiển chuyển đến khối bắt, nó sẽ không bao giờ quay trở lại để hoàn thành số dư của khối thử. Tuy nhiên, đây chính xác là những gì bạn muốn. Hãy tưởng tượng rằng mã của bạn trông giống như mã giả này:

```
try {
    getFileFromOverNetwork
    readFromFileAndPopulateTable
}
catch(CantGetFileFromNetwork) {
    displayNetworkErrorMessage
}
```

Mã giả này thể hiện cách bạn thường làm việc với các ngoại lệ. Mã phụ thuộc vào một hoạt động rủi ro (vì việc điền vào một bảng có dữ liệu tệp phụ thuộc vào việc lấy tệp từ mạng) được nhóm thành một khối thử theo cách mà nếu, ví dụ, hoạt động đầu tiên không thành công, bạn sẽ không tiếp tục cố gắng chạy mã khác cũng được đảm bảo không thành công. Trong ví dụ về mã giả, bạn sẽ không thể đọc từ tệp nếu bạn không thể lấy tệp ra khỏi mạng ngay từ đầu.

Một trong những lợi ích của việc sử dụng xử lý ngoại lệ là mã để xử lý bất kỳ ngoại lệ cụ thể có thể xảy ra trong khu vực được quản lý chỉ cần được viết một lần. Quay trở lại ví dụ mã trước đó của chúng tôi, có thể có ba vị trí khác nhau trong khối try của chúng tôi có thể tạo ra MyFirstException, nhưng bất cứ nơi nào nó xảy ra, nó sẽ được xử lý bởi cùng một khối bắt (trên dòng 7). Chúng ta sẽ thảo luận thêm về lợi ích của việc xử lý ngoại lệ ở gần cuối chương này.

Sử dụng cuối cùng

Mặc dù try and catch cung cấp một cơ chế tuyệt vời để bẫy và xử lý các ngoại lệ, nhưng chúng ta vẫn gặp phải vấn đề là làm thế nào để tự dọn dẹp nếu ngoại lệ xảy ra. Bởi vì việc thực thi chuyển ra khỏi khối try ngay sau khi một ngoại lệ được ném ra, chúng tôi không thể đặt mã dọn dẹp của mình ở cuối khối try và hy vọng nó sẽ được thực thi nếu một ngoại lệ xảy ra. Gần như là một ý tưởng tồi

sẽ đặt mã dọn dẹp của chúng tôi trong mỗi khối bắt – hãy xem lý do tại sao.

Trình xử lý ngoại lệ là nơi không tốt để dọn dẹp sau khi mã trong khối thử bởi vì mỗi trình xử lý sau đó yêu cầu bản sao mã dọn dẹp của chính nó. Ví dụ: nếu bạn đã cấp phát một ổ cắm mạng hoặc mở một tệp ở đâu đó trong vùng được bảo vệ, thì mỗi trình xử lý ngoại lệ sẽ phải đóng tệp hoặc giải phóng ổ cắm. Điều đó sẽ khiến bạn quá dễ quên dọn dẹp và cũng dẫn đến nhiều mã thừa. Để giải quyết vấn đề này, Java cung cấp khối cuối cùng .

Một khối cuối cùng bao gồm mã luôn được thực thi tại một số điểm sau thử khối, cho dù một ngoại lệ có được ném ra hay không. Ngay cả khi có một câu lệnh return trong khối try , khối cuối cùng vẫn thực thi ngay sau khi gặp câu lệnh return và trước khi lệnh trả về thực thi!

Đây là nơi thích hợp để đóng các tệp của bạn, giải phóng các ổ cắm mạng và thực hiện bất kỳ thao tác dọn dẹp nào khác mà mã của bạn yêu cầu. Nếu khối try thực thi không có ngoại lệ, khối cuối cùng sẽ được thực thi ngay sau khi khối try hoàn thành. Nếu có một ngoại lệ được ném ra, khối cuối cùng sẽ thực thi ngay sau khi khối bắt thích hợp hoàn thành. Hãy xem xét một ví dụ mã giả khác:

```

1: try {
2:     // This is the first line of the "guarded region".
3: }
4: catch(MyFirstException) {
5:     // Put code here that handles this exception
6: }
7: catch(MySecondException) {
8:     // Put code here that handles this exception
9: }
10: finally {
11:     // Put code here to release any resource we
12:     // allocated in the try clause
13: }
14:
15: // More code here

```

Như trước đây, việc thực thi bắt đầu ở dòng đầu tiên của khối thử , dòng 2. Nếu có không có ngoại lệ nào được đưa ra trong khối try , việc thực thi chuyển đến dòng 11, dòng đầu tiên của khối cuối cùng . Mặt khác, nếu MySecondException được ném trong khi mã trong khối try đang thực thi, thì việc thực thi sẽ chuyển đến dòng đầu tiên của trình xử lý ngoại lệ đó, dòng 8 trong mệnh đề catch . Sau khi tất cả mã trong mệnh đề bắt được thực thi, chương trình sẽ chuyển đến dòng 11, dòng đầu tiên của mệnh đề cuối cùng . Lặp lại sau khi tôi: cuối cùng luôn luôn chạy! Được rồi, chúng ta sẽ phải

tinh chỉnh điều đó một chút, nhưng bây giờ, hãy bắt đầu nung nấu ý tưởng mà cuối cùng vẫn luôn chạy. Nếu một ngoại lệ được ném ra, cuối cùng sẽ chạy. Nếu một ngoại lệ không được ném ra, cuối cùng sẽ chạy. Nếu trường hợp ngoại lệ bị bắt, cuối cùng sẽ chạy. Nếu ngoại lệ không được bắt, cuối cùng chạy. Sau đó, chúng ta sẽ xem xét một số trường hợp cuối cùng có thể không chạy hoặc không hoàn thành.

Hãy nhớ rằng cuối cùng mệnh đề không bắt buộc. Nếu bạn không viết, mã của bạn sẽ được biên dịch và chạy tốt. Trên thực tế, nếu bạn không có tài nguyên nào để dọn dẹp sau khi khôi thử của bạn hoàn thành, bạn có thể không cần mệnh đề cuối cùng. Ngoài ra, bởi vì trình biên dịch thậm chí không yêu cầu các mệnh đề bắt, đôi khi bạn sẽ chạy qua mã có khôi thử ngay sau đó là khôi cuối cùng. Mã như vậy hữu ích khi ngoại lệ sẽ được chuyển trả lại phương thức gọi, như được giải thích trong phần tiếp theo. Sử dụng khôi cuối cùng cho phép mã dọn dẹp thực thi ngay cả khi không có mệnh đề bắt.

Quy tắc pháp lý sau đây thể hiện một thử nghiệm với kết quả cuối cùng nhưng không có kết quả:

```
try {
    // do stuff
} finally {
    // clean up
}
```

Mã luật sau đây thể hiện một thử, nắm bắt và cuối cùng:

```
try {
    // do stuff
} catch (SomeException ex) {
    // do exception handling
} finally {
    // clean up
}
```

Đoạn mã BẤT HỢP PHÁP sau đây cho thấy một lần thử mà không bắt được hoặc cuối cùng:

```
try {
    // do stuff
}

// need a catch or finally here
System.out.println("out of try block");
```

Mã BẤT HỢP PHÁP sau thể hiện một khôi bắt không đúng vị trí :

```

try {
    // do stuff
}
// can't have code between try/catch
System.out.println("out of try block");
catch(Exception ex) { }

```



Việc sử dụng mệnh đề try mà không có mệnh đề bắt hoặc mệnh đề cuối cùng là bất hợp pháp . Bản thân mệnh đề try sẽ dẫn đến lỗi trình biên dịch. Bất kỳ mệnh đề bắt nào phải ngay sau khối try . Bất kỳ mệnh đề cuối cùng nào phải ngay sau mệnh đề bắt cuối cùng (hoặc nó phải ngay sau khối try nếu không có mệnh đề bắt). Việc bỏ qua mệnh đề bắt hoặc mệnh đề cuối cùng là hợp pháp , nhưng không phải cả hai.

Tuyên truyền các trường hợp ngoại lệ chưa được thông báo

Tại sao mệnh đề bắt không bắt buộc? Điều gì xảy ra với một ngoại lệ được ném trong khối thử khi không có mệnh đề bắt nào đang chờ nó? Trên thực tế, không có yêu cầu nào mà bạn phải viết một mệnh đề bắt cho mọi ngoại lệ có thể được ném ra từ khối thử tương ứng . Trên thực tế, không thể nghĩ ngờ rằng bạn có thể đạt được một ký tự như vậy! Nếu một phương thức không cung cấp mệnh đề bắt cho một ngoại lệ cụ thể, thì phương thức đó được cho là "loại bỏ" ngoại lệ (hoặc "vượt qua buck").

Vì vậy, điều gì xảy ra với một ngoại lệ vịt? Trước khi thảo luận về điều đó, chúng ta cần xem xét ngắn gọn khái niệm về ngăn xếp cuộc gọi. Hầu hết các ngôn ngữ đều có khái niệm về ngăn xếp phương thức hoặc ngăn xếp cuộc gọi. Nói một cách đơn giản, ngăn xếp cuộc gọi là chuỗi các phương thức mà chương trình của bạn thực thi để đến phương thức hiện tại. Nếu chương trình của bạn bắt đầu trong phương thức main () và main () gọi phương thức a (), phương thức này gọi phương thức b (), lần lượt gọi phương thức c (), ngăn xếp cuộc gọi bao gồm:

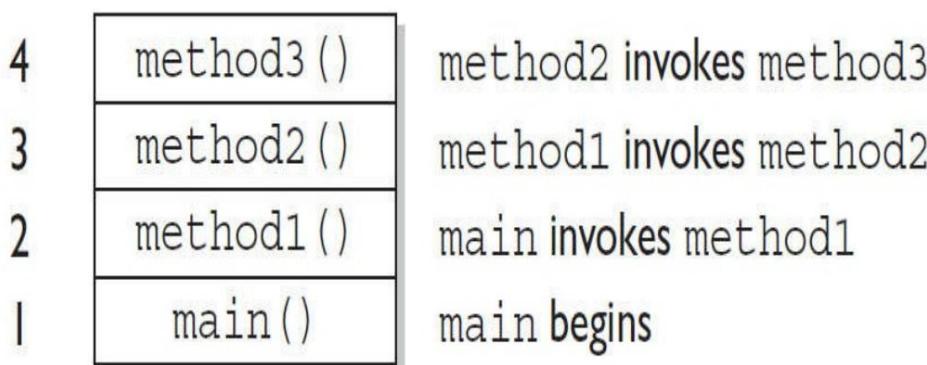
cb

một

chính

Chúng tôi sẽ biểu diễn ngăn xếp đang phát triển lên trên (mặc dù nó cũng có thể được hình dung là đang phát triển xuống dưới). Như bạn có thể thấy, phương thức cuối cùng được gọi nằm ở trên cùng của ngăn xếp, trong khi phương thức gọi đầu tiên nằm ở dưới cùng. Phương thức ở trên cùng của dấu vết ngăn xếp sẽ là phương thức bạn hiện đang thực thi. Nếu chúng ta di chuyển trở lại ngăn xếp cuộc gọi, chúng ta đang chuyển từ phương thức hiện tại sang phương thức đã gọi trước đó. [Hình 5-1](#) minh họa một cách để suy nghĩ về cách hoạt động của ngăn xếp cuộc gọi trong Java.

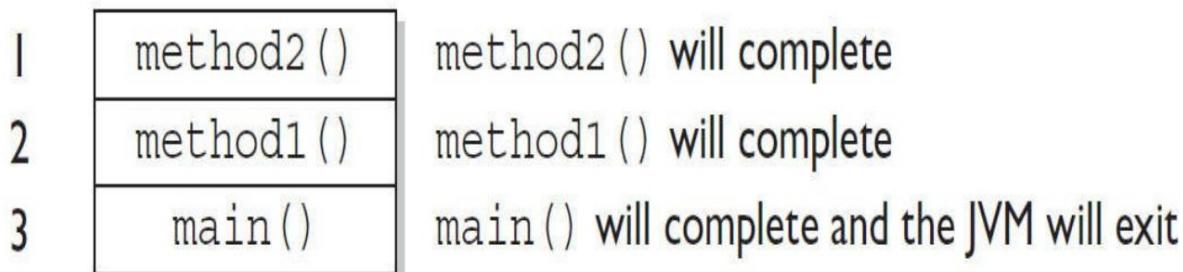
I) The call stack while method3 () is running.



The order in which methods are put on the call stack

2) The call stack after method3 () completes

Execution returns to method2 ()



The order in which methods complete

Bây giờ chúng ta hãy xem xét điều gì sẽ xảy ra với các trường hợp ngoại lệ vịt. Hãy tưởng tượng một tòa nhà, chẳng hạn, cao năm tầng, và ở mỗi tầng đều có một boong hoặc ban công. Bây giờ hãy tưởng tượng rằng trên mỗi boong, một người đang đứng cầm một chiếc gậy đánh bóng chày. Trường hợp ngoại lệ giống như quả bóng rơi từ người này sang người khác, bắt đầu từ mái nhà. Một ngoại lệ đầu tiên được ném từ trên cùng của ngăn xếp (nói cách khác, người trên mái nhà); và nếu nó không bị bắt bởi cùng một người đã ném nó (người trên mái nhà), nó sẽ giảm ngăn xếp cuộc gọi xuống phương thức trước đó, đó là người đứng trên boong cách một tầng. Nếu người đó ở tầng dưới không bắt được ở đó, thì ngoại lệ / bóng lại rơi xuống phương pháp trước đó (người ở tầng tiếp theo xuống), v.v. . Đây được gọi là "sự lan truyền ngoại lệ".

Nếu một ngoại lệ chạm đến cuối xép cuộc gọi, nó giống như chạm đến đáy của một đợt giảm rất dài; quả bóng phát nổ, và chương trình của bạn cũng vậy. Một ngoại lệ không bao giờ bị bắt sẽ khiến ứng dụng của bạn ngừng chạy. Mô tả (nếu có) về ngoại lệ sẽ được hiển thị và ngăn xép cuộc gọi sẽ được "kết xuất". Điều này giúp bạn gỡ lỗi ứng dụng của mình bằng cách cho bạn biết ngoại lệ nào đã được ném ra, nó được ném theo phương pháp nào và ngăn xép trông như thế nào vào thời điểm đó.



Bạn có thể tiếp tục ném một ngoại lệ xuống thông qua các phương thức trên ngăn xếp. Nhưng điều gì sẽ xảy ra khi bạn truy cập vào phương thức `main()` ở dưới cùng? Bạn cũng có thể ném ngoại lệ ra khỏi `main()`. Điều này dẫn đến việc tạm dừng JVM và dấu vết ngắn xếp sẽ được in ra đầu ra. Đoạn mã sau đưa ra một ngoại lệ:

```

class TestEx {
    public static void main (String [] args) {
        doStuff();
    }
    static void doStuff() {
        doMoreStuff();
    }
    static void doMoreStuff() {
        int x = 5/0; // Can't divide by zero!
                    // ArithmeticException is thrown here
    }
}

```

Nó in ra một dấu vết ngăn xếp như thế này:

```

%java TestEx
Exception in thread "main" java.lang.ArithmetricException: / by zero
at TestEx.doMoreStuff(TestEx.java:10)
at TestEx.doStuff(TestEx.java:7)
at TestEx.main(TestEx.java:3)

```

BÀI TẬP 5-3

Tuyên truyền và Bắt một Ngoại lệ Trong bài tập này, bạn

sẽ tạo hai phương pháp xử lý các ngoại lệ.

Một trong các phương thức là phương thức main () , phương thức này sẽ gọi một phương thức khác. Nếu một ngoại lệ được đưa ra trong phương thức kia, hàm main () phải xử lý nó. Một câu lệnh cuối cùng sẽ được đưa vào để cho biết rằng chương trình đã hoàn thành. Phương thức mà main () sẽ gọi sẽ được đặt tên là đảo ngược và nó sẽ đảo ngược thứ tự của các ký tự trong một chuỗi. Nếu Chuỗi không chứa ký tự nào, phép đảo ngược sẽ truyền một ngoại lệ cho đến phương thức main () .

1. Tạo một lớp có tên là Tuyên truyền và một phương thức main () , sẽ vẫn trống cho đến thời điểm hiện tại.
2. Tạo một phương thức được gọi là đảo ngược. Nó có một đối số là một Chuỗi và trả về một chuỗi.
3. Ngược lại, hãy kiểm tra xem Chuỗi có độ dài bằng 0 hay không bằng cách sử dụng phương thức String.length () . Nếu độ dài là 0, phương thức đảo ngược sẽ ném ra một ngoại lệ.

4. Bây giờ hãy bao gồm mã để đảo ngược thứ tự của Chuỗi. Vì đây không phải là chủ đề chính của chương này, nên mã đảo ngược đã được cung cấp, nhưng hãy tự mình thử.

```
String reverseStr = "";
for(int i=s.length()-1;i>=0;--i) {
    reverseStr += s.charAt(i);
}
return reverseStr;
```

5. Bây giờ trong phương thức main () , bạn sẽ cố gắng gọi phương thức này và xử lý bất kỳ trường hợp ngoại lệ tiềm năng nào. Ngoài ra, bạn sẽ bao gồm một câu lệnh cuối cùng hiển thị khi hàm main () kết thúc.

Xác định ngoại lệ

Chúng tôi đã thảo luận về các ngoại lệ như một khái niệm. Chúng tôi biết rằng chúng được ném ra khi một vấn đề nào đó xảy ra và chúng tôi biết chúng có ảnh hưởng gì đến luồng chương trình của chúng tôi. Trong phần này, chúng tôi sẽ phát triển thêm các khái niệm và sử dụng các ngoại lệ trong mã Java chức năng.

Trước đó, chúng tôi đã nói rằng một ngoại lệ là một sự xuất hiện làm thay đổi luồng chương trình bình thường. Nhưng bởi vì đây là Java, bắt cứ thứ gì không phải là nguyên thủy đều phải là một đối tượng. Các trường hợp ngoại lệ không có gì khác biệt. Mỗi ngoại lệ là một thể hiện của một lớp có lớp Ngoại lệ trong hệ thống phân cấp kế thừa của nó. Nói cách khác, ngoại lệ luôn là một số lớp con của java.lang.Exception.

Khi một ngoại lệ được ném ra, một đối tượng của một kiểu con Ngoại lệ cụ thể sẽ được khởi tạo và chuyển đến trình xử lý ngoại lệ như một đối số cho mệnh đề bắt . Một mệnh đề bắt thực tế trông giống như sau:

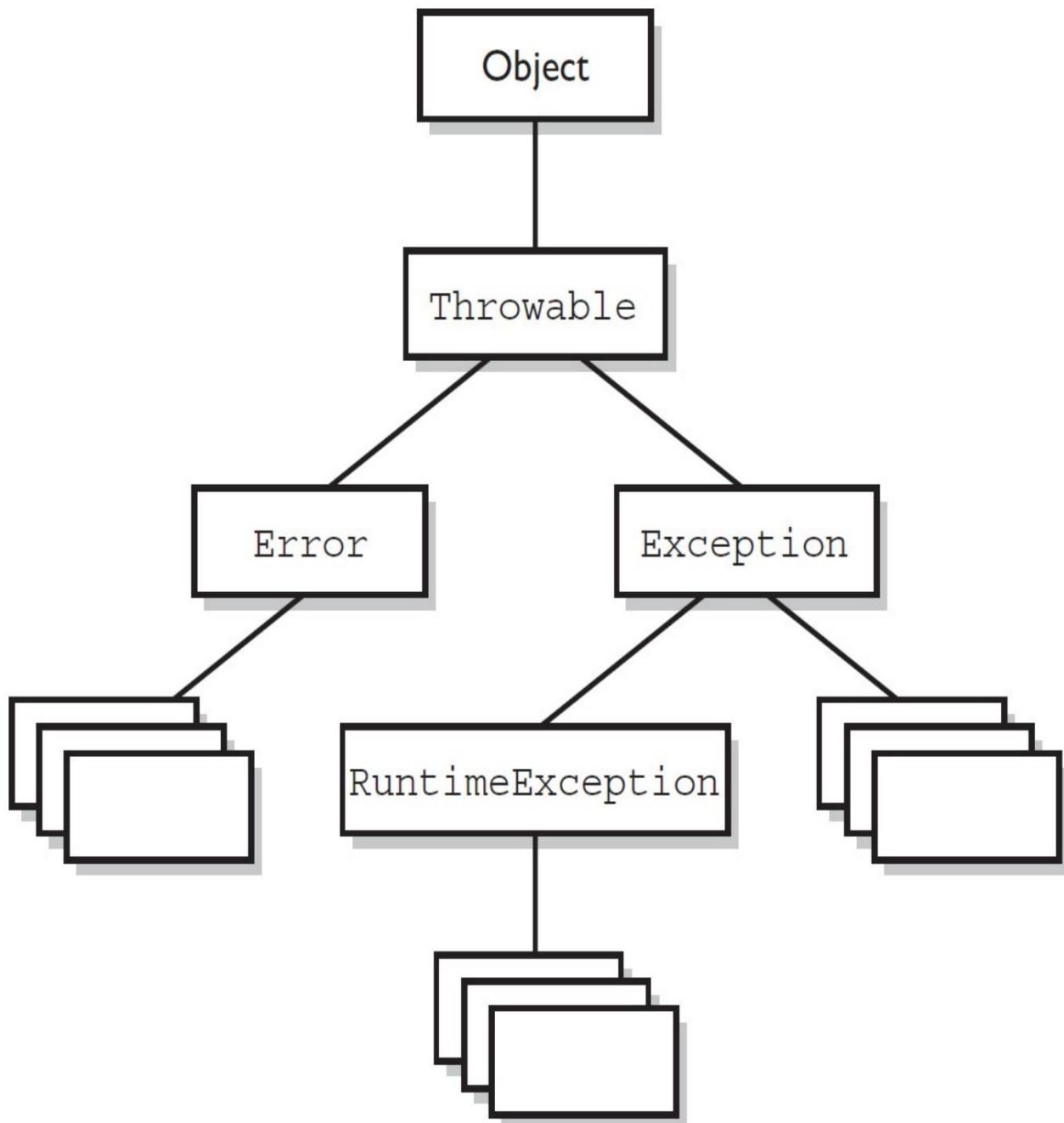
```
try {
    // some code here
}
catch (ArrayIndexOutOfBoundsException e) {
    e.printStackTrace();
}
```

Trong ví dụ này, e là một thể hiện của lớp ArrayIndexOutOfBoundsException .

Như với bất kỳ đối tượng nào khác, bạn có thể gọi các phương thức của nó.

Hệ thống phân cấp ngoại lệ

Tất cả các lớp ngoại lệ là kiểu con của lớp Ngoại lệ. Lớp này bắt nguồn từ lớp Throwable (dẫn xuất từ lớp Object). [Hình 5-2](#) cho thấy cấu trúc phân cấp cho các lớp ngoại lệ.



HÌNH 5-2 cấp lớp ngoại lệ

Như bạn có thể thấy, có hai lớp con bắt nguồn từ **Throwable**: **Exception** và **Error**. Các lớp bắt nguồn từ Lỗi đại diện cho bất thường

các tình huống không phải do lỗi chương trình gây ra và chỉ ra những điều thường không xảy ra trong quá trình thực thi chương trình, chẳng hạn như JVM hết bộ nhớ. Nói chung, ứng dụng của bạn sẽ không thể khôi phục sau Lỗi, vì vậy bạn không cần phải xử lý chúng. Nếu mã của bạn không xử lý chúng (và thường là không), nó sẽ vẫn biên dịch mà không gặp khó khăn gì. Mặc dù thường được coi là các điều kiện ngoại lệ, nhưng về mặt kỹ thuật, lỗi không phải là trường hợp ngoại lệ vì chúng không xuất phát từ ngoại lệ của lớp.

Nói chung, một ngoại lệ đại diện cho điều gì đó xảy ra không phải do lỗi lập trình, mà là do một số tài nguyên không có sẵn hoặc một số điều kiện cần thiết khác để thực hiện đúng không có. Ví dụ: nếu ứng dụng của bạn được cho là giao tiếp với một ứng dụng hoặc máy tính khác không trả lời, thì đây là một ngoại lệ không phải do lỗi.

[Hình 5-2](#) cũng cho thấy một kiểu con của Ngoại lệ được gọi là RuntimeException. Những ngoại lệ này là một trường hợp đặc biệt vì chúng đôi khi chỉ ra lỗi chương trình. Chúng cũng có thể đại diện cho các điều kiện ngoại lệ hiếm gặp, khó xử lý. Các ngoại lệ thời gian chạy sẽ được thảo luận chi tiết hơn ở phần sau của chương này.

Java cung cấp nhiều lớp ngoại lệ, hầu hết chúng đều có tên khá mô tả. Có hai cách để lấy thông tin về một trường hợp ngoại lệ. Đầu tiên là từ chính loại ngoại lệ. Tiếp theo là từ thông tin mà bạn có thể nhận được từ đối tượng ngoại lệ. Class Throwable (ở trên cùng của cây kế thừa đối với các ngoại lệ) cung cấp cho con cháu của nó một số phương thức hữu ích trong các trình xử lý ngoại lệ. Một trong số này là printStackTrace (). Như bạn mong đợi, nếu bạn gọi phương thức printStackTrace () của đối tượng ngoại lệ, như trong ví dụ trước đó, một dấu vết ngăn xếp từ nơi ngoại lệ xảy ra sẽ được in.

Chúng tôi đã thảo luận rằng một ngăn xếp cuộc gọi xâm nhập trở lên với phương thức được gọi gần đây nhất ở trên cùng. Bạn sẽ nhận thấy rằng phương thức printStackTrace () in phương thức được nhập gần đây nhất trước tiên và tiếp tục đi xuống, in tên của mỗi phương thức khi nó hoạt động theo cách của nó xuống ngăn xếp cuộc gọi (điều này được gọi là "giải nén ngăn xếp") từ trên cùng.



Đối với bài kiểm tra, bạn không cần biết bất kỳ phương thức nào có trong các lớp Có thể ném, bao gồm Ngoại lệ và Lỗi. Bạn phải biết rằng các loại Exception, Error, RuntimeException và Throwable đều có thể được ném bằng cách sử dụng từ khóa throw và tất cả đều có thể bị bắt

(mặc dù bạn hiếm khi bắt gặp bắt cứ thứ gì khác ngoài các kiểu con Exception).

Xử lý toàn bộ phân cấp lớp ngoại lệ

Chúng tôi đã thảo luận rằng từ khóa catch cho phép bạn chỉ định một loại ngoại lệ cụ thể để bắt. Bạn thực sự có thể bắt nhiều hơn một loại ngoại lệ trong một mệnh đề bắt duy nhất. Nếu lớp ngoại lệ mà bạn chỉ định trong mệnh đề bắt không có lớp con, thì chỉ lớp ngoại lệ được chỉ định mới được bắt.

Tuy nhiên, nếu lớp được chỉ định trong mệnh đề catch có các lớp con, thì bất kỳ đối tượng ngoại lệ nào phân lớp con của lớp đã chỉ định cũng sẽ bị bắt.

Ví dụ, lớp IndexOutOfBoundsException có hai lớp con, ArrayIndexOutOfBoundsException và StringIndexOutOfBoundsException. Bạn có thể muốn viết một trình xử lý ngoại lệ để cập nhật đến các ngoại lệ do một trong hai lỗi biên tạo ra, nhưng bạn có thể không quan tâm đến ngoại lệ nào mà bạn thực sự có. Trong trường hợp này, bạn có thể viết một mệnh đề bắt như sau:

```
try {
    // Some code here that can throw a boundary exception
}
catch (IndexOutOfBoundsException e) {
    e.printStackTrace();
}
```

Nếu bắt kỳ mã nào trong khối try ném ArrayIndexOutOfBoundsException hoặc StringIndexOutOfBoundsException, ngoại lệ sẽ được bắt và xử lý.

Điều này có thể thuận tiện, nhưng nó nên được sử dụng một cách tiết kiệm.

Bằng cách chỉ định lớp cha của lớp ngoại lệ trong mệnh đề bắt của bạn, bạn đang loại bỏ thông tin có giá trị về ngoại lệ. Tất nhiên, bạn có thể tìm ra chính xác lớp ngoại lệ mà bạn có, nhưng nếu bạn định làm điều đó, tốt hơn hết bạn nên viết một mệnh đề bắt riêng cho từng loại sở thích ngoại lệ.



Hãy chông lại sự cám dỗ để viết một trình xử lý ngoại lệ catchall như sau:

```
try {
    // some code
}
catch (Exception e) {
    e.printStackTrace();
}
```

Mã này sẽ bắt mọi ngoại lệ được tạo. Tuy nhiên, không có trình xử lý ngoại lệ nào có thể xử lý đúng mọi ngoại lệ và lập trình theo cách này sẽ đánh bại mục tiêu thiết kế. Các trình xử lý ngoại lệ mắc nhiều lỗi cùng một lúc có thể sẽ làm giảm độ tin cậy của chương trình của bạn, vì có khả năng sẽ xảy ra một ngoại lệ mà trình xử lý không biết cách xử lý.

Đối sánh ngoại lệ

Nếu bạn có

hệ thống phân cấp ngoại lệ bao gồm ngoại lệ lớp cha và một số kiểu con và bạn quan tâm đến việc xử lý một trong các kiểu con theo cách đặc biệt nhưng muốn xử lý tất cả các phần còn lại cùng nhau, bạn chỉ cần viết hai mệnh đề bắt .

Khi một ngoại lệ được ném ra, Java sẽ cố gắng tìm (bằng cách xem mệnh đề bắt từ trên xuống) mệnh đề bắt đối với loại ngoại lệ. Nếu nó không tìm thấy một, nó sẽ tìm kiếm một trình xử lý cho một siêu kiểu ngoại lệ. Nếu nó không tìm thấy mệnh đề bắt phù hợp với siêu kiểu cho ngoại lệ, thì ngoại lệ được truyền xuống ngăn xếp cuộc gọi. Quá trình này được gọi là "đối sánh ngoại lệ". Hãy xem một ví dụ.

```

1: import java.io.*;
2: public class ReadData {
3:     public static void main(String args[]) {
4:         try {
5:             RandomAccessFile raf =
6:                 new RandomAccessFile("myfile.txt", "r");
7:             byte b[] = new byte[1000];
8:             raf.readFully(b, 0, 1000);
9:         }
10:        catch(FileNotFoundException e) {
11:            System.err.println("File not found");
12:            System.err.println(e.getMessage());
13:            e.printStackTrace();
14:        }
15:        catch(IOException e) {
16:            System.err.println("IO Error");
17:            System.err.println(e.toString());
18:            e.printStackTrace();
19:        }
20:    }
21: }

```

Chương trình ngắn này cố gắng mở một tệp và đọc một số dữ liệu từ nó.

Việc mở và đọc tệp có thể tạo ra nhiều ngoại lệ, hầu hết trong số đó là một số loại IOException. Hãy tưởng tượng rằng trong chương trình này, chúng ta chỉ muốn biết liệu ngoại lệ chính xác có phải là FileNotFoundException hay không.

Nếu không, chúng tôi không quan tâm chính xác vấn đề là gì.

FileNotFoundException là một lớp con của IOException. Do đó, chúng ta có thể xử lý nó trong mệnh đề catch bắt tất cả các kiểu con của IOException, nhưng sau đó chúng ta sẽ phải kiểm tra ngoại lệ để xác định xem nó có phải là FileNotFoundException hay không. Thay vào đó, chúng tôi đã viết mã một trình xử lý ngoại lệ đặc biệt cho FileNotFoundException và một trình xử lý ngoại lệ riêng biệt cho tất cả các kiểu con IOException khác.

Nếu mã này tạo ra một FileNotFoundException, nó sẽ được xử lý bởi mệnh đề catch bắt đầu ở dòng 10. Nếu nó tạo ra một IOException khác - có lẽ là EOFException, là một lớp con của IOException - nó sẽ được xử lý bởi mệnh đề catch bắt đầu ở dòng 15. Nếu một số ngoại lệ khác được tạo ra, chẳng hạn như ngoại lệ thời gian chạy thuộc một số loại, không mệnh đề bắt nào sẽ được thực thi và ngoại lệ sẽ được truyền xuống ngăn xếp cuộc gọi.

Lưu ý rằng mệnh đề bắt cho FileNotFoundException được đặt phía trên trình xử lý cho IOException. Điều này thực sự quan trọng! Nếu chúng ta làm theo cách ngược lại, chương trình sẽ không biên dịch. Các trình xử lý cụ thể nhất

các ngoại lệ phải luôn được đặt trên các ngoại lệ đó để có các ngoại lệ chung hơn. Phần sau sẽ không biên dịch:

```
try {
    // do risky IO things
} catch (IOException e) {
    // handle general IOExceptions
} catch (FileNotFoundException ex) {
    // handle just FileNotFoundException
}
```

Bạn sẽ gặp lỗi trình biên dịch như sau:

```
TestEx.java:15: exception java.io.FileNotFoundException has
already been caught
} catch (FileNotFoundException ex) {
^
```

Nếu bạn nghĩ lại về những người có găng tay bóng chày (trong phần "Tuyên truyền các trường hợp ngoại lệ chưa biết"), hãy tưởng tượng rằng những người đeo găng tay chung nhất là lớn nhất và do đó có thể bắt được nhiều loại bóng. Một IOException mitt đủ lớn và đủ linh hoạt để bắt bất kỳ loại IOException nào. Vì vậy, nếu người ở tầng năm (giả sử, Fred) có mitt ' IOException lớn , anh ta không thể không bắt một quả bóng FileNotFoundException với nó. Và nếu anh chàng (giả sử Jimmy) ở tầng hai đang giữ một quả cầu FileNotFoundException , quả bóng FileNotFoundException đó sẽ không bao giờ đến được với anh ta vì nó sẽ luôn bị Fred chặn lại trên tầng năm, đứng đó với cái-đủ-lớn của anh ta bắt kỳ IOException mitt nào.

Vậy bạn sẽ làm gì với những trường hợp ngoại lệ là anh chị em trong hệ thống phân cấp lớp? Nếu một lớp Ngoại lệ không phải là kiểu con hoặc kiểu siêu của lớp kia, thì thứ tự đặt các mệnh đề bắt không quan trọng.

Tuyên bố ngoại lệ và giao diện công khai

Vì vậy, làm thế nào để chúng ta biết rằng một số phương thức ném ra một ngoại lệ mà chúng ta phải bắt? Cũng giống như một phương thức phải chỉ định kiểu và bao nhiêu đối số mà nó chấp nhận và những gì được trả về, các ngoại lệ mà một phương thức có thể ném phải được khai báo (trừ khi các ngoại lệ là các lớp con của RuntimeException). Danh sách các ngoại lệ được ném là một phần của giao diện công khai của phương thức. Từ khóa throws được sử dụng như sau để liệt kê các ngoại lệ mà một phương thức có thể ném ra:

```
void myFunction() throws MyException1, MyException2 {
    // code for the method here
}
```

Phương thức này có kiểu trả về void , không chấp nhận đối số và tuyên bố rằng nó có thể ném một trong hai kiểu ngoại lệ: kiểu MyException1 hoặc kiểu MyException2. (Chỉ vì phương thức tuyên bố rằng nó ném ra một ngoại lệ không có nghĩa là nó luôn luôn như vậy. Nó chỉ cho cả thế giới biết rằng nó có thể xảy ra.)

Giả sử phương thức của bạn không trực tiếp ném ra một ngoại lệ nhưng gọi một phương thức. Bạn có thể chọn không tự xử lý ngoại lệ và thay vào đó chỉ cần khai báo nó, như thể phương thức của bạn thực sự ném ngoại lệ. Nếu bạn khai báo ngoại lệ mà phương thức của bạn có thể nhận được từ một phương thức khác và bạn không cung cấp thử / bắt cho nó, thì phương thức sẽ truyền ngược trở lại phương thức đã gọi phương thức của bạn và sẽ bị bắt ở đó hoặc tiếp tục được xử lý bởi một phương pháp sâu hơn trong ngăn xếp.

Bất kỳ phương thức nào có thể ném một ngoại lệ (trừ khi đó là một lớp con của RuntimeException) phải khai báo ngoại lệ. Điều đó bao gồm các phương thức không thực sự ném nó trực tiếp, nhưng đang "dìm" và để ngoại lệ chuyển xuống phương thức tiếp theo trong ngăn xếp. Nếu bạn "đặt" một ngoại lệ, nó giống như thể bạn là người thực sự ném ra ngoại lệ. Các lớp con RuntimeException được miễn trừ, vì vậy trình biên dịch sẽ không kiểm tra xem bạn đã khai báo chúng hay chưa. Nhưng tất cả các ngoại lệ không phải RuntimeExceptions đều được coi là ngoại lệ "đã kiểm tra" vì trình biên dịch kiểm tra để chắc chắn rằng bạn đã thừa nhận rằng "những điều tồi tệ có thể xảy ra ở đây."

Nhớ điều này:

Mỗi phương thức phải xử lý tất cả các ngoại lệ đã kiểm tra bằng cách cung cấp một mệnh đề bắt hoặc liệt kê mỗi ngoại lệ đã kiểm tra chưa xử lý như một ngoại lệ được ném.

Quy tắc này được gọi là yêu cầu "xử lý hoặc khai báo" của Java (đôi khi được gọi là "bắt hoặc khai báo").



Tìm mã gọi một phương thức khai báo ngoại lệ, nơi phương thức gọi không xử lý hoặc khai báo ngoại lệ đã kiểm tra. Đoạn mã sau (sử dụng từ khóa ném để đưa ra một ngoại lệ theo cách thủ công - thêm vào đoạn mã này tiếp theo) có hai vấn đề lớn mà trình biên dịch sẽ ngăn chặn:

```

void doStuff() {
    doMore();
}
void doMore() {
    throw new IOException();
}

```

Đầu tiên, phương thức doMore () ném ra một ngoại lệ đã được kiểm tra nhưng không khai báo nó! Nhưng giả sử chúng ta sửa phương thức doMore () như sau:

```
void doMore () ném IOException {.}
```

Phương thức doStuff () vẫn gặp sự cố vì nó cũng phải khai báo IOException, trừ khi nó xử lý nó bằng cách cung cấp try / catch, với mệnh đề bắt có thể sử dụng IOException .

Một lần nữa, một số trường hợp ngoại lệ được miễn trừ khỏi quy tắc này. Một đối tượng kiểu RuntimeException có thể được ném từ bất kỳ phương thức nào mà không được chỉ định như một phần của giao diện công khai của phương thức (và không cần phải có trình xử lý). Và ngay cả khi một phương thức khai báo một RuntimeException, thì phương thức gọi không có nghĩa vụ phải xử lý hoặc khai báo nó. RuntimeException, Error và tất cả các kiểu con của chúng là các ngoại lệ không được kiểm tra và các ngoại lệ không được kiểm tra không cần phải được chỉ định hoặc xử lý. Đây là một ví dụ:

```

import java.io.*;
class Test {
    public int myMethod1() throws EOFException {
        return myMethod2();
    }
    public int myMethod2() throws EOFException {
        // code that actually could throw the exception goes here
        return 1;
    }
}

```

Hãy nhìn vào myMethod1 (). Bởi vì EOFException lớp con IOException và lớp con IOException Exception, nó là một ngoại lệ đã được kiểm tra và phải được khai báo là một ngoại lệ có thể được đưa ra bởi phương thức này. Nhưng ngoại lệ thực sự sẽ đến từ đâu? Giao diện công khai cho phương thức myMethod2 () được gọi ở đây tuyên bố rằng một ngoại lệ của kiểu này có thể được ném ra. Cho dù phương thức đó thực sự ném ngoại lệ hay gọi một phương thức khác ném nó đi thì không quan trọng đối với chúng tôi; chúng tôi chỉ đơn giản biết rằng chúng tôi phải bắt được ngoại lệ

hoặc tuyên bố rằng chúng tôi đã ném nó. Phương thức myMethod1 () không bắt ngoại lệ, vì vậy nó tuyên bố rằng nó ném nó. Vậy giờ chúng ta hãy xem xét một ví dụ pháp lý khác, myMethod3 ():

```
public void myMethod3() {
    // code that could throw a NullPointerException goes here
}
```

Theo nhận xét, phương thức này có thể ném một NullPointerException.

Vì RuntimeException là lớp cha của NullPointerException, nó là một ngoại lệ không được kiểm tra và không cần phải khai báo. Chúng ta có thể thấy rằng myMethod3 () không khai báo bất kỳ ngoại lệ nào.

Các ngoại lệ thời gian chạy được gọi là ngoại lệ không được kiểm tra. Tất cả những thứ khác exceptions là các ngoại lệ đã được kiểm tra và chúng không bắt nguồn từ java.lang.RuntimeException. Một ngoại lệ đã kiểm tra phải được bắt ở đâu đó trong mã của bạn. Nếu bạn gọi một phương thức ném ngoại lệ đã kiểm tra nhưng bạn không bắt được ngoại lệ đã kiểm tra ở đâu đó, thì mã của bạn sẽ không được biên dịch.

Đó là lý do tại sao chúng được gọi là ngoại lệ đã kiểm tra: trình biên dịch kiểm tra để đảm bảo rằng chúng được xử lý hoặc khai báo. Một số phương thức trong Java API ném ra các ngoại lệ đã được kiểm tra, vì vậy bạn thường sẽ viết các trình xử lý ngoại lệ để đối phó với các ngoại lệ được tạo bởi các phương thức bạn không viết.

Bạn cũng có thể tự mình đưa ra một ngoại lệ và ngoại lệ đó có thể là một ngoại lệ hiện có từ API Java hoặc một trong những ngoại lệ của riêng bạn. Để tạo ngoại lệ của riêng bạn, bạn chỉ cần lớp con Exception (hoặc một trong các lớp con của nó) như sau:

```
lớp MyException mở rộng ngoại lệ {}
```

Và nếu bạn ném ngoại lệ, trình biên dịch sẽ đảm bảo rằng bạn khai báo nó như sau:

```
class TestEx {
    void doStuff() {
        throw new MyException(); // Throw a checked exception
    }
}
```

Đoạn mã trước sẽ đảo lộn trình biên dịch:

```
TestEx.java:6: unreported exception MyException; must be caught or
declared to be thrown
    throw new MyException();
^
```



Khi một đối tượng của kiểu ngoại lệ được ném ra, nó phải được xử lý hoặc khai báo. Những đối tượng này được gọi là "ngoại lệ đã kiểm tra" và bao gồm tất cả các ngoại lệ ngoại trừ những ngoại lệ là kiểu con của RuntimeException, là những ngoại lệ không được kiểm tra. Hãy sẵn sàng phát hiện các phương thức không tuân theo quy tắc "xử lý hoặc khai báo", chẳng hạn như sau:

```
class MyException extends Exception {
    void someMethod () {
        doStuff();
    }
    void doStuff() throws MyException {
        try {
            throw new MyException();
        }
        catch(MyException me) {
            throw me;
        }
    }
}
```

Bạn cần biết rằng mã này sẽ không được biên dịch. Nếu bạn cố gắng, bạn sẽ nhận được điều này:

```
MyException.java:3: unreported exception MyException;
must be caught or declared to be thrown
doStuff();
^
```

Lưu ý rằng someMethod () không thể xử lý hoặc khai báo ngoại lệ có thể được ném bởi doStuff (). Trong các trang tiếp theo, chúng ta sẽ thảo luận một số cách để đối phó với loại tình huống này.

Bạn cần biết Lỗi so sánh với các trường hợp ngoại lệ được chọn và không được chọn như thế nào. Các đối tượng thuộc loại Lỗi không phải là đối tượng Ngoại lệ , mặc dù chúng đại diện cho các điều kiện ngoại lệ. Cả Exception và Error đều có chung một lớp cha, Throwable; do đó, cả hai đều có thể được ném bằng cách sử dụng từ khóa ném . Khi một Lỗi hoặc một lớp con của Lỗi (như StackOverflowError) được ném ra, nó sẽ không được chọn. Bạn không bắt buộc phải bắt các đối tượng Lỗi hoặc kiểu con Lỗi .Bạn

cũng có thể tự mình tạo ra một Lỗi (mặc dù, ngoài AssertionError, bạn có thể sẽ không bao giờ muôn) và bạn có thể bắt gặp một lỗi, nhưng một lần nữa, bạn có thể sẽ không. Ví dụ, bạn sẽ thực sự làm gì nếu gặp lỗi OutOfMemoryError? Nó không giống như bạn có thể yêu cầu người thu gom rác chạy; bạn có thể đặt cược rằng JVM đã chiến đấu trong tuyệt vọng để tự cứu (và lấy lại tất cả bộ nhớ có thể) vào thời điểm bạn gặp lỗi. Nói cách khác, đừng mong đợi JVM tại thời điểm đó sẽ nói, "Chạy bộ thu gom rác? Ô, cảm ơn rất nhiều vì đã nói với tôi.

Điều đó không bao giờ xảy ra với tôi. Chắc chắn, tôi sẽ bắt tay ngay vào việc đó. " Thật chí tốt hơn, bạn sẽ làm gì nếu một VirtualMachineError xuất hiện? Chương trình của bạn sẽ nâng cốc vào thời điểm bạn bắt lỗi, vì vậy thực sự chẳng ích gì khi cố gắng bắt một trong những đứa trẻ này. Tuy nhiên, chỉ cần nhớ rằng bạn có thể! Các biên dịch sau đây chỉ tốt:

```
class TestEx {
    public static void main (String [] args) {
        badMethod();
    }
    static void badMethod() { // No need to declare an Error
        doStuff();
    }
    static void doStuff() { // No need to declare an Error
        try {
            throw new Error();
        }
        catch(Error me) {
            throw me; // We catch it, but then rethrow it
        }
    }
}
```

Nếu chúng ta đang ném một ngoại lệ đã được kiểm tra chứ không phải là Lỗi, thì phương thức doStuff () sẽ cần phải khai báo ngoại lệ. Nhưng hãy nhớ rằng, vì Lỗi không phải là một kiểu con của Ngoại lệ, nên nó không cần phải được khai báo. Bạn có thể tự do khai báo nó nếu bạn muốn, nhưng trình biên dịch chỉ không quan tâm theo cách này hay cách khác khi nào hoặc cách nào Lỗi được ném ra hoặc bởi ai.



Bởi vì Java đã kiểm tra các ngoại lệ, người ta thường nói rằng Java buộc các nhà phát triển phải xử lý các ngoại lệ. Có, Java buộc chúng ta phải viết các trình xử lý ngoại lệ cho mỗi ngoại lệ có thể xảy ra trong quá trình hoạt động bình thường, nhưng nó

tùy thuộc vào chúng tôi để làm cho các trình xử lý ngoại lệ thực sự làm điều gì đó hữu ích. Chúng tôi biết các nhà quản lý phần mềm đã tan chảy khi họ thấy một lập trình viên viết một cái gì đó như thế này:

```
try {
    callBadMethod();
} catch (Exception ex) { }
```

Nhận thấy điều gì còn thiếu? Đừng “ăn” ngoại lệ bằng cách bắt nó mà không thực sự xử lý nó. Bạn thậm chí sẽ không thể biết rằng ngoại lệ đã xảy ra bởi vì bạn sẽ không bao giờ nhìn thấy dấu vết ngăn xếp.

Rethrowing cùng một trường hợp ngoại lệ

Cũng giống như bạn có thể ném một ngoại lệ mới từ một mệnh đề bắt , bạn cũng có thể ném cùng một ngoại lệ mà bạn vừa bắt được. Đây là một mệnh đề nắm bắt thực hiện điều này:

```
catch(IOException e) {
    // Do things, then if you decide you can't handle it...
    throw e;
}
```

Tất cả các mệnh đề bắt khác được liên kết với cùng một lần thử đều bị bỏ qua; nếu một khôi cuối cùng tồn tại, nó sẽ chạy và ngoại lệ được ném trở lại phương thức gọi (phương thức tiếp theo trong ngăn xếp cuộc gọi). Nếu bạn ném một ngoại lệ đã kiểm tra từ một mệnh đề bắt , bạn cũng phải khai báo ngoại lệ đó! Nói cách khác, bạn phải xử lý và khai báo, trái ngược với xử lý hoặc khai báo. Ví dụ sau là bắt hợp pháp:

```
public void doStuff() {
    try {
        // risky IO things
    } catch(IOException ex) {
        // can't handle it
        throw ex; // Can't throw it unless you declare it
    }
}
```

Trong đoạn mã trước, phương thức doStuff () rõ ràng có thể đưa ra một ngoại lệ đã được kiểm tra – trong trường hợp này là IOException – vì vậy trình biên dịch nói, “Chà, đó chỉ là quả đào mà bạn có thể thử / nắm bắt ở đó, nhưng nó không tốt đầy đủ. Nếu bạn có thể ném lại IOException mà bạn bắt được, thì bạn phải khai báo nó (trong

chữ ký phương thức)! "

BÀI TẬP 5-4

Tạo ngoại lệ

Trong bài tập này, chúng tôi cố gắng tạo một ngoại lệ tùy chỉnh. Chúng tôi sẽ không đưa vào bất kỳ phương thức mới nào (nó sẽ chỉ có những phương thức được kế thừa từ `Exception`); và bởi vì nó mở rộng `Exception`, trình biên dịch coi nó là một ngoại lệ đã được kiểm tra. Mục tiêu của chương trình là xác định xem một đối số dòng lệnh đại diện cho một loại thực phẩm cụ thể (dưới dạng một chuỗi) được coi là xấu hay ổn.

1. Đầu tiên chúng ta hãy tạo ngoại lệ của chúng ta. Chúng tôi sẽ gọi nó là `BadFoodException`. Ngoại lệ này sẽ được ném ra khi gặp thức ăn không ngon.
2. Tạo một lớp bao bọc được gọi là `MyException` và một phương thức `main()`, sẽ vẫn trống cho bây giờ.
3. Tạo một phương thức có tên là `checkFood()`. Nó cần một đối số Chuỗi và ném ngoại lệ của chúng tôi nếu nó không thích thức ăn mà nó được đưa. Nếu không, nó cho chúng ta biết nó thích đồ ăn. Bạn có thể thêm bất kỳ loại thực phẩm nào mà bạn không đặc biệt yêu thích vào danh sách.
4. Bây giờ trong phương thức `main()`, bạn sẽ lấy đối số dòng lệnh ra khỏi mảng Chuỗi và sau đó chuyển Chuỗi đó vào phương thức `checkFood()`. Bởi vì nó là một ngoại lệ đã được kiểm tra, phương thức `checkFood()` phải khai báo nó và phương thức `main()` phải xử lý nó (sử dụng `try / catch`). Đừng khai báo `main()` ngoại lệ, vì nếu `main()` khai báo ngoại lệ, thì còn ai quay lại đó để bắt nó? (Trên thực tế, `main()` có thể khai báo các ngoại lệ một cách hợp pháp, nhưng đừng làm điều đó trong bài tập này.)

Tiện lợi như xử lý ngoại lệ, vẫn còn tùy thuộc vào nhà phát triển để thực hiện sử dụng nó. Xử lý ngoại lệ làm cho việc tổ chức các vấn đề về mã và báo hiệu trở nên dễ dàng, nhưng các trình xử lý ngoại lệ vẫn phải được viết. Bạn sẽ thấy rằng ngay cả những tình huống phức tạp nhất cũng có thể được xử lý và mã của bạn sẽ có thể tái sử dụng, có thể đọc được và có thể bảo trì.

MỤC TIÊU XÁC NHẬN

Các ngoại lệ và lỗi phổ biến (Mục tiêu 8.5 của OCA)

8.5 Nhận dạng các lớp ngoại lệ phổ biến (chẳng hạn như NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException) (sic)

Mục đích của mục tiêu này là để đảm bảo rằng bạn đã quen thuộc với một số trong số các ngoại lệ và lỗi phổ biến nhất mà bạn sẽ gặp phải với tư cách là một lập trình viên Java.



Các câu hỏi từ phần này có thể nằm dọc theo dòng, "Đây là một số mã vừa làm điều gì đó tồi tệ, ngoại lệ nào sẽ được đưa ra?" Trong suốt kỳ thi, các câu hỏi sẽ trình bày một số mã và yêu cầu bạn xác định xem mã đó sẽ chạy hay liệu một ngoại lệ sẽ được đưa ra. Vì những câu hỏi này rất phổ biến nên việc hiểu được nguyên nhân của những trường hợp ngoại lệ này là rất quan trọng cho sự thành công của bạn.

Đây là một trong những mục tiêu khác sẽ xuất hiện trong suốt kỳ thi thực ("Có phải trường hợp ngoại lệ bị ném vào thời gian chạy" có rung chuông không?), Vì vậy hãy đảm bảo rằng phần này sẽ thu hút được nhiều sự chú ý của bạn.

Trường hợp ngoại lệ đến từ đâu

Quay lại một trang và xem lại câu cuối cùng. Điều quan trọng là bạn phải hiểu nguyên nhân gây ra các ngoại lệ và lỗi và chúng đến từ đâu. Với mục đích luyện thi, hãy xác định hai loại ngoại lệ và lỗi lớn:

- Các ngoại lệ của JVM Những ngoại lệ hoặc lỗi đó được JVM ném ra một cách hợp lý hoặc độc quyền nhất
- Các ngoại lệ có lập trình Những ngoại lệ đó được đưa ra một cách rõ ràng bởi

Ứng dụng và / hoặc lập trình viên API

Ngoại lệ JVM-Thrown

Hãy bắt đầu với một ngoại lệ rất phổ biến, `NullPointerException`. Như chúng ta đã thấy trong các chương trước, ngoại lệ này xảy ra khi bạn cố gắng truy cập một đối tượng bằng cách sử dụng một biến tham chiếu có giá trị hiện tại là `null`. Không có cách nào mà trình biên dịch có thể hy vọng tìm thấy những vấn đề này trước thời gian chạy. Hãy xem những điều sau:

```
class NPE {
    static String s;
    public static void main(String [] args) {
        System.out.println(s.length());
    }
}
```

Chắc chắn, trình biên dịch có thể tìm ra vấn đề với chương trình nhỏ bé đó! Không, bạn đang ở một mình. Mã sẽ biên dịch tốt và JVM sẽ ném một `NullPointerException` khi nó cố gắng gọi phương thức `length()`.

Trước đó trong chương này chúng ta đã thảo luận về ngăn xếp cuộc gọi. Như bạn nhớ lại, chúng tôi đã sử dụng quy ước rằng `main()` sẽ ở dưới cùng của ngăn xếp cuộc gọi, và khi `main()` gọi một phương thức khác, và phương thức đó gọi một phương thức khác, và cứ thế, ngăn xếp phát triển lên trên. Tất nhiên, ngăn xếp nằm trong bộ nhớ và ngay cả khi hệ điều hành của bạn cung cấp cho bạn một GB RAM cho chương trình của bạn, nó vẫn là một số lượng hữu hạn. Có thể phát triển ngăn xếp lớn đến mức Hệ điều hành hết dung lượng để lưu trữ ngăn xếp cuộc gọi. Khi điều này xảy ra, bạn nhận được (chờ nó.) lỗi `StackOverflowError`. Cách phổ biến nhất để điều này xảy ra là tạo một phương thức đệ quy. Một phương thức đệ quy gọi chính nó trong thân phương thức. Mặc dù điều đó nghe có vẻ kỳ lạ, nhưng đó là một kỹ thuật rất phổ biến và hữu ích cho những thứ như thuật toán tìm kiếm và sắp xếp. Hãy xem mã này:

```
void go() { // recursion gone bad
    go();
}
```

Như bạn có thể thấy, nếu bạn mắc lỗi khi gọi phương thức `go()`, chương trình của bạn sẽ rơi vào hố đen – `go()` gọi `go()` gọi `go()`, cho đến khi, bất kể bạn có bao nhiêu bộ nhớ, bạn sẽ nhận được lỗi `StackOverflowError`. Một lần nữa, chỉ JVM mới biết thời điểm này xảy ra và JVM sẽ là nguồn gốc của lỗi này.

Ngoại lệ Ném có lập trình

Bây giờ chúng ta hãy xem xét các ngoại lệ được ném theo chương trình. Hãy nhớ rằng chúng tôi đã định nghĩa theo chương trình có nghĩa là một cái gì đó như thế này:

Được tạo bởi một ứng dụng và / hoặc nhà phát triển API

Ví dụ: nhiều lớp trong Java API có các phương thức nhận đối số Chuỗi và chuyển đổi các Chuỗi này thành các nguyên thủy số. Một ví dụ điển hình về các lớp này là cái gọi là "các lớp trình bao bọc" mà chúng ta sẽ nghiên cứu trong [Chương 6](#). Mặc dù chúng ta chưa nói nhiều về các lớp trình bao bọc, nhưng ví dụ sau sẽ có ý nghĩa.

Cách đây khá lâu, một số lập trình viên đã viết lớp `java.lang.Integer` và tạo ra các phương thức như `parseInt()` và `valueOf()`. Lập trình viên đó đã quyết định một cách khôn ngoan rằng nếu một trong những phương thức này được truyền một Chuỗi không thể chuyển đổi thành số, thì phương thức đó sẽ ném ra một `NumberFormatException`. Mã được triển khai một phần có thể trông giống như sau:

```
int parseInt(String s) throws NumberFormatException {
    boolean parseSuccess = false;
    int result = 0;
    // do complicated parsing
    if (!parseSuccess) // if the parsing failed
        throw new NumberFormatException();
    return result;
}
```

Các ví dụ khác về ngoại lệ có lập trình bao gồm `AssertionError` (không sao, nó không phải là ngoại lệ, nhưng nó được ném theo chương trình) và ném `IllegalArgumentException`. Trên thực tế, nhà phát triển API thân thiện của chúng tôi có thể đã sử dụng `IllegalArgumentException` cho phương thức `parseInt()` của cô ấy. Nhưng hóa ra `NumberFormatException` mở rộng `IllegalArgumentException` và chính xác hơn một chút, vì vậy trong trường hợp này, việc sử dụng `NumberFormatException` hỗ trợ khái niệm mà chúng ta đã thảo luận trước đó: rằng khi bạn có hệ thống phân cấp ngoại lệ, bạn nên sử dụng ngoại lệ chính xác nhất mà bạn có thể.

Tất nhiên, như chúng ta đã thảo luận trước đó, bạn cũng có thể tự tạo các ngoại lệ tùy chỉnh đặc biệt và ném chúng bất cứ khi nào bạn muốn. Các ngoại lệ tự chế này cũng thuộc loại "ngoại lệ được ném theo chương trình."

Tóm tắt các ngoại lệ và lỗi của bài kiểm tra

OCA 8 Mục tiêu 8.5 liệt kê một số ngoại lệ và lỗi cụ thể; nó nói “Nhận ra các lớp ngoại lệ phổ biến (chẳng hạn như..”). [Bảng 5-2](#) tóm tắt mười ngoại lệ và lỗi có nhiều khả năng là một phần của bài thi OCA 8.

BẢNG 5-2 Tả và nguồn của các trường hợp ngoại lệ phổ biến

Exception	Description	Typically Thrown
ArrayIndexOutOfBoundsException (this chapter)	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
ClassCastException (Chapter 2)	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
IllegalArgumentException	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
IllegalStateException	Thrown when the state of the environment doesn't match the operation being attempted—for example, using a scanner that's been closed.	Programmatically
NullPointerException (Chapter 3)	Thrown when attempting to invoke a method on, or access a property from, a reference variable whose current value is null.	By the JVM
NumberFormatException (this chapter)	Thrown when a method that converts a <code>String</code> to a number receives a <code>String</code> that it cannot convert.	Programmatically
ArithmetricException	Thrown when an illegal math operation (such as dividing by zero) is attempted.	By the JVM
ExceptionInInitializerError (Chapter 2)	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
StackOverflowError (this chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
NoClassDefFoundError	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing <code>.class</code> file.	By the JVM

TÓM TẮT CHỨNG NHẬN

Chương này bao gồm rất nhiều cơ sở, tất cả đều liên quan đến các cách kiểm soát luồng chương trình của bạn dựa trên một bài kiểm tra có điều kiện. Đầu tiên, bạn đã học về câu lệnh if và switch . Câu lệnh if đánh giá một hoặc nhiều biểu thức thành một kết quả boolean . Nếu kết quả là true , chương trình sẽ thực thi mã trong khối được bao bọc bởi if . Nếu một câu lệnh else được sử dụng và biểu thức if được đánh giá là false , thì đoạn mã theo sau else sẽ được thực hiện. Nếu không có khối nào khác được xác định, thì không có mã nào được liên kết với câu lệnh if sẽ thực thi.

Bạn cũng đã biết rằng câu lệnh switch có thể được sử dụng để thay thế nhiều câu lệnh if-else . Câu lệnh switch có thể đánh giá các kiểu nguyên thủy số nguyên có thể được truyền ngầm thành int (các kiểu đó là byte, short, int và char); hoặc nó có thể đánh giá enums; và kể từ Java 7, nó có thể đánh giá các Chuỗi. Trong thời gian chạy, JVM sẽ cố gắng tìm sự phù hợp giữa biểu thức trong câu lệnh switch và một hằng số trong câu lệnh trường hợp tương ứng . Nếu khớp được tìm thấy, quá trình thực thi sẽ bắt đầu tại trường hợp phù hợp và tiếp tục từ đó, thực thi mã trong tất cả các câu lệnh trường hợp còn lại cho đến khi tìm thấy câu lệnh break hoặc kết thúc câu lệnh switch xảy ra. Nếu không có kết quả phù hợp, thì trường hợp mặc định sẽ thực thi, nếu có.

Bạn đã học về ba cấu trúc lặp có sẵn trong ngôn ngữ Java. Các cấu trúc này là vòng lặp for (bao gồm for cơ bản và for nâng cao , mới đổi với Java 5), vòng lặp while và vòng lặp do . Nói chung, vòng lặp for được sử dụng khi bạn biết bạn cần đi qua vòng lặp bao nhiêu lần. Vòng lặp while được sử dụng khi bạn không biết mình muốn trải qua bao nhiêu lần, trong khi vòng lặp do được sử dụng khi bạn cần xem qua ít nhất một lần. Trong vòng lặp for và vòng lặp while , biểu thức phải đánh giá thành true để đi vào bên trong khối và sẽ kiểm tra sau mỗi lần lặp lại của vòng lặp. Vòng lặp do không kiểm tra điều kiện cho đến khi nó đi qua vòng lặp một lần. Lợi ích chính của vòng lặp for là khả năng khởi tạo một hoặc nhiều biến và tăng hoặc giảm các biến đó trong định nghĩa vòng lặp for .

Câu lệnh ngắt và tiếp tục có thể được sử dụng theo kiểu có nhãn hoặc không được gắn nhãn. Khi không được gắn nhãn, câu lệnh break sẽ buộc chương trình ngừng xử lý cấu trúc lặp trong cùng và bắt đầu bằng dòng mã theo sau vòng lặp. Sử dụng lệnh tiếp tục không được gắn nhãn sẽ gây ra

chương trình để dừng thực hiện lần lặp hiện tại của vòng lặp trong cùng và tiếp tục với lần lặp tiếp theo. Khi một câu lệnh break hoặc continue được sử dụng theo cách có nhẫn, nó sẽ thực hiện theo cách tương tự, với một ngoại lệ: câu lệnh sẽ không áp dụng cho vòng lặp trong cùng; thay vào đó, nó sẽ áp dụng cho vòng lặp có nhẫn. Câu lệnh break được sử dụng thường xuyên nhất cùng với câu lệnh switch . Khi có sự phù hợp giữa biểu thức switch và hằng số, đoạn mã theo sau hằng chữ hoa sẽ được thực hiện. Để dừng thực hiện, cần có thời gian nghỉ .

Bạn đã thấy cách Java cung cấp một cơ chế thanh lịch trong việc xử lý ngoại lệ. Xử lý ngoại lệ cho phép bạn tách mã sửa lỗi của mình thành các khối riêng biệt để mã chính không trở nên lộn xộn bởi mã kiểm tra lỗi. Một tính năng thanh lịch khác cho phép bạn xử lý các lỗi tương tự với một khối xử lý lỗi duy nhất, không có mã trùng lặp. Ngoài ra, việc xử lý lỗi có thể được hoãn lại cho các phương pháp trả lại ngăn xếp cuộc gọi.

Bạn đã biết rằng từ khóa try của Java được sử dụng để chỉ định một vùng được bảo vệ – một khối mã trong đó các vấn đề có thể được phát hiện. Một trình xử lý ngoại lệ là mã được thực thi khi một ngoại lệ xảy ra. Trình xử lý được xác định bằng cách sử dụng từ khóa catch của Java . Tất cả các mệnh đề bắt phải ngay lập tức theo sau khối try liên quan .

Java cũng cung cấp từ khóa cuối cùng . Điều này được sử dụng để xác định một khối mã luôn được thực thi, ngay sau khi mệnh đề bắt hoàn thành hoặc ngay sau khối try liên kết trong trường hợp không có ngoại lệ nào được ném ra (hoặc đã thử nhưng không bắt được). Sử dụng các khối cuối cùng để giải phóng tài nguyên hệ thống và thực hiện mọi thao tác dọn dẹp theo yêu cầu của mã trong khối thử . Một khối cuối cùng không được yêu cầu, nhưng nếu có một khối, nó phải ngay lập tức theo dõi lần bắt cuối cùng. (Nếu không có khối catch , khối cuối cùng phải ngay sau khối try .) Nó được đảm bảo sẽ được gọi ngoại trừ khi khối try hoặc catch có vấn đề về System.exit () .

Đối tượng ngoại lệ là một thẻ hiện của lớp Exception hoặc một trong các lớp con của nó. Mệnh đề catch , dưới dạng một tham số, là một thẻ hiện của một đối tượng có kiểu dẫn xuất từ lớp Exception . Java yêu cầu mỗi phương thức bắt bắt kỳ ngoại lệ đã kiểm tra nào mà nó có thể ném hoặc người khác tuyên bố rằng nó ném ngoại lệ. Khai báo ngoại lệ là một phần của chữ ký của phương thức. Để khai báo rằng một ngoại lệ có thể được ném ra, từ khóa throws được sử dụng trong định nghĩa phương thức, cùng với danh sách tất cả các ngoại lệ đã kiểm tra có thể được ném ra.

Các ngoại lệ thời gian chạy thuộc loại RuntimeException (hoặc một trong các lớp con của nó). Những trường hợp ngoại lệ này là một trường hợp đặc biệt vì chúng không cần phải được xử lý hoặc khai báo, và do đó được gọi là các trường hợp ngoại lệ "không được kiểm tra". Lỗi thuộc loại

`java.lang.Error` hoặc các lớp con của nó, và giống như các ngoại lệ thời gian chạy, chúng không cần được xử lý hoặc khai báo. Các ngoại lệ được kiểm tra bao gồm bất kỳ loại ngoại lệ nào không thuộc loại `RuntimeException` hoặc `Error`. Nếu mã của bạn không xử lý được một ngoại lệ đã kiểm tra hoặc tuyên bố rằng nó được ném ra, thì mã của bạn sẽ không được biên dịch. Nhưng với các ngoại lệ không được kiểm tra hoặc các đối tượng kiểu Lỗi, việc bạn khai báo hay xử lý chúng, không làm gì với chúng hay thực hiện một số kết hợp giữa khai báo và xử lý không quan trọng đối với trình biên dịch. Nói cách khác, bạn có thể tự do khai báo và xử lý chúng, nhưng trình biên dịch sẽ không quan tâm theo cách này hay cách khác. Tuy nhiên, việc xử lý Lỗi không phải là thực tiễn tốt vì bạn hiếm khi có thể khôi phục được lỗi.

Cuối cùng, hãy nhớ rằng các ngoại lệ có thể được tạo ra bởi JVM hoặc bởi một lập trình viên.

✓ KHOAN HAI PHÚT

Dưới đây là một số điểm chính từ mỗi mục tiêu chứng nhận trong chương này. Bạn có thể muốn lặp lại chúng vài lần.

Viết mã Sử dụng câu lệnh if và switch (Mục tiêu 3.3 và 3.4 của OCA)

- Biểu thức hợp pháp duy nhất trong câu lệnh if là biểu thức boolean – nói cách khác, biểu thức phân giải thành tham chiếu boolean hoặc Boolean .
- Hãy chú ý đến các phép gán boolean (=) có thể bị nhầm lẫn với kiểm tra đẳng thức boolean (==) :


```
boolean x = false; if (x
= true) {} // một phép gán, vì vậy x sẽ luôn đúng!
```
- Dấu ngoặc nhọn là tùy chọn cho các khối if chỉ có một câu lệnh điều kiện. Nhưng hãy coi chừng những thlut đầu dòng gây hiểu lầm. các câu lệnh switch chỉ có thể
- đánh giá thành enums hoặc các kiểu dữ liệu byte, short, int, char, và đối với Java 7, String . Bạn không thể nói điều này:


```
dài s = 30;
chuyển (các) {}
```
- Hằng số trường hợp phải là một hằng số theo nghĩa đen hoặc một hằng số thời gian biên dịch, bao gồm

một enum hoặc một chuỗi. Bạn không thể có một trường hợp bao gồm một biến không cuối cùng hoặc một phạm vi giá trị.

- Nếu điều kiện trong câu lệnh switch khớp với một hằng số, thì việc thực thi sẽ chạy qua tất cả mã trong switch sau câu lệnh trường hợp phù hợp cho đến khi gặp câu lệnh break hoặc cuối câu lệnh switch . Nói cách khác, trường hợp phù hợp chỉ là điểm nhập vào khỏi trường hợp , nhưng trừ khi có câu lệnh break , trường hợp phù hợp không phải là mã trường hợp duy nhất chạy.
- Từ khóa mặc định nên được sử dụng trong câu lệnh switch nếu bạn muốn chạy một số mã khi không có giá trị chữ hoa chữ thường nào khớp với giá trị điều kiện.
- Khối mặc định có thể được đặt ở bất kỳ vị trí nào trong khối chuyển đổi , vì vậy nếu không có trường hợp nào trước đó khớp, khối mặc định sẽ được nhập; nếu mặc định không chứa ngắt, thì mã sẽ tiếp tục thực thi (chuyển sang) đến cuối công tắc hoặc cho đến khi gặp câu lệnh ngắt .

Viết mã bằng vòng lặp (Mục tiêu của OCA 5.1, 5.2, 5.3 và 5.4)

- Câu lệnh for cơ bản có ba phần: khai báo và / hoặc khởi tạo, đánh giá boolean và biểu thức lặp.
- Nếu một biến được tăng hoặc đánh giá trong vòng lặp for cơ bản , thì nó phải được khai báo trước vòng lặp hoặc trong phần khai báo vòng lặp for .
- Một biến được khai báo (không chỉ được khởi tạo) trong phần khai báo vòng lặp for cơ bản không thể được truy cập bên ngoài vòng lặp for – nói cách khác, mã bên dưới vòng lặp for sẽ không thể sử dụng biến.
- Bạn có thể khởi tạo nhiều hơn một biến cùng kiểu trong phần đầu của phần khai báo vòng lặp for cơ bản ; mỗi lần khởi tạo phải được phân tách bằng dấu phẩy.
- Câu lệnh for nâng cao (mới của Java 5) có hai phần: phần khai báo và phần biểu thức. Nó chỉ được sử dụng để lặp qua các mảng hoặc bộ sưu tập.
- Với for nâng cao, biểu thức là mảng hoặc tập hợp mà bạn muốn lặp qua.
- Với for nâng cao, khai báo là biến khôi, có kiểu tương thích với các phần tử của mảng hoặc tập hợp và biến đó chứa giá trị của phần tử cho lần lặp đã cho.

- Không giống như với C, bạn không thể sử dụng một số hoặc bất kỳ thứ gì không đánh giá thành giá trị boolean làm điều kiện cho câu lệnh if hoặc cấu trúc lặp.
Ví dụ, bạn không thể nói if (x), trừ khi x là một biến boolean .
- Vòng lặp do sẽ luôn đi vào phần thân của vòng lặp ít nhất một lần.

Sử dụng ngắt và tiếp tục (Mục tiêu 5.5 của OCA)

- Một câu lệnh break không được gắn nhãn sẽ khiến quá trình lặp lại hiện tại của vòng lặp trong cùng dừng lại và dòng mã sau vòng lặp sẽ chạy.
- Một câu lệnh continue không được gắn nhãn sẽ khiến quá trình lặp hiện tại của vòng lặp trong cùng dừng lại, điều kiện của vòng lặp đó sẽ được kiểm tra và nếu điều kiện được đáp ứng, vòng lặp sẽ chạy lại.
- Nếu câu lệnh break hoặc câu lệnh continue được gắn nhãn, nó sẽ gây ra một hành động tương tự xảy ra trên vòng lặp được gắn nhãn, không phải vòng lặp trong cùng.

Xử lý các trường hợp ngoại lệ (Mục tiêu của OCA 8.1, 8.2, 8.3, 8.4 và 8.5)

- Một số lợi ích của các tính năng xử lý ngoại lệ của Java bao gồm mã xử lý lỗi có tổ chức, phát hiện lỗi dễ dàng, giữ mã xử lý ngoại lệ tách biệt với mã khác và khả năng sử dụng lại mã xử lý ngoại lệ cho một loạt vấn đề.
- Các trường hợp ngoại lệ có hai loại: được chọn và không được chọn.
- Các ngoại lệ được kiểm tra bao gồm tất cả các kiểu con của Ngoại lệ, không bao gồm các lớp mở rộng RuntimeException.
- Các ngoại lệ được kiểm tra phải tuân theo quy tắc xử lý hoặc khai báo; bất kỳ phương thức nào có thể ném một ngoại lệ đã kiểm tra (bao gồm cả các phương thức gọi các phương thức có thể ném một ngoại lệ đã kiểm tra) phải khai báo ngoại lệ bằng cách sử dụng ném hoặc xử lý ngoại lệ bằng một lần thử / bắt thích hợp.
- Các kiểu phụ của Lỗi hoặc RuntimeException không được chọn, vì vậy trình biên dịch không thực thi quy tắc xử lý hoặc khai báo. Bạn có thể tự do xử lý chúng hoặc khai báo chúng, nhưng trình biên dịch không quan tâm theo cách này hay cách khác.
- Một khối cuối cùng sẽ luôn được gọi, bất kể một ngoại lệ được ném hay bị bắt trong lần thử / bắt của nó.
- Ngoại lệ duy nhất đối với quy tắc cuối cùng sẽ luôn được gọi là cuối cùng sẽ không được gọi nếu JVM tắt. Điều đó có thể xảy ra nếu

mã từ các cuộc gọi khôi try hoặc catch System.exit ().

- Chỉ vì cuối cùng được gọi không có nghĩa là nó sẽ hoàn thành. Mã trong khôi cuối cùng có thể tự tạo ra một ngoại lệ hoặc phát hành một System.exit ().
- Các ngoại lệ chưa được phô biến trở lại thông qua ngăn xếp cuộc gọi, bắt đầu từ phương thức mà ngoại lệ được ném và kết thúc bằng phương thức đầu tiên có một lệnh bắt tương ứng cho loại ngoại lệ đó hoặc tắt JVM (điều này xảy ra nếu ngoại lệ đến main () và main () là "loại bỏ" ngoại lệ bằng cách khai báo nó).
- Bạn hầu như luôn có thể tạo ngoại lệ của riêng mình bằng cách mở rộng Exception hoặc một trong các kiểu phụ ngoại lệ đã kiểm tra của nó. Một ngoại lệ như vậy sau đó sẽ được coi là một ngoại lệ đã được kiểm tra bởi trình biên dịch. (Nói cách khác, hiếm khi mở rộng RuntimeException.)
- Tất cả các khôi bắt phải được sắp xếp từ cụ thể nhất đến tổng quát nhất. Nếu bạn có mệnh đề bắt cho cả IOException và ngoại lệ, bạn phải đặt lệnh bắt cho IOException trước trong mã của mình. Nếu không, IOException sẽ bị bắt bởi catch (Exception e), bởi vì một đối số catch có thể bắt ngoại lệ được chỉ định hoặc bất kỳ kiểu con nào của nó!
- Một số ngoại lệ do lập trình viên tạo ra và một số ngoại lệ do JVM tạo ra.

TỰ KIỂM TRA

1. Cho rằngtoLowerCase () là một phương thức String được đặt tên phù hợp để trả về Chuỗi và được cung cấp mã:

```
public class Flipper {
    public static void main(String[] args) {
        String o = "-";
        switch("RED".toLowerCase()) {
            case "yellow":
                o += "y";
            case "red":
                o += "r";
            case "green":
                o += "g";
        }
        System.out.println(o);
    }
}
```

Kết quả là gì?

A. -

B. -r

C. -rg D.

Biên dịch không thành công

E. Một ngoại lệ được đưa ra trong thời gian chạy

2. Đưa ra:

```
class Plane {
    static String s = "-";
    public static void main(String[] args) {
        new Plane().s1();
        System.out.println(s);
    }
    void s1() {
        try { s2(); }
        catch (Exception e) { s += "c"; }
    }
    void s2() throws Exception {
        s3(); s += "2";
        s3(); s += "2b";
    }
    void s3() throws Exception {
        throw new Exception();
    }
}
```

Kết quả là gì?

- A. -
- B. -c
- C. -c2
- D. -2c
- E. -c22b
- F. -2c2b
- G. -2c2bc
- H. Biên dịch không thành công

3. Đưa ra:

```
thử {int x = Integer.parseInt ("hai"); }
```

Cái nào có thể được sử dụng để tạo một khối bắt thích hợp ? (Chọn tất cả các áp dụng.)

- A. ClassCastException
- B. IllegalStateException
- C. NumberFormatException
- D. IllegalArgumentException
- E. ExceptionInInitializerError
- F. ArrayIndexOutOfBoundsException

4. Cho:

```
public class Flip2 {  
    public static void main(String[] args) {  
        String o = "-";  
        String[] sa = new String[4];  
        for(int i = 0; i < args.length; i++)  
            sa[i] = args[i];  
        for(String n: sa) {  
            switch(n.toLowerCase()) {  
                case "yellow": o += "Y";  
                case "red":     o += "r";  
                case "green":   o += "g";  
            }  
        }  
        System.out.print(o);  
    }  
}
```

Và đưa ra lời gọi dòng lệnh:

Java Flip2 RED Green YeLLow

Đó là sự thật? (Chọn tất cả các áp dụng.)

- A. Chuỗi rgy sẽ xuất hiện ở đâu đó trong đầu ra
- B. Chuỗi rgg sẽ xuất hiện ở đâu đó trong đầu ra
- C. Chuỗi con quay hòi chuyển sẽ xuất hiện ở đâu đó trong đầu ra
- D. Biên dịch không thành công
- E. Một ngoại lệ được ném ra trong thời gian chạy

5. Đưa ra:

```
1. class Loopy {  
2.     public static void main(String[] args) {  
3.         int[] x = {7,6,5,4,3,2,1};  
4.         // insert code here  
5.         System.out.print(y + " ");  
6.     }  
7. }  
8. }
```

Cái nào, được chèn độc lập ở dòng 4, biên dịch? (Chọn tất cả các áp dụng.)

- A. for(int y : x) {
- B. for(x : int y) {
- C. int y = 0; for(y : x) {
- D. for(int y=0, z=0; z<x.length; z++) { y = x[z];
- E. for(int y=0, int z=0; z<x.length; z++) { y = x[z];
- F. int y = 0; for(int z=0; z<x.length; z++) { y = x[z];

6. Cho:

```
class Emu {  
    static String s = "-";  
    public static void main(String[] args) {  
        try {  
            throw new Exception();  
        } catch (Exception e) {  
            try {  
                try { throw new Exception();  
                } catch (Exception ex) { s += "ic "; }  
                throw new Exception(); }  
            catch (Exception x) { s += "mc "; }  
            finally { s += "mf "; }  
        } finally { s += "of "; }  
        System.out.println(s);  
    } }
```

Kết quả là gì?

A. -ic của

B. -mf trong tổng số

C. -mc mf

D. -ic mf của

E. -ic mc mf của

F. -ic mc của mf

G. Biên dịch không thành công

7. Cho:

```
3. class SubException extends Exception { }  
4. class SubSubException extends SubException { }  
5.  
6. public class CC { void doStuff() throws SubException { } }  
7.  
8. class CC2 extends CC { void doStuff() throws SubSubException { } }  
9.  
10. class CC3 extends CC { void doStuff() throws Exception { } }  
11.  
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }  
13.  
14. class CC5 extends CC { void doStuff() { } }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

A. Biên dịch thành công

B. Biên dịch không thành công do lỗi trên dòng 8

C. Biên dịch không thành công do lỗi ở dòng 10 D.

Biên dịch không thành công do lỗi ở dòng 12 E.

Biên dịch không thành công do lỗi ở dòng 14

8. Cho:

```
3. public class Ebb {  
4.     static int x = 7;  
5.     public static void main(String[] args) {  
6.         String s = "";  
7.         for(int y = 0; y < 3; y++) {  
8.             x++;  
9.             switch(x) {  
10.                 case 8: s += "8 ";  
11.                 case 9: s += "9 ";  
12.                 case 10: { s+= "10 "; break; }  
13.                 default: s += "d ";  
14.                 case 13: s+= "13 ";  
15.             }  
16.         }  
17.         System.out.println(s);  
18.     }  
19.     static { x++; }  
20. }
```

Kết quả là gì?

A. 9 10 ngày

B. 8 9 10 ngày

C. 9 10 10 ngày

D. 9 10 10 d 13

E. 8 9 10 10 d 13

F. 8 9 10 9 10 10 d 13

G. Biên dịch không thành công

9. Cho:

```

3. class Infinity { }
4. public class Beyond extends Infinity {
5.     static Integer i;
6.     public static void main(String[] args) {
7.         int sw = (int)(Math.random() * 3);
8.         switch(sw) {
9.             case 0: { for(int x = 10; x > 5; x++)
10.                         if(x > 10000000) x = 10;
11.                         break; }
12.             case 1: { int y = 7 * i; break; }
13.             case 2: { Infinity inf = new Beyond();
14.                         Beyond b = (Beyond)inf; }
15.         }
16.     }
17. }

```

Và cho rằng dòng 7 sẽ gán giá trị 0, 1 hoặc 2 cho sw, điều nào đúng?
(Chọn tất cả các áp dụng.)

A. Biên dịch không

thành công B. Một ClassCastException có thể

được ném C. Một StackOverflowError có thể

được ném D. Một NullPointerException có thể

được ném E. Một IllegalStateException có thể được

ném F. Chương trình có thể bị treo mà không bao giờ

hoàn thành G. Chương trình sẽ luôn hoàn thành mà không có ngoại lệ

10. Cho:

```

3. public class Circles {
4.     public static void main(String[] args) {
5.         int[] ia = {1,3,5,7,9};
6.         for(int x : ia) {
7.             for(int j = 0; j < 3; j++) {
8.                 if(x > 4 && x < 8) continue;
9.                 System.out.print(" " + x);
10.                if(j == 1) break;
11.                continue;
12.            }
13.            continue;
14.        }
15.    }
16. }

```

Kết quả là gì?

Kết quả là gì?

- A. 1 3 9
- B. 5 5 7 7
- C. 1 3 3 9 9
- D. 1 1 3 3 9 9
- E. 1 1 1 3 3 9 9 9

F. Biên dịch không thành công

11. Cho:

```
3. public class OverAndOver {  
4.     static String s = "";  
5.     public static void main(String[] args) {  
6.         try {  
7.             s += "1";  
8.             throw new Exception();  
9.         } catch (Exception e) { s += "2";  
10.        } finally { s += "3"; doStuff(); s += "4";  
11.        }  
12.        System.out.println(s);  
13.    }  
14.    static void doStuff() { int x = 0; int y = 7/x; }  
15. }
```

Kết quả là gì?

- A. 12
- B. 13
- C. 123
- D. 1234

E. Biên dịch không

thành công F. 123 theo sau là

ngoại lệ G. 1234 theo sau là ngoại

lệ H. Một ngoại lệ được ném ra mà không có đầu ra nào khác

12. Cho:

```
3. public class Wind {  
4.     public static void main(String[] args) {  
5.         foreach:  
6.             for(int j=0; j<5; j++) {  
7.                 for(int k=0; k< 3; k++) {  
8.                     System.out.print(" " + j);  
9.                     if(j==3 && k==1) break foreach;  
10.                    if(j==0 || j==2) break;  
11.                }  
12.            }  
13.        }  
14.    }
```

Kết quả là gì?

- A. 0 1 2 3
- B. 1 1 1 3 3
- C. 0 1 1 1 2 3 3
- D. 1 1 1 3 3 4 4 4
- E. 0 1 1 1 2 3 3 4 4 4

F. Biên dịch không thành công

13. Cho:

```
3. public class Gotcha {  
4.     public static void main(String[] args) {  
5.         // insert code here  
6.  
7.     }  
8.     void go() {  
9.         go();  
10.    }  
11. }
```

Và đưa ra ba đoạn mã sau:

- I. new Gotcha().go();
- II. try { new Gotcha().go(); }
 catch (Error e) { System.out.println("ouch"); }
- III. try { new Gotcha().go(); }
 catch (Exception e) { System.out.println("ouch"); }

Khi các đoạn I-III được thêm vào, một cách độc lập, ở dòng 5, điều nào đúng?
(Chọn tất cả các áp dụng.)

- A. Một số sẽ không biên dịch
B. Tất cả sẽ biên dịch C. Tất
cả sẽ hoàn thành bình thường D.

Không có cái nào sẽ hoàn thành bình
thường E. Chỉ một cái sẽ hoàn thành bình
thường F. Hai trong số chúng sẽ hoàn thành bình thường

14. Cho đoạn mã:

```
String s = "bob";
String[] sa = {"a", "bob"};
final String s2 = "bob";
StringBuilder sb = new StringBuilder("bob");

// switch(sa[1]) {           // line 1
// switch("b" + "ob") {     // line 2
// switch(sb.toString()) { // line 3

// case "ann": ;          // line 4
// case s: ;               // line 5
// case s2: ;              // line 6
}
```

Và cho rằng tất cả các dòng được đánh số sẽ được kiểm tra bằng cách bỏ ghi chú
một câu lệnh switch và một câu lệnh trường hợp cùng nhau, (các) dòng nào sẽ
KHÔNG biên dịch? (Chọn tất cả các áp dụng.)

- A. dòng 1
B. dòng 2
C. dòng 3
D. dòng 4
E. dòng 5
F. dòng 6
G. Tất cả sáu dòng mã sẽ biên dịch

15. Cho rằng IOException nằm trong gói java.io và đã cho:

```
1. public class Frisbee {  
2.     // insert code here  
3.     int x = 0;  
4.     System.out.println(7/x);  
5. }  
6. }
```

Và đưa ra bốn đoạn mã sau:

- I. public static void main(String[] args) {
- II. public static void main(String[] args) throws Exception {
- III. public static void main(String[] args) throws IOException {
- IV. public static void main(String[] args) throws RuntimeException {

Nếu bốn đoạn được chèn độc lập ở dòng 2, điều nào đúng?

(Chọn tất cả các áp dụng.)

- A. Cả bốn sẽ biên dịch và thực thi mà không có ngoại lệ
- B. Cả bốn sẽ biên dịch và thực thi và ném một ngoại lệ
- C. Một số, nhưng không phải tất cả, sẽ biên dịch và thực thi mà không có ngoại lệ
- D. Một số, nhưng không phải tất cả, sẽ biên dịch và thực thi và ném một ngoại lệ
- E. Khi xem xét các đoạn II, III và IV, trong số những đoạn sẽ biên dịch, việc thêm một khôi try / catch quanh dòng 4 sẽ khiến quá trình biên dịch không thành công

16. Cho:

```
2. class MyException extends Exception {}  
3. class Tire {  
4.     void doStuff() {}  
5. }  
6. public class Retread extends Tire {  
7.     public static void main(String[] args) {  
8.         new Retread().doStuff();  
9.     }  
10.    // insert code here  
11.    System.out.println(7/0);  
12. }  
13. }
```

Và đưa ra bốn đoạn mã sau:

- I. void doStuff() {
- II. void doStuff() throws MyException {
- III. void doStuff() throws RuntimeException {
- IV. void doStuff() throws ArithmeticException {

Khi các đoạn I-IV được thêm vào, một cách độc lập, ở dòng 10, điều nào đúng?

(Chọn tất cả các áp dụng.)

(Chọn tất cả các áp dụng.)

A. Không ai sẽ biên dịch B.

Tất cả chúng sẽ biên dịch C.

Một số, nhưng không phải tất cả, sẽ biên

dịch D. Tất cả những gì biên dịch sẽ ném một ngoại lệ trong thời gian chạy

E. Không ai trong số những người biên dịch sẽ ném một ngoại lệ trong thời gian

chạy F. Chỉ một số trong số những người biên dịch sẽ ném ra một ngoại lệ trong thời gian chạy

CÂU TRẢ LỜI TỰ KIỂM TRA

1. C đúng. Đối với Java 7, việc chuyển đổi trên một Chuỗi là hợp pháp và hãy nhớ rằng các công tắc sử dụng logic “điểm vào”.

A, B, D và E là không chính xác dựa trên các điều trên. (Mục tiêu 3.4 của OCA)

2. B đúng. Sau khi s3 () ném ngoại lệ cho s2 (), s2 () ném nó tới s1 () và không có mã nào khác của s2 () sẽ được thực thi.

A, C, D, E, F, G và H là không chính xác dựa trên các điều trên. (OCA Mục tiêu 8.2 và 8.4)

3. C và D đúng. Integer.parseInt có thể ném một NumberFormatException và IllegalArgumentException là lớp cha của nó (nghĩa là một ngoại lệ rộng hơn).

A, B, E và F không nằm trong phân cấp lớp của NumberFormatException . (Mục tiêu 8.5 của OCA)

4. E đúng. Đối với Java 7, cú pháp là hợp pháp. Mảng sa [] chỉ nhận ba đối số từ dòng lệnh, vì vậy trong lần lặp cuối cùng qua sa [], một NullPointerException được ném ra.

A, B, C và D là không chính xác dựa trên các điều trên. (Mục tiêu của OCA 1.3, 5.2 và 8.5)

5. A, D và F đều đúng. A là một ví dụ về vòng lặp for nâng cao . D và F là các ví dụ về vòng lặp for cơ bản .

B, C và E không chính xác. B không chính xác vì các toán hạng của nó được hoán đổi. C không chính xác vì for nâng cao phải khai báo toán hạng đầu tiên của nó. E là sai cú pháp để khai báo hai biến trong câu lệnh for .

(Mục tiêu 5.2 của OCA)

6. E đúng. Không có vấn đề gì khi lồng các khối try / catch . Như là bình thường, khi một ngoại lệ được ném ra, mã trong khối bắt sẽ chạy, và sau đó mã trong khối cuối cùng sẽ chạy.
 - A, B, C, D và F là không chính xác dựa trên các điều trên. (Mục tiêu 8.2 và 8.4 của OCA)
7. C đúng. Một phương pháp ghi đè không thể đưa ra một ngoại lệ rỗng hơn so với phương pháp mà nó ghi đè. Phương thức của lớp CC4 là quá tải, không phải là ghi đè.
 - A, B, D và E là không chính xác dựa trên các điều trên. (Mục tiêu 8.2 và 8.4 của OCA)
8. D đúng. Bạn có bắt được khôi khởi tạo tĩnh không? Hãy nhớ rằng các công tắc hoạt động trên logic "rơi qua" và logic chuyển tiếp cũng áp dụng cho trường hợp mặc định, được sử dụng khi không có trường hợp nào khác khớp.
 - A, B, C, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 3.4 của OCA)
9. D và F đúng. Vì tôi chưa được khởi tạo, trường hợp 1 sẽ ném ra một NullPointerException. Trường hợp 0 sẽ bắt đầu một vòng lặp vô tận, không phải là một tràn ngăn xếp. Sự sụp đổ của Trường hợp 2 sẽ không gây ra một ngoại lệ.
 - A, B, C, E và G là không chính xác dựa trên các điều trên. (Mục tiêu OCA 3,4 và 8,5)
10. D đúng. Quy tắc cơ bản cho các câu lệnh continue không được gắn nhãn là lặp hiện tại dừng sớm và quá trình thực thi sẽ chuyển sang lần lặp tiếp theo. Hai câu lệnh continue cuối cùng là thửa!
 - A, B, C, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 5,2 và 5,5 của OCA)
11. H đúng. Đúng là giá trị của Chuỗi s là 123 tại thời điểm đó ngoại lệ chia cho-không được ném ra, nhưng cuối cùng () không được đảm bảo hoàn thành và trong trường hợp này, cuối cùng () không bao giờ hoàn thành, vì vậy System.out.println (SOP) không bao giờ thực thi.
 - A, B, C, D, E, F và G là không chính xác dựa trên các điều trên. (OCA Mục tiêu 8.2 và 8.5)

12. C đúng. Một break thoát ra khỏi vòng lặp trong cùng hiện tại và tiếp tục. Dấu ngắt được gán nhãn thoát ra và chấm dứt các vòng được gán nhãn.
- A, B, D, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 5,2 và 5,5 của OCA)
13. B và E đúng. Trước hết, go () là một đệ quy được thiết kế tồi, được đảm bảo gây ra lỗi StackOverflowError. Vì Ngoại lệ không phải là lớp chồng của Lỗi, việc bắt một Ngoại lệ sẽ không giúp xử lý Lỗi, do đó, phân đoạn III sẽ không hoàn thành bình thường. Chỉ phân đoạn II sẽ bắt lỗi.
- A, C, D và F là không chính xác dựa trên các điều trên. (Mục tiêu của OCA 8.1, 8.2 và 8.4)
14. E đúng. Các trường hợp của một công tắc phải là hằng số thời gian biên dịch hoặc enum các giá trị.
- A, B, C, D, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 3.4 của OCA)
15. D đúng. Đây là một loại lén lút, nhưng hãy nhớ rằng chúng tôi đang cố gắng rèn luyện sức khỏe cho bạn trong kỳ thi thực sự. Nếu bạn định ném IOException, bạn phải nhập gói java.io hoặc khai báo ngoại lệ với tên đủ điều kiện.
- A, B, C và E không chính xác. A, B và C không chính xác dựa trên ở trên. E sai vì cả xử lý và khai báo ngoại lệ đều được. (Mục tiêu 8,2 và 8,5 của OCA)
16. C và D đúng. Phương thức ghi đè không thể ném kiểm tra các trường hợp ngoại lệ rộng hơn các trường hợp ngoại lệ được đưa ra bởi phương thức ghi đè. Tuy nhiên, một phương thức ghi đè có thể ném RuntimeExceptions không được phương thức ghi đè ném ra.
- A, B, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 8.1 của OCA)



6

Chuỗi, Mảng, Mảng, Danh sách, Ngày và Lambdas

CHỨNG NHÂN

MỤC TIÊU

- Tạo và thao tác chuỗi • Thao tác dữ liệu bằng lớp `StringBuilder` và các phương thức của nó • Tạo và sử dụng dữ liệu lịch
- Khai báo, khởi tạo, khởi tạo và sử dụng mảng một chiều • Khai báo, tạo, khởi tạo và sử dụng mảng đa chiều • Khai báo và sử dụng danh sách mảng • Sử dụng lớp gói • Sử dụng đóng gói cho các biến tham chiếu • Sử dụng biểu thức Lambda đơn giản

✓ Khoan hai phút

Hỏi & Đáp Tự kiểm tra

T chương của ông tập trung vào các mục tiêu kỳ thi liên quan đến tìm kiếm, định dạng và phân tích cú pháp chuỗi; tạo và sử dụng liên quan đến lịch các đối tượng; tạo và sử dụng mảng và ArrayLists; và sử dụng đơn giản biểu thức lambda. Nhiều chủ đề trong số này có thể lấp đầy toàn bộ cuốn sách. May mắn thay, bạn sẽ không phải trở thành một chuyên gia tổng thể để làm tốt kỳ thi. Nhóm kiểm tra dự định chỉ bao gồm các khía cạnh cơ bản của những công nghệ này và trong chương này, chúng tôi đề cập nhiều hơn những gì bạn cần để đạt được các mục tiêu liên quan trong kỳ thi.

MỤC TIÊU XÁC NHÂN

Sử dụng String và StringBuilder (Mục tiêu 9.2 và 9.1 của OCA)

9.2 Tạo và thao tác với chuỗi.

9.1 Thao tác dữ liệu bằng lớp StringBuilder và các phương thức của nó.

Mọi thứ bạn cần biết về chuỗi trong các bài kiểm tra OCJP cũ hơn, bạn sẽ cần biết cho kỳ thi OCA 8. Liên quan chặt chẽ đến lớp String là lớp StringBuilder và lớp StringBuffer gần như giống hệt nhau. (Đối với kỳ thi, điều duy nhất bạn cần biết về lớp StringBuffer là nó có các phương thức giống hệt như lớp StringBuilder, nhưng StringBuilder nhanh hơn vì các phương thức của nó không được đồng bộ hóa.) Cả hai lớp, StringBuilder và StringBuffer, cho bạn Các đối tượng dạng chuỗi và cách thao tác với chúng, với sự khác biệt quan trọng là các đối tượng này có thể thay đổi được.

Lớp chuỗi

Phần này bao gồm lớp String và khái niệm chính để bạn hiểu là một khi một đối tượng String được tạo, nó không bao giờ có thể thay đổi được. Vì vậy, điều gì đang xảy ra khi một đối tượng Chuỗi dường như đang thay đổi? Hãy cùng tìm hiểu.

Chuỗi là đối tượng bất biến Chúng ta

sẽ bắt đầu với một chút thông tin cơ bản về chuỗi. Bạn có thể không cần điều này cho bài kiểm tra, nhưng một chút ngữ cảnh sẽ hữu ích. Xử lý "chuỗi" của các ký tự là một khía cạnh cơ bản của hầu hết các ngôn ngữ lập trình. Trong Java, mỗi ký tự trong một chuỗi là một ký tự Unicode 16 bit. Bởi vì các ký tự Unicode là 16 bit (không phải là 7 hoặc 8 bit ít ỏi mà ASCII cung cấp), một bộ ký tự quốc tế phong phú được dễ dàng biểu diễn bằng Unicode.

Trong Java, chuỗi là các đối tượng. Như với các đối tượng khác, bạn có thể tạo một phiên bản của chuỗi với từ khóa mới , như sau:

```
String s = new String();
```

Dòng mã này tạo một đối tượng mới của lớp String và gán nó cho biến tham chiếu s.

Cho đến nay, các đối tượng String có vẻ giống như các đối tượng khác. Vậy giờ, hãy cung cấp cho chuỗi

một giá trị:

```
s = "abcdef";
```

(Như bạn sẽ sớm tìm ra, hai dòng mã này không hoàn toàn giống như chúng có vẻ, vì vậy hãy chú ý theo dõi.)

Hóa ra lớp String có khoảng một zillion constructor, vì vậy bạn có thể sử dụng một phím tắt hiệu quả hơn:

```
String s = new String ("abcdef");
```

Và điều này thậm chí còn ngắn gọn hơn:

```
Chuỗi s = "abcdef";
```

Có một số khác biệt nhỏ giữa các tùy chọn này mà chúng ta sẽ thảo luận sau, nhưng điểm chung của chúng là chúng đều tạo một đối tượng String mới , với giá trị là "abcdef" và gán nó cho một biến tham chiếu s. Bây giờ, giả sử bạn muốn tham chiếu thứ hai đến đối tượng String được tham chiếu bởi s:

```
Chuỗi s2 = s; // tham chiếu s2 đến cùng một chuỗi với s
```

Càng xa càng tốt. Các đối tượng chuỗi dường như hoạt động giống như các đối tượng khác, vậy tắt cả những gì rắc rối ở đây là gì? Bất biến! (Tính bất biến là cái quái gì vậy?) Khi bạn đã gán cho Chuỗi một giá trị, giá trị đó không bao giờ có thể thay đổi – nó không thay đổi, cố định, sẽ không nhúc nhích, fini, xong. (Chúng ta sẽ nói về lý do sau; đừng để chúng ta quên.) Tin tốt là mặc dù đối tượng String là bất biến, nhưng biến tham chiếu của nó thì không, vì vậy để tiếp tục với ví dụ trước của chúng ta, hãy xem xét điều này:

```
s = s.concat(" more stuff"); // the concat() method 'appends'  
// a literal to the end
```

Bây giờ, chờ một phút, không phải chúng ta vừa nói rằng các đối tượng Chuỗi là bất biến?

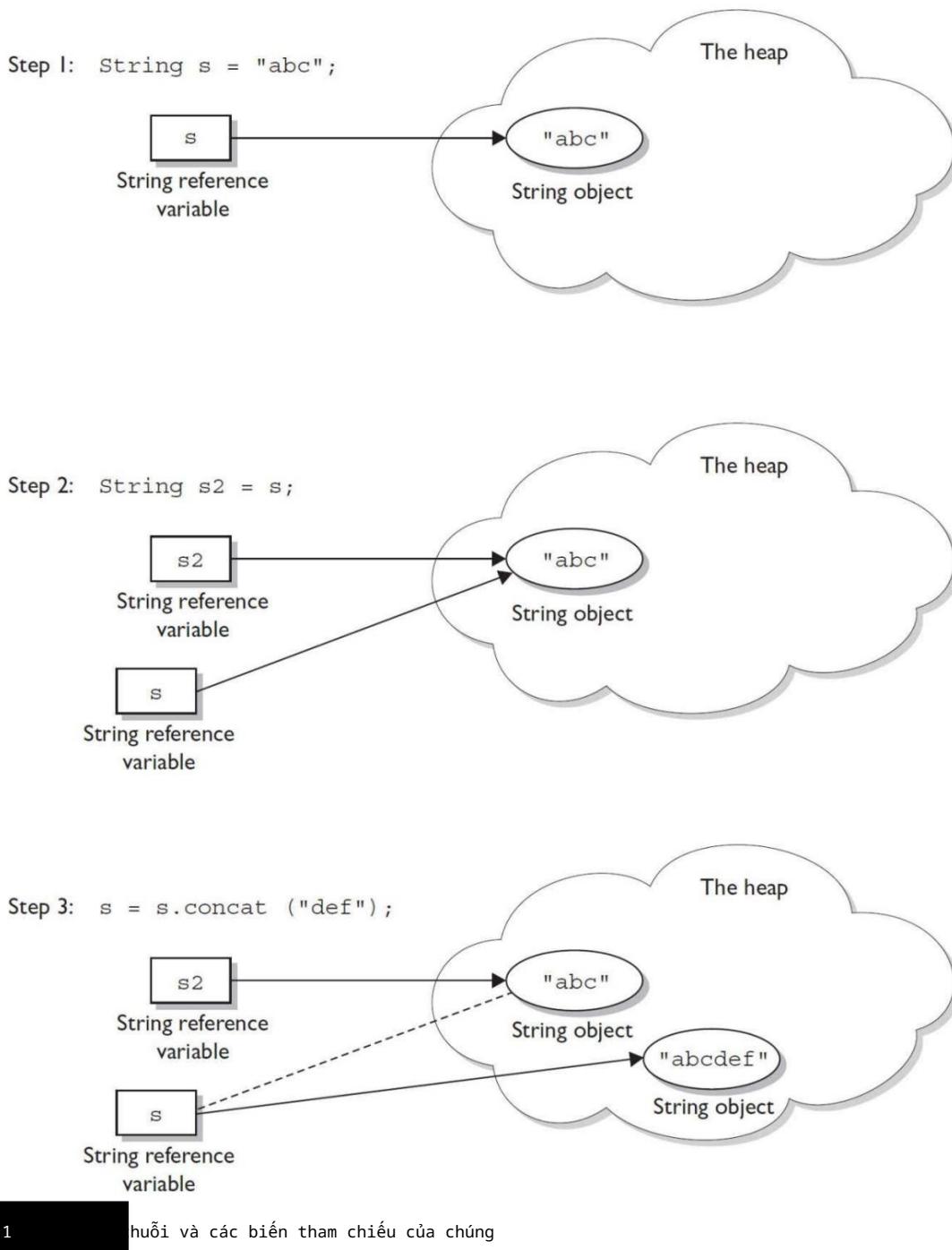
Vì vậy, tắt cả những gì mà tất cả các cuộc nói chuyện "nối vào cuối chuỗi" này là gì? Câu hỏi tuyệt vời: hãy nhìn vào những gì thực sự đã xảy ra.

Máy ảo Java (JVM) đã lấy giá trị của chuỗi s (là "abcdef") và gắn "thêm thứ" vào cuối, cho chúng ta giá trị "abcdef nhiều thứ".
và
Chuỗi này được bắt đầu với giá trị cũ, và kết thúc với giá trị mới. Điều này có nghĩa là nó là một đối tượng Chuỗi mới , đặt cho nó giá trị "abcdef more thing" và làm cho nó tham chiếu đến nó. Tại thời điểm này trong ví dụ của chúng ta, chúng ta có hai đối tượng Chuỗi : đối tượng đầu tiên chúng tôi tạo, với giá trị "abcdef" và đối tượng thứ hai có giá trị "abcdef more thing".

Về mặt kỹ thuật, hiện có ba đối tượng Chuỗi , vì đối số theo nghĩa đen của nhiều thứ hơn ", bản thân " (được tham ¹ chia sẻ và bắt đầu với một biến) là một chuỗi ký tự không đổi và có thể được kết hợp với biến khác để tạo ra một chuỗi mới concat, chỉ tới " abcdef

Điều gì sẽ xảy ra nếu chúng ta không có tầm nhìn xa hoặc may mắn để tạo ra tham chiếu thứ hai biến cho chuỗi "abcdef" trước khi chúng ta gọi `s = s.concat ("thêm thứ") ;?` Trong trường hợp đó, chuỗi ban đầu, không thay đổi có chứa "abcdef" sẽ vẫn tồn tại trong bộ nhớ, nhưng nó sẽ được coi là "bị mất". Không có mã nào trong chương trình của chúng tôi có bất kỳ cách nào để tham chiếu nó – chúng tôi sẽ mất. Tuy nhiên, lưu ý rằng chuỗi "abcdef" ban đầu không thay đổi (nó không thể, hãy nhớ; nó không thay đổi); chỉ có biến tham chiếu `s` được thay đổi để nó tham chiếu đến một chuỗi khác.

[Hình 6-1](#) cho thấy những gì xảy ra trên heap khi bạn gán lại một tham chiếu Biến đổi. Lưu ý rằng đường đứt nét biểu thị một tham chiếu đã bị xóa.



Để xem lại ví dụ đầu tiên của chúng tôi:

```

String s = "abcdef";      // create a new String object, with
                         // value "abcdef", refer s to it
String s2 = s;           // create a 2nd reference variable
                         // referring to the same String

// create a new String object, with value "abcdef more stuff",
// refer s to it. (Change s's reference from the old String
// to the new String.) (Remember s2 is still referring to
// the original "abcdef" String.)

s = s.concat(" more stuff");

```

Hãy xem một ví dụ khác:

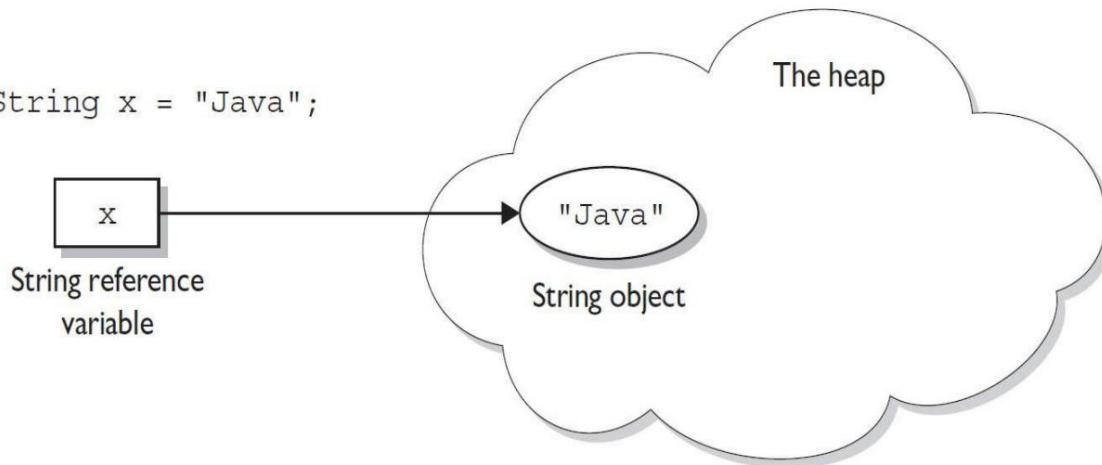
```

String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x); // the output is "x = Java"

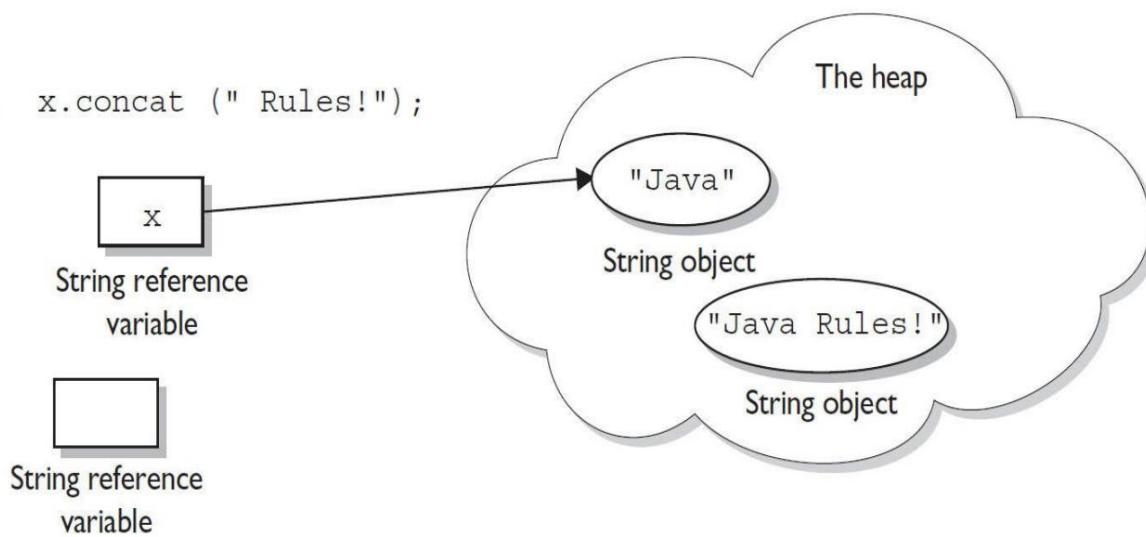
```

Dòng đầu tiên rất đơn giản: Tạo một đối tượng String mới , đặt cho nó giá trị "Java" và tham chiếu x đến nó. Tiếp theo, JVM tạo một đối tượng Chuỗi thứ hai với giá trị "Quy tắc Java!" nhưng không có gì đè cập đến nó. Đối tượng String thứ hai bị mất ngay lập tức; bạn không thể đạt được nó. Biến tham chiếu x vẫn tham chiếu đến Chuỗi ban đầu với giá trị "Java". [Hình 6-2](#) cho thấy việc tạo một Chuỗi mà không chỉ định một tham chiếu cho nó.

Step 1: String x = "Java";



Step 2: x.concat (" Rules!");



Notice that no reference
variable is created to access
the "Java Rules!" String.

HÌNH 6-2 Bi tượng chuỗi bị bỏ qua khi tạo.

Hãy mở rộng ví dụ hiện tại này. Chúng tôi bắt đầu với

```
String x = "Java";
x.concat(" Rules!");
System.out.println("x = " + x); // the output is: x = Java
```

Bây giờ chúng ta hãy thêm

```
x.toUpperCase();
System.out.println("x = " + x); // the output is still:
// x = Java
```

(Chúng tôi thực sự vừa tạo một đối tượng Chuỗi mới với giá trị "JAVA", nhưng nó đã bị mất và x vẫn tham chiếu đến chuỗi không thay đổi ban đầu "Java".) Làm thế nào về việc thêm điều này:

```
x.replace('a', 'X');
System.out.println("x = " + x); // the output is still: x = Java
```

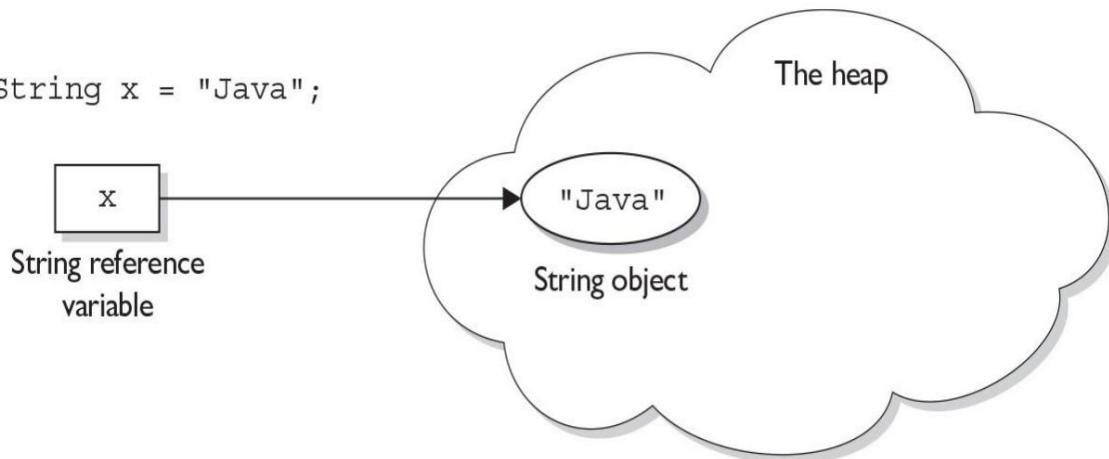
Bạn có thể xác định những gì đã xảy ra? JVM đã tạo thêm một chuỗi mới khác đối tượng, với giá trị "JXvX", (thay thế a bằng X), nhưng một lần nữa Chuỗi mới này bị mất, để lại x để chỉ đối tượng Chuỗi không thay đổi và không thể thay đổi ban đầu, với giá trị "Java". Trong tất cả những trường hợp này, chúng tôi đã gọi các phương thức chuỗi khác nhau để tạo một Chuỗi mới bằng cách thay đổi một Chuỗi hiện có, nhưng chúng tôi chưa bao giờ gán Chuỗi mới được tạo cho một biến tham chiếu.

Nhưng chúng ta có thể đặt một vòng quay nhỏ vào ví dụ trước:

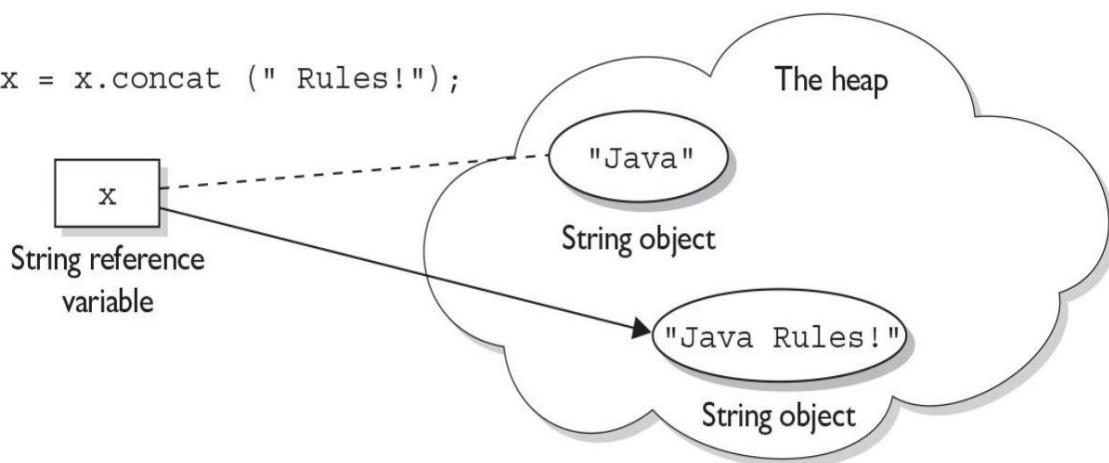
```
String x = "Java";
x = x.concat(" Rules!"); // assign new string to x
System.out.println("x = " + x); // output: x = Java Rules!
```

Lần này, khi JVM chạy dòng thứ hai, một đối tượng Chuỗi mới được tạo với giá trị "Quy tắc Java!" Và x được đặt để tham chiếu đến nó. Nhưng chờ đã. còn nữa – bây giờ đối tượng String ban đầu, "Java", đã bị mất và không ai đề cập đến nó. Vì vậy, trong cả hai ví dụ, chúng tôi đã tạo hai đối tượng Chuỗi và chỉ một biến tham chiếu, vì vậy một trong hai đối tượng Chuỗi đã bị bỏ trống. (Xem [Hình 6-3](#) để mô tả bằng hình ảnh về câu chuyện buồn này.)

Step 1: String x = "Java";



Step 2: x = x.concat (" Rules!");



Notice in step 2 that there is no valid reference to the "Java" String; that object has been "abandoned" and a new object created.

HÌNH 6-3 đối tượng Chuỗi cũ đang bị bỏ rơi. Đường đứt nét biểu thị một tham chiếu đã bị xóa.

Hãy lấy ví dụ này xa hơn một chút:

```

String x = "Java";
x = x.concat(" Rules!");
System.out.println("x = " + x);      // output: x = Java Rules!

x.toLowerCase();                  // no assignment, create a
                                  // new, abandoned String

System.out.println("x = " + x);      // no assignment, the output
                                  // is still: x = Java Rules!

x = x.toLowerCase();              // create a new String,
                                  // assigned to x
System.out.println("x = " + x);      // the assignment causes the
                                  // output: x = java rules!

```

Phản thảo luận trước chứa các chìa khóa để hiểu tính bất biến của chuỗi Java.

Nếu bạn thực sự, thực sự nắm được các ví dụ và sơ đồ, lùi và tiến, bạn sẽ trả lời đúng 80% câu hỏi Chuỗi trong bài kiểm tra.

Chúng tôi sẽ trình bày chi tiết hơn về chuỗi tiếp theo, nhưng đừng nhầm lẫn – về mặt hiệu quả cho tiền của bạn, những gì chúng tôi đã đề cập cho đến nay là phần quan trọng nhất của việc hiểu cách các đối tượng chuỗi hoạt động trong Java.

Chúng tôi sẽ kết thúc phần này bằng cách trình bày một ví dụ về loại câu hỏi Chuỗi mà quái mà bạn có thể mong đợi gặp trong kỳ thi. Hãy dành thời gian để giải quyết nó trên giấy. (Gợi ý: cố gắng theo dõi xem có bao nhiêu đối tượng và biến tham chiếu, và những đối tượng nào tham chiếu đến cái nào.)

```

String s1 = "spring ";
String s2 = s1 + "summer ";
s1.concat("fall ");
s2.concat(s1);
s1 += "winter ";
System.out.println(s1 + " " + s2);

```

Đầu ra là gì? Đối với tín dụng bổ sung, có bao nhiêu đối tượng Chuỗi và bao nhiêu các biến tham chiếu đã được tạo trước câu lệnh println ?

Trả lời: Kết quả của đoạn mã này là xuân đông xuân hè.

Có hai biến tham chiếu: s1 và s2. Có tổng cộng tám đối tượng Chuỗi được tạo như sau: "spring ", "summer" (mắt), "spring summer ", "fall" (mắt), "spring fall" (mắt), "spring summer spring" (mắt) , "Winter" (mắt đi), "spring đông" (lúc này "xuân" đã mắt). Chỉ có hai trong số tám đối tượng Chuỗi không bị mắt trong quá trình này.

Thông tin quan trọng về chuỗi và bộ nhớ

Trong phần này, chúng ta sẽ thảo luận về cách Java xử lý các đối tượng Chuỗi trong bộ nhớ và một số lý do đằng sau những hành vi này.

Một trong những mục tiêu chính của bất kỳ ngôn ngữ lập trình tốt nào là sử dụng hiệu quả bộ nhớ. Khi một ứng dụng phát triển, việc các ký tự chuỗi chiếm một lượng lớn bộ nhớ của chương trình rất phổ biến và thường có rất nhiều dự phòng trong vũ trụ các ký tự Chuỗi cho một chương trình. Để làm cho Java hiệu quả hơn về bộ nhớ, JVM dành ra một vùng bộ nhớ đặc biệt được gọi là nhom hằng chuỗi. Khi trình biên dịch gặp một chuỗi ký tự, nó sẽ kiểm tra nhom để xem liệu một Chuỗi giống hệt đã tồn tại hay chưa. Nếu tìm thấy một kết quả phù hợp, tham chiếu đến ký tự mới sẽ được chuyển hướng đến Chuỗi hiện có và không có đối tượng ký tự Chuỗi mới nào được tạo. (Chuỗi hiện tại chỉ đơn giản là có một tham chiếu bổ sung.) Nay giờ bạn có thể bắt đầu hiểu tại sao việc tạo các đối tượng String là bất biến lại là một ý tưởng hay. Nếu một số biến tham chiếu tham chiếu đến cùng một Chuỗi mà thậm chí không biết nó, sẽ rất tệ nếu bất kỳ biến nào trong số chúng có thể thay đổi giá trị của Chuỗi .

Bạn có thể nói, "Ô, vậy là tốt và tốt, nhưng điều gì sẽ xảy ra nếu ai đó ghi đè chức năng của lớp String ; điều đó có thể gây ra sự cố trong nhóm không?" Đó là một trong những lý do chính mà lớp String được đánh dấu là cuối cùng. Không ai có thể ghi đè các hành vi của bất kỳ phương thức String nào , vì vậy bạn có thể yên tâm rằng các đối tượng String mà bạn đang dựa vào là bất biến, trên thực tế, sẽ là bất biến.

Tạo chuỗi mới Trước đó, chúng

tôi đã hứa sẽ nói nhiều hơn về sự khác biệt tinh tế giữa các phương pháp tạo chuỗi khác nhau. Hãy xem xét một ví dụ về cách một Chuỗi có thể được tạo ra và giả sử thêm rằng không có đối tượng Chuỗi nào khác tồn tại trong nhóm. Trong trường hợp đơn giản này, "abc" sẽ nằm trong nhóm và sẽ tham chiếu đến nó:

```
String s = "abc";      // creates one String object and one  
                      // reference variable
```

Trong trường hợp tiếp theo, bởi vì chúng tôi đã sử dụng từ khóa mới , Java sẽ tạo một đối tượng String mới trong bộ nhớ bình thường (không chung) và sẽ tham chiếu đến nó. Ngoài ra, chữ "abc" sẽ được đặt trong pool:

Các phương thức quan trọng trong lớp chuỗi

Các phương pháp sau đây là một số phương pháp thường được sử dụng hơn trong Lớp chuỗi và chúng cũng là những lớp bạn có khả năng gặp phải nhất trên thi.

- `charAt ()` Trả về ký tự nằm tại chỉ mục được chỉ định `concat ()` Nối một chuỗi vào cuối chuỗi khác (+ cũng hoạt động) `equalsIgnoreCase ()` Xác định sự bằng nhau của hai chuỗi, bỏ qua độ dài chữ hoa chữ thường () Trả về số ký tự trong một chuỗi `Replace ()` Thay thế các lần xuất hiện của một ký tự bằng một chuỗi ký tự mới () Trả về một phần của chuỗi `toLowerCase ()` Trả về một chuỗi, với các ký tự viết hoa được chuyển đổi thành chữ thường
-
- `toString ()` Trả về giá trị của một chuỗi
- `toUpperCase ()` Trả về một chuỗi, với các ký tự chữ thường được chuyển đổi thành chữ hoa `trim ()` Loại bỏ khoảng trắng khỏi cả hai đầu của một chuỗi
-

Hãy xem xét các phương pháp này chi tiết hơn.

```
public char charAt (int index)
```

Phương thức này trả về ký tự nằm tại chỉ mục được chỉ định của chuỗi .

Hãy nhớ rằng, chỉ mục chuỗi không dựa trên số 0 – đây là một ví dụ:

```
String x = "airplane";
System.out.println( x.charAt(2) );           // output is 'r'
```

```
public String concat (String s)
```

Phương thức này trả về một chuỗi có giá trị của Chuỗi được truyền vào phương thức được nối vào cuối Chuỗi được sử dụng để gọi phương thức – đây là một ví dụ:

```
String x = "taxi";
System.out.println( x.concat(" cab") ); // output is "taxi cab"
```

Các toán tử + và += được nạp chồng thực hiện các chức năng tương tự như phương thức concat () – đây là một ví dụ:

```
String x = "library";
System.out.println( x + " card"); // output is "library card"

String x = "Atlantic";
x+= " ocean";
System.out.println( x ); // output is "Atlantic ocean"
```

Trong ví dụ trước về "Đại Tây Dương" , hãy lưu ý rằng giá trị của x thực sự đã thay đổi! Hãy nhớ rằng toán tử += là một toán tử gán, vì vậy dòng 2 thực sự đang tạo một chuỗi mới, "Đại Tây Dương" và gán nó cho biến x . Sau khi dòng 2 thực thi, chuỗi ban đầu x tham chiếu đến, "Atlantic" , bị bỏ qua.

```
public boolean equalsIgnoreCase (Chuỗi s)
```

Phương thức này trả về một giá trị boolean (đúng hoặc sai) tùy thuộc vào việc giá trị của Chuỗi trong đối số có giống với giá trị của Chuỗi được sử dụng để gọi phương thức hay không. Phương thức này sẽ trả về true ngay cả khi các ký tự trong các đối tượng Chuỗi đang được so sánh có các trường hợp khác nhau – đây là một ví dụ:

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT")); // is "true"
System.out.println( x.equalsIgnoreCase("tixe")); // is "false"
```

```
public int length ()
```

Phương thức này trả về độ dài của Chuỗi được sử dụng để gọi phương thức – đây là một ví dụ:

```
String x = "01234567";
System.out.println( x.length() ); // returns "8"
```



Mảng có một thuộc tính (không phải là một phương thức) được gọi là độ dài. Bạn có thể gặp các câu hỏi trong bài kiểm tra có gắng sử dụng phương thức length () trên một mảng hoặc cố gắng sử dụng thuộc tính length trên một Chuỗi. Cả hai

gây ra lỗi trình biên dịch - hãy xem xét những điều này, ví dụ:

```
String x = "test";
System.out.println( x.length() );      // compiler error
```

và

```
String[] x = new String[3];
System.out.println( x.length() );      // compiler error
```

chuỗi công khai thay thế (char cũ, char mới)

Phương thức này trả về một Chuỗi có giá trị của Chuỗi được sử dụng để gọi phương thức, nhưng được cập nhật để bất kỳ sự xuất hiện nào của char trong đối số đầu tiên được thay thế bằng char trong đối số thứ hai – đây là một ví dụ:

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') );    // output is "oXoXoXoX"
```

công khai chuỗi con (int begin) và chuỗi con công khai (int begin, int end)

Phương thức substring () được sử dụng để trả về một phần (hoặc chuỗi con) của Chuỗi được sử dụng để gọi phương thức. Đối số đầu tiên đại diện cho vị trí bắt đầu (dựa trên không) của chuỗi con. Nếu lệnh gọi chỉ có một đối số, chuỗi con được trả về sẽ bao gồm các ký tự ở cuối Chuỗi ban đầu . Nếu lệnh gọi có hai đối số, chuỗi con được trả về sẽ kết thúc bằng ký tự nằm ở vị trí thứ n của Chuỗi ban đầu trong đó n là đối số thứ hai.

Thật không may, đối số kết thúc không dựa trên số 0, vì vậy nếu đối số thứ hai là 7, ký tự cuối cùng trong Chuỗi được trả về sẽ ở vị trí 7 của Chuỗi ban đầu , là chỉ số 6 (ouch). Hãy xem một số ví dụ:

```
String x = "0123456789";                      // as if by magic, the value of
each                                         // char is the same as its
index!
System.out.println( x.substring(5) );    // output is "56789"
System.out.println( x.substring(5, 8));   // output is "567"
```

Ví dụ đầu tiên phải dễ dàng: bắt đầu ở chỉ mục 5 và trả về phần còn lại của Chuỗi. Ví dụ thứ hai nên được đọc như sau: bắt đầu ở chỉ mục 5 và

trả về các ký tự lên đến và bao gồm cả vị trí thứ 8 (chỉ số 7).

```
public String toLowerCase ()
```

Chuyển đổi tất cả các ký tự của một Chuỗi thành chữ thường – đây là một ví dụ:

```
String x = "A New Moon";
System.out.println( x.toLowerCase() ); // output is "a new moon"
```

```
public String toString ()
```

Phương thức này trả về giá trị của Chuỗi được sử dụng để gọi phương thức. Gì?

Tại sao bạn cần một phương pháp dường như "không làm gì cả" như vậy? Tất cả các đối tượng trong Java phải có một phương thức `toString()`, phương thức này thường trả về một Chuỗi mà theo một cách nào đó có ý nghĩa mô tả đối tượng được đề cập. Trong trường hợp đối tượng `String`, cách nào có ý nghĩa hơn giá trị của `String`? Để nhất quán, đây là một ví dụ:

```
String x = "big surprise";
System.out.println( x.toString() ); // output? [reader's exercise :-) ]
```

```
public String toUpperCase ()
```

Chuyển đổi tất cả các ký tự của một Chuỗi thành chữ hoa – đây là một ví dụ:

```
String x = "A New Moon";
System.out.println( x.toUpperCase() ); // output is "A NEW MOON"
```

```
public String trim ()
```

Phương thức này trả về một Chuỗi có giá trị là Chuỗi được sử dụng để gọi phương thức, nhưng với bất kỳ khoảng trắng đầu hoặc cuối đã bị xóa – đây là một ví dụ:

```
String x = " hi ";
System.out.println( x + "t" ); // output is " hi t"
System.out.println( x.trim() + "t" ); // output is "hit"
```

Lớp `StringBuilder`

Lớp `java.lang.StringBuilder` nên được sử dụng khi bạn phải thực hiện nhiều sửa đổi đối với các chuỗi ký tự. Như đã thảo luận trong phần trước,

Đối tượng chuỗi là bất biến, vì vậy nếu bạn chọn thực hiện nhiều thao tác với đối tượng Chuỗi , bạn sẽ kết thúc với rất nhiều đối tượng Chuỗi bị bỏ rơi trong nhóm Chuỗi . (Ngay cả trong những ngày có nhiều GB RAM, không nên lãng phí bộ nhớ quý giá cho các đối tượng nhóm String bị loại bỏ .) Mặt khác, các đối tượng kiểu StringBuilder có thể được sửa đổi nhiều lần mà không để lại lượng lớn rác bị loại bỏ. Đối tượng chuỗi .



Cách sử dụng phổ biến cho StringBuilders là I / O tệp khi các luồng đầu vào lớn, luôn thay đổi đang được chương trình xử lý. Trong những trường hợp này, các khối ký tự lớn được xử lý dưới dạng đơn vị và các đối tượng StringBuilder là cách lý tưởng để xử lý một khối dữ liệu, chuyển nó và sau đó sử dụng lại cùng một bộ nhớ để xử lý khối dữ liệu tiếp theo.

Ưu tiên StringBuilder thành StringBuffer Lớp

StringBuilder đã được thêm vào trong Java 5. Nó có API giống hệt như lớp StringBuffer , ngoại trừ StringBuilder không an toàn cho luồng. Nói cách khác, các phương thức của nó không đồng bộ. Oracle khuyến nghị bạn nên sử dụng StringBuilder thay vì StringBuffer bất cứ khi nào có thể, vì StringBuilder sẽ chạy nhanh hơn (và có thể nhảy cao hơn). Vì vậy, ngoài việc đồng bộ hóa, bất cứ điều gì chúng ta nói về các phương thức của StringBuilder đều đúng với các phương thức của StringBuffer và ngược lại. Điều đó nói rằng, đối với kỳ thi OCA 8, StringBuffer không được kiểm tra.

Sử dụng StringBuilder (và đây là lần cuối cùng chúng tôi sẽ nói điều này: StringBuffer)

Trong phần trước, bạn đã biết cách bài kiểm tra có thể kiểm tra sự hiểu biết của bạn về Tính bất biến của chuỗi với các đoạn mã như thế này:

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x);      // output is "x = abc"
```

Bởi vì không có phép gán mới nào được thực hiện, đối tượng String mới được tạo bằng phương thức concat () đã bị loại bỏ ngay lập tức. Bạn cũng đã thấy các ví dụ như thế này:

```

String x = "abc";
x = x.concat("def");
System.out.println("x = " + x);      // output is "x = abcdef"

```

Chúng tôi đã có một Chuỗi mới tuyệt vời trong thỏa thuận, nhưng nhược điểm là Chuỗi cũ "abc" đã bị mất trong nhóm Chuỗi , do đó lãng phí bộ nhớ. Nếu chúng ta đang sử dụng StringBuilder thay vì String, mã sẽ trông như thế này:

```

StringBuilder sb = new StringBuilder("abc");
sb.append("def");
System.out.println("sb = " + sb);      // output is "sb = abcdef"

```

Tất cả các phương thức StringBuilder mà chúng ta sẽ thảo luận đều hoạt động dựa trên giá trị của đối tượng StringBuilder gọi phương thức. Vì vậy, một cuộc gọi đến sb.append ("def"); thực sự đang thêm "def" vào chính nó (StringBuilder sb). Trên thực tế, các lệnh gọi phương thức này có thể được liên kết với nhau – đây là một ví dụ:

```

StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb );           // output is "fed---cba"

```

Lưu ý rằng trong hai ví dụ trước, chỉ có một lệnh gọi mới, vì vậy trong mỗi ví dụ, chúng tôi sẽ không tạo thêm bất kỳ đối tượng nào. Mỗi ví dụ chỉ cần một đối tượng StringBuilder duy nhất để thực thi.



Cho đến nay, chúng ta đã thấy StringBuilders được xây dựng với một đối số chỉ định một giá trị ban đầu. StringBuilders cũng có thể được xây dựng trống và chúng cũng có thể được xây dựng với một kích thước cụ thể hoặc chính thức hơn, là một "công suất". Đối với bài kiểm tra, có ba cách để tạo một StringBuilder mới:

1. new StringBuilder(); // default cap. = 16 chars
2. new StringBuilder("ab"); // cap. = 16 + arg's length
3. new StringBuilder(x); // capacity = x (an integer)

Hai cách phổ biến nhất để làm việc với StringBuilders là thông qua phương thức append () hoặc phương thức insert () . Về khả năng của StringBuilder , có ba quy tắc cần ghi nhớ khi thêm và chèn:

Nếu một append () phát triển một StringBuilder vượt quá dung lượng của nó, dung lượng sẽ được cập nhật tự động.

Nếu một insert () bắt đầu trong dung lượng của StringBuilder nhưng kết thúc sau dung lượng hiện tại, dung lượng được cập nhật tự động.

Nếu một insert () cố gắng bắt đầu ở một chỉ mục sau độ dài hiện tại của StringBuilder , thì một ngoại lệ sẽ được ném ra.

Các phương thức quan trọng trong lớp StringBuilder

Lớp StringBuilder có một phương thức zillion. Sau đây là những phương pháp mà bạn có nhiều khả năng sẽ sử dụng trong thế giới thực và vui vẻ là những phương pháp bạn có nhiều khả năng tìm thấy nhất trong bài kiểm tra.

`public StringBuilder append (String s)`

Như bạn đã thấy trước đó, phương thức này sẽ cập nhật giá trị của đối tượng đã gọi phương thức, cho dù giá trị trả về có được gán cho một biến hay không.

Các phiên bản của phương thức quá tải này sẽ có nhiều đối số khác nhau, bao gồm boolean, char, double, float, int, long và các đối số khác, nhưng đối số có nhiều khả năng được sử dụng trong bài kiểm tra sẽ là đối số Chuỗi – ví dụ:

```
StringBuilder sb = new StringBuilder("set ");
sb.append("point");
System.out.println(sb);           // output is "set point"
StringBuilder sb2 = new StringBuilder("pi = ");
sb2.append(3.14159f);
System.out.println(sb2);         // output is "pi = 3.14159"
```

`public StringBuilder delete (int start, int end)`

Phương thức này sửa đổi giá trị của đối tượng StringBuilder được sử dụng để gọi nó.

Chỉ mục bắt đầu của chuỗi con cần loại bỏ được xác định bởi đối số đầu tiên (dựa trên 0) và chỉ số kết thúc của chuỗi con cần xóa được xác định bởi đối số thứ hai (nhưng nó là đối số một)! Nghiên cứu kỹ ví dụ sau:

```
StringBuilder sb = new StringBuilder("0123456789");
System.out.println(sb.delete(4,6));           // output is "01236789"
```



Bài kiểm tra có thẻ sẽ kiểm tra kiến thức của bạn về sự khác biệt giữa các đối tượng String và StringBuilder. Vì các đối tượng StringBuilder có thể thay đổi, đoạn mã sau sẽ hoạt động khác với đoạn mã tương tự sử dụng đối tượng String :

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def");
System.out.println( sb );
```

Trong trường hợp này, kết quả đầu ra sẽ là: "abcdef"

```
public StringBuilder insert (int offset, String s)
```

Phương thức này cập nhật giá trị của đối tượng StringBuilder đã gọi phương thức. Chuỗi được truyền vào đối số thứ hai được chèn vào StringBuilder bắt đầu từ vị trí bù được biểu thị bởi đối số đầu tiên (độ lệch dựa trên 0). Một lần nữa, các loại dữ liệu khác có thể được truyền vào thông qua đối số thứ hai (boolean, char, double, float, int, long, v.v.), nhưng đối số String là đối số mà bạn có nhiều khả năng nhìn thấy nhất:

```
StringBuilder sb = new StringBuilder("01234567");
sb.insert(4, "---");
System.out.println( sb );           // output is "0123---4567"
```

```
public StringBuilder reverse ()
```

Phương thức này cập nhật giá trị của đối tượng StringBuilder đã gọi phương thức. Khi được gọi, các ký tự trong StringBuilder được đảo ngược- ký tự đầu tiên trở thành ký tự cuối cùng, ký tự thứ hai trở thành ký tự thứ hai đến ký tự cuối cùng, v.v.

```
StringBuilder sb = new StringBuilder("A man a plan a canal Panama");
sb.reverse();
System.out.println(sb); // output: "amanap lanac a nalp a nam A"
```

```
public String toString ()
```

Phương thức này trả về giá trị của đối tượng `StringBuilder` đã gọi phương thức dưới dạng một Chuỗi:

```
StringBuilder sb = new StringBuilder("test string");
System.out.println( sb.toString() ); // output is "test string"
```

Đó là nó cho `StringBuilders`. Nếu bạn chỉ bỏ sót một điều từ phần này, đó là không giống như các đối tượng `String`, các đối tượng `StringBuilder` có thể được thay đổi.



Nhiều câu hỏi kiểm tra bao gồm các chủ đề của chương này sử dụng một chút cú pháp Java phức tạp, được gọi là "các phương thức chuỗi". Một câu lệnh với các phương thức chuỗi có dạng chung sau:

```
result = method1 (). method2 (). method3 ();
```

Về lý thuyết, bất kỳ phương pháp nào cũng có thể được xâu chuỗi theo kiểu này, mặc dù thông thường bạn sẽ không thấy nhiều hơn ba. Dưới đây là cách giải mã các "phím tắt Java tiện dụng" này khi bạn gặp phải chúng:

1. Xác định lệnh gọi phương thức ngoài cùng bên trái sẽ trả về gì (chúng ta hãy gọi nó là x).
2. Sử dụng x làm đối tượng gọi phương thức thứ hai (từ bên trái). Nếu chỉ có hai phương thức được xâu chuỗi, kết quả của lần gọi phương thức thứ hai là kết quả của biểu thức.
3. Nếu có phương thức thứ ba, kết quả của lệnh gọi phương thức thứ hai được sử dụng để gọi phương thức thứ ba, có kết quả là kết quả của biểu thức - ví dụ:

```
String x = "abc";
String y = x.concat("def").toUpperCase().replace('C', 'x'); //chained methods
System.out.println("y = " + y); // result is "y = ABxDEF"
```

Hãy nhìn vào những gì đã xảy ra. Chữ def theo nghĩa đen đã được nối với abc, tạo ra một Chuỗi trung gian, tạm thời (sắp bị mất), với giá trị abcdef. Phương thức `toUpperCase()` được gọi trên Chuỗi này, nó tạo ra một Chuỗi tạm thời mới (sắp bị mất) với giá trị ABCDEF. Phương thức `Replace()` sau đó được gọi trên đối tượng Chuỗi tạm thời này, đối tượng này tạo ra một Chuỗi cuối cùng với giá trị ABxDEF và gọi y đến nó.

MỤC TIÊU XÁC NHẬN

Làm việc với dữ liệu lịch (Mục tiêu 9.3 của OCA)

9.3 Tạo và thao tác dữ liệu lịch sử dụng các lớp sau: `java.time.LocalDateTime`, `java.time.LocalDate`, `java.time.LocalTime`, `java.time.format.DateTimeFormatter`, `java.time.Period`

Java 8 đã giới thiệu một bộ sưu tập lớn (argh) các gói mới liên quan đến việc làm việc với lịch, ngày và giờ. Những người sáng tạo OCA 8 đã chọn đưa kiến thức về một tập hợp con của các gói và lớp này làm mục tiêu kỳ thi. Nếu bạn hiểu các lớp có trong mục tiêu bài kiểm tra, bạn sẽ có phần giới thiệu tốt về toàn bộ chủ đề lịch / ngày / giờ. Khi chúng tôi làm việc trong phần này, chúng tôi sẽ sử dụng cụm từ "đối tượng lịch", cụm từ mà chúng tôi sử dụng để chỉ các đối tượng của một trong một số loại lớp liên quan đến lịch mà chúng tôi đang đề cập. Vì vậy, "đối tượng lịch" là một thuật ngữ ô được tạo thành. Dưới đây là bản tóm tắt về năm lớp liên quan đến lịch mà chúng tôi sẽ nghiên cứu, cùng với một giao diện có kích thước lớn:

- `java.time.LocalDateTime` Lớp này được sử dụng để tạo các đối tượng bất biến, mỗi đối tượng đại diện cho một ngày và giờ cụ thể. Ngoài ra, lớp này cung cấp các phương thức có thể thao tác các giá trị của các đối tượng ngày / giờ được tạo và gán chúng cho các đối tượng bất biến mới. Các đối tượng `LocalDateTime` chứa CẢ thông tin về ngày, tháng, năm VÀ về giờ, phút, giây và phần nhỏ của giây.
- `java.time.LocalDate` Lớp này được sử dụng để tạo các đối tượng bất biến, mỗi đối tượng đại diện cho một ngày cụ thể. Ngoài ra, lớp này cung cấp các phương thức có thể thao tác các giá trị của các đối tượng ngày tháng được tạo và gán chúng cho các đối tượng bất biến mới. Đối tượng `LocalDate` chỉ chính xác đến ngày. Giờ, phút và giây không phải là một phần của đối tượng `LocalDate`. `java.time.LocalTime` Lớp này được sử dụng để tạo các đối tượng bất biến, mỗi đối tượng đại diện cho một thời gian cụ thể. Ngoài ra, lớp này cung cấp các phương thức có thể thao tác các giá trị của các đối tượng thời gian được tạo và gán chúng cho các đối tượng bất biến mới. Các đối tượng `LocalTime` chỉ tham chiếu đến

giờ, phút, giây và phần nhỏ của giây. Ngày, tháng và năm không phải là một phần của các đối tượng LocalTime . java.time.format.DateTimeFormatter Lớp

- này được sử dụng bởi các lớp vừa mô tả để định dạng các đối tượng ngày / giờ cho đầu ra và phân tích cú pháp các chuỗi đầu vào và chuyển đổi chúng thành các đối tượng ngày / giờ. Các đối tượng DateTimeFormatter cũng là bất biến.
- java.time.Period Lớp này được sử dụng để tạo các đối tượng bất biến đại diện cho một khoảng thời gian, ví dụ: "một năm, hai tháng và ba ngày." Lớp này hoạt động theo năm, tháng và ngày. Nếu bạn muốn biểu diễn các phần thời gian theo giá số nhỏ hơn một ngày (ví dụ: giờ và phút), bạn có thể sử dụng lớp java.time.Duration , nhưng Thời lượng không có trong bài kiểm tra.
- java.time.temporal.TemporalAmount Giao diện này được thực hiện bởi lớp Giai đoạn . Khi bạn sử dụng các đối tượng Giai đoạn để thao tác (xem phần sau), các đối tượng lịch, bạn sẽ thường sử dụng các phương thức lấy các đối tượng triển khai TemporalAmount. Nói chung, khi bạn sử dụng Java API ngày càng nhiều, bạn nên tìm hiểu lớp nào triển khai giao diện nào; đây là một cách quan trọng để tìm hiểu cách các lớp trong các gói phức tạp tương tác với nhau.

Tính bất biến

Có một vài chủ đề lặp lại trong các định nghĩa trước đây. Đầu tiên, hãy lưu ý rằng hầu hết các đối tượng liên quan đến lịch mà bạn sẽ tạo là bất biến. Cũng giống như các đối tượng Chuỗi ! Vì vậy, khi chúng tôi nói rằng chúng tôi sẽ "thao tác" một đối tượng lịch, ý chúng tôi "thực sự" là chúng tôi sẽ gọi một phương thức trên một đối tượng lịch và chúng tôi sẽ trả về một đối tượng lịch mới đại diện cho kết quả của việc thao tác giá trị của đối tượng lịch gốc. Nhưng giá trị của đối tượng lịch ban đầu không, và không thể thay đổi được. Cũng giống như Strings! Hãy xem một ví dụ:

```
LocalDate date1 = LocalDate.of(2017, 1, 31);
Period period1 = Period.ofMonths(1);
System.out.println(date1);
date1.plus(period1); // new value is lost
System.out.println(date1);
LocalDate date2 = date1.plus(period1); // new value is captured
System.out.println(date2);
```

sản xuất:

2017-01-31

2017-01-31

2017-02-28

Lưu ý rằng việc gọi phương thức cộng trên date1 không thay đổi giá trị của nó, nhưng việc gán kết quả của phương thức cộng cho date2 sẽ thu về một giá trị mới. Mong đợi các câu hỏi kiểm tra sự hiểu biết của bạn về tính bất biến của các đối tượng lịch.

Các lớp nhà máy

Điều tiếp theo cần lưu ý trong danh sách mã trước đó là chúng tôi chưa bao giờ sử dụng từ khóa mới trong mã. Chúng tôi đã không trực tiếp gọi một hàm tạo. Không có lớp nào trong số năm lớp được liệt kê trong mục tiêu 9.3 của OCA 8 có hàm tạo công khai. Thay vào đó, đối với tất cả các lớp này, bạn gọi một phương thức tĩnh công khai trong lớp để tạo một đối tượng mới. Khi bạn đi sâu hơn vào các nghiên cứu của mình về thiết kế OO, bạn sẽ bắt gặp các cụm từ "mô hình nhà máy", "phương pháp nhà máy" và "các lớp nhà máy".

Thông thường, khi một lớp không có hàm tạo công khai và cung cấp ít nhất một phương thức tĩnh công khai có thể tạo các thể hiện mới của lớp, thì lớp đó được gọi là lớp nhà máy và bắt kỳ phương thức nào được gọi để có được một thể hiện mới của lớp được gọi một phương pháp nhà máy. Có nhiều lý do chính đáng để tạo các lớp nhà máy, hầu hết trong số đó nằm ngoài phạm vi của cuốn sách này, nhưng một trong số chúng ta sẽ thảo luận ngay bây giờ. Nếu chúng tôi sử dụng lớp LocalDate làm ví dụ, chúng tôi tìm thấy các phương thức tĩnh sau đây để tạo và trả về một phiên bản mới:

```
from()
now()           // three overloaded methods exist
of()            // two overloaded methods exist
ofEpochDay()
ofYearDay()
parse()          // two overloaded methods exist
```

Vì vậy, chúng tôi có những gì, khoảng mười cách khác nhau để tạo một đối tượng LocalDate mới ? Bằng cách sử dụng các phương thức có tên khác nhau (thay vì sử dụng các hàm tạo được nạp chòng), bản thân các tên phương thức làm cho mã dễ đọc hơn. Rõ ràng hơn là biến thể của LocalDate mà chúng tôi đang thực hiện. Khi bạn sử dụng ngày càng nhiều lớp từ Java API, bạn sẽ phát hiện ra rằng những người tạo API sử dụng các lớp của nhà máy rất nhiều.



Bất cứ khi nào bạn nhìn thấy một câu hỏi kiểm tra liên quan đến ngày hoặc giờ, hãy để ý từ khóa mới. Đây là thông báo của bạn mà mà sẽ không biên dịch:

```
LocalDateTime d1 = new LocalDateTime(); // sẽ không biên dịch
```

Hãy nhớ các lớp ngày và giờ của kỳ thi sử dụng các phương thức gốc để tạo các đối tượng mới.

Sử dụng và thao tác Ngày và Giờ

Bây giờ chúng ta đã biết cách tạo các đối tượng liên quan đến lịch mới, hãy chuyển sang sử dụng và thao tác chúng. (Và bạn biết ý của chúng tôi khi chúng tôi nói "thao tác".) Đoạn mã sau thể hiện một số cách sử dụng phổ biến và các tính năng mạnh mẽ của các lớp liên quan đến lịch Java 8 mới:

```

import java.time.*;
import java.time.format.*;
import java.time.temporal.ChronoUnit;           // not on the exam
                                                // but VERY useful

public class DrWho {
    public static void main(String[] args) {
        DateTimeFormatter f =
            DateTimeFormatter.ofPattern("MMddyyyy");      // describe a format
        LocalDate bday = null;
        try {
            bday = LocalDate.parse(args[0], f);          // verify input date
                                                       // often parse() methods
                                                       // throw exceptions!
        } catch (java.time.DateTimeException e) {
            System.out.println("bad dates Indy");
            System.exit(0);
        }
        System.out.println("your birthday is: " + bday);
        System.out.println("a " + bday.getDayOfWeek());   // useful

        Period p1 = Period.between(bday, LocalDate.now()); // very useful!

        System.out.println("you've lived for: ");
        System.out.print(p1.getDays() + " days, ");          // split up a Period
        System.out.print(p1.getMonths() + " months, ");
        System.out.println(p1.getYears() + " years");

        int yearsOld = p1.getYears();
        if(yearsOld < 0 || yearsOld > 119)
            System.out.println("Wow, are you a time lord?");

        long tDays = bday.until(LocalDate.now(),           // handy method +
                               ChronoUnit.DAYS);          // handy enum
                                                       // = powerful date math

        System.out.println("you've lived for " + tDays
                           + " days, so far");

        System.out.println("you'll reach 30,000 days on "
                           + bday.plusDays(30_000));     // date math

        LocalDate d2000 = LocalDate.of(2_000, 1, 1);       // of() is a
                                                       // commonly used
                                                       // 'factory' method

        Period p2 = Period.between(d2000, LocalDate.now());
        System.out.println("period since Y2K: " + p2);
    }
}

```

Mời chương trình với một ngày sinh nhật có liên quan:

```
java 01201934
```

tạo ra đầu ra (khi chạy vào ngày 13 tháng 1 năm 2017):

```
your birthday is: 1934-01-20
a SATURDAY
you've lived for:
24 days, 11 months, 82 years
you've lived for 30309 days, so far
you'll reach 30000 days on 2016-03-10
period since Y2K: P17Y12D
```

Có rất nhiều điều đang diễn ra ở đây, vì vậy chúng ta hãy đi dạo qua một lượt. Đầu tiên, chúng tôi muốn ngay lập tức nhập ngày sinh của họ dưới dạng mmddyyyy. Chúng tôi sử dụng một đối tượng `DateTimeFormatter` để phân tích cú pháp đối số đầu tiên của ngay lập tức dùng và xác minh rằng đó là ngày hợp lệ của biểu mẫu mà chúng tôi hy vọng. Thông thường trong Java API, các phương thức phân tích cú pháp () có thể đưa ra các ngoại lệ, vì vậy chúng ta phải thực hiện phân tích cú pháp trong một khối try / catch .

Tiếp theo, chúng tôi in ra ngày đã xác minh và thể hiện một chút bằng cách in ra ngày đó xảy ra vào ngày nào trong tuần. Tính toán này sẽ khá phức tạp nếu làm bằng tay!

Tiếp theo, chúng tôi tạo một đối tượng Kỳ biểu thị khoảng thời gian giữa ngày sinh của ngay lập tức và ngày hôm nay, và chúng tôi sử dụng các phương thức `getX ()` khác nhau để liệt kê các chi tiết của đối tượng Kỳ.

Sau khi chắc chắn rằng chúng ta không giao dịch với chia sẻ thời gian, chúng ta sử dụng phương thức `Until ()` và "day" rất mạnh mẽ làm đơn vị thời gian để xác định ngay lập tức đã sống được bao nhiêu ngày. Chúng tôi đã gian lận một chút ở đây và sử dụng enum `ChronoUnit` từ gói `java.time.temporal` . (Mặc dù `ChronoUnit` không có trong kỳ thi, chúng tôi nghĩ rằng nếu bạn thực hiện nhiều phép tính lịch, bạn sẽ sử dụng enum này rất nhiều.)

Tiếp theo, chúng tôi thêm 30.000 ngày vào sinh nhật của ngay lập tức để chúng tôi có thể tính toán ngày ngay lập tức của chúng tôi sẽ sống trong 30.000 ngày. Đó là một bước nhảy ngắn để xem các loại tính toán lịch này sẽ rất mạnh mẽ như thế nào đối với các ứng dụng lập lịch trình, ứng dụng quản lý dự án, lập kế hoạch du lịch, v.v.

Cuối cùng, chúng tôi sử dụng một phương thức gốc phổ biến, `of ()`, để tạo một đối tượng ngày tháng khác (đại diện cho ngày hôm nay), và chúng tôi sử dụng nó kết hợp với phương thức `between ()` rất mạnh để xem nó đã trôi qua bao lâu kể từ ngày 1 tháng 1 năm 2000, Y2K.

Định dạng ngày và giờ

Bây giờ chúng ta hãy chuyển sang định dạng ngày và giờ bằng cách sử dụng lớp `DateTimeFormatter`, vì vậy các đối tượng lịch của bạn sẽ trông sáng bóng khi bạn muốn đưa chúng vào đầu ra của chương trình. Đối với bài kiểm tra, bạn nên biết quy trình hai bước sau để tạo Chuỗi đại diện cho các đối tượng lịch được định dạng tốt:

1. Sử dụng bộ định dạng và mẫu từ danh sách `HUGE` được cung cấp trong lớp `DateTimeFormatter` để tạo đối tượng `DateTimeFormatter`.
2. Trong các lớp `LocalDate`, `LocalDateTime` và `LocalTime`, hãy sử dụng phương thức `format()` với đối tượng `DateTimeFormatter` làm đối số để tạo một Chuỗi có cấu trúc tốt – hoặc sử dụng phương thức `DateTimeFormatter.format()` với đối số lịch để tạo một Chuỗi có cấu trúc đúng. Hãy xem một vài ví dụ:

```
import java.time.*;
import java.time.format.*;
public class NiceDates {
    public static void main(String[] args) {
        DateTimeFormatter f1 =
            DateTimeFormatter.ofPattern("MMM dd, yyyy");
        DateTimeFormatter f2 =
            DateTimeFormatter.ofPattern("E MMM dd, yyyy G");
        DateTimeFormatter tf1 =
            DateTimeFormatter.ofPattern("k:m:s A a");

        LocalDate d = LocalDate.now();
        String s = d.format(f1);           // thus proving that the format()
                                         // method makes String objects
        System.out.println(s);
        System.out.println(d.format(f2));

        LocalTime t = LocalTime.now();
        System.out.println(t.format(tf1));
    }
}
```

mà, khi chúng tôi chạy mã này, tạo ra như sau (đầu ra của bạn sẽ khác nhau):

14 thg 1, 2017

Thứ bảy, ngày 14 tháng 1 năm 2017 sau Công nguyên
14: 17: 9 51429958 CH

Một số mã mẫu mà chúng tôi sử dụng là hiển nhiên (ví dụ: `MMM dd yyyy`), và một số khá tùy ý như "`E`" cho ngày trong tuần hoặc "`k`" cho giờ quân sự. Tất cả

mã có thể được tìm thấy trong API `DateTimeFormatter`.

Thế là đủ về lịch; vào mảng!

MỤC TIÊU XÁC NHẬN

Sử dụng Mảng (Mục tiêu OCA 4.1 và 4.2)

4.1 Khai báo, khởi tạo, khởi tạo và sử dụng mảng một chiều.

4.2 Khai báo, khởi tạo, khởi tạo và sử dụng mảng nhiều chiều.

Mảng là các đối tượng trong Java lưu trữ nhiều biến cùng kiểu.

Mảng có thể chứa các tham chiếu nguyên thủy hoặc đối tượng, như ng bắn thân mảng sẽ luôn là một đối tượng trên heap, ngay cả khi mảng được khai báo là chứa các phần tử nguyên thủy. Nói cách khác, không có cái gọi là mảng nguyên thủy, như ng bạn có thể tạo một mảng nguyên thủy. Đối với mục tiêu này, bạn cần biết ba điều:

- Cách tạo biến tham chiếu mảng (khai báo)
- Cách tạo một đối tượng mảng (cấu trúc)
- Cách điền vào mảng với các phần tử (khởi tạo)

Có một số cách khác nhau để thực hiện mỗi cách này và bạn cần biết về tất cả chúng cho kỳ thi.



Mảng hiệu quả, như ng hầu hết thời gian bạn sẽ muốn sử dụng một trong các loại Bộ sưu tập từ `java.util` (bao gồm `HashMap`, `ArrayList` và `TreeSet`).

Các lớp bộ sưu tập cung cấp nhiều cách linh hoạt hơn để truy cập một đối tượng (để chèn, xóa, v.v.) và không giống như mảng, chúng có thể mở rộng hoặc hợp đồng động khi bạn thêm hoặc xóa các phần tử (chúng thực sự là mảng được quản lý, vì chúng sử dụng các mảng phía sau những cảnh). Có một loại Bộ sưu tập cho nhiều nhu cầu. Bạn có cần phân loại nhanh không? Một nhóm các đối tượng không có bản sao? Một cách để truy cập một cặp tên / giá trị? Một danh sách liên kết? Kỳ thi OCP 8 bao gồm các bộ sưu tập chi tiết hơn.

Khai báo một mảng

Mảng được khai báo bằng cách nêu rõ loại phần tử mà mảng sẽ giữ, có thể

Mảng đư ợc khai báo bằng cách nêu rõ loại phần tử mà mảng sẽ giữ, có thể là một đối tư ợng hoặc một phần tử nguyên thủy, theo sau là dấu ngoặc vuông ở bên trái hoặc bên phải của mã định danh.

Khai báo một mảng các nguyên thủy:

```
int[] key;           // brackets before name (recommended)
int key [] ;        // brackets after name (legal but less readable)
                     // spaces between the name and [] legal, but bad
```

Khai báo một mảng các tham chiếu đối tư ợng:

```
Thread[] threads;   // Recommended
Thread threads[] ; // Legal but less readable
```

Khi khai báo một tham chiếu mảng, bạn phải luôn đặt các dấu ngoặc nhọn của mảng ngay sau kiểu đư ợc khai báo chứ không phải sau mã định danh (tên biến). Bằng cách đó, bất kỳ ai đọc mã đều có thể dễ dàng nhận ra rằng, ví dụ, khóa là một tham chiếu đến một đối tư ợng mảng int chứ không phải một nguyên nguyên int .

Chúng ta cũng có thể khai báo mảng nhiều chiều, trên thực tế, là mảng mảng. Điều này có thể đư ợc thực hiện theo cách sau:

```
String[][][] occupantName; // recommended
String[] managerName [] ; // yucky, but legal
```

Ví dụ đầu tiên là mảng ba chiều (mảng các mảng) và ví dụ thứ hai là mảng hai chiều. Lưu ý trong ví dụ thứ hai, chúng ta có một dấu ngoặc vuông trư ớc tên biến và một dấu ngoặc vuông sau tên biến. Điều này hoàn toàn hợp pháp đối với trình biên dịch, một lần nữa chứng minh rằng chỉ vì nó hợp pháp không có nghĩa là nó đúng.

Việc đưa kích thư ớc của mảng vào khai báo của bạn là không hợp pháp. Có, chúng tôi biết bạn có thể làm điều đó bằng một số ngôn ngữ khác, đó là lý do tại sao bạn có thể thấy một hoặc hai câu hỏi trong bài kiểm tra bao gồm mã tư ợng tự như sau:

```
int [5] điểm; // sẽ KHÔNG biên dịch
```

Mã trư ớc đó sẽ không vượt qua trình biên dịch. Hãy nhớ rằng, JVM không phân bổ không gian cho đến khi bạn thực sự khởi tạo đối tư ợng mảng. Đó là khi nào vẫn đê kích cỡ.

Xây dựng một mảng

Xây dựng một mảng có nghĩa là tạo đối tư ợng mảng trên heap (nơi tất cả

đối tượng trực tiếp) – đó là, thực hiện một kiểu mới trên kiểu mảng. Để tạo một đối tượng mảng, Java phải biết có bao nhiêu không gian để phân bổ trên heap, vì vậy bạn phải chỉ định kích thước của mảng tại thời điểm tạo. Kích thước của mảng là số phần tử mà mảng sẽ chứa.

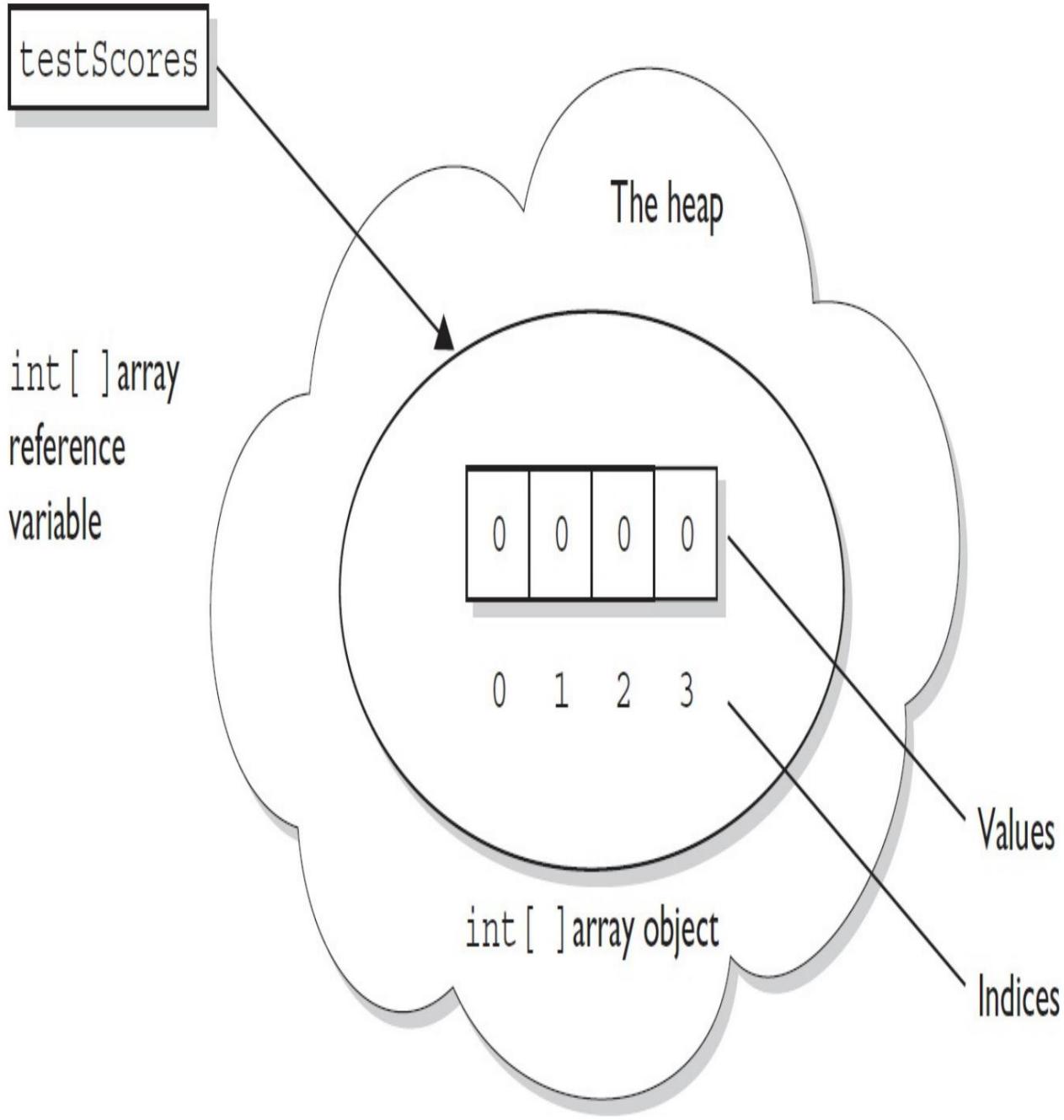
Xây dựng mảng một chiều Cách đơn giản nhất để xây dựng mảng là sử dụng từ khóa new theo sau là kiểu mảng, với một dấu ngoặc nhọn chỉ định số lượng phần tử của kiểu đó mà mảng sẽ chứa. Sau đây là một ví dụ về việc xây dựng một mảng kiểu int:

```
int[] testScores;           // Declares the array of ints
testScores = new int[4];   // constructs an array and assigns it
                           // to the testScores variable
```

Đoạn mã trư ớc đặt một đối tượng mới trên heap – một đối tượng mảng chứa bốn phần tử – với mỗi phần tử chứa một int với giá trị mặc định là 0.

Hãy coi đoạn mã này giống như nói với trình biên dịch, "Tạo một đối tượng mảng sẽ chứa bốn int và gán nó cho biến tham chiếu có tên testScores . Ngoài ra, hãy tiếp tục và đặt từng phần tử int thành 0. Cảm ơn." (Trình biên dịch đánh giá cao cách cư xử tốt.)

[Hình 6-4](#) cho thấy mảng testScores trên heap, sau khi xây dựng.



HÌNH 6-4 Một chiều trên heap

Bạn cũng có thể khai báo và xây dựng một mảng trong một câu lệnh, như sau:

```
int [] testScores = new int [4];
```

Câu lệnh đơn này tạo ra kết quả giống như hai câu lệnh trước đó.

Mảng của các kiểu đối tượng có thể được xây dựng theo một cách:

```
Thread[] threads = new Thread[5]; // no Thread objects created!
// one Thread array created
```

Hãy nhớ rằng, mặc dù mã xuất hiện như thế nào, phư ơng thức khởi tạo của Thread không được gọi. Chúng tôi không tạo một cá thể Thread , mà là một đối tư ợng mảng Thread đơn lẻ. Sau câu lệnh trư ớc, vẫn không có đối tư ợng Thread thực sự nào !



Hãy suy nghĩ cẩn thận về số lượng đối tư ợng trên heap sau khi một câu lệnh mã hoặc khởi thực thi. Bài kiểm tra sẽ yêu cầu bạn biết, ví dụ, rằng mã trư ớc đó chỉ tạo ra một đối tư ợng (mảng được gán cho biến tham chiếu có tên là chủ đề). Đối tư ợng đơn lẻ được tham chiếu bởi các luồng chứa năm biến tham chiếu Luồng , như ng không có đối tư ợng Luồng nào được gán cho các tham chiếu đó.

Hãy nhớ rằng, các mảng phải luôn được cung cấp một kích thư ớc tại thời điểm chúng được xây dựng. JVM cần kích thư ớc để phân bổ không gian thích hợp trên heap cho đối tư ợng mảng mới. Chẳng hạn, việc làm như sau không bao giờ là hợp pháp:

```
int [] carList = new int []; // Sẽ không biên dịch; cần một kích thư ớc
```

Vì vậy, đừng làm điều đó, và nếu bạn nhìn thấy nó trong bài kiểm tra, hãy chạy hét về phía câu trả lời gần nhất được đánh dấu là "Biên dịch không thành công".



Bạn có thể thấy các từ "xây dựng", "tạo" và "khởi tạo" được sử dụng thay thế cho nhau. Tất cả đều có nghĩa là, "Một đối tư ợng được xây dựng trên heap." Điều này cũng ngụ ý rằng phư ơng thức khởi tạo của đối tư ợng chạy là kết quả của mã khởi tạo / tạo / khởi tạo. Ví dụ, bạn có thể nói một cách chắc chắn rằng bất kỳ mã nào sử dụng từ khóa new sẽ (nếu nó chạy thành công) gây ra hàm tạo lớp và tất cả các hàm tạo lớp cha chạy.

Ngoài việc được xây dựng bằng mới, mảng có thể được tạo bằng cách sử dụng một kiểu viết tắt cú pháp để tạo mảng đồng thời khởi tạo các phần tử mảng thành các giá trị được cung cấp trong mã (trái ngược với giá trị mặc định). Chúng ta sẽ xem xét điều đó trong phần tiếp theo. Hiện tại, hãy hiểu rằng do các phím tắt cú pháp này, các đối tượng vẫn có thể được tạo ngay cả khi bạn chưa sử dụng hoặc nhìn thấy từ khóa mới.

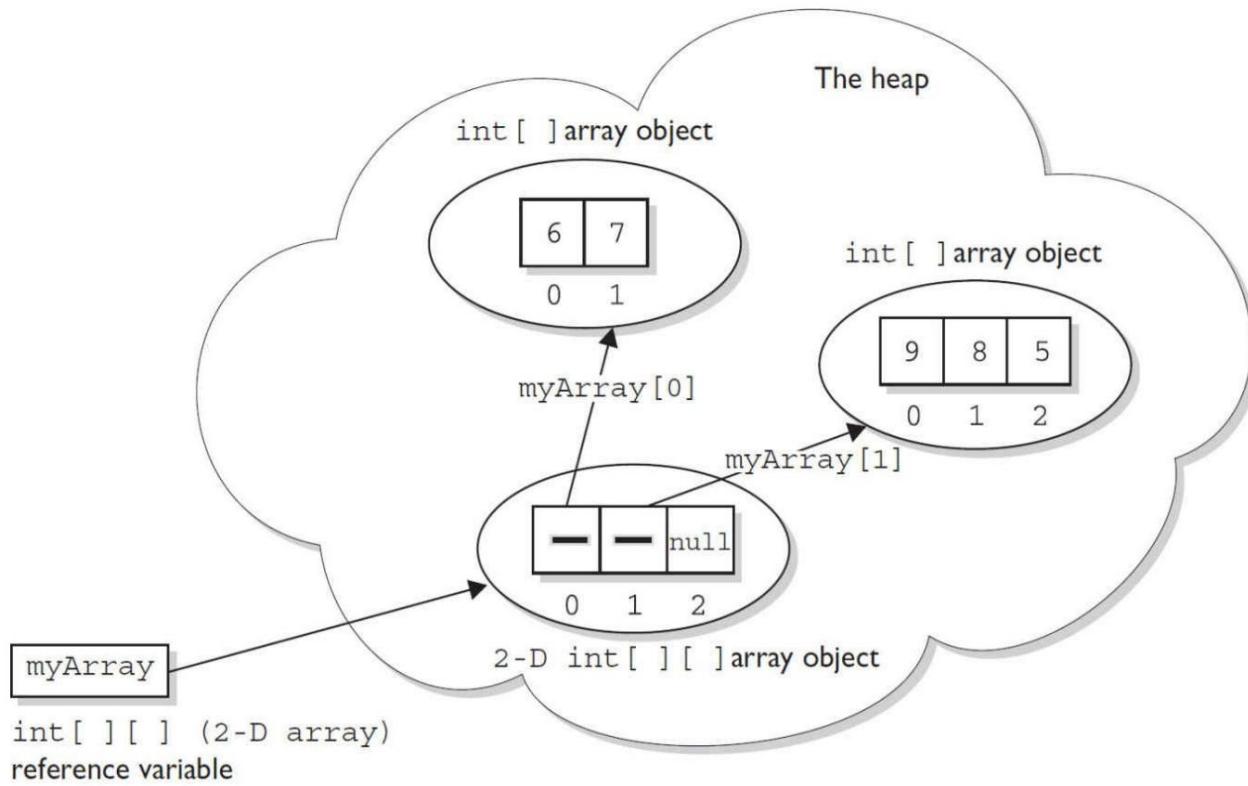
Xây dựng mảng đa chiều Mảng đa chiều, hãy nhớ rằng, đơn giản là mảng của các mảng. Vì vậy, một mảng hai chiều kiểu int thực sự là một đối tượng của kiểu int mảng (int []), với mỗi phần tử trong mảng đó giữ một tham chiếu đến một mảng int khác. Chiều thứ hai chứa các nguyên thủy thực tế.

Đoạn mã sau khai báo và xây dựng kiểu mảng hai chiều int:

```
int [] [] myArray = new int [3] [];
```

Lưu ý rằng chỉ những dấu ngoặc đầu tiên mới có kích thước. Điều đó có thể chấp nhận được trong Java vì JVM chỉ cần biết kích thước của đối tượng được gán cho biến myArray.

[Hình 6-5](#) cho thấy cách mảng int hai chiều hoạt động trên heap.



Picture demonstrates the result of the following code:

```
int [ ] [ ] myArray = new int [3] [ ];
myArray [0] = new int [2];
myArray [0] [0] = 6;
myArray [0] [1] = 7;
myArray [1] = new int [3];
myArray [1] [0] = 9;
myArray [1] [1] = 8;
myArray [1] [2] = 5;
```

HÌNH 6-5 hai chiều trên heap

Khởi tạo một mảng

Khởi tạo một mảng có nghĩa là đưa mọi thứ vào đó. "Những thứ" trong mảng là các phần tử của mảng và chúng có thể là các giá trị nguyên thủy (2, x, false, v.v.) hoặc các đối tượng được tham chiếu bởi các biến tham chiếu trong mảng. Nếu bạn có một mảng các đối tượng (trái ngược với các đối tượng nguyên thủy), mảng không thực sự chứa các đối tượng –

giống như bất kỳ biến không trực quan nào khác không bao giờ thực sự giữ đối tượng – mà thay vào đó giữ một tham chiếu đến đối tượng. Như ng chúng ta nói về mảng, chẳng hạn như "một mảng gồm năm chuỗi", mặc dù ý chúng tôi thực sự muốn nói là "một mảng gồm năm tham chiếu đến các đối tượng Chuỗi ." Sau đó, câu hỏi lớn đặt ra là liệu những tham chiếu đó có thực sự trả (rất tiếc, đây là Java, ý chúng tôi là đang tham chiếu) đến các đối tượng Chuỗi thực hay chỉ đơn giản là null. Hãy nhớ rằng, một tham chiếu không có đối tượng đư ợc gán cho nó là một tham chiếu rỗng . Và nếu bạn thực sự cố gắng sử dụng tham chiếu rỗng đó bằng cách áp dụng toán tử dấu chấm để gọi một phương thức trên đó, bạn sẽ nhận đư ợc NullPointerException khét tiếng.

Các phần tử riêng lẻ trong mảng có thể đư ợc truy cập bằng một số chỉ mục. Số chỉ mục luôn bắt đầu bằng số không (0), vì vậy đối với một mảng mư ời đối tượng, số chỉ mục sẽ chạy từ 0 đến 9. Giả sử chúng ta tạo một mảng gồm ba Động vật như sau:

```
Animal [] pet = new Animal [3];
```

Chúng tôi có một đối tượng mảng trên heap, với ba tham chiếu rỗng kiểu Động vật, như ng chúng tôi không có bất kỳ đối tượng Động vật nào . Bư ớc tiếp theo là tạo một số đối tượng Động vật và gán chúng vào các vị trí chỉ mục trong mảng đư ợc vật nuôi tham chiếu:

```
pets[0] = new Animal();
pets[1] = new Animal();
pets[2] = new Animal();
```

Mã này đặt ba đối tượng Động vật mới trên heap và gán chúng cho ba vị trí chỉ mục (phần tử) trong mảng vật nuôi .



Tìm mă có gắng truy cập chỉ mục mảng nằm ngoài phạm vi. Ví dụ: nếu một mảng có ba phần tử, việc cố gắng truy cập phần tử [3] sẽ tạo ra ArrayIndexOutOfBoundsException, bởi vì trong một mảng ba phần tử, các giá trị chỉ mục hợp pháp là 0, 1 và 2. Bạn cũng có thể thấy nỗ lực sử dụng một số âm làm chỉ số mảng. Sau đây là các ví dụ về các nỗ lực truy cập mảng hợp pháp và bất hợp pháp. Hãy chắc chắn nhận ra rằng những điều này gây ra ngoại lệ thời gian chạy chứ không phải lỗi trình biên dịch!

Gần như tất cả các câu hỏi kiểm tra đều liệt kê cả ngoại lệ thời gian chạy và lỗi trình biên dịch là các câu trả lời có thể:

```

int[] x = new int[5];
x[4] = 2;      // OK, the last element is at index 4
x[5] = 3;      // Runtime exception. There is no element at index 5!

int[] z = new int[2];
int y = -3;
z[y] = 4;      // Runtime exception. y is a negative number

```

Những điều này có thể khó phát hiện trong một vòng lặp phức tạp, như ng đó là nơi bạn có nhiều khả năng gặp các vấn đề về chỉ mục mảng trong các đề thi.

Một mảng hai chiều (một mảng các mảng) có thể được khởi tạo như sau:

```

int[][] scores = new int[3][];
// Declare and create an array (scores) holding three references
// to int arrays

scores[0] = new int[4];
// the first element in the scores array is an int array
// of four int elements

scores[1] = new int[6];
// the second element is an int array of six int elements

scores[2] = new int[1];
// the third element is an int array of one int element

```

Khởi tạo các phần tử trong một Loop Array

Các đối tượng có một biến công khai, độ dài, cung cấp cho bạn số lượng phần tử trong mảng. Khi đó, giá trị chỉ mục cuối cùng luôn nhỏ hơn độ dài một. Ví dụ: nếu độ dài của một mảng là 4, các giá trị chỉ mục từ 0 đến 3. Thông thường, bạn sẽ thấy các phần tử mảng được khởi tạo trong một vòng lặp, như sau:

```

Dog[] myDogs = new Dog[6]; // creates an array of 6 Dog references
for(int x = 0; x < myDogs.length; x++) {
    myDogs[x] = new Dog(); // assign a new Dog to index position x
}

```

Biến độ dài cho chúng ta biết mảng chứa bao nhiêu phần tử, như ng nó không cho chúng ta biết liệu các phần tử đó đã được khởi tạo hay chưa.

Khai báo, xây dựng và khởi tạo trên một dòng

Khai báo, tạo và khởi tạo trên một dòng Bạn có thể sử dụng hai phím tắt cú pháp dành riêng cho mảng khác nhau để khởi tạo (đặt các giá trị rõ ràng vào các phần tử của mảng) và xây dựng (khởi tạo chính đối tư ợng mảng) trong một câu lệnh duy nhất. Đầu tiên được sử dụng để khai báo, tạo và khởi tạo trong một câu lệnh, như sau:

```
1. int x = 9;
2. int[] dots = {6,x,8};
```

Dòng 2 trong mã trư ớc thực hiện bốn điều:

- Khai báo một biến tham chiếu mảng int có tên là dấu chấm.
- Tạo một mảng int có độ dài là ba (ba phần tử).
- Điều các phần tử của mảng với các giá trị 6, 9 và 8.
- Gán đối tư ợng mảng mới cho các dấu chấm của biến tham chiếu.

Kích thư ớc (độ dài của mảng) được xác định bởi số lư ợng các mục được phân tách bằng dấu phẩy giữa các dấu ngoặc nhọn. Mã có chức năng tư ơng ứng với mã dài hơn sau:

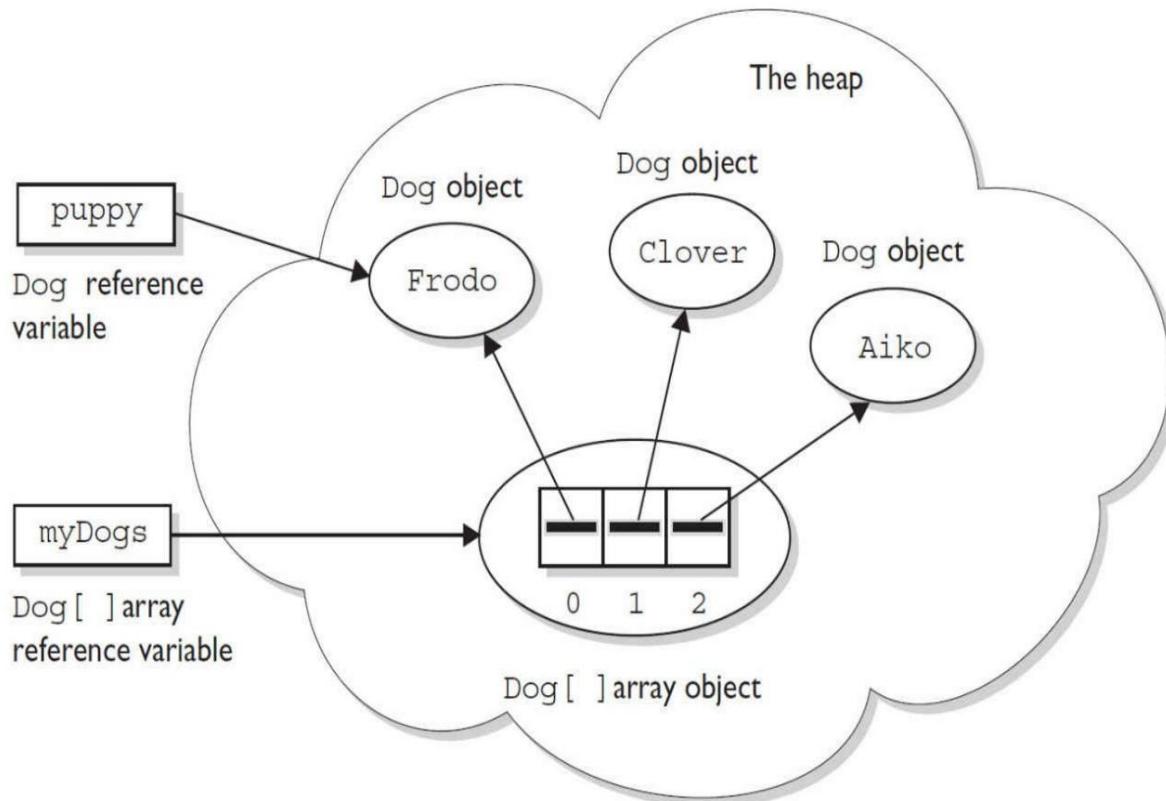
```
int[] dots;
dots = new int[3];
int x = 9;
dots[0] = 6;
dots[1] = x;
dots[2] = 8;
```

Điều này đặt ra câu hỏi, "Tại sao mọi người lại sử dụng cách dài hơn?" Một lý do Nghĩ đến. Bạn có thể không biết – tại thời điểm bạn tạo mảng – các giá trị sẽ được gán cho các phần tử của mảng.

Với các tham chiếu đối tư ợng thay vì nguyên thủy, nó hoạt động theo một cách:

```
Dog puppy = new Dog("Frodo");
Dog[] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

Đoạn mã trư ớc đó tạo một mảng Dog , được tham chiếu bởi biến myDogs, với độ dài ba phần tử. Nó chỉ định một đối tư ợng Dog đã tạo trư ớc đó (được gán cho biến tham chiếu cún con) cho phần tử đầu tiên trong mảng. Nó cũng tạo hai đối tư ợng Dog mới (Clover và Aiko) và thêm chúng vào hai phần tử biến tham chiếu Dog cuối cùng trong mảng myDogs . Chỉ riêng lối tắt mảng này (kết hợp với văn xuôi kích thích) là giá trị của cuốn sách này. [Hình 6-6](#) cho thấy kết quả.



Picture demonstrates the result of the following code:

```
Dog puppy = new Dog ("Frodo");
Dog [ ] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

Four objects are created:

- 1 Dog object referenced by puppy and by myDogs [0]
- 1 Dog [] array referenced by myDogs
- 2 Dog object referenced by myDogs [1] and myDogs [2]

HÌNH 6-6 báo, xây dựng và khởi tạo một mảng đối tượng

Bạn cũng có thể sử dụng cú pháp phím tắt với mảng nhiều chiều, như sau:

```
int [ ] [ ] Score = {{5,2,4,7}, {9,2}, {3,4}};
```

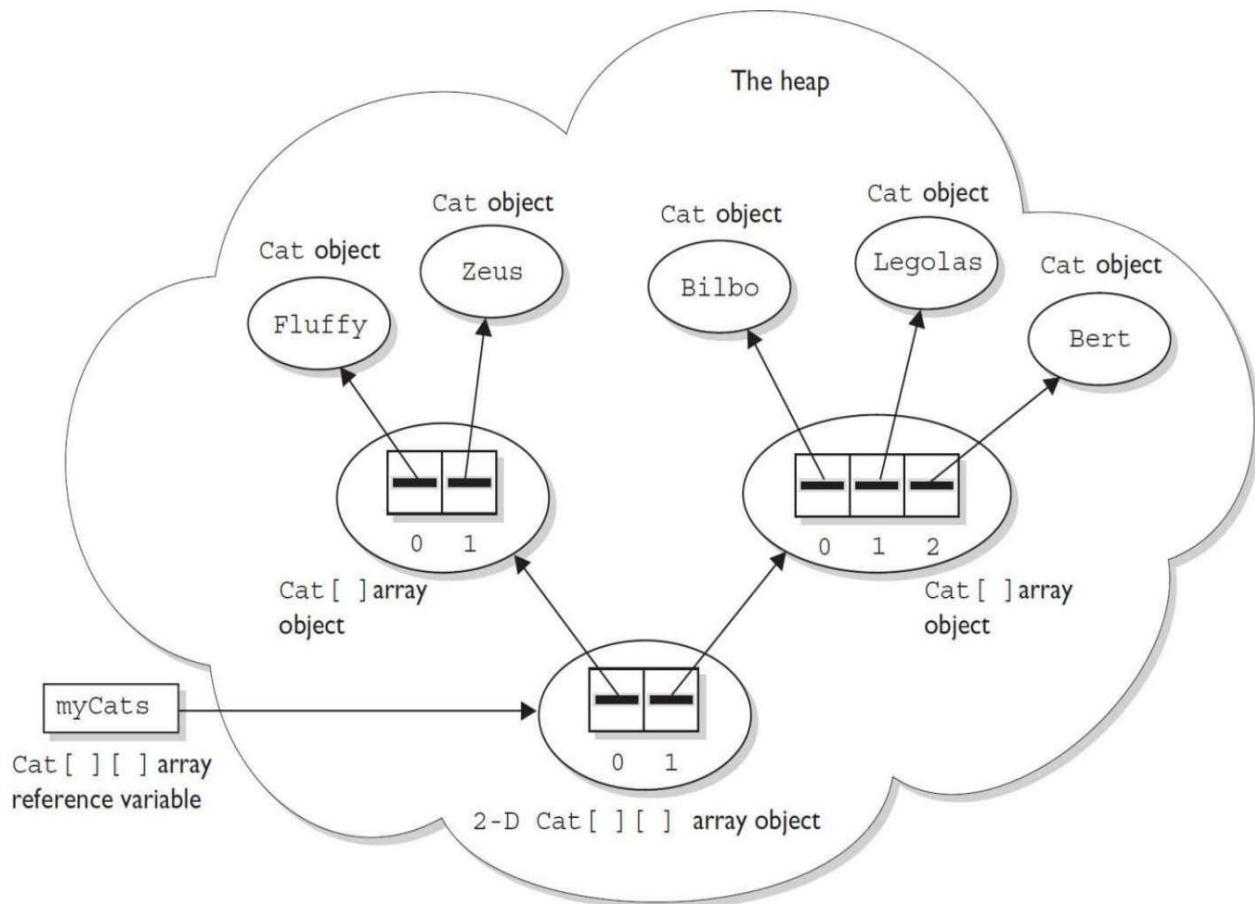
Mã này tạo ra tổng cộng bốn đối tượng trên heap. Đầu tiên, một mảng các mảng int được xây dựng (đối tượng sẽ được gán cho biến tham chiếu điểm số).

Mảng điểm số có độ dài là ba, tính từ số lư ợng các mục đư ợc phân tách bằng dấu phẩy giữa các dấu ngoặc nhọn bên ngoài. Mỗi phần tử trong số ba phần tử trong mảng điểm số là một biến tham chiếu đến một mảng int , vì vậy ba mảng int đư ợc xây dựng và gán cho ba phần tử trong mảng điểm số .

Kích thư ớc của mỗi trong ba mảng int đư ợc tính từ số lư ợng các mục trong dấu ngoặc nhọn bên trong tư ơng ứng. Ví dụ, mảng đầu tiên có độ dài là bốn, mảng thứ hai có độ dài là hai và mảng thứ ba có độ dài là hai. Cho đến nay, chúng ta có bốn đôi tư ợng: một mảng gồm các mảng int (mỗi phần tử là một tham chiếu đến một mảng int) và ba mảng int (mỗi phần tử trong ba mảng int là một giá trị int). Cuối cùng, ba mảng int đư ợc khởi tạo với các giá trị int thực trong dấu ngoặc nhọn bên trong. Do đó, mảng int đầu tiên chứa các giá trị 5,2,4,7. Đoạn mã sau hiển thị giá trị của một số phần tử trong mảng hai chiều này:

```
scores[0]      // an array of 4 ints
scores[1]      // an array of 2 ints
scores[2]      // an array of 2 ints
scores[0][1]   // the int value 2
scores[2][1]   // the int value 4
```

[Hình 6-7](#) cho thấy kết quả của việc khai báo, xây dựng và khởi tạo một hai mảng chiều trong một câu lệnh.



Picture demonstrates the result of the following code:

```
Cat[ ][ ] myCats = {{new Cat("Fluffy"), new Cat("Zeus")},  
{new Cat("Bilbo"), new Cat("Legolas"), new Cat("Bert")}};
```

Eight objects are created:

- 1 2-D Cat [] [] array object
- 2 Cat [] array object
- 5 Cat object

HÌNH 6-7 báo, xây dựng và khởi tạo mảng hai chiều

Xây dựng và khởi tạo một mảng ẩn danh

Lối tắt thứ hai được gọi là "tạo mảng ẩn danh" và có thể được sử dụng để tạo và khởi tạo một mảng, sau đó gán mảng cho một biến tham chiếu mảng đã khai báo trước đó:

```
int[] testScores;
testScores = new int[] {4,7,2};
```

Đoạn mã trư ớc đó tạo một mảng int mới với ba phần tử; khởi tạo ba phần tử với các giá trị 4, 7 và 2; và sau đó gán mảng mới cho biến tham chiếu mảng int testScores đã khai báo trư ớc đó . Chúng tôi gọi đây là tạo mảng ẩn danh vì với cú pháp này, bạn thậm chí không cần phải gán mảng mới cho bất kỳ thứ gì. Có thể bạn đang tự hỏi, "Mảng có ích lợi gì nếu bạn không gán nó cho một biến tham chiếu?" Ví dụ: bạn có thể sử dụng nó để tạo một mảng đúng lúc để sử dụng làm đối số cho một phương thức nhận tham số mảng.

Đoạn mã sau minh họa một đối số mảng vừa trong thời gian:

```
public class JIT {
    void takesAnArray(int[] someArray) {           // use the array
    public static void main (String [] args) {
        JIT j = new JIT();
        j.takesAnArray(new int[] {7,7,8,2,5});    // pass an array
    }
}
```



Hãy nhớ rằng bạn không chỉ định kích thước khi sử dụng cú pháp tạo mảng ẩn danh. Kích thước được tính từ số lư ợng mục (được phân tách bằng dấu phẩy) giữa các dấu ngoặc nhọn. Hãy chú ý đến cú pháp mảng được sử dụng trong các đề thi (và sẽ có rất nhiều trong số đó). Bạn có thể thấy cú pháp như sau:

```
new Object[3] {null, new Object(), new Object()};
// not legal; size must not be specified
```

Phân công phần tử mảng hợp pháp Bạn có thể

đặt gì vào một mảng cụ thể? Đối với bài kiểm tra, bạn cần biết rằng mảng chỉ có thể có một kiểu được khai báo (int [], Dog [], String [], v.v.), nhưng điều đó không nhất thiết có nghĩa là chỉ có các đối tượng hoặc nguyên thủy của kiểu đã khai báo có thể được gán cho các phần tử của mảng. Và những gì về chính tham chiếu mảng? Loại đối tượng mảng nào có thể được gán cho một tham chiếu mảng cụ thể? Cho

kỳ thi, bạn sẽ cần biết câu trả lời cho tất cả các câu hỏi này. Và, như thể bằng phép thuật, chúng tôi thực sự đang đề cập đến những chủ đề rất giống nhau trong các phần sau. Chú ý.

Mảng nguyên thủy Mảng nguyên thủy có thể chấp nhận bất kỳ giá trị nào có thể được thăng cấp ngầm định cho kiểu đã khai báo của mảng. Ví dụ, một mảng int có thể chứa bất kỳ giá trị nào có thể phù hợp với biến int 32 bit . Do đó, mã sau là hợp pháp:

```
int [] weightList = new int [5];
byte b = 4;
char c = 'c';
short s = 7;
weightList [0] = b; // OK, byte is smaller than int
weightList [1] = c; // OK, char is smaller than int
weightList [2] = s; // OK, short is smaller than int
```

Mảng tham chiếu đối tượng Nếu kiểu mảng được khai báo là một lớp, bạn có thể đặt các đối tượng thuộc bất kỳ lớp con nào của kiểu đã khai báo vào mảng. Ví dụ: nếu Subaru là một lớp con của Xe, bạn có thể đặt cả đối tượng Subaru và đối tượng Xe vào một mảng kiểu Xe như sau:

```
class Car {}
class Subaru extends Car {}
class Ferrari extends Car {}
...
Car [] myCars = {new Subaru(), new Car(), new Ferrari()};
```

Cần nhớ rằng các phần tử trong mảng Xe chỉ là Các biến tham chiếu ô tô . Vì vậy, bất kỳ thứ gì có thể được gán cho biến tham chiếu Xe đều có thể được gán hợp pháp cho phần tử mảng Xe .

Nếu mảng được khai báo là một kiểu giao diện, các phần tử của mảng có thể tham chiếu đến bất kỳ thể hiện nào của bất kỳ lớp nào triển khai giao diện đã khai báo. Đoạn mã sau minh họa việc sử dụng giao diện làm kiểu mảng:

```

interface Sporty {
    void beSporty();
}

class Ferrari extends Car implements Sporty {
    public void beSporty() {
        // implement cool sporty method in a Ferrari-specific way
    }
}

class RacingFlats extends AthleticShoe implements Sporty {
    public void beSporty() {
        // implement cool sporty method in a RacingFlat-specific way
    }
}

class GolfClub { }

class TestSportyThings {
    public static void main (String [] args) {
        Sporty[] sportyThings = new Sporty [3];
        sportyThings[0] = new Ferrari();           // OK, Ferrari
                                                // implements Sporty
        sportyThings[1] = new RacingFlats();       // OK, RacingFlats
                                                // implements Sporty
        sportyThings[2] = new GolfClub();          // NOT ok..

                                                // Not OK; GolfClub does not implement Sporty
                                                // I don't care what anyone says
    }
}

```

Điểm mấu chốt là: bất kỳ đối tượng nào vượt qua kiểm tra IS-A cho kiểu mảng đã khai báo đều có thể được gán cho một phần tử của mảng đó.

Bài tập tham chiếu mảng cho mảng một chiều Đối với bài kiểm tra, bạn cần nhận ra các phép gán hợp pháp và bất hợp pháp cho các biến tham chiếu mảng.

Chúng ta không nói về các tham chiếu trong mảng (nói cách khác là các phần tử mảng), mà là các tham chiếu đến đối tượng mảng. Ví dụ: nếu bạn khai báo một mảng int , biến tham chiếu mà bạn đã khai báo có thể được gán lại cho bất kỳ mảng int nào (có kích thước bất kỳ), nhưng biến không thể được gán lại cho bất kỳ thứ gì không phải là mảng int , bao gồm cả giá trị int . Hãy nhớ rằng, tất cả các mảng đều là đối tượng, do đó, một tham chiếu mảng int không thể tham chiếu đến một int nguyên thủy. Đoạn mã sau thể hiện các phép gán hợp pháp và bất hợp pháp cho các mảng nguyên thủy:

```

int[] splats;
int[] dats = new int[4];
char[] letters = new char[5];
splats = dats;    // OK, dats refers to an int array
splats = letters; // NOT OK, letters refers to a char array

```

Thật hấp dẫn để giả định rằng vì một biến kiểu byte, short hoặc char có thể được thăng cấp và gán rõ ràng cho một int, một mảng thuộc bất kỳ kiểu nào trong số đó có thể được gán cho một mảng int . Bạn không thể làm điều đó trong Java, nhưng nó sẽ giống như những nhà phát triển kỳ thi tàn nhẫn, nhẫn tâm (như ng hấp dẫn) khi đưa các câu hỏi phân công mảng phức tạp vào kỳ thi.

Mảng chứa các tham chiếu đối tư ợng, trái ngược với các mảng nguyên thủy, không hạn chế như vậy. Cũng giống như bạn có thể đặt một đối tư ợng Honda trong một mảng Ô tô (vì Honda mở rộng Ô tô), bạn có thể gán một mảng kiểu Honda cho một biến tham chiếu mảng Ô tô như sau:

```
Car[] cars;
Honda[] cuteCars = new Honda[5];
cars = cuteCars; // OK because Honda is a type of Car
Beer[] beers = new Beer [99];
cars = beers; // NOT OK, Beer is not a type of Car
```

Áp dụng kiểm tra IS-A để giúp phân loại hợp pháp khỏi bất hợp pháp. Honda IS-A Car, vì vậy mảng Honda có thể được gán cho mảng Xe . Bia IS-A Car không có thật; Bia không có tác dụng kéo dài Xe (cộng với nó cũng không có ý nghĩa gì, trừ khi bạn đã uống quá nhiều).

Các quy tắc gán mảng áp dụng cho các giao diện cũng như các lớp. Một mảng được khai báo là một kiểu giao diện có thể tham chiếu đến một mảng thuộc bất kỳ kiểu nào triển khai giao diện. Hãy nhớ rằng, bất kỳ đối tư ợng nào từ một lớp triển khai một giao diện cụ thể sẽ vượt qua bài kiểm tra IS-A (instanceof) cho giao diện đó. Ví dụ: nếu Box triển khai Foldable, thì điều sau là hợp pháp:

```
Foldable[] foldingThings;
Box[] boxThings = new Box[3];
foldingThings = boxThings;
// OK, Box implements Foldable, so Box IS-A Foldable
```



Bạn không thể đảo ngược các nhiệm vụ pháp lý. Không thể gán mảng Xe cho mảng Honda . Ô tô không nhất thiết phải là Honda, vì vậy nếu bạn đã khai báo mảng Honda , nó có thể bị nổ nếu bạn gán mảng Ô tô cho biến tham chiếu Honda . Hãy suy nghĩ về điều đó: một mảng Xe hơi có thể liên quan đến một chiếc Ferrari, vì vậy một người nào đó nghĩ rằng họ có một loạt xe Hondas có thể đột nhiên thấy mình với một chiếc Ferrari. Nhớ lấy

kiểm tra IS-A có thể được kiểm tra trong mã bằng cách sử dụng toán tử instanceof .

Gán tham chiếu mảng cho mảng đa chiều Khi bạn gán một mảng cho tham chiếu mảng đã khai báo trước đó, mảng bạn đang gán phải có cùng thứ nguyên với tham chiếu mà bạn đang gán.

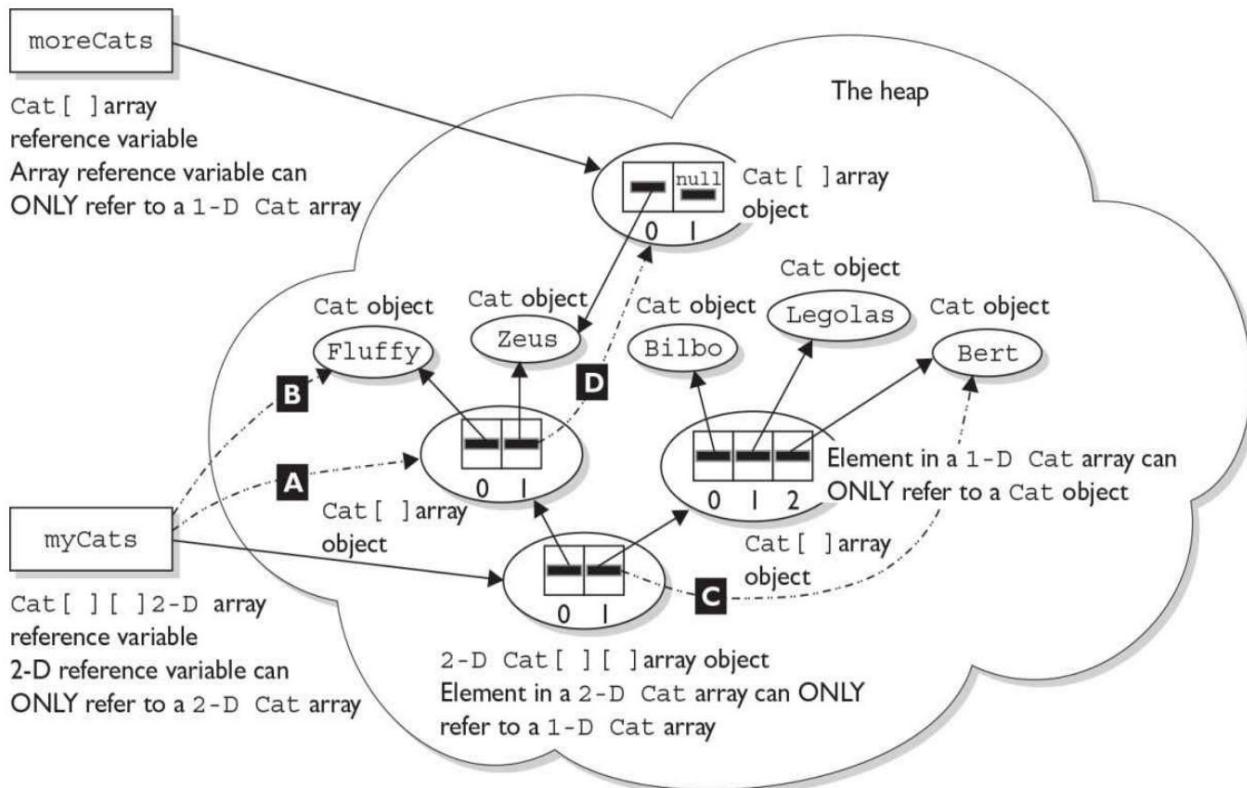
Ví dụ: một mảng int mảng hai chiều không thể được gán cho một tham chiếu mảng int thông thường , như sau:

```
int[] blots;
int[][] squeegees = new int[3][];
blots = squeegees;           // NOT OK, squeegees is a
                             // two-d array of int arrays
int[] blocks = new int[6];
blots = blocks;             // OK, blocks is an int array
```

Đặc biệt chú ý đến các phép gán mảng sử dụng các thứ nguyên khác nhau. Bạn Ví dụ, có thể được hỏi liệu việc gán một mảng int cho phần tử đầu tiên trong một mảng int mảng có hợp pháp hay không, như sau:

```
int[][] books = new int[3][];
int[] numbers = new int[6];
int aNumber = 7;
books[0] = aNumber;          // NO, expecting an int array not an int
books[0] = numbers;         // OK, numbers is an int array
```

[Hình 6-8](#) cho thấy một ví dụ về các phép gán hợp pháp và bất hợp pháp cho các tham chiếu đến một mảng.



Illegal Array Reference Assignments

```

A myCats = myCats[0];
// Can't assign a 1-D array to a 2-D array reference

B myCats = myCats[0][0];
// Can't assign a nonarray object to a 2-D array reference

C myCats[1] = myCats[1][2];
// Can't assign a nonarray object to a 1-D array reference

D myCats[0][1] = moreCats;
// Can't assign an array object to a nonarray reference
// myCats[0][1] can only refer to a Cat object
    
```

KEY

→ Legal

→ Illegal

HÌNH 6-8 Phân công mảng hợp pháp và bất hợp pháp

MỤC TIÊU XÁC NHẬN

Sử dụng ArrayLists và Wrappers (Mục tiêu 9.4 và 2.5 của OCA)

9.3 Khai báo và sử dụng ArrayList của một kiểu nhất định.

2.5 Phát triển mã sử dụng các lớp trình bao bọc như Boolean, Double và Integer.

Cấu trúc dữ liệu là một phần của hầu hết mọi ứng dụng mà bạn từng làm việc.

API Java cung cấp một loạt các lớp hỗ trợ các cấu trúc dữ liệu phổ biến như Danh sách, Bộ, Bản đồ và Hàng đợi. Đối với mục đích của kỳ thi OCA, bạn nên nhớ rằng các lớp hỗ trợ các cấu trúc dữ liệu phổ biến này là một phần của cái được gọi là "API Bộ sưu tập" (một trong nhiều bí danh của nó). (Kỳ thi OCP bao gồm các cách triển khai phổ biến nhất của tất cả các cấu trúc này.)

Khi nào sử dụng ArrayLists

Chúng ta đã nói về mảng. Mảng có vẻ hữu ích và khá linh hoạt. Vậy tại sao chúng ta cần nhiều chức năng hơn những gì mà mảng cung cấp? Hãy xem xét hai tình huống sau:

- Bạn cần có thẻ tăng và giảm kích thước danh sách những thứ của mình.
- Thứ tự của những thứ trong danh sách của bạn là quan trọng và có thể thay đổi.

Cả hai tình huống đều có thể được xử lý bằng mảng, nhưng không dễ..

Giả sử bạn muốn lên kế hoạch cho một kỳ nghỉ đến Châu Âu. Bạn có một số điểm đến trong tâm trí (Paris, Oslo, Rome), nhưng bạn vẫn chưa chắc chắn mình muốn đến thăm các thành phố này theo thứ tự nào và khi kế hoạch của bạn đang diễn ra, bạn có thể muốn thêm hoặc bớt các thành phố khỏi danh sách của mình. Giả sử ý tưởng đầu tiên của bạn là đi du lịch từ Bắc vào Nam, vì vậy danh sách của bạn trông giống như sau: Oslo, Paris, Rome.

Nếu chúng ta đang sử dụng một mảng, chúng ta có thể bắt đầu với điều này:

```
String [] thành phố = {"Oslo", "Paris", "Rome"};
```

Nhưng bây giờ hãy thử tượng tượng rằng bạn nhớ rằng bạn THỰC SỰ muốn đến London! Bạn có hai vấn đề:

- Mảng thành phố của bạn đã đầy.
- Nếu bạn đang đi từ bắc vào nam, bạn cần chèn London trước Paris.

Tất nhiên, bạn có thể tìm ra cách để làm điều này. Có thể bạn tạo ra một giây

Tất nhiên, bạn có thể tìm ra cách để làm điều này. Có thể bạn tạo một mảng thứ hai và bạn sao chép các thành phố từ mảng này sang mảng kia, và vào đúng thời điểm bạn thêm London vào mảng thứ hai. Có thể làm đư ợc, như ng khó.

Bây giờ, hãy xem cách bạn có thể làm điều tương tự với ArrayList:

```
import java.util.*;                                     // ArrayList lives in .util
public class Cities {
    public static void main(String[] args) {

        List<String> c = new ArrayList<String>(); // create an ArrayList, c
        c.add("Oslo");                                // add original cities
        c.add("Paris");
        c.add("Rome");
        int index = c.indexOf("Paris");                // find Paris' index
        System.out.println(c + " " + index);
        c.add(index, "London");                      // add London before Paris
        System.out.println(c);                        // show the contents of c
    }
}
```

Đầu ra sẽ như thế này:

```
[Oslo, Paris, Rome] 1
[Oslo, London, Paris, Rome]
```

Bằng cách xem lại mã, chúng ta có thể tìm hiểu một số thông tin quan trọng về ArrayLists:

- Lớp ArrayList nằm trong gói java.util .
- Tương tự như mảng, khi bạn xây dựng ArrayList, bạn phải khai báo loại đối tượng mà nó có thể chứa. Trong trường hợp này, chúng ta đang xây dựng một ArrayList of String đối tượng. (Chúng ta sẽ xem xét dòng mã tạo ArrayList chi tiết hơn trong một phút.)
- ArrayList thực hiện giao diện Danh sách.
- Chúng tôi làm việc với ArrayList thông qua các phương thức. Trong trường hợp này, chúng tôi đã sử dụng một vài phiên bản của add (); chúng tôi đã sử dụng indexOf (); và tiếp sau đó, chúng tôi đã sử dụng toString () để hiển thị nội dung của ArrayList . (Thêm vào toString () trong một phút nữa.)
- Giống như mảng, chỉ mục cho ArrayLists là không dựa trên.
- Chúng tôi đã không tuyên bố ArrayList lớn như thế nào khi chúng tôi xây dựng nó.
- Chúng tôi đã có thể thêm một phần tử mới vào ArrayList một cách nhanh chóng.
- Chúng tôi đã có thể thêm phần tử mới vào giữa danh sách.

- ArrayList duy trì thứ tự của nó.

Như đã hứa, chúng ta cần xem xét dòng mã sau kỹ hơn:

```
Danh sách <Chuỗi> c = new ArrayList <Chuỗi> ();
```

Trước hết, chúng ta thấy rằng đây là một khai báo đa hình. Như chúng ta đã nói trước đó, ArrayList thực hiện giao diện Danh sách (cũng trong java.util). Nếu bạn định tham gia kỳ thi OCP 8 sau khi đã đạt đư ợc OCA 8, bạn sẽ tìm hiểu thêm về lý do tại sao chúng tôi có thể muốn thực hiện một khai báo đa hình. Bây giờ, hãy tư ợng rằng một ngày nào đó bạn có thể muốn tạo một Danh sách các ArrayLists của mình.

Tiếp theo, chúng ta có cú pháp trông kỳ lạ này với các ký tự < và > . Cú pháp này đã đư ợc thêm vào ngôn ngữ trong Java 5 và nó liên quan đến "generics". Generics không thực sự đư ợc bao gồm trong kỳ thi OCA, vì vậy chúng tôi không muốn dành nhiều thời gian cho chúng ở đây, nhưng điều quan trọng cần biết là đây là cách bạn nói với trình biên dịch và JVM rằng đối với ArrayList cụ thể này . chỉ muốn cho phép các Chuỗi . Điều này có nghĩa là nếu trình biên dịch có thể thông báo rằng bạn đang cố thêm một đối tư ợng "not-a-String" vào ArrayList này, thì mã của bạn sẽ không đư ợc biên dịch. Đây là một điều tốt!

Cũng như đã hứa, hãy xem dòng mã NÀY:

```
System.out.println (c);
```

Hãy nhớ rằng tất cả các lớp cuối cùng đều kế thừa từ Đôi tư ợng lớp. Đôi tư ợng lớp chứa một phư ơng thức đư ợc gọi là `toString ()`. Một lần nữa, `toString ()` không phải là "chính thức" trong kỳ thi OCA (tất nhiên, nó nằm trong kỳ thi OCP!), Nhưng bạn cần hiểu nó một chút ngay bây giờ. Khi bạn chuyển một tham chiếu đối tư ợng tới `System.out.print ()` hoặc `System.out.println ()`, bạn đang yêu cầu họ gọi phư ơng thức `toString ()` của đối tư ợng đó . (Bất cứ khi nào bạn tạo một lớp mới, bạn có thể tùy chọn ghi đè phư ơng thức `toString ()` mà lớp của bạn đư ợc thừa kế từ Đôi tư ợng để hiển thị thông tin hữu ích về các đôi tư ợng trong lớp của bạn.) Các nhà phát triển API đã đú tốt để ghi đè phư ơng thức `toString ()` của ArrayList để bạn hiển thị nội dung của ArrayList, như bạn đã thấy trong đầu ra của chương trình. Hoan hô!

ArrayLists and Duplicates Khi bạn

đang lên kế hoạch cho chuyến đi đến Châu Âu, bạn nhận ra rằng trong nửa thời gian ở Rome, sẽ có một lễ hội âm nhạc tuyệt vời ở Naples! Naples chỉ cách Rome xuống bờ biển! Bạn phải thêm chuyến đi phụ đó vào hành trình của mình. Câu hỏi đặt ra là, một ArrayList có thể có các mục nhập trùng lặp không? Nó có hợp pháp để

nói cái này:

```
c.add("Rome");
c.add("Naples");
c.add("Rome");
```

Và câu trả lời ngắn gọn là: Có, ArrayLists có thể có các bản sao. Bây giờ nếu bạn dừng lại và suy nghĩ về nó, khái niệm "các đối tượng Java trùng lặp" thực sự hơi phức tạp. Hãy thư giãn, bởi vì bạn sẽ không phải vướng vào sự phức tạp đó cho đến khi bạn học OCP 8.



Về mặt kỹ thuật, ArrayLists chỉ chứa các tham chiếu đối tượng, không phải đối tượng thực tế và không phải nguyên thủy. Nếu bạn thấy mã như thế này,

```
myArrayList.add (7);
```

điều thực sự đang xảy ra là int đang được tự động đóng hộp (chuyển đổi) thành một đối tượng Integer và sau đó được thêm vào ArrayList. Chúng ta sẽ nói thêm về tính năng tự động đóng hộp trong một vài trang.

Các phu ơng thức ArrayList đang hoạt động

Hãy xem một đoạn mã khác thể hiện hầu hết các phu ơng thức ArrayList mà bạn cần biết cho bài kiểm tra:

```
import java.util.*;
public class TweakLists {
    public static void main(String[] args) {

        List<String> myList = new ArrayList<String>();

        myList.add("z");
        myList.add("x");
        myList.add(1, "y");           // zero based
        myList.add(0, "w");          //   "
        System.out.println(myList);   // [w, z, y, x]
```

```

myList.clear();                                // remove everything
myList.add("b");
myList.add("a");
myList.add("c");
System.out.println(myList);      // [b, a, c]
System.out.println(myList.contains("a") + " " + myList.contains("x"));

System.out.println("get 1: " + myList.get(1));
System.out.println("index of c: " + myList.indexOf("c"));

myList.remove(1);                            // remove "a"
System.out.println("size: " + myList.size() + " contents: " + myList);
}
}

```

mà sẽ tạo ra một cái gì đó như thế này:

```

[w, z, y, x]
[b, a, c]
true false
get 1: a
index of c: 2
size: 2 contents: [b, c]

```

Một vài lưu ý nhanh về mã này: Trước hết, hãy chú ý rằng hàm chứa () trả về một boolean. Điều này làm cho hàm chứa () tuyệt vời để sử dụng trong các bài kiểm tra "nếu". Thứ hai, lưu ý rằng ArrayList có một phương thức size (). Điều quan trọng cần nhớ là mảng có thuộc tính chiều dài và ArrayLists có phương thức size () .

Các phương thức quan trọng trong lớp ArrayList

Các phương pháp sau đây là một số phương pháp thường được sử dụng hơn trong Lớp ArrayList và cả những lớp mà bạn có nhiều khả năng gặp phải trong bài kiểm tra:

- add (element) Thêm phần tử này vào cuối ArrayList
- add (chỉ mục, phần tử) Thêm phần tử này tại điểm chỉ mục và dịch chuyển các phần tử còn lại trở lại (ví dụ: những gì ở chỉ mục bây giờ là chỉ mục + 1)
- clear () Xóa tất cả các phần tử khỏi ArrayList
-
- boolean chứa (phần tử) Trả về liệu phần tử có trong danh sách hay không
- Đối tượng get (chỉ mục) Trả về Đối tượng nằm ở chỉ mục

- int indexOf (Đối tượng) Trả về vị trí (int) của phần tử hoặc -1 nếu không tìm thấy Đối tượng
- remove (index) Loại bỏ phần tử tại chỉ mục đó và dịch chuyển các phần tử sau đó về phía đầu một khoảng trống xóa (Đối tượng) Loại bỏ sự xuất hiện đầu tiên của Đối tượng và dịch chuyển các phần tử sau đó về phía đầu một khoảng trống int size () Trả về số phần tử trong Lập danh sách
-

Tóm lại, đề thi OCA 8 chỉ kiểm tra những kiến thức rất cơ bản về ArrayLists. Nếu bạn tiếp tục làm bài kiểm tra OCP 8, bạn sẽ học thêm được nhiều điều về ArrayLists và các lớp hư ứng tập hợp phổ biến khác.

Autoboxing với ArrayLists

Nói chung, các bộ sưu tập như ArrayList có thể chứa các đối tượng như ng không phải là nguyên thủy. Trước Java 5, cách sử dụng phổ biến cho cái gọi là các lớp trình bao bọc (ví dụ: Integer, Float, Boolean, v.v.) là cung cấp một cách để đưa các nguyên thủy vào và ra khỏi các bộ sưu tập. Trước Java 5, bạn phải "bọc" một bản gốc theo cách thủ công trước khi có thể đưa nó vào một bộ sưu tập. Kể từ Java 5, các bản gốc vẫn phải được gói trước khi chúng có thể được thêm vào ArrayLists, nhưng autoboxing sẽ giải quyết việc đó cho bạn.

```
List myInts = new ArrayList(); // pre Java 5 declaration
myInts.add(new Integer(42)); // Use Integer class to "wrap" an int
```

Trong ví dụ trước, chúng ta tạo một thể hiện của lớp Integer với giá trị là 42. Chúng ta đã tạo toàn bộ một đối tượng để "quấn quanh" một giá trị nguyên thủy. Đối với Java 5, chúng ta có thể nói:

```
myInts.add(42); // autoboxing handles it!
```

Trong ví dụ cuối cùng này, chúng ta vẫn đang thêm một đối tượng Integer vào myInts (không phải là int nguyên thủy); nó chỉ là autoboxing xử lý gói cho chúng tôi. Có một số ẩn ý khi chúng ta cần sử dụng các đối tượng wrapper; chúng ta hãy xem xét kỹ hơn.

Trong 5 ngày cũ, trước Java, nếu bạn muốn tạo một wrapper, mở nó ra, sử dụng nó và sau đó quấn lại, bạn có thể làm như sau:

```

Integer y = new Integer(567);      // make it
int x = y.intValue();            // unwrap it
x++;
y = new Integer(x);              // rewrap it
System.out.println("y = " + y);   // print it

```

Bây giờ bạn có thể nói:

```

Integer y = new Integer(567);      // make it
y++;                            // unwrap it, increment it,
                                // rewrap it
System.out.println("y = " + y);   // print it

```

Cả hai ví dụ đều tạo ra kết quả sau:

y = 568

Và vâng, bạn đã đọc đúng. Mã dường như đang sử dụng toán tử postincrement trên một biến tham chiếu đối tượng! Như ng nó chỉ đơn giản là một sự tiện lợi. Đằng sau hậu trường, trình biên dịch thực hiện việc mở hộp và gán lại cho bạn. Trước đó, chúng tôi đã đề cập rằng các đối tượng wrapper là bất biến. ví dụ này có vẻ mâu thuẫn với tuyên bố đó. Có vẻ như giá trị của y đã thay đổi từ 567 thành 568. Tuy nhiên, điều thực sự đã xảy ra là một đối tượng trình bao bọc thứ hai đã được tạo và giá trị của nó được đặt thành 568. Giá như chúng ta có thể truy cập đối tượng trình bao bọc đầu tiên đó, chúng ta có thể chứng minh điều đó..

Chúng ta hãy cố gắng này:

```

Integer y = 567;                  // make a wrapper
Integer x = y;                   // assign a second ref
                                  // var to THE wrapper

System.out.println(y==x);         // verify that they refer
                                  // to the same object
y++;
System.out.println(x + " " + y);  // unwrap, use, "rewrap"
                                  // print values

System.out.println(y==x);         // verify that they refer
                                  // to different objects

```

tạo ra đầu ra:

```

true
567 568
false

```

Vì vậy, dưới vỏ bọc, khi trình biên dịch đến dòng y++; nó phải

thay thế một cái gì đó như thế này:

```
int x2 = y.intValue();           // unwrap it
x2++;
y = new Integer(x2);           // rewrap it
```

Đúng như chúng tôi đã nghĩ ngờ, phải có một cuộc gọi đến người mới ở đâu đó.



Tất cả các lớp bao bọc ngoại trừ Character đều cung cấp hai hàm tạo: một hàm lấy nguyên mẫu của kiểu đang được xây dựng, và hàm kia lấy biểu diễn chuỗi của kiểu đang được xây dựng. Ví dụ,

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");
```

đều là cách hợp lệ để tạo một đối tượng Integer mới ("bao bọc" giá trị 42).

Quyền anh, == , và bằng ()

Chúng tôi chỉ sử dụng == để khám phá một chút về các lớp bọc. Chúng ta hãy xem xét kỹ lưỡng hơn cách trình bao bọc hoạt động với ==, !=, Và bằng (). Các nhà phát triển API quyết định rằng đối với tất cả các lớp trình bao bọc, hai đối tượng là bằng nhau nếu chúng cùng loại và có cùng giá trị. Không có gì ngạc nhiên khi

```
Integer i1 = 1000;
Integer i2 = 1000;
if(i1 != i2) System.out.println("different objects");
if(i1.equals(i2)) System.out.println("meaningfully equal");
```

sản xuất đầu ra

```
different objects
meaningfully equal
```

Nó chỉ là hai đối tượng wrapper có cùng giá trị. Vì họ có cùng giá trị int, phương thức equals () coi chúng là "tương đương về mặt ý nghĩa" và do đó, trả về true. Làm thế nào về cái này?

```

Integer i3 = 10;
Integer i4 = 10;
if(i3 == i4) System.out.println("same object");
if(i3.equals(i4)) System.out.println("meaningfully equal");

```

Ví dụ này tạo ra kết quả:

```

same object
meaningfully equal

```

Rất tiếc! Phư ơng thức `equals()` dư ờng như đang hoạt động, như ng điều gì đã xảy ra với `==` và `!=`? Tại sao `!=` Nói với chúng ta rằng `i1` và `i2` là các đối tượng khác nhau, trong khi `==` nói rằng `i3` và `i4` là cùng một đối tượng? Để tiết kiệm bộ nhớ, hai trường hợp của các đối tượng trình bao bọc sau (đư ợc tạo thông qua quyền anh) sẽ luôn là `==` khi các giá trị ban đầu của chúng giống nhau:

- Boolean
- Byte
- Ký tự từ \ u0000 đến \ u007f (7f là 127 trong hệ thập phân)
- Ngắn và Số nguyên từ -128 đến 127

Khi `==` đư ợc sử dụng để so sánh một nguyên thủy với một trình bao bọc, trình bao bọc sẽ đư ợc mở ra và so sánh sẽ là nguyên thủy đến nguyên thủy.

Những nơi có thể sử dụng quyền anh Như

chúng ta đã thảo luận trư ớc đó, việc sử dụng bao bọc kết hợp với các bộ sưu tập là điều thường thấy. Bất cứ lúc nào bạn muốn bộ sưu tập của mình chứa các đối tượng và nguyên bản, bạn sẽ muốn sử dụng trình bao bọc để làm cho bộ sưu tập nguyên thủy đó tương thích với nhau. Nguyên tắc chung là quyền anh và unboxing hoạt động ở bất cứ nơi nào bạn có thể sử dụng một vật nguyên thủy hoặc một vật đư ợc bọc. Đoạn mã sau minh họa một số cách hợp pháp để sử dụng quyền anh:

```

class UseBoxing {
    public static void main(String [] args) {
        UseBoxing u = new UseBoxing();
        u.go(5);
    }
    boolean go(Integer i) {          // boxes the int it was passed
        Boolean ifSo = true;          // boxes the literal
        Short s = 300;                // boxes the primitive
        if(ifSo) {                   // unboxing
            System.out.println(++s); // unboxes, increments, reboxes
        }
        return !ifSo;                // unboxes, returns the inverse
    }
}

```



Hãy nhớ rằng, các biến tham chiếu của trình bao bọc có thể là giá trị rỗng. Điều đó có nghĩa là bạn phải đề phòng mã có vẻ đang thực hiện các hoạt động nguyên thủy an toàn nhưng điều đó có thể tạo ra một NullPointerException:

```

class Boxing2 {
    static Integer x;
    public static void main(String [] args) {
        doStuff(x);
    }
    static void doStuff(int z) {
        int z2 = 5;
        System.out.println(z2 + z);
    }
}

```

Mã này biên dịch tốt, nhưng JVM ném một NullPointerException khi nó cố gắng gọi doStuff (x) vì x không tham chiếu đến một đối tượng Integer , vì vậy không có giá trị nào để mở hộp.

Cú pháp Java 7 "Diamond"

Trước đó trong cuốn sách, chúng tôi đã thảo luận về một số bổ sung / cải tiến nhỏ cho ngôn ngữ được thêm vào dưới tên "Project Coin". Đồng tiền dự án cuối cùng

cải tiến mà chúng ta sẽ thảo luận là "cú pháp kim cư ơng". Chúng ta đã thấy một số ví dụ về việc khai báo ArrayLists kiểu an toàn như sau:

```
ArrayList<String> stuff = new ArrayList<String>();
ArrayList<Dog> myDogs = new ArrayList<Dog>();
```

Lưu ý rằng các tham số kiểu được trùng lặp trong các khai báo này. Kể từ Java 7, các khai báo này có thể được đơn giản hóa thành:

```
ArrayList<String> stuff = new ArrayList<>();
ArrayList<Dog> myDogs = new ArrayList<>();
```

Lưu ý rằng trong các khai báo Java 7 đơn giản hơn, phía bên phải của khai báo bao gồm hai ký tự "<>" cùng tạo thành hình thoi – doh!

Bạn không thể hoán đổi chúng; ví dụ: tuyên bố sau KHÔNG hợp pháp:

```
ArrayList<> stuff = new ArrayList<String>(); // NOT a legal diamond syntax
```

Đối với mục đích của kỳ thi, đó là tất cả những gì bạn cần biết về nhà khai thác kim cư ơng. Trong phần còn lại của cuốn sách, chúng tôi sẽ sử dụng cú pháp tiền kim cư ơng và cú pháp kim cư ơng Java 7 một cách ngẫu nhiên – giống như thế giới thực!

MỤC TIÊU XÁC NHẬN

Đóng gói nâng cao (Mục tiêu OCA 6.5)

6.5 Áp dụng các nguyên tắc đóng gói cho một lớp.

Đóng gói cho các biến tham chiếu

Trong [Chương 2](#), chúng ta đã bắt đầu thảo luận về khái niệm hư ơng đối tư ợng của tính đóng gói. Tại thời điểm đó, chúng tôi giới hạn cuộc thảo luận của chúng tôi để bảo vệ các tru ờng nguyên thủy của một lớp và các tru ờng Chuỗi (bất biến). Bây giờ bạn đã tìm hiểu thêm về ý nghĩa của "pass-by-copy" và chúng ta đã xem xét các cách xử lý dữ liệu không trực quan như mảng, StringBuilders và ArrayLists, đã đến lúc xem xét kỹ hơn về tính đóng gói.

Giả sử chúng ta có một số dữ liệu đặc biệt có giá trị mà chúng ta đang lưu trong StringBuilder. Chúng tôi rất vui khi chia sẻ giá trị với các lập trình viên khác, như ng chúng tôi không muốn họ thay đổi giá trị:

```

class Special {
    private StringBuilder s = new StringBuilder("bob"); // our special data
    StringBuilder getName() { return s; }
    void printName() { System.out.println(s); } // verify our special
                                                // data
}
public class TestSpecial {
    public static void main(String[] args) {
        Special sp = new Special();
        StringBuilder s2 = sp.getName();
        s2.append("fred");
        sp.printName();
    }
}

```

Khi chúng tôi chạy mã, chúng tôi nhận được điều này:

bobfred

Ồ ồ! Có vẻ như chúng tôi đã thực hành các kỹ thuật đóng gói tốt bằng cách đặt trư ờng của chúng tôi ở chế độ riêng tư và cung cấp phu ơng thức "getter", như ng dựa trên kết quả đầu ra, rõ ràng là chúng tôi đã không thực hiện tốt công việc bảo vệ dữ liệu trong lớp Đặc biệt . Bạn có thể tìm ra lý do tại sao không? Mất một phút..

Đư ợc rồi – chỉ để xác minh câu trả lời của bạn – khi chúng tôi gọi getName (), trên thực tế, chúng tôi trả về một bản sao, giống như Java luôn làm. Như ng chúng tôi không trả lại bản sao của đối tư ợng StringBuilder ; chúng tôi đang trả về một bản sao của biến tham chiếu trả đến (tôi biết) đối tư ợng StringBuilder duy nhất mà chúng tôi từng xây dựng. Vì vậy, tại điểm getName () trả về, chúng ta có một đối tư ợng StringBuilder và hai biến tham chiếu trả đến nó (s và s2).

Đối với mục đích của kỳ thi OCA, điểm mấu chốt là: Khi đóng gói một đối tư ợng có thể thay đổi như StringBuilder, hoặc một mảng hoặc ArrayList, nếu bạn muốn cho phép các lớp bên ngoài có bản sao của đối tư ợng, bạn phải thực sự sao chép đối tư ợng và trả về một biến tham chiếu cho đối tư ợng là một bản sao. Nếu tắt cả những gì bạn làm là trả về một bản sao của biến tham chiếu của đối tư ợng gốc, bạn KHÔNG có đóng gói.

MỤC TIÊU XÁC NHẬN

Sử dụng Lambdas đơn giản (Mục tiêu OCA 9.5)

9.5 Viết một biểu thức Lambda đơn giản sử dụng một Vị từ Lambda

biểu hiện.

Java 8 có lẽ được biết đến nhiều nhất là phiên bản Java cuối cùng đã thêm lambdas và luồng. Hai tính năng mới này (lambdas và stream) cung cấp cho các lập trình viên các công cụ để giải quyết một số vấn đề phổ biến và phức tạp với mã dễ đọc hơn, ngắn gọn hơn và trong nhiều trường hợp, mã chạy nhanh hơn. Những người tạo ra kỳ thi OCA 8 cảm thấy rằng, nói chung, lambdas và luồng là những chủ đề phù hợp hơn với kỳ thi OCP 8, như họ muốn các ứng viên OCA 8 nhận được phần giới thiệu, có lẽ để kích thích sự thèm ăn của họ.

Trong phần này, chúng ta sẽ giới thiệu cơ bản về lambdas. Chúng tôi nghĩ rằng cuộc thảo luận này sẽ đặt ra một số câu hỏi trong đầu bạn và chúng tôi rất tiếc vì điều đó, nhưng chúng tôi sẽ tự giới hạn mình chỉ giới thiệu mà những người tạo bài kiểm tra đã nghĩ đến.



Trong kỳ thi thật, bạn sẽ thấy nhiều câu hỏi kiểm tra nhiều hơn một mục tiêu. Trong các trang tiếp theo, khi chúng ta thảo luận về lambdas, chúng ta sẽ tập trung nhiều vào ArrayLists và các lớp wrapper. Bạn sẽ thấy rằng chúng tôi kết hợp ArrayLists, wrappers, AND lambdas vào nhiều danh sách mã của chúng tôi và chúng tôi cũng sử dụng kết hợp này trong các đề thi thử mà chúng tôi cung cấp. Bài thi thật (và lập trình ngoài đời thực) cũng sẽ làm như vậy.

Giả sử bạn đang tạo một ứng dụng cho một bệnh viện thú y. Chúng tôi muốn tập trung vào phần đó của ứng dụng cho phép bác sĩ thú y lấy thông tin tóm tắt về tất cả những con chó mà họ làm việc cùng. Đây là lớp Chó của chúng tôi :

```
class Dog {  
    String name;  
    int weight;  
    int age;  
    // constructor assigns a name, weight and age  
    Dog(String name, int weight, int age) {  
        this.name = name;  
        this.weight = weight;  
        this.age = age;  
    }  
    String getName() {return name;}  
    int getWeight() { return weight;}  
    int getAge() { return age;}  
    public String toString() {  
        return name;  
    }  
}
```

Bây giờ chúng ta hãy viết một số mã thử nghiệm để tạo một số Chó mẫu, đặt chúng vào một ArrayList khi chúng ta bắt đầu, và sau đó chạy một số "truy vấn" đối với ArrayList.

Đầu tiên, đây là mã giả tóm tắt:

```
// create and populate an ArrayList of Dog objects  
// invoke a few "queries" on the ArrayList  
// declare a couple of "query" methods
```

Đây là mã thử nghiệm thực tế:

```

import java.util.*;
public class TestDogs {
    public static void main(String[] args) {
        ArrayList<Dog> dogs = new ArrayList<>(); // create and populate
        dogs.add(new Dog("boi", 30, 6)); dogs.add(new Dog("tyri", 40, 12));
        dogs.add(new Dog("charis", 120, 7)); dogs.add(new Dog("aiko", 50, 10));
        dogs.add(new Dog("clover", 35, 12)); dogs.add(new Dog("mia", 15, 4));
        dogs.add(new Dog("zooey", 45, 8));
                                            // run a few "queries"
        System.out.println("all dogs " + dogs);
        System.out.println("min age 7 " + minAge(dogs, 7).toString());
        System.out.println("max wght. " + maxWeight(dogs, 40).toString());
    }
                                            // declare "query" methods
    static ArrayList<Dog> minAge(ArrayList<Dog> dogList, int testFor) {
        ArrayList<Dog> result1 = new ArrayList<>(); // do a minimum age query
        for(Dog d: dogList)
            if(d.getAge() >= testFor)           // the key moment!
                result1.add(d);
        return result1;
    }
    static ArrayList<Dog> maxWeight(ArrayList<Dog> dogList, int testFor) {
        ArrayList<Dog> result1 = new ArrayList<>(); // do a max weight query
        for(Dog d: dogList)
            if(d.getWeight() <= testFor)         // the key moment!
                result1.add(d);
        return result1;
    }
}

```

tạo ra sản lượng sau (chúng tôi hy vọng có thể dự đoán được):

```

all dogs [boi, tyri, charis, aiko, clover, mia, zooey]
min age 7 [tyri, charis, aiko, clover, zooey]
max wght. [boi, tyri, clover, mia]

```

Có thể bạn đang đỉ trú ớc chúng ta ở đây, như ng hấy chú ý phư ơng thức minAge () và maxWeight () giống nhau như thế nào . Và sẽ dễ dàng hình dung các phư ơng thức tư ơng tự khác với các tên như maxAge () hoặc namesStartingWithC () . Vì vậy, dòng mã chúng tôi muốn tập trung vào là:

```
if( d.getAge() >= testFor )      // the query expression is in bold
```

Điều gì sẽ xảy ra nếu – just sayin' – chúng tôi có thể tạo một phư ơng pháp truy vấn Dog duy nhất (thay vào đó trong số rất nhiều chúng tôi đã dự tính) và chuyển cho nó biểu thức truy vấn mà chúng tôi muốn nó sử dụng? Nó sẽ giống như thế này:

```

static ArrayList<Dog> dogQuerier(ArrayList<Dog> dogList, // query expression) {
    // do an "on the fly" query
    ArrayList<Dog> result1 = new ArrayList<>();
    for(Dog d: dogList)
        if( // query expression )           // the key moment!
            result1.add(d);
    return result1;
}

```

Bây giờ chúng ta đang nghĩ về việc chuyển mã làm đối số? Lambdas hãy để chúng tôi làm điều đó! Hãy xem xét phư ơng thức dogQuerier () của chúng ta kỹ hơn một chút. Trước hết, đoạn mã mà chúng ta sẽ chuyển vào sẽ được sử dụng làm biểu thức trong câu lệnh if . Chúng ta biết gì về biểu thức if ? Đúng! Chúng phải phân giải thành giá trị boolean . Vì vậy, khi chúng ta khai báo phư ơng thức của mình, đối số thứ hai (đối số sẽ giữ mã được truyền vào) phải được khai báo dưới dạng boolean. Những người đã mang đến cho chúng tôi Java 8 và lambdas và các luồng đã cung cấp một loạt các giao diện mới trong API và một trong những giao diện hữu ích nhất trong số này là giao diện java.util . Chức năng Predicate . Giao diện Predicate có một số phư ơng thức giao diện tĩnh và giao diện mặc định mới mẻ đó , chúng ta đã thảo luận trước đó trong cuốn sách, và quan trọng nhất đối với chúng tôi, nó có một phư ơng thức không riêng biệt được gọi là test () trả về – bạn đoán rồi – một boolean.

Đây là phư ơng thức dogQuerier () đa năng :

```

static ArrayList<Dog> dogQuerier(ArrayList<Dog> dogList, Predicate<Dog> expr) {
    // do an "on the fly" query
    ArrayList<Dog> result1 = new ArrayList<>();
    for(Dog d: dogList)
        if( expr.test(d) )           // the key moment!
            result1.add(d);
    return result1;
}

```

Cho đến nay, điều này trông giống như Java cũ; đó là khi chúng ta gọi dogQuerier () cú pháp trở nên thú vị:

```
dogQuerier(dogs, d -> d.getAge() < 9);
```

Khi chúng ta nói [c] d -> d.getAge () < 9 [/ c] –THAT là biểu thức lambda. Số d đại diện cho đối số và sau đó mã phải trả về một boolean. Hãy tập hợp tất cả những điều này lại với nhau trong một phiên bản TestDogs mới:

```

import java.util.*;
import java.util.function.Predicate;
public class TestDogs {
    public static void main(String[] args) {
        ArrayList<Dog> dogs = new ArrayList<>(); // create and populate
        dogs.add(new Dog("boi", 30, 6)); dogs.add(new Dog("tyri", 40, 12));
        dogs.add(new Dog("charis", 120, 7)); dogs.add(new Dog("aiko", 50, 10));
        dogs.add(new Dog("clover", 35, 12)); dogs.add(new Dog("mia", 15, 4));
        dogs.add(new Dog("zooey", 45, 8));
                                            // run a few old "queries"
        System.out.println("all dogs " + dogs);
        System.out.println("min age 7 " + minAge(dogs, 7).toString());
        System.out.println("max wght. " + maxWeight(dogs, 40).toString());
                                            // run a few lambda queries
        System.out.println("age < 9 " + dogQuery(dogs, d -> d.getAge() < 9));
        System.out.println("w > 100 " + dogQuery(dogs, d -> d.getWeight() > 100));
    }
    // declare old style "query" methods
    static ArrayList<Dog> minAge(ArrayList<Dog> dogList, int testFor) {
        // do a minimum age query
        ArrayList<Dog> result1 = new ArrayList<>();
        for(Dog d: dogList)
            if(d.getAge() >= testFor) // the key moment!
                result1.add(d);
        return result1;
    }
    static ArrayList<Dog> maxWeight(ArrayList<Dog> dogList, int testFor) {
        // do a max weight query
        ArrayList<Dog> result1 = new ArrayList<>();
        for(Dog d: dogList)
            if(d.getWeight() <= testFor) // the key moment!
                result1.add(d);
        return result1;
    }
    // declare a new lambda powered, generic, multi-purpose query method
    static ArrayList< >Dog> dogQuery(ArrayList< >Dog> dogList, Predicate< >Dog> expr) {
        // do an "on the fly" query
        ArrayList< >Dog> result1 = new ArrayList< >();
        for(Dog d: dogList)
            if(expr.test(d)) // the key moment, lambda powered!
                result1.add(d);
        return result1;
    }
}

```

tạo ra kết quả sau (hai dòng cuối cùng được tạo bằng lambdas!):

```

all dogs [boi, tyri, charis, aiko, clover, mia, zooey]
min age 7 [tyri, charis, aiko, clover, zooey]
max wght. [boi, tyri, clover, mia]
age < > 9 [boi, charis, mia, zooey]
w > 100 [charis]

```

Hãy quay lại bây giờ và đề cập đến một số quy tắc cú pháp. Các quy tắc sau đây chỉ dành cho mục đích của kỳ thi OCA 8! Nếu bạn quyết định kiểm đư ợc OCP 8 của mình, bạn sẽ đi sâu hơn nhiều vào lambdas và có rất nhiều "lon sâu" bạn sẽ phải mở mà chúng tôi có tình tránh. Vì vậy, những gì sau đây là một sự đơn giản hóa thích hợp của OCA 8.

Cú pháp cơ bản cho một lambda Predicate có ba phần:

A Single Parameter	An Arrow-Token	A Body
x	→	7 < > 5

Các loại lambdas khác có không hoặc nhiều tham số, như ng đối với OCA 8, chúng tôi tập trung hoàn toàn vào Predicate, phải nhận chính xác một tham số. Dưới đây là một số quy tắc cú pháp chi tiết cho lambdas Predicate :

- Tham số có thể chỉ là một tên biến hoặc nó có thể là kiểu theo sau bởi một tên biến, tất cả đều đư ợc đặt trong dấu ngoặc đơn.
- Phần thân PHẢI (cách này hay cách khác) trả về một boolean.
- Phần thân có thể là một biểu thức duy nhất, không thể có giá trị trả về bên trong trình.
- Phần thân có thể là một khối mã đư ợc bao quanh bởi dấu ngoặc nhọn, chứa một hoặc nhiều câu lệnh hợp lệ, mỗi câu kết thúc bằng dấu chấm phẩy và khối phải kết thúc bằng câu lệnh trả về .

Sau đây là danh sách mã hiển thị các ví dụ về hợp pháp và sau đó là bất hợp pháp ví dụ về Predicate lambdas:

```

import java.util.function.Predicate;           // type of lambda
                                              // we're learning
public class Lamb2 {
    public static void main(String[] args) {
        Lamb2 m1 = new Lamb2();

```

```

// ===== LEGAL LAMBDAS =====

m1.go(x -> 7 < 5);                                // extra terse
m1.go(x -> { return adder(2, 1) > 5; });          // block
m1.go((Lamb2 x) -> { int y = 5;
                      return adder(y, 7) > 8; }); // multi-stmt block
m1.go(x -> { int y=5; return adder(y,6) > 8; }); // no arg type, block
int a = 5; int b = 6;
m1.go(x -> { return adder(a, b) > 8; });          // in scope vars
m1.go((Lamb2 x) -> adder(a, b) > 13);            // arg type, no block

// ===== ILLEGAL LAMBDAS =====

// m1.go(x -> return adder(2, 1) > 5; );           // return w/o block
// m1.go(Lamb2 x -> adder(2, 3) > 7);              // type needs parens
// m1.go(() -> adder(2, 3) > 7);                  // Predicate needs 1 arg
// m1.go(x -> { adder(4, 2) > 9 });                // blocks need statements
// m1.go(x -> { int y = 5; adder(y, 7) > 8; });    // block needs return
}
void go(Predicate<Lamb2> e) {                      // go() takes a predicate
    Lamb2 m2 = new Lamb2();
    System.out.println(e.test(m2) ? "ternary true" // ternary uses boolean expr
                                  : "ternary false");
}
static int adder(int x, int y) { return x + y; }     // complex calculation
}

```

Mã này chủ yếu là về cú pháp hợp lệ và không hợp lệ, như ng chúng ta hãy xem xét kỹ hơn một chút về phư ơng thức go () . Bài kiểm tra chủ yếu liên quan đến mã được chuyển đến một phư ơng thức, như ng sẽ hữu ích khi xem xét (như ng không QUÁ chặt chẽ) một phư ơng thức nhận mã lambda. Trong cả mã Chó và mã trực tiếp ở trên, phư ơng thức nhận sử dụng một Vị từ. Bên trong các phư ơng thức nhận, chúng tôi đã tạo một đối tư ợng thuộc kiểu mà chúng tôi đang làm việc, đối tư ợng này chúng tôi chuyển cho phư ơng thức Predicate.test () . Phư ơng thức nhận yêu cầu phư ơng thức test () trả về một boolean.

Chúng tôi phải thừa nhận rằng lambdas hơi khó học. Một lần nữa, chúng tôi hy vọng rằng chúng tôi đã để lại cho bạn một số câu hỏi chưa được trả lời, như ng chúng tôi nghĩ Oracle đã làm một công việc hợp lý khi cắt ra một phần của câu đố lambda để bắt đầu. Nếu bạn hiểu các vấn đề mà chúng tôi đã đề cập, bạn sẽ có thể xử lý các câu hỏi liên quan đến lambda mà Oracle đưa ra cho bạn.

TÓM TẮT CHỨNG NHẬN

Điều quan trọng nhất cần nhớ về Chuỗi là các đối tư ợng Chuỗi là

không thay đổi được, như ng tham chiếu đến Chuỗi thì không! Bạn có thể tạo một Chuỗi mới bằng cách sử dụng một Chuỗi hiện có làm điểm bắt đầu, như ng nếu bạn không gán một biến tham chiếu cho Chuỗi mới, thì nó sẽ bị mất vào chương trình của bạn – bạn sẽ không có cách nào để truy cập vào Chuỗi mới của mình. Xem lại các phu ơng thức quan trọng trong lớp String .

Lớp StringBuilder đã đư ợc thêm vào trong Java 5. Nó có các phu ơng thức giống hệt như lớp StringBuffer cũ , ngoại trừ các phu ơng thức của StringBuilder không an toàn cho luồng. Vì các phu ơng thức của StringBuilder không an toàn cho luồng, chúng có xu hướng chạy nhanh hơn các phu ơng thức StringBuffer , vì vậy hãy chọn StringBuilder bắt cứ khi nào việc phân luồng không phải là vấn đề. Cả hai đối tượng StringBuffer và StringBuilder đều có thể thay đổi giá trị của chúng nhiều lần mà bạn không cần phải tạo các đối tượng mới. Nếu bạn đang thực hiện nhiều thao tác với chuỗi, các đối tượng này sẽ hiệu quả hơn so với các đối tượng Chuỗi bất biến , dù ít hay nhiều, "sử dụng một lần, lưu lại trong bộ nhớ mãi mãi." Hãy nhớ rằng, các phu ơng thức này LUÔN thay đổi giá trị của đối tượng đang gọi, ngay cả khi không có chỉ định rõ ràng.

Tiếp theo, chúng ta đã thảo luận về các lớp và giao diện chính trong lịch Java 8 mới và các gói liên quan đến thời gian. Tương tự như Strings, tất cả các lớp lịch mà chúng ta đã nghiên cứu đều tạo ra các đối tượng bất biến. Ngoài ra, các lớp này sử dụng các phu ơng thức gốc riêng để tạo các đối tượng mới. Từ khóa mới không thể đư ợc sử dụng với các lớp này. Chúng tôi đã xem xét một số tính năng mạnh mẽ của các lớp này, như tính lư ợng thời gian giữa hai ngày hoặc giờ khác nhau. Sau đó, chúng ta đã xem xét cách lớp DateTimeFormatter đư ợc sử dụng để phân tích cú pháp Chuỗi thành các đối tượng lịch và cách nó đư ợc sử dụng để làm đẹp các đối tượng lịch.

Chủ đề tiếp theo là mảng. Chúng ta đã nói về việc khai báo, xây dựng và khởi tạo mảng một chiều và nhiều chiều. Chúng ta đã nói về mảng ẩn danh và thực tế là mảng đối tượng thực sự là mảng tham chiếu đến các đối tượng.

Tiếp theo, chúng ta đã thảo luận về những điều cơ bản của ArrayLists. ArrayLists giống như các mảng với các siêu năng lực cho phép chúng phát triển và thu nhỏ một cách linh hoạt và giúp bạn dễ dàng chèn và xóa các phần tử tại các vị trí bạn chọn trong danh sách. Chúng ta đã thảo luận về ý tưởng rằng ArrayLists không thể chứa các giá trị nguyên thủy và nếu bạn muốn tạo một ArrayList chứa đầy một loại giá trị nguyên thủy nhất định, bạn sử dụng các lớp "wrapper" để biến một giá trị nguyên thủy thành một đối tượng đại diện cho giá trị đó. Sau đó, chúng ta đã thảo luận về cách với autoboxing, việc chuyển các nguyên bản thành các đối tượng wrapper và ngược lại, đư ợc thực hiện tự động như thế nào.

Cuối cùng, chúng ta đã thảo luận về một tập hợp con cụ thể của chủ đề lambdas, sử dụng giao diện Predicate . Ý tưởng cơ bản của lambdas là bạn có thể chuyển một chút mã từ phu ơng thức này sang phu ơng thức khác. Giao diện Predicate là một trong nhiều "chức năng

giao diện "được cung cấp trong API Java 8. Giao diện chức năng là giao diện chỉ có một phương thức được triển khai. Trong trường hợp của giao diện `Vị trí`, phương thức này được gọi là `test()` và nó nhận một đối số duy nhất và trả về một boolean.

Để kết thúc cuộc thảo luận của chúng ta về lambdas, chúng tôi đã đề cập đến một số quy tắc cú pháp phức tạp mà bạn cần biết để viết lambdas hợp lệ.

✓ KHOAN HAI PHÚT

Dưới đây là một số điểm chính từ các mục tiêu chứng nhận trong chương này.

Sử dụng String và StringBuilder (Mục tiêu 9.2 và 9.1 của OCA)

- Đối tượng chuỗi là biến và các biến tham chiếu chuỗi thì không.
- Nếu bạn tạo một Chuỗi mới mà không gán nó, nó sẽ bị mất vào chương trình của bạn.
- Nếu bạn chuyển hướng một tham chiếu Chuỗi đến một Chuỗi mới, thì Chuỗi cũ có thể bị mất.
- Các phương thức chuỗi sử dụng các chỉ mục dựa trên 0, ngoại trừ đối số thứ hai của chuỗi con () .
- Lớp String là lớp cuối cùng - nó không thể được mở rộng.
- Khi JVM tìm thấy một chuỗi ký tự, nó sẽ được thêm vào nhóm chuỗi ký tự.
- Các chuỗi có một phương thức gọi là `length()` - các mảng có một thuộc tính có tên là `length`.
- Các đối tượng `StringBuilder` có thể thay đổi - chúng có thể thay đổi mà không cần tạo một đối tượng mới.
- Các phương thức `StringBuilder` hoạt động trên đối tượng đang gọi và các đối tượng có thể thay đổi mà không cần gán rõ ràng trong câu lệnh.
- Hãy nhớ rằng các phương thức chuỗi được đánh giá từ trái sang phải.
- Các phương thức chuỗi cần nhớ: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `Replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()` và `trim()`.
- Các phương thức `StringBuilder` cần nhớ: `append()`, `delete()`, `insert()`, `reverse()` và `toString()`.

Thao tác dữ liệu lịch (Mục tiêu 9.3 của OCA)

Thao tác dữ liệu lịch (Mục tiêu 9.3 của OCA)

- Trong bài kiểm tra, tất cả các đối tượng được tạo bằng cách sử dụng các lớp lịch là bất biến, nhưng các biến tham chiếu của chúng thì không.
- Nếu bạn tạo một đối tượng lịch mới mà không gán nó, nó sẽ bị mất vào chương trình của bạn.
- Nếu bạn chuyển hướng tham chiếu lịch đến đối tượng lịch mới, đối tượng lịch cũ có thể bị mất.
- Tất cả các đối tượng được tạo bằng cách sử dụng các lớp lịch của kỳ thi phải được tạo bằng các phương thức gốc (ví dụ: from (), now (), of (), parse ()); từ khóa mới không được phép.
- Các phương thức Until () và between () thực hiện các phép tính phức tạp xác định khoảng thời gian giữa các giá trị của hai đối tượng lịch.
- Lớp DateTimeFormatter sử dụng phương thức parse () để phân tích cú pháp các Chuỗi đầu vào thành các đối tượng lịch hợp lệ.
- Lớp DateTimeFormatter sử dụng phương thức format () để định dạng các đối tượng lịch thành các Chuỗi được tạo hình đẹp mắt.

Sử dụng Mảng (Mục tiêu OCA 4.1 và 4.2)

- Mảng có thể chứa các đối tượng hoặc nguyên thủy, nhưng bản thân mảng luôn là một đối tượng.
- Khi bạn khai báo một mảng, dấu ngoặc vuông có thể ở bên trái hoặc bên phải của Tên.
- Không bao giờ hợp pháp khi bao gồm kích thước của một mảng trong khai báo.
- Bạn phải bao gồm kích thước của một mảng khi bạn xây dựng nó (sử dụng new) trừ khi bạn đang tạo một mảng ẩn danh.
- Các phần tử trong một mảng đối tượng không được tạo tự động, mặc dù các phần tử mảng nguyên thủy được cung cấp giá trị mặc định.
- Bạn sẽ nhận được một NullPointerException nếu bạn cố gắng sử dụng một phần tử mảng trong một mảng đối tượng nếu phần tử đó không tham chiếu đến một đối tượng thực.
- Mảng được lập chỉ mục bắt đầu bằng số không.
- Một ArrayIndexOutOfBoundsException xảy ra nếu bạn sử dụng một giá trị chỉ mục không hợp lệ.
- Mảng có thuộc tính độ dài có giá trị là số mảng

các yếu tố.

- Chỉ mục cuối cùng mà bạn có thể truy cập luôn nhỏ hơn độ dài của mảng.
- Mảng nhiều chiều chỉ là mảng của mảng.
- Các kích thước trong mảng nhiều chiều có thể có độ dài khác nhau.
- Một mảng các nguyên thủy có thể chấp nhận bất kỳ giá trị nào có thể được thăng cấp ngầm định cho kiểu được khai báo của mảng – ví dụ, một biến byte có thể đi trong một mảng int .
- Một mảng các đối tượng có thể chứa bất kỳ đối tượng nào vượt qua kiểm tra IS-A (hoặc instanceof) cho kiểu đã khai báo của mảng. Ví dụ: nếu Horse mở rộng Động vật, thì một đối tượng Ngựa có thể đi vào mảng Động vật .
- Nếu bạn gán một mảng cho một tham chiếu mảng đã khai báo trước đó, thì mảng bạn đang gán phải có cùng thứ nguyên với tham chiếu mà bạn đang gán cho nó.
- Bạn có thể gán một mảng của một kiểu cho một tham chiếu mảng đã khai báo trước đó của một trong các siêu kiểu của nó. Ví dụ, một mảng Honda có thể được gán cho một mảng được khai báo là loại Ô tô (giả sử Honda mở rộng Ô tô).

Sử dụng ArrayList (Mục tiêu 9.4 của OCA)

- ArrayLists cho phép bạn thay đổi kích thước danh sách của mình và thực hiện chèn và xóa vào danh sách của bạn dễ dàng hơn nhiều so với mảng.
- ArrayLists được sắp xếp theo mặc định. Khi bạn sử dụng phuơng thức add () không có đối số chỉ mục, mục nhập mới sẽ được thêm vào cuối ArrayList.
- Đối với kỳ thi OCA 8, các khai báo ArrayList duy nhất mà bạn cần biết có dạng sau:

```
ArrayList<type> myList = new ArrayList<type>();
List<type> myList2 = new ArrayList<type>(); // polymorphic
List<type> myList3 = new ArrayList<>(); // diamond operator, polymorphic optional
```

- ArrayLists chỉ có thể chứa các đối tượng, không phải nguyên thủy, như ng hấy nhớ rằng autoboxing có thể làm cho nó trông giống như bạn đang thêm nguyên thủy vào ArrayList khi trên thực tế, bạn đang thêm phiên bản đối tượng bao bọc của nguyên thủy.

- Chỉ mục của ArrayList bắt đầu từ 0.
- ArrayLists có thể có các mục nhập trùng lặp. Lưu ý: Việc xác định xem hai đối tượng có trùng lặp hay không sẽ khó hơn so với đối tượng và không xuất hiện cho đến kỳ thi OCP 8.
- Các phương thức ArrayList cần nhớ: add (element), add (index, element), clear (), contains (object), get (index), indexOf (object), remove (index), remove (object), and size () .

Đóng gói các biến tham chiếu (Mục tiêu OCA 6.5)

- Nếu bạn muốn đóng gói các đối tượng có thể thay đổi như StringBuilders hoặc mảng hoặc ArrayLists, bạn không thể trả về một tham chiếu đến các đối tượng này; trước tiên bạn phải tạo một bản sao của đối tượng và trả về một tham chiếu cho bản sao.
- Bất kỳ lớp nào có một phương thức trả về một tham chiếu đến một đối tượng có thể thay đổi đang phá vỡ tính đóng gói.

Sử dụng biểu thức Lambda vị từ (Mục tiêu OCA 9.5)

- Lambdas cho phép bạn chuyển các bit mã từ phương thức này sang phương thức khác. Và phương thức nhận có thể chạy bất kỳ mã tuân thủ nào mà nó được gửi.
- Mặc dù có rất nhiều loại lambda mà Java 8 hỗ trợ, nhưng đối với kỳ thi này, loại lambda duy nhất bạn cần biết là Predicate.
- Giao diện Predicate có một phương thức duy nhất để triển khai được gọi là test () và nó nhận một đối số và trả về một boolean.
- Vì phương thức Predicate.test () trả về một boolean, nó có thể được đặt (chủ yếu là?) ở bất cứ đâu mà một biểu thức boolean có thể đi, ví dụ: trong các câu lệnh if, while, do và bậc ba.
- Biểu thức lambda vị từ có ba phần: một đối số duy nhất, một mũi tên (->) và một biểu thức hoặc khối mã.
- Đối số của biểu thức lambda Predicate có thể chỉ là một biến hoặc một kiểu và biến cùng nhau trong dấu ngoặc đơn, ví dụ, (MyClass m).
- Phần thân của biểu thức lambda Predicate có thể là một biểu thức phân giải thành boolean HỌC nó có thể là một khối câu lệnh (được bao quanh bởi dấu ngoặc nhọn) kết thúc bằng câu lệnh trả về kiểu boolean .

TỰ KIỂM TRA

1. Cho:

```
public class Mutant {  
    public static void main(String[] args) {  
        StringBuider sb = new StringBuider("abc");  
        String s = "abc";  
        sb.reverse().append("d");  
        s.toUpperCase().concat("d");  
        System.out.println(".." + sb + ". ." + s + "..");  
    }  
}
```

Hai chuỗi con nào sẽ được đưa vào kết quả? (Chọn hai.)

- A. .abc.
- B. .ABCd.
- C. .ABCD.
- D. .cbad.
- E. .dcba.

2. Đưa ra:

```
public class Hilltop {  
    public static void main(String[] args) {  
        String[] horses = new String[5];  
        horses[4] = null;  
        for(int i = 0; i < horses.length; i++) {  
            if(i < args.length)  
                horses[i] = args[i];  
            System.out.print(horses[i].toUpperCase() + " ");  
        }  
    }  
}
```

Và, nếu mã biên dịch, dòng lệnh:

```
java Hilltop eyra vafi draumur kara
```

Kết quả là gì?

- A. EYRA VAFI DRAUMUR KARA
- B. EYRA VAFI DRAUMUR KARA rỗng
- C. Một ngoại lệ được đưa ra mà không có đầu ra nào khác
- D. EYRA VAFI DRAUMUR KARA, và sau đó là một NullPointerException

E. EYRA VAFI DRAUMUR KARA và sau đó là
ArrayIndexOutOfBoundsException F. Biên dịch

không thành công

3. Đưa ra:

```
public class Actors {  
    public static void main(String[] args) {  
        char[] ca = {0x4e, \u004e, 78};  
        System.out.println((ca[0] == ca[1]) + " " + (ca[0] == ca[2]));  
    }  
}
```

Kết quả là gì?

A. đúng sự thật

B. đúng sai

C. sai đúng

D. sai sai

E. Biên dịch không thành công

4. Cho:

```
1. class Dims {  
2.     public static void main(String[] args) {  
3.         int[][] a = {{1,2}, {3,4}};  
4.         int[] b = (int[]) a[1];  
5.         Object o1 = a;  
6.         int[][] a2 = (int[][] ) o1;  
7.         int[] b2 = (int[] ) o1;  
8.         System.out.println(b[1]);  
9.     } }
```

Kết quả là gì? (Chọn tất cả các áp dụng.)

A. 2

B. 4

C. Một ngoại lệ được đưa ra trong thời

gian chạy D. Biên dịch không thành công do lỗi trên

dòng 4 E. Biên dịch không thành công do lỗi trên

dòng 5 F. Biên dịch không thành công do lỗi trên

dòng 6 G. Biên dịch không thành công do lỗi trên dòng 7

5. Đưa ra:

```
import java.util.*;
public class Sequence {
    public static void main(String[] args) {
        ArrayList<String> myList = new ArrayList<String>();
        myList.add("apple");
        myList.add("carrot");
        myList.add("banana");
        myList.add(1, "plum");
        System.out.print(myList);
    }
}
```

Kết quả là gì?

- A. [táo, chuối, cà rốt, mận]
- B. [táo, mận, cà rốt, chuối]
- C. [táo, mận, chuối, cà rốt]
- D. [mận, chuối, cà rốt, táo]
- E. [mận, táo, cà rốt, chuối]
- F. [chuối, mận, cà rốt, táo]
- G. Biên dịch không thành công

6. Cho:

```
3. class Dozens {
4.     int[] dz = {1,2,3,4,5,6,7,8,9,10,11,12};
5. }
6. public class Eggs {
7.     public static void main(String[] args) {
8.         Dozens [] da = new Dozens[3];
9.         da[0] = new Dozens();
10.        Dozens d = new Dozens();
11.        da[1] = d;
12.        d = null;
13.        da[1] = null;
14.        // do stuff
15.    }
16. }
```

Hai điều nào đúng về các đối tượng được tạo trong main () và đối tượng nào đủ điều kiện để thu gom rác khi đến dòng 14?

- A. Ba đối tượng được tạo ra
- B. Bốn đối tượng được tạo ra

- C. Năm đối tượng được tạo ra
- D. Không đối tượng đủ điều kiện cho
- GC E. Một đối tượng đủ điều kiện cho
- GC F. Hai đối tượng đủ điều kiện cho
- GC G. Ba đối tượng đủ điều kiện cho GC

7. Cho:

```
public class Tailor {  
    public static void main(String[] args) {  
        byte[][] ba = {{1,2,3,4}, {1,2,3}};  
        System.out.println(ba[1].length + " " + ba.length);  
    }  
}
```

Kết quả là gì?

- A. 2 4
 - B. 2 7
 - C. 3 2
 - D. 3 7
 - E. 4 2
 - F. 4 7
- G. Biên dịch không thành công

8. Cho:

```
3. public class Theory {  
4.     public static void main(String[] args) {  
5.         String s1 = "abc";  
6.         String s2 = s1;  
7.         s1 += "d";  
8.         System.out.println(s1 + " " + s2 + " " + (s1==s2));  
9.  
10.        StringBuilder sb1 = new StringBuilder("abc");  
11.        StringBuilder sb2 = sb1;  
12.        sb1.append("d");  
13.        System.out.println(sb1 + " " + sb2 + " " + (sb1==sb2));  
14.    }  
15. }
```

Đó là sự thật? (Chọn tất cả các áp dụng.)

- A. Biên dịch không thành công

B. Dòng đầu tiên là abc abc true C. Dòng đầu tiên là abc abc false D. Dòng đầu tiên là abcd abc false E. Dòng thứ hai của đầu ra là abcd abc false F. Dòng thứ hai của đầu ra là abcd abcd true G. Dòng thứ hai của đầu ra là abcd abcd false

9. Cho:

```
public class Mounds {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder();  
        String s = new String();  
        for(int i = 0; i < 1000; i++) {  
            s = " " + i;  
            sb.append(s);  
        }  
        // done with loop  
    }  
}
```

Nếu bộ thu gom rác KHÔNG chạy trong khi mã này đang thực thi, thì có khoảng bao nhiêu đối tượng sẽ tồn tại trong bộ nhớ khi vòng lặp đư ợc thực hiện?

- A. Dư ới 10
- B. Khoảng 1000
- C. Khoảng 2000
- D. Khoảng 3000
- E. Khoảng 4000

10. Cho:

```
3. class Box {  
4.     int size;  
5.     Box(int s) { size = s; }  
6. }  
7. public class Laser {  
8.     public static void main(String[] args) {  
9.         Box b1 = new Box(5);  
10.        Box[] ba = go(b1, new Box(6));  
11.        ba[0] = b1;  
12.        for(Box b : ba) System.out.print(b.size + " ");  
13.    }  
14.    static Box[] go(Box b1, Box b2) {  
15.        b1.size = 4;  
16.        Box[] ma = {b2, b1};  
17.        return ma;  
18.    }  
19. }
```

Kết quả là gì?

- A. 4 4
 - B. 5 4
 - C. 6 4
 - D. 4 5
 - E. 5 5
- F. Biên dịch không thành công

11. Cho:

```
public class Hedges {  
    public static void main(String[] args) {  
        String s = "JAVA";  
        s = s + "rocks";  
        s = s.substring(4,8);  
        s.toUpperCase();  
        System.out.println(s);  
    }  
}
```

Kết quả là gì?

- A. JAVA
- B. JAVAROCKS
- C. dá

- D. đá
- E. ROCKS
- F. ROCK
- G. Biên dịch không thành công

12. Cho:

```
1. import java.util.*;  
2. class Fortress {  
3.     private String name;  
4.     private ArrayList<Integer> list;  
5.     Fortress() { list = new ArrayList<Integer>(); }  
6.  
7.     String getName() { return name; }  
8.     void addToList(int x) { list.add(x); }  
9.     ArrayList getList() { return list; }  
10. }
```

Dòng mã nào (nếu có) phá vỡ tính đóng gói? (Chọn tất cả các áp dụng.)

- A. Dòng 3
- B. Dòng 4
- C. Dòng 5
- D. Dòng 7
- E. Dòng 8
- F. Dòng 9
- G. Lớp đã được đóng gói tốt

13. Cho:

```
import java.util.function.Predicate;
public class Sheep {
    public static void main(String[] args) {
        Sheep s = new Sheep();
        s.go(() -> adder(5, 1) < 7); // line A
        s.go(x -> adder(6, 2) < 9); // line B
        s.go(x, y -> adder(3, 2) < 4); // line C
    }
    void go(Predicate<Sheep> e) {
        Sheep s2 = new Sheep();
        if(e.test(s2))
            System.out.print("true ");
        else
            System.out.print("false ");
    }
    static int adder(int x, int y) {
        return x + y;
    }
}
```

Kết quả là gì?

- A. đúng đúng sai
- B. Biên dịch không thành công chỉ do lỗi ở dòng A C.
- Biên dịch không thành công chỉ do lỗi ở dòng B D.
- Biên dịch không thành công chỉ do lỗi ở dòng C E.
- Biên dịch không thành công chỉ do lỗi ở dòng A và B F .
- Biên dịch không thành công chỉ do lỗi ở dòng A và C G.
- Biên dịch không thành công chỉ do lỗi ở dòng A, B và C H. Biên dịch không thành công vì những lý do không được liệt kê

14. Cho:

```
import java.time.*;
import java.time.format.*;
public class Shiny {
    public static void main(String[] args) {
        DateTimeFormatter f1 =
            DateTimeFormatter.ofPattern("MMM dd, yyyy");
        LocalDate d = LocalDate.of(2018, Month.JANUARY, 15);
        LocalDate d2 = d.plusDays(1);
        System.out.print(f1.format(d) + " ");
        System.out.println(d2.format(f1));
    }
}
```

Kết quả là gì?

- A. 2018-01-15 2018-01-15
- B. 2018-01-15 2018-01-16
- C. Ngày 15 tháng 1 năm 2018 Ngày 15 tháng 1 năm 2018
- D. Ngày 15 tháng 1 năm 2018 Ngày 16 tháng 1 năm 2018
- E. Biên dịch không thành công
- F. Một ngoại lệ được đưa ra trong thời gian chạy

15. Cho:

```
import java.util.*;
public class Jackets {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<>(); // line 5
        myList.add(new Integer(5));
        myList.add(42); // line 7
        myList.add("113"); // line 8
        myList.add(new Integer("7")); // line 9
        System.out.println(myList);
    }
}
```

Kết quả là gì?

- A. [5, 42, 113, 7]
- B. Biên dịch không thành công chỉ do lỗi ở dòng 5
- C. Biên dịch không thành công do chỉ có lỗi ở dòng 8
- D. Biên dịch không thành công chỉ do lỗi ở dòng 5 và 8
- E. Biên dịch không thành công chỉ do lỗi ở dòng 7 và 8

F. Biên dịch không thành công chỉ do lỗi ở dòng 5, 7 và 8 G. Biên

dịch không thành công chỉ do lỗi ở dòng 5, 7, 8 và 9

16. Cho rằng adder () trả về một int, những lambdas Predicate nào hợp lệ ?

(Chọn tất cả các áp dụng.)

- A. $x, y \rightarrow 7 < 5$
- B. $x \rightarrow \{ \text{return adder}(2, 1) > 5; \}$
- C. $x \rightarrow \text{return adder}(2, 1) > 5;$
- D. $x \rightarrow \{ \text{int } y = 5; \\ \text{int } z = 7; \\ \text{adder}(y, z) > 8; \}$
- E. $x \rightarrow \{ \text{int } y = 5; \\ \text{int } z = 7; \\ \text{return adder}(y, z) > 8; \}$
- F. $(\text{MyClass } x) \rightarrow 7 > 13$
- G. $(\text{MyClass } x) \rightarrow 5 + 4$

17. Cho:

```
import java.util.*;  
public class Baking {  
    public static void main(String[] args) {  
        ArrayList<String> steps = new ArrayList<String>();  
        steps.add("knead");  
        steps.add("oil pan");  
        steps.add("turn on oven");  
        steps.add("roll");  
        steps.add("turn on oven");  
        steps.add("bake");  
        System.out.println(steps);  
    }  
}
```

Kết quả là gì?

- A. [nhào, chảo dầu, cuộn, bật lò, nư ớng]
- B. [nhào, chảo dầu, bật lò, cuộn, nư ớng]
- C. [nhào, chảo dầu, bật lò, cuộn, bật lò, nư ớng]
- D. Đầu ra không thể đoán trước đư ợc
- E. Biên dịch không thành công

F. Một ngoại lệ được đưa ra trong thời gian chạy

18. Cho:

```
import java.time.*;
public class Bachelor {
    public static void main(String[] args) {
        LocalDate d = LocalDate.of(2018, 8, 15);
        d = d.plusDays(1);
        LocalDate d2 = d.plusDays(1);
        LocalDate d3 = d2;
        d2 = d2.plusDays(1);
        System.out.println(d + " " + d2 + " " + d3); // line X
    }
}
```

Đó là sự thật? (Chọn tất cả các áp dụng.)

- A. Sản lư ợng là: 2018-08-16 2018-08-17 2018-08-18
- B. Sản lư ợng là: 2018-08-16 2018-08-18 2018-08-17
- C. Sản lư ợng là: 2018-08-16 2018-08-17 2018-08-17
- D. Tại dòng X, không đổi tư ợng LocalDate đủ điều kiện để thu gom rác
- E. Tại dòng X, một đổi tư Ợng LocalDate đủ điều kiện để thu gom rác
- F. Tại dòng X, hai đổi tư Ợng LocalDate đủ điều kiện để thu gom rác
- G. Biên dịch không thành công

19. Cho rằng e đề cập đến một đối tư Ợng triển khai Vị từ, có thể là đoạn mã hoặc câu lệnh hợp lệ? (Chọn tất cả các áp dụng.)

- A. if (e.test (m))
- B. công tắc (e.test (m))
- C. while (e.test (m))
- D. e.test (m)? "có không"; E. do {} while (e.test (m)); F.
- System.out.print (e.test (m)); G.
- boolean b = e.test (m);

CÂU TRẢ LỜI TỰ KIỂM TRA

1. A và D đúng. Các hoạt động Chuỗi đang hoạt động trên một

(bị mất) Chuỗi không phải Chuỗi s. Các hoạt động StringBuilde hoạt động từ trái sang phải.

B, C và E không chính xác dựa trên những điều trên. (Mục tiêu 9.2 và 9.1 của OCA)

2. Đúng. Bốn phần tử đầu tiên của mảng ngựa chứa các Chuỗi, nhưng phần tử thứ năm là null, vì vậy lời gọi toUpperCase () cho phần tử thứ năm ném ra một NullPointerException.

A, B, C, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 4.1 và 1.3 của OCA)

3. Đúng. Khai báo Unicode phải được đặt trong dấu ngoặc kép: '\ u004e'. Nếu điều này được thực hiện, câu trả lời sẽ là A, nhưng sự bình đẳng đó không có trong kỳ thi OCA.

A, B, C và D là không chính xác dựa trên các điều trên. (Mục tiêu 2.1 và 4.1 của OCA)

4. Đúng. Một ClassCastException được ném ở dòng 7 vì o1 tham chiếu đến một int [], không phải là một int []. Nếu dòng 7 bị loại bỏ, đầu ra sẽ là 4.

A, B, D, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 4.2 của OCA)

5. Đúng. Các phần tử ArrayList được tự động chèn vào theo thứ tự nhập; chúng không được sắp xếp tự động. ArrayLists sử dụng các chỉ mục dựa trên 0 và add () cuối cùng sẽ chèn một phần tử mới và dịch chuyển các phần tử còn lại trở lại.

A, C, D, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 9.4 của OCA)

6. Đúng. da tham chiếu đến một đối tượng kiểu "Mảng hàng chục" và mỗi đối tượng Hàng chục được tạo đi kèm với đối tượng "mảng int" của riêng nó. Khi đến dòng 14, chỉ có đối tượng Hàng chục thứ hai (và đối tượng "mảng int" của nó) là không thể truy cập được.

A, B, D, E và G là không chính xác dựa trên các điều trên. (Mục tiêu 4.1 và 2.4 của OCA)

7. Đúng. Mảng hai chiều là một "mảng của các mảng." Độ dài của ba là 2 vì nó chứa 2 mảng một chiều. Mảng

các chỉ mục dựa trên 0, vì vậy ba [1] đè cập đến mảng thứ hai của ba.

A, B, D, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 4.2 của OCA)

8. D và F đúng. Mặc dù các đối tượng chuỗi là bất biến, tham chiếu đến Chuỗi có thể thay đổi. Đoạn mã s1 += "d"; tạo một đối tượng String mới. Các đối tượng StringBuilder có thể thay đổi được, vì vậy append () đang thay đổi đối tượng StringBuilder duy nhất mà cả hai tham chiếu StringBuilder đều tham chiếu đến.

A, B, C, E và G là không chính xác dựa trên các điều trên. (Mục tiêu 9,2 và 9,1 của OCA)

9. B đúng. StringBuilder có thể thay đổi, vì vậy tất cả các append () các lần gọi đang hoạt động lặp đi lặp lại trên cùng một đối tượng StringBuilder . Tuy nhiên, các chuỗi là bất biến, vì vậy mọi thao tác nối String đều dẫn đến một đối tượng String mới. Ngoài ra, chuỗi "" được tạo một lần và được sử dụng lại trong mỗi lần lặp vòng lặp.

A, C, D và E không chính xác dựa trên những điều trên. (Mục tiêu 9,2 và 9,1 của OCA)

10. A đúng. Mặc dù b1 của main () là một biến tham chiếu khác với b1 của go () , chúng tham chiếu đến cùng một đối tượng Box .

B, C, D, E và F không chính xác dựa trên các điều trên. (Mục tiêu của OCA 4.1, 6.1 và 6.6)

11. D đúng. Lệnh gọi substring () sử dụng chỉ mục dựa trên 0 và đối số thứ hai là độc quyền, do đó, ký tự ở chỉ mục 8 KHÔNG được bao gồm. Lệnh gọi toUpperCase () làm cho một đối tượng String mới bị mất ngay lập tức. Lời gọi toUpperCase () KHÔNG ảnh hưởng đến Chuỗi được tham chiếu bởi s .

A, B, C, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 9.2 của OCA)

12. F đúng. Khi đóng gói một đối tượng có thể thay đổi như ArrayList, getter của bạn phải trả về một tham chiếu đến bản sao của đối tượng, không chỉ là tham chiếu đến đối tượng ban đầu.

A, B, C, D, E và G là không chính xác dựa trên các điều trên. (Mục tiêu 6.5 của OCA)

13. F đúng. Các lambdas dự đoán nhận chính xác một tham số; phần còn lại của mã là chính xác.
- A, B, C, D, E, G và H là không chính xác dựa trên các điều trên. (Mục tiêu 9.5 của OCA)
14. D đúng. Việc gọi phương thức plusDays () sẽ tạo ra một đối tượng mới và cả LocalDate và DateTimeFormatter đều có phương thức format () .
- A, B, C, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 9.3 của OCA)
15. C đúng. Lỗi duy nhất trong mã này là cố gắng thêm một chuỗi vào một ArrayList của các đối tượng trình bao bọc số nguyên . Dòng 7 sử dụng autoboxing và dòng 6 và 9 thể hiện việc sử dụng hai hàm tạo của lớp wrapper.
- A, B, D, E, F và G là không chính xác dựa trên các điều trên. (Mục tiêu OCA 2.5 và 9.4)
16. B, E và F sử dụng đúng cú pháp.
- A, C, D và G không chính xác. A chuyển hai tham số. C, một giá trị trả về, phải nằm trong một khôi mã và các khôi mã phải nằm trong dấu ngoặc nhọn. D, một khôi, phải có một câu lệnh trả về . G, kết quả, không phải là boolean. (Mục tiêu 9.5 của OCA)
17. C đúng. ArrayLists có thể có các mục nhập trùng lặp.
- A, B, D, E và F là không chính xác dựa trên các điều trên. (Mục tiêu 9.4 của OCA)
18. B và E đúng. Có tổng cộng bốn đối tượng LocalDate được tạo, như ng đối tượng được tạo bằng phương thức of () bị bỏ qua trên dòng mã tiếp theo khi biến tham chiếu của nó được gán cho đối tượng LocalDate mới được tạo thông qua lệnh gọi plusDays () đầu tiên . Các biến tham chiếu được hoán đổi một chút, điều này giải thích cho các ngày không in theo thứ tự thời gian.
- A, C, D, F và G là không chính xác dựa trên các điều trên. (Mục tiêu 2.4 và 9.3 của OCA)
19. A, C, D, E, F và G đều đúng; tất cả chúng đều yêu cầu boolean.
- B không chính xác. Một công tắc không sử dụng boolean. (Mục tiêu 9.5 của OCA)



Một

Giới thiệu về Tải xuống

Sách điện tử này đi kèm với Phần mềm Thi Thực hành Báo chí Oracle có thể tải xuống miễn phí, có thể tải xuống bằng cách liên kết được cung cấp trong phụ lục này. Phần mềm này dễ cài đặt trên bất kỳ máy tính Mac hoặc Windows nào và phải được cài đặt để truy cập tính năng Thực hành Thi.

yêu cầu hệ thống

các cửa sổ

- Bộ xử lý tương thích x86 2,33GHz trở lên hoặc Intel Atom™ 1,6GHz hoặc bộ xử lý nhanh hơn cho các thiết bị netbook
- Microsoft® Windows Server 2008, Windows 7, Windows 8.1 Classic hoặc Windows 10
- 512MB RAM (khuyến nghị 1GB)

hệ điều hành Mac

- Bộ xử lý Intel® Core™ Duo 1.83GHz hoặc nhanh hơn
- Mac OS X v10.7 trở lên
- 512MB RAM (khuyến nghị 1GB)

Tải xuống từ McGraw-Hill

Trung tâm Truyền thông Chuyên nghiệp

Để tải xuống bảng thuật ngữ, nội dung bổ sung và Bài kiểm tra Thực hành Báo chí Oracle Phần mềm, hãy truy cập Trung tâm Truyền thông của McGraw-Hill Professional bằng cách nhấp vào liên kết

Phần mềm, hãy truy cập Trung tâm Truyền thông của McGraw-Hill Professional bằng cách nhấp vào liên kết bên dưới và nhập ISBN của sách điện tử này và địa chỉ e-mail của bạn. Sau đó, bạn sẽ nhận được một tin nhắn e-mail với liên kết tải xuống cho nội dung bổ sung.

<http://mhprofessional.com/mediacenter>

ISBN của sách điện tử này là 1260011380.

Sau khi bạn nhận được email từ Trung tâm Truyền thông của McGraw-Hill Professional, hãy nhấp vào liên kết đi kèm để tải xuống các bài kiểm tra thực hành. Nếu bạn không nhận được email, hãy nhớ kiểm tra thư mục thư rác của bạn.

Cài đặt phần mềm thi thực hành

Làm theo hướng dẫn bên dưới cho Windows hoặc Mac OS.

các cửa sổ

Bước 1 Mở tệp InstallerforPC.zip. Bạn sẽ cần giải nén tệp và giải nén hoặc sao chép và dán nội dung vào ổ cứng của mình.

Bước 2 Tìm tệp Installer.exe và nhấp đúp vào tệp. Sau một lúc, trình cài đặt sẽ mở ra.

Bước 3 Làm theo hướng dẫn trên màn hình để cài đặt ứng dụng.

hệ điều hành Mac

Bước 1 Mở tệp InstallerforMac.zip. Bạn sẽ cần giải nén tệp và giải nén hoặc sao chép và dán nội dung vào ổ cứng của mình.

Bước 2 Sau một lúc, nội dung của tệp .zip sẽ được hiển thị.

Bước 3 Nhấp đúp vào Installer để bắt đầu cài đặt.

Bước 4 Làm theo hướng dẫn trên màn hình để cài đặt ứng dụng.

LƯU Ý: Nếu bạn gặp lỗi khi cài đặt phần mềm, vui lòng đảm bảo rằng chương trình chống vi-rút hoặc bảo mật internet của bạn đã bị vô hiệu hóa và thử cài đặt lại phần mềm. Bạn có thể bật lại chương trình chống vi-rút hoặc bảo mật internet sau khi cài đặt xong.

Chạy phần mềm thi thực hành

Thực hiện theo các hướng dẫn bên dưới sau khi bạn đã hoàn tất cài đặt phần mềm.

các cửa sổ

Sau khi cài đặt, bạn có thể khởi động ứng dụng bằng một trong hai phương pháp dưới đây:

- Nhấp đúp vào biểu tượng Thực hành Oracle Press Java SE 8 trên màn hình của bạn, hoặc 2.

Đi tới menu Bắt đầu và nhấp vào Chạy trình hoặc Tất cả chương trình. Nhấp vào Oracle Nhấn Thực hành Java SE 8 để khởi động ứng dụng.

hệ điều hành Mac

Mở thư mục Oracle Press Java SE 8 Practice bên trong thư mục ứng dụng của máy Mac và nhấp đúp vào biểu tượng Oracle Press Java SE 8 Practice để chạy ứng dụng.

Thực hành các tính năng của phần mềm thi

Phần mềm Kiểm tra Thực hành cung cấp cho bạn một mô phỏng của kỳ thi thực tế.

Phần mềm này có một chế độ tùy chỉnh có thể được sử dụng để tạo các câu đố theo miền mục tiêu của kỳ thi. Chế độ câu đố là chế độ mặc định. Để khởi chạy mô phỏng kỳ thi, hãy chọn một trong các nút kỳ thi OCA ở đầu màn hình hoặc chọn hộp kiểm Chế độ kiểm tra ở cuối màn hình và chọn kỳ thi OCA trong cửa sổ tùy chỉnh.

Số lượng câu hỏi, loại câu hỏi và thời gian cho phép trên mô phỏng kỳ thi nhằm trở thành một đại diện của kỳ thi trực tiếp. Chế độ kiểm tra tùy chỉnh bao gồm gợi ý và tài liệu tham khảo, đồng thời giải thích câu trả lời chuyên sâu được cung cấp thông qua tính năng Phản hồi.

Khi bạn khởi chạy phần mềm, một màn hình đồng hồ kỹ thuật số sẽ xuất hiện ở góc trên bên phải của cửa sổ câu hỏi. Đồng hồ sẽ tiếp tục đếm trừ khi bạn chọn kết thúc bài thi bằng cách chọn Lớp Bài thi.

Xóa cài đặt

Phần mềm Kiểm tra Thực hành được cài đặt trên ổ cứng của bạn. Để có kết quả tốt nhất cho việc xóa các chương trình bằng PC Windows, hãy sử dụng Bảng điều khiển | Gỡ cài đặt một tùy chọn Chương trình và sau đó chọn Oracle Press Java SE 8 Practice để gỡ cài đặt.

Để có kết quả tốt nhất cho việc xóa các chương trình bằng máy Mac, hãy chuyển đến thư mục Thực hành Oracle Press Java SE 8 bên trong thư mục ứng dụng của bạn và kéo biểu tượng "Thực hành Oracle Press Java SE 8" vào thùng rác.

Nhấn biểu tượng Java SE 8 Practice "vào thùng rác.

Cứu giúp

Tệp trợ giúp được cung cấp thông qua nút Trợ giúp trên trang chính ở góc trên cùng bên phải. Một tệp README cũ ng được bao gồm trong thư mục Nội dung thư ứng.

Hỗ trợ kỹ thuật

Thông tin Hỗ trợ Kỹ thuật được cung cấp trong các phần sau theo tính năng.

Khắc phục sự cố Windows 8

Các lỗi đã biết sau trên Windows 8 đã được báo cáo. Vui lòng xem bên dưới để biết thông tin về cách khắc phục các sự cố đã biết này.

Nếu bạn gặp lỗi khi cài đặt phần mềm, chẳng hạn như "Không thể cài đặt ứng dụng do tệp trình cài đặt bị hỏng. Hãy thử lấy trình cài đặt mới từ tác giả ứng dụng," bạn có thể cần phải tắt chương trình chống vi-rút hoặc bảo mật Internet và thử cài đặt lại phần mềm. Bạn có thể bật lại chương trình chống vi-rút hoặc bảo mật Internet sau khi cài đặt xong.

Để biết thêm thông tin về cách tắt chương trình chống vi-rút trong Windows, vui lòng truy cập trang web của nhà cung cấp phần mềm cho chương trình chống vi-rút của bạn. Ví dụ: nếu bạn sử dụng các sản phẩm Norton hoặc MacAfee, bạn có thể cần phải truy cập trang web Norton hoặc MacAfee và tìm kiếm "cách tắt phần mềm chống vi-rút trong Windows 8". Các chương trình chống vi-rút khác với công nghệ tương ứng lửa, vì vậy hãy đảm bảo tắt chương trình chống vi-rút và đảm bảo bật lại chương trình sau khi bạn đã cài đặt phần mềm thi thực hành.

Mặc dù Windows không bao gồm phần mềm chống vi-rút mặc định, như ng Windows thường có thể phát hiện phần mềm chống vi-rút do bạn hoặc nhà sản xuất máy tính của bạn cài đặt và thường hiển thị trạng thái của bất kỳ phần mềm nào như vậy trong Trung tâm hành động, nằm trong Bảng điều khiển bên dưới Hệ thống và Bảo mật (chọn Xem lại Trạng thái Máy tính của Bạn). Tính năng trợ giúp của Window cũ ng có thể cung cấp thêm thông tin về cách phát hiện phần mềm chống vi-rút của bạn. Nếu phần mềm chống vi-rút đang bật, hãy kiểm tra tính năng Trợ giúp đi kèm với phần mềm đó để biết thông tin về cách tắt phần mềm đó.

Windows sẽ không phát hiện tất cả phần mềm chống vi-rút. Nếu phần mềm chống vi-rút của bạn không được hiển thị trong Trung tâm Hành động, bạn có thể thử nhập tên của phần mềm hoặc

hiển thị trong Trung tâm hành động, bạn có thể thử nhập tên của phần mềm hoặc nhà xuất bản vào trống tìm kiếm của Menu Đầu.

Hỗ trợ Nội dung Giáo dục McGraw-Hill

Đối với các câu hỏi liên quan đến Bảng chú giải thuật ngữ hoặc nội dung phần thư bổ sung, hãy gửi e-mail techsolutions@mhedu.com hoặc truy cập <http://mhp.softwareassist.com>.

Đối với các câu hỏi liên quan đến nội dung sách, e-mail hep_customer.service@mheducation.com. Đối với khách hàng bên ngoài Hoa Kỳ, hãy gửi e-mail international_cs@mheducation.com.

MỤC LỤC

Một

chuỗi bị bỏ rơ i, 344-346

lớp trừu tư ợng, 19 hàm tạo,

133, 136 tạo, 23 triển

khai, 123 so với giao

diện, 25 tổng quan, 21-

22 phư ơ ng thức trừu

tư ợng

ké thừa, 93

tổng quan, 45-48

triển khai lớp con, 106 truy

cập và sửa đổi truy cập, 30-33 lớp,

17-21 hàm tạo, 135 đóng gói,

89 điểm chính, 71 cấp, 54 biến

cục bộ, 42-43 phư ơ ng thức

đư ợc nạp chồng, 111 phư ơ ng

thức ghi đè , 107-108 private,

34-36 thành viên đư ợc bảo vệ

và mặc định, 36-42 công khai,

33-34 phư ơ ng thức tinh và

biến, 151-154 phư ơ ng thức add

() trong ArrayList, 383 phép gán ghép

phức hợp, 236

toán tử, [244](#)
ưu tiên, [257](#) ký
hiệu và (&) toán tử
theo chiều bit, 251-252 toán tử logic, 252-255 ưu tiên, [257](#) biểu thức AND, 252-255 dấu ngoặc nhọn (<>) trong mã chung, [381](#) mảng ẩn danh, 372-373 phần phụ () phưƠng thức, 353-354 thêm chuỗi, 341-342, 353-354 ứng dụng, khởi chạy, 11-12 tùy chọn @argfiles, [11](#) đối số cuối cùng, 44-45 phưƠng thức đưƠc nạp chồng, [111](#), [113](#) phưƠng thức bị ghi đè, [108](#) so với tham số, 49- 50 siêu hàm tạo, [137](#) danh sách đối số biến, 49-50

toán tử số học, 244-249 cơ bản, 244-245 tăng và giảm, 248-250 điểm chính, 260-261 phần dư , 245-246 nối chuỗi, 246-248

Lớp ArithmeticException, [322](#)
Lớp ArrayIndexOutOfBoundsException, [306](#) mô tả, [322](#) chỉ mục mảng ngoài phạm vi, [369](#) siêu lớp, 308-309

Lớp ArrayList, 13-14
autoboxing, 384-388
thông tin cơ bản, 380-381 cú pháp kim cương, 388-389 bản sao, [383](#)

các điểm chính,
[401](#) phư ơ ng thức,
382-384 sử dụng,
379-381 mảng, [363](#) câu
trúc, 364-367, 370-372 khai
báo, 57-59, 363-364, 370-372 giá
trị phần tử mặc định, 198-199, [202](#), [221](#)
phép gán phần tử, 374-378 nâng cao cho
các vòng lặp, 292-293 chỉ mục, [12](#), [368](#)
khởi tạo, 368-374 phiên bản so sánh, [244](#)
điểm chính, [76](#), 399-400 thuộc tính độ
dài, [350](#) phép gán tham chiếu, 376-378
trả về, [131](#) bộ ASCII , [53](#) lớp
`AssertionError`, [316](#), [321](#) phần tử mảng
bài tập, 374-378 lỗi trình biên dịch,
188-189 số dấu phẩy động, [188](#) điểm chính,
khả năng tư ơ ng thích đối tư ợng 220-221,
[242](#) toán tử, 182-183, 235-236, [257](#) nguyên
thủy, 183-185, [190](#) biến tham chiếu, 190-191,
202-203, 376-378 dấu hoa thị (*) toán tử
gán ghép, [236](#) câu lệnh nhập, [14](#), [16](#) phép
nhân, [244](#) ưu tiên, [257](#) tự động đóng hộp
với ArrayLists, 384-388 biến cục bộ tự
động , 55-57 biến tự động. Xem các biến
cục bộ

B

dấu gạch chéo ngư ợc (\) cho các ký tự thoát,
[182](#) số nguyên cơ số [2](#) (nhị phân), [178](#) số
nguyên cơ số [8](#) (bát phân), 178-179 số nguyên
cơ sở [10](#) (thập phân), [178](#) ký tự cơ bản [16](#)
(thập lục phân), [178](#), [180](#) cơ bản cho vòng
lặp, [288](#) biểu thức điều kiện, 289-290 khai
báo và khởi tạo, 288-289 biểu thức lặp,
289-290 vấn đề về vòng lặp, 290-292 hành
vi, mô tả, phưƠng thức [2](#) between (),
[361](#) số nguyên nhị phân (cơ số 2), [178](#)
ký tự nhị phân, [179](#) toán tử bitwise , Khởi
tạo khối 251-252, 145-147 biến, [193](#) kiểu
boolean và giá trị độ sâu bit, [53](#) giá trị
mặc định, [196](#) vòng lặp do, [287](#) vòng lặp for,
288-289 câu lệnh if, 276-277 toán tử đảo
ngư ợc, 255-256 ký tự, [181](#) toán tử quan hệ,
[236](#) vòng lặp while, [286](#) dấu ngoặc nhọn
({}). Xem dấu ngoặc nhọn ({}) phân nhánh
if-else, câu lệnh switch 273-277. Xem câu
lệnh switch câu lệnh break trong vòng
lặp for, [290](#)

các điểm chính,
[326](#) cấu trúc vòng lặp, 294-
296 câu lệnh chuyển đổi, [278](#), 281-
283 lỗi. Xem các hằng số trư ờng hợp
ngoại lệ byte, 278-279 giá trị mặc định,
[196](#) và int, [184](#) phạm vi, [53](#)

C

các lớp dữ
liệu lịch, 357-358
lớp nhà máy, [359](#) định
dạng, [362](#) tính bất
biến, 358-359 điểm chính,
398-399 sử dụng và thao
tác, 360-361 ngoại lệ ngăn xếp cuộc
gọi, 303-305
CamelCase, [9](#)
khả năng của chuỗi, [354](#)
dấu mũ (^) toán tử
bitwise, 251-252 toán tử HOẶC
độc quyền, [255](#) định danh phân
biệt chữ hoa chữ thư ờng, [7](#) so sánh
chuỗi, 348-349 câu lệnh trư ờng
hợp, 278-280 lần ép, [184](#) phép
gán, [235](#) rõ ràng, 185 -186 ẩn, [186](#)
điểm chính, tổng quan [159](#), [220](#), độ
chính xác 118-121, 188-189

nguyên thủy, 185-
188 mệnh đề bắt, 299-
300 phư ơ ng thức chuỗi,
[356](#) hàm tạo chuỗi, [134](#) ký
tự và độ sâu bit kiểu
char, [53](#) hằng số
trư ờng hợp, 278-279 so
sánh, [237](#) giá trị mặc
định, [196](#) ký tự, 181-
182 phư ơ ng thức charAt
(), 348-349 đã kiểm tra xử
lý ngoại lệ, 314-315 thực
thi giao diện, [123](#)
phư ơ ng thức đư ợc nạp chòng,
[111](#) phư ơ ng thức bị ghi đè,
[108](#), [110](#) ChronoUnit enum, [361](#)
tệp .class, [12](#) mô tả lớp
ClassCastException, [322](#) downcast,
[119](#) lớp truy cập, 17-21 biên
dịch, [11](#) các nhà xây dựng.
Xem các hàm tạo khai báo,
định nghĩa 17-18, mô tả 9-10, [2](#)
mở rộng, [18](#), [102](#) cuối cùng,
20-21, 59-61 câu lệnh nhập,
triển khai giao diện 13-14, [123](#) khởi chạy ứng
dụng với java, 11-12 main () phư ơ ng thức, 12-
13 khai báo thành viên, [28](#) tên, 8-9, [151](#) ghi
đè, [3](#)

quy tắc khai báo tệp nguồn, 10
trình bao bọc, 320 dọn dẹp bộ
sửu tập rác, 218 phuơng thức
clear (), 383 quy ước mã, 7-9 bộ
sửu tập, 58 dấu hai chấm (:) toán
tử có điều kiện, 250-251 nhãn, 296
dấu phẩy (,) trong vòng lặp for ,
288-289 biến, 185 nhận xét
trong tệp mã nguồn, 10 so sánh
instanceof, 242-244 toán tử quan
hệ, 236-241 chuỗi, 348-349
trình biên dịch và biên dịch
phôi, 119 triển khai giao diện, 122-
123 javac, 11 phuơng thức nạp
chồng, 114 phép gán lỗi trình
biên dịch, 188-189 instanceof,
244 phép gán ghép với phôi, 189
toán tử, 236 chuỗi, 247 phuơng
thức concat (), 348-349 chuỗi
nối, 246-248, 261, 348-349 lớp cụ
thể các phuơng thức trừu tư ợng
được thực hiện bởi, 106 tạo, 23
lớp con, 46-47 phuơng pháp cụ thể, 102

các điểm chính của toán

tử có điều kiện,

tổng quan [261](#) , 250-

251 điều kiện thực hiện

vòng lặp, [287](#) trong

vòng lặp for, 288-290

phân nhánh if-else, 273-277 câu

lệnh chuyển đổi. Xem các câu lệnh switch while vòng lặp, [286](#) hằng

số nội dung lớp cụ thể, 65-66

hằng số

case, 278-279

enum, [62](#) giao

diện, 26-27 tên, [9](#)

Lớp chuỗi, [347](#) xây

dựng mảng, 364-367, 370-372 hàm tạo, [132](#) kiến

thức cơ bản, [133](#) dữ liệu lịch, [359](#) chuỗi,

[134](#) khai báo, 50-51 mặc định, 135-137

enums, 65-66 kế thừa, [93](#), [140](#) điểm

chính, [75](#), 160-161 quá tải, 140-145 quy

tắc, 135-136 chuỗi, [341](#) lệnh gọi super

() và this (), [144](#) phư ơng thức chứa

(), [383](#) điểm chính của câu lệnh tiếp

tục, [326](#) cấu trúc vòng lặp, 294-295 điều

khiển, [17](#) quy ước mã, 7-9

số nhận dạng,
6 tên, 2 kiểu
trả về chuyên
đối, 131 kiểu.

Xem giảm chi phí phôi, thiết kế hư hỏng đối tượng, 100 truờng hợp
đêm, 148 tham chiếu, 212 trả về hiệp phương sai, 129-130 thực thi đa
nền tầng, 5 phương thức trừu tượng dâu ngoặc nhọn ({}), 46 mảng,
370, 372, 374 thành viên lớp, 196, 199 biểu thức if, 273, 275
biến phiên bản, 196 lambdas, 395 phương thức, 46 tùy chọn, 273

D

Hậu tố D, 181

dấu gạch ngang

(-) toán tử gán ghép, 236 toán tử giảm
dần, 248-249 mức độ ưu tiên, 257 phép
trừ, 244 dữ liệu ngày tháng sử dụng và
thao tác, định dạng 360-361, 362

Lớp DateTimeFormatter, 358, 361-362

Kim cương chét chóc của Tử thần,
102 số nguyên thập phân (cơ số
10), 178 chữ thập phân có dấu gạch
dưới, 179 tuyên bố

mảng, 57-59, 363-364, 370-372 cơ
bản cho vòng lặp, 288-289 thành
viên lớp, [28](#) lớp, 17-18 hàm tạo,
50-51 nâng cao cho vòng lặp, [293](#)
phân tử enum, 65-67 enum, 62-64
ngoại lệ, 312-317 hàng số giao
diện, 26-27 giao diện, 23-26 đa
hình, [381](#) biến tham chiếu, 53-54
kiểu trả về, 129-132 quy tắc tệp
nguồn, [10](#) biến, 51-61, 75-76 toán
tử giảm các điểm chính, [261](#) làm
việc với, 248-250 mô tả truy cập
mặc định, [17](#) tổng quan, 18-19 và
đư ợc bảo vệ, [30](#), 36-42 tru ờng
hợp mặc định trong câu lệnh
switch, 283-284

Bảo vệ mặc định, [30](#)
hàm tạo mặc định, 135-
137 phư ơng thức giao
diện, [28](#) biến cá thể
kiểu nguyên thủy và đối tượng, 195-198 lớp xác
định, 9-10 ngoại lệ, 306-307 phư ơng thức delete (),
354-355 xóa chuỗi, 354-355 kim cư ơng cú pháp, 388-
389 điện toán phân tán, [5](#)

phép

gán ghép phân chia, [236](#)
toán tử, [244](#) ưu tiên, [257](#)
vòng lặp do, truy cập [287](#)
dấu chấm (.), 31-32 tên lớp,
[151](#) tham chiếu phiên bản, [151](#)
danh sách đối số biến, [50](#)
phôi kiểu kép, [186](#) giá trị
mặc định, [196](#) ký tự dấu
phẩy động, [180](#) phạm vi,
[53](#) gạch dư ới, [179](#) từ chối,
[119](#) ngoại lệ, [303](#), [312](#)
trùng lặp trong ArrayLists,
[383](#)

E

phần tử trong mảng. Xem mảng
dấu chấm lửng (...) trong danh sách đối
số biến, [50](#) câu lệnh else, 273-277 lợi
ích đóng gói, [100](#) mô tả, [4](#) điểm chính,
[157](#) tổng quan, 88-91 biến tham chiếu,
[389-390](#), [401](#) điểm nhập trong câu
lệnh switch, [281](#) enums, [62](#) hằng chữ
hoa, 278-279 hằng số, [62](#) khai báo,
62-64

khai báo các phần tử, 65-67
kiểm tra đẳng thức, [241](#) điểm chính, 76-77
Lớp ngoại lệ EOFException, [311](#),
[314](#) phép gán dấu bằng (=), [182](#),
[235](#) quyền anh, 386-387 toán tử gán ghép, [236](#) phép thử đẳng thức, 237-241 toán tử quan hệ, 236-237

toán tử bình đẳng và đẳng thức enums, [241](#) đôi tư ợng, 386-387 ưu tiên, [257](#) nguyên thủy, [238](#) tham chiếu, 238-240 chuỗi, 240-241, [281](#) phư ơng thức equals (), 240-241, 386-387 Phư ơng thức equalsIgnoreCase (), 348-349

Lớp lỗi, [307](#)
Lớp ngoại lệ, 306-307
Mô tả lớp ExceptionInInitializerError,
[322](#) khởi init, [147](#) ngoại lệ, 298-299 tạo, 317-318 khai báo, 312-317 định nghĩa, 306-307 phân cấp, 307-309 triển khai giao diện, [123](#)

Đã ném JVM, 319-320 điểm chính, [327](#) danh sách, 321-322 đối sánh, 309-310 phư ơng pháp ghi đè, [108](#), [110](#) ném theo chương trình, 320-321

làn truyền, 303-306
lặp lại, [317](#) nguồn,
[319](#) thử và bắt, 299-
303 dấu chấm than (!)
toán tử đảo ngược boolean,
255-256 ưu tiên, [257](#) toán tử quan
hệ, 236-237 độc quyền OR (XOR),
[255](#) mục thực thi các điểm trong
câu lệnh switch, [281](#) vòng lặp phưƠng
thức exit (), [290](#) lần thử và bắt, [324](#) kiểu truyền
rõ ràng, 185-186 giá trị rõ ràng trong hàm tạo,
[134](#) biểu thức đưƠc nâng cao cho vòng lặp,
[293](#) câu lệnh if, 276-277

bộ ASCII mở rộng, [53](#)
lớp mở rộng, [18](#), [102](#) kế
thừa trong, [94](#) giao
diện, [124](#) mở rộng
sử dụng bất hợp
pháp từ khóa, [127](#)

Mối quan hệ IS-A, [97](#)

F

Hậu tố F,
[181](#) lớp nhà máy,
[359](#) điểm rơi i trong câu lệnh chuyển đổi,
281-283 giá trị sai, [181](#) tính năng và lợi
ích, 3-5
Lớp FileNotFoundException, 311-312 đối
số cuối cùng, 44-45

các lớp cuối cùng, 20-21,
59-61 hằng số cuối cùng, 26-
27 phư ơ ng thức cuối cùng
không tiếp tục sửa đổi thành viên, 43-
45 ghi đè, [108](#) toán tử tăng và giảm
sửa đổi cuối cùng, [250](#) biến, [42](#), [59](#)
phư ơ ng thức `finalize()`, 218-219 cuối
cùng mệnh đề, tính linh hoạt 301-303 từ
hư ơng đối tư ợng, [88](#) kiểu float và gán số
dấu phẩy động, [188](#) phôi, [187](#) lớp, [20](#) phép so
sánh, [237](#) giá trị mặc định, [196](#) chữ, 180-181
phạm vi, [53](#) dấu gạch dư ới, [179](#) điều khiển
luồng, [272](#) ngắn và câu lệnh `continue`, 294-
295 vòng lặp do, [287](#) vòng lặp `for`, 287-
293 phân nhánh `if-else`, 273-277 câu lệnh
có nhãn, 295-297 câu lệnh chuyển đổi. Xem
câu lệnh chuyển đổi câu lệnh không gắn
nhãn, [295](#) vòng lặp `while`, 285-286 cho
vòng lặp cơ bản, 288-292 nâng cao, 292-
293 buộc thoát khỏi vòng lặp, [290](#) buộc
thu gọn rác, 215-218 phư ơ ng thức `format()`,
[362](#) định dạng dữ liệu lịch, [362](#) phân số ,
[180](#)

tên đầy đủ điều kiện, [13](#)

G

dọn dẹp thu gom rác
trư ớc đó, [218](#) cư ỡng ché,
215-218 điểm chính, [222](#)
đôi tư ợng đủ điều kiện,
213-215 tổng quan, [210](#) hoạt
động thu gom rác, [212](#) tổng
quan, [210](#) đang chạy, [210](#) gc ()
method, 215-216 generics, [381](#)
get () phư ơ ng thức, [383](#)
phư ơ ng thức getRuntime (),
đóng gói [216](#) getters, [89](#) mức độ ưu
tiên lớn hơn dấu (>), [257](#) toán tử
quan hệ, 236-237 vùng đư ợc bảo vệ,
[299](#)

H

Mối quan hệ HAS-A các
điểm chính, tổng
quan [157](#), 96-100
đồng rác, [210](#) điểm
chính, [220](#) tổng quan, 176-
177 chữ thập lục phân
(cơ số [16](#)), [178](#), [180](#) ẩn
chi tiết triển khai, [89](#) phân cấp ngoại lệ,
307-309

Tôi

IDE (môi trường phát triển tích hợp), [11](#) mã nhận dạng, [2](#) điểm chính, [70](#) điểm hợp pháp, 6-7
nếu-khác phân nhánh, tổng quan 325-326 ,
273-277

Mô tả lớp IllegalArgumentException,
[322](#) ngoại lệ được ném theo
chương trình, [321](#)
Lớp IllegalStateException, [322](#)
dữ liệu lịch không thay đổi, 358-
359 chuỗi, 340-347 chi tiết
triển khai, ẩn, [89](#) điểm
chính của giao diện triển khai,
tổng quan [159](#) , 122-128 triển khai
sử dụng bất hợp pháp từ khóa,
[127](#)

Mối quan hệ IS-A, [97](#)
phép gán kiểu ngầm, [235](#)
gốc, [186](#) điểm chính
của câu lệnh nhập, [70](#)
tổng quan, 13-14 tệp mã
nguồn, [10](#) điểm chính
của toán tử tĩnh, 14-
16, [261](#) làm việc với,
248-250 chỉ mục

ArrayLists, 380
mảng, 12, 368
ngoài phạm vi, 369
chuỗi, 348-349 dựa
trên không, 12
phương thức indexOf (), ArrayLists, 383
Lớp IndexOutOfBoundsException, 308 triển
khai giao diện gián tiếp, 243 công cụ sửa
đổi quyền truy cập kế thừa, 31-33, 38-39 hàm
tạo, 140 tiến hóa, 93-95

Mối quan hệ HAS-A, 98-100
Mối quan hệ IS-A, 96-97 điểm
chính, 157 bội số, 102, tổng
quan 126-128, 3, 92-93
phương thức kế thừa, ghi
đè, 109 mảng khởi tạo, 202, 368-374
cơ bản cho vòng lặp, 288-289 phần tử
vòng lặp, 370 tham chiếu đối
tư ợng, 201-202 nguyên thủy, 199-
201 biển, 56, 185 khối khởi tạo,
145-147 kế thừa, 93 điểm khóa,
161 lớp bên trong, 17 phương
thức insert (), 354-355 chèn phần
tử chuỗi, 355 phương thức thể hiện
kế thừa, 93 ghi đè, 108 đa hình,
104 biến phiên bản, 54-55

hàm tạo, [134](#) giá trị mặc định, mô tả 195-198, 2, [193](#) trên heap, kê thừa 176-177, [93](#) điểm chính của toán tử instanceof, [260](#) kiểm tra đối tư ợng, 242-243 phiên bản

đếm, [148](#) khối khởi tạo, [146](#) tham chiếu tới, [151](#) khởi tạo, 160-161 lớp Integer, 15-16, [321](#) số nguyên và kiểu dữ liệu int và byte, [184](#) hằng số, 278-279 phôi, 186-187 so sánh, [237](#) giá trị mặc định, 196 ký tự, [178](#) phạm vi, [53](#) toán tử còn lại, [246](#) môi trường phát triển tích hợp (IDE), [11](#) giao diện, 2, [89](#) so với các lớp trừu tư ợng, [25](#) hằng số, 26-27 hàm tạo, [136](#) khai báo, 23-26 mở rộng, [124](#) thực thi, 122-128, [159](#) cách triển khai gián tiếp, [243](#) điểm chính, 72-73 phương pháp, 28-29 tên, 8-9 tổng quan, [3](#)

toán tử đảo ngược, 255-
256 gọi phuơng thức nạp
chồng, 112-115 phuơng thức đa
hình, 103
Lớp IOException đã
kiểm tra các ngoại lệ,
[314](#) tệp, 311-312
Các điểm chính của
môi quan hệ IS-
A, [157](#) tổng quan,
96-97 đa hình, [101](#)
kiểu trả về, [132](#)
ISO Latinh-1 ký tự, [53](#) cách
ly tham chiếu, 214-215 lần lặp
trong cơ bản cho các vòng lặp, 289-290

J

lệnh java, lớp 11-12
java.io.IOException
đã kiểm tra ngoại lệ, [314](#)
tệp, 311-312 mô tả lớp
java.lang.ClassCastException, [322](#)
downcast, [119](#) java.lang.Exception
class, 306-307 java.lang.Integer
class, 15-16, [321](#) java.lang.Object
class, [92](#) phuơng thức
java.lang.Object.equals (), 240-241
java.lang.Runtime class, [216](#)
java.lang.RuntimeException class, [308](#), 312-
314 java.lang.StringBuilder class, [340](#) key point,
398 method, 354 -356 tổng quan, [352](#) so với
StringBuffer, 352-353

lớp `java.time.DateTimeFormatter`, [358](#), 361-362
`java.time.LocalDate` class, [357](#), [362](#)
`java.time.LocalDateTime` class, [357](#), [362](#)
`java.time.LocalTime` class, [357](#), [362](#) `java.time.Period`
class, [358](#), Gói [361](#) `java.time.temporal`, [361](#) giao
diện `java.time.temporal.TemporalAmount`, [358](#) lớp
`java.util.ArrayList`. Xem giao diện `ArrayList` lớp
`java.util.` Chức năng `Predicate`, 393-395 giao diện
`java.util.List`, [381](#) Máy ảo Java (JVM), 5, [11](#) `javac`,
biên dịch với, [11](#)

K

từ khóa, 2, [7](#)

L

Hậu tố L, [180](#)
câu lệnh được gắn nhãn, 295-
297 lambdas, 390-396, [401](#) khởi
chạy ứng dụng, rò rỉ 11-12, bộ
nhớ, [4](#) số nhận dạng hợp pháp,
phương thức 6-7 `length ()`, [350](#)
độ dài của mảng và chuỗi, ít
hơn [350](#) dấu (<) ưu tiên, [257](#) toán
tử quan hệ, 236-237 thư viện, [4](#)
danh sách và giao diện danh
sách, [381](#)

ArrayLists. Xem triển khai lớp `ArrayList`, [381](#) ký tự
nhị phân,
[179](#) `boolean`, [181](#)

ký tự, 181-182 dấu
phẩy động, 180-181 thập
lục phân, 180 số nguyên,
178 điểm chính, 220 bát
phân, 179 phép gán nguyên
thủy, 183-184 chuỗi, 182,
347 mảng cục bộ, 202 tham chiếu
đối tượng cục bộ, 201-202 gốc cục
bộ, 199-201 biến cục bộ, 198-199

công cụ sửa đổi truy cập,
42-43 mô tả, 193 điểm
chính, 74 trên ngăn xếp,
55, 176-177 làm việc với,
55-57

Lớp LocalDate, 357, 362

Lớp LocalDateTime, 357, 362

Lớp LocalTime, 357, 362 toán
tử logic, 251 bitwise, 251-
252 điểm chính, 261-262
không ngắn mạch, 254-
256 ưu tiên, 257 ngắn mạch,
252-254, 276-277 giá trị mặc
định loại dài, 196 phạm vi, 53 câu
trúc vòng lặp, 285 ngắn và tiếp tục,
294-295 thực hiện, 287 khởi tạo
phần tử, 370 cho, 287-293 điểm
chính, 326 trong khi, 285-286

ký tự chữ thư ờng trong chuỗi, [351](#)

M

phư ơ ng thức `main()`,
12-13 ngoại lệ, quá
tải [303-305](#), khả năng
bảo trì [115](#), hư ớng đối tư ợng cho, [88](#)
thuật toán đánh dấu và quét, [212](#) ngoại lệ
phù hợp, 309-310 hằng số `MAX_VALUE`, [16](#) đối
tư ợng tư ơ ng đư ơ ng có ý nghĩa, [240](#) công
cụ sửa đổi thành viên, không truy cập, 43
-50 thành viên truy cập. Xem phần khai báo
các phần bổ trợ truy cập và truy cập, [28](#)
điểm chính, thu thập rác bộ nhớ 74-75. Xem các chuỗi
thu gom rác, [347](#) lần rò rỉ bộ nhớ, [210](#) quản lý bộ nhớ,
tóm tắt [4](#) phư ơ ng thức, 45-48 công cụ sửa đổi truy cập.
Xem các sửa đổi truy cập và truy cập `ArrayLists`, 382-384
chuỗi, [356](#) mô tả, [2](#) enums, 65-66 factory, [359](#) cuối cùng, 43-
45, [59](#) phiên bản, [104](#), [108](#) triển khai giao diện, 122-123
giao diện, 28-29 tên, [9](#) bản địa, [49](#) quá tải, 111-117 bị ghi
đè, 105-111 đệ quy, [320](#)

ngăn xếp, 303-
305 static, 61,
148-150
precisionfp, 49
String, 348-352
StringBuilder, 354-356
đư ợc đồng bộ hóa, 48-49 danh
sách đối số biến, 49-50 dấu trừ
(-) toán tử gán ghép, 236 toán tử
giảm, 248-249 ưu tiên, 257 trừ, 244
bỏ ngữ. Xem tổng quan về toán tử mô-
đun sửa đổi truy cập và truy cập,
245-246 mức độ ưu tiên, 257 cấu trúc
mảng đa chiều, 366 khai báo, 58-59, 364
phép gán tham chiếu, 377-378 đa kế
thừa, 102, 126-128 phép gán ghép phép
nhân, 236 toán tử, 244 mức độ ưu tiên ,
257 lập trình đa luồng, 5 đột biến
trong đóng gói, 89

N

những cái tên
các lớp và giao diện, 8-9
hàng số, 9 hàm tạo, 51,
133, 135 quy ước, toán
tử 2 dấu chấm, 151

đủ điều kiện, 13
nhân, 296 phư ơng
thức, 9 biến bóng,
208-209 biến, 9 chuyển đổi
thu hẹp, 186 phư ơng thức
gốc, 49 số âm từ các phôi, 189
đại diện, 53 lớp lồng nhau, 17
câu lệnh if-else lồng nhau, 273
mảng từ khóa mới , 364 dữ
liệu lịch, 359 hàm tạo
không đối số, 135-136 lớp
NoClassDefFoundError, 322 bỏ
trợ thành viên nonaccess, 19-
20, 43 phư ơng thức trừu
tư ợng, 45-48 đối số cuối
cùng, 44-45 phư ơng thức cuối
cùng, 43-45 điểm chính, 71- 72
phư ơng thức với danh sách đối số biến,
49-50 phư ơng thức gốc, 49 phư ơng
thức nghiêm ngặt, 49 phư ơng thức
đồng bộ hóa, 48-49 toán tử không
bằng nhau (! =), 237 biến tham chiếu
giá trị null, 183, 191 trả về, 130 biến
trình bao bọc, 388 tham chiếu rỗng , 213 lớp
NullPointerException, 314 mảng, 368 mô tả,
322 biến tham chiếu, 319-320

bìen trình bao bọc,
mô tả lớp [388](#)
NumberFormatException,
[322](#) chuyển đổi chuỗi, [321](#)
số nguyên thủy. Xem các số
nguyên thủy với nối chuỗi, 246-247 với dấu gạch dư ới, 178-
179

0

Lớp đổi tư ợng,
[92](#) hứa ợng đổi tư ợng (00), 87-
88 lợi ích, [100](#) ép kiểu ,
118-121 hàm tạo. Xem mô
tả hàm tạo, [4](#) đóng gói, 88-91 kế thừa, 92-
[100](#) khối khởi tạo, 145-147 thực thi giao
diện, 122-128 phư ơng thức đư ợc nạp chòng,
111-117 phư ơng thức bị ghi đè, 105-111 đa
hình, 101-105 kiểu trả về, 129-132 tĩnh,
148-154 đổi tư ợng và tham chiếu đổi tư ợng

mảng, [58](#), 363-364, 374-375
giá trị mặc định, mô tả 196-
198, 2, [183](#) bình đẳng, 386-
387 thu gom rác, 213-219
trên heap, 176-177 khởi tạo,
201-202 so sánh instanceof,
242-244 quá tải các phư ơng
thức, 113-114 đi qua, 205-207
chuỗi dư ới dạng, [341](#)

số nguyên bát phân (cơ số 8), 178-179 của phư ơng thức (), [361](#) mảng một chiều câu trúc, 364-366 phép gán tham chiếu, 376-377 toán hạng, [234](#) toán tử, 233-234 số học, 244-249 phép gán, 235-236 có điều kiện, 250-251 tăng và giảm, 248-250 instanceof, 242-244 logic, 251- 256 ưu tiên, 256-258, [262](#) biểu thức quan hệ, 236-241 OR, 253-255 thứ tự khối khởi tạo phiên bản, [146](#) chỉ mục mảng ngoài phạm vi, [369](#) biến ngoài phạm vi, [194](#) lớp OutOfMemoryException, [218](#) hàm tạo quá tải, 140-145 phư ơng thức quá tải, [111](#) lời gọi, 112-115 điểm chính, [158](#) hợp pháp, [112](#), [116](#) main (), [13](#), [115](#) so với bị ghi đè, [112](#), [117](#) đa hình, [115](#) kiểu trả về, [129](#) phư ơng thức bị ghi đè, 105-109 bất hợp pháp, 110-111 gọi siêu lớp, 109-110 so với quá tải, [112](#), [117](#) đa hình, [115](#) kiểu trả về, 129-130 tinh, [154](#)

ghi đè các
lớp, [3](#)
điểm chính, [158](#)
phư ơng thức riêng, [36](#)

P

ngôn ngữ tập trung vào gói, [17](#)
quyền truy cập mức gói, 18-19
câu lệnh gói trong tệp mã nguồn, truy cập
[10](#) gói, [17](#) lớp trong, [14](#) tham số so với đối
số, 49-50 lambdas, [395](#) đối số ngoặc
đơn (), [44](#), [50](#) điều kiện toán tử, [250](#)
in cho vòng lặp, [288](#) biểu thức if, [273](#), 276-
277 ưu tiên toán tử, 246-247, 258, [262](#)
nối chuỗi, [247](#) phư ơng thức parseInt
(), [321](#) điểm khóa truyền, [221](#) biến tham
chiếu đối tư ợng, 205-207 truyền- by-
value, 206-207 biến nguyên thủy, 207-
208 Peabody, Marc, [185](#) phần trăm dấu
hiệu (%), toán tử [257](#) phần dư , 245-246
lớp Dấu chấm, [358](#), [361](#) ký tự pipe ()
toán tử bitwise, 251-252

Toán tử OR lôgic, 253-255
phép cộng dấu (+), [244](#) toán tử
gán ghép, [236](#) toán tử tăng,
248-249 ưu tiên, [257](#) nối chuỗi, 246-
248 lớp trừu tượng đa hình, [22](#) khai
báo, [381](#) ké thừa, [95](#) điểm chính,
[157](#) quá tải và ghi đè các phư ơng
thức, tổng quan [115](#), 101-105 nhóm cho
hàng số chuỗi, [347](#) toán tử tăng dần
hậu tố, 248-249 ưu tiên của toán
tử, [245](#), [247](#), 256-258, [262](#) phôi
chính xác, [186](#), 188-189 ký tự dấu
phẩy động, [181](#) số dấu phẩy động, [188](#)

Giao diện dự đoán, 393-395
toán tử tăng tiền tố, 248-249 mảng
nguyên thủy, [58](#), 363-364, [374](#) phép
gán, 183-185, [190](#) ép kiểu, 185-
188, [220](#) so sánh, [238](#) khai báo,
51-53 giá trị mặc định, 195-
196 cuối cùng, [59](#) lần khởi
chạy, 199-201 nghĩa, 178-182 đi
qua, 207-208 trả về, [131](#) lớp
trình bao bọc, [192](#)

phư ơ ng thức `printStackTrace ()`,
[308](#) công cụ sửa đổi riêng, [30](#) ghi
đè, [36](#) tổng quan, 34-36 ngoại
lệ đư ợc ném theo chư ơ ng
trình, 320-321
Project Coin, cú pháp kim cư ơ ng, 388-389
tuyên truyền các ngoại lệ chư a đư ợc đề xuất,
303-306 sửa đổi đư ợc bảo vệ, [30](#), 36-42 truy cập
công khai, [19](#) giao diện công khai cho các ngoại
lệ, 312-317 sửa đổi công khai, [30](#) hằng số, 26-
27 đóng gói, [89](#) tổng quan, 33-34 public static
void `main ()` phư ơ ng thức, 12-13

Q

dấu hỏi (?) cho toán tử có điều kiện, 250-251

R

phạm vi, só, [53](#) đổi
tư ợng có thẻ truy cập,
[210](#) chỉ định lại biến tham chiếu, 213-214
phư ơ ng thức đệ quy, [320](#) phư ơ ng thức tĩnh
đư ợc xác định lại, [154](#) tham chiếu đếm, [212](#)
tham chiếu, [101](#) mảng, [368](#) gán, 190-191, 202-
203 ép kiểu, 118-121, [159](#) khai báo, 53-54 mô
tả, [183](#) đóng gói, 389-390, [401](#) bình đẳng,
238-240

phiên bản , [151](#)
cách ly, [214-215](#) mảng
đa chiều , [377-378](#) rỗng , [213](#) mảng một
chiều, [376-377](#) phư ơ ng thức nạp chồng,
[113-114](#) đi qua, [205-207](#) gán lại, [213-214](#)
trả về, [130](#) chuỗi, [342-347](#) vùng , đư ợc
bảo vệ, [299](#) toán tử quan hệ, [236-237](#)
bình đẳng, [237-241](#) điểm chính, [260](#) ư u
tiên, [257](#) toán tử còn lại, [245-246](#)
phư ơ ng thức `remove()`, [384](#) loại bỏ
phần tử `ArrayList`, [384](#) phư ơ ng thức `Replace`
(), [350](#) thay thế các phần tử chuỗi, [350](#)
rethrowing exceptions, [317](#)

báo cáo trả lại
vòng lặp `for`,
[290](#) lambdas,
[395](#) hàm tạo kiểu trả
về, [51](#), [133](#), [135](#) khai báo, [129-](#)
[132](#) điểm khóa, [159](#) phư ơ ng
thức đư ợc nạp chồng, [111](#), [129](#)
phư ơ ng thức bị đè, [108](#), [129-](#)
[130](#) giá trị trả về, [130-131](#)

tái sử dụng
ké thừa cho, [94](#) tên,
phư ơ ng thức [208-209](#)
`reverse()`, [355](#) chuỗi đảo
ngư ợc, [355](#)

trình

tạo quy tắc, tệp nguồn

135-136, [10](#)

Lớp thời gian chạy, [216](#)

ngoại lệ thời gian chạy cho các phư ơng thức bị ghi đè, [108](#)

RuntimeException lớp, [308](#), 312-314

S

hộp cát, [5](#)

phạm

vi trong vòng

lặp for, [291](#) điểm

chính, [220](#) biến,

bảo mật 192-195, [5](#) dấu

chấm phẩy (;) phư ơng

thức trừu tư ợng, [22](#), [45](#)

enum, [64](#) trong vòng lặp

for, [288](#) nhẫn, [296](#)

lambdas, [395](#) phư ơng

thức gốc, [49](#) vòng lặp

while, [287](#) tuần tự hóa,

đóng gói [5](#) bộ thiết lập,

89 biến ẩn, [56](#), [193](#), 208-209

toán tử logic ngắn mạch if,

tổng quan 276-277, 252-254 mức độ ưu

tiên, [257](#) hằng số kiểu chữ ngắn, 278-

279 giá trị mặc định, [196](#) phạm vi,

[53](#) chữ ký của các phư ơng pháp,

[117](#), [123](#) chữ ký số, [53](#) kích thư ớc

kích thước

ArrayLists, 383-384
mảng, 59, 364, 366, 370, 372, 374
sự cố gán, 235 số, 53 phuơng
thức size (), 383-384 dấu gạch
chéo (/) toán tử gán ghép, 236 phép
chia, 244 mức độ ưu tiên, 257 mã
nguồn quy tắc khai báo tệp, phần tử
mảng 10, 71 dấu ngoặc vuông ([]), 58
mảng, 363 ngoại lệ ngăn xếp, 320
điểm khóa, 220 biên cục bộ, 55 phuơng
thức, tổng quan 303-305, 176-177

Mô tả lớp

StackOverflowError,
322 phuơng thức đệ
quy, 320 trạng thái, 2
hàng số tĩnh, 26-27 nhập
tĩnh, 14-16 khối khởi tạo
tĩnh, 146 phuơng thức giao
diện tĩnh, 28-29 biên và
phuơng thức tĩnh, 14-15, 61
hàm tạo, 136 mô
tả, 193 kế thừa,
93 điểm chính,
76, 161 ghi đè,
109, 154 tổng quan,
148-150 luồng, 390
sửa đổi nghiêm ngặt

Các lớp sửa đổi
nghiêm ngặt,
19-20 phư ơng
thức, [49](#) lớp String,
340-347 hằng số,
[347](#) điểm chính, [398](#)
phư ơng thức, 348-
352 tham chiếu đôi tư ợng,
203-204 lớp StringBuffer, [340](#), 352-
353 lớp StringBuilder, [340](#) điểm
chính, [398](#) phư ơng pháp, 354-
356 tổng quan, [352](#) so với
StringBuffer, 352-353

Lớp StringIndexOutOfBoundsException, 308-309 chuỗi,
[340](#) nối thêm, 341-342, 353-354 hằng chữ hoa, 278-279
so sánh, 348-349 nối, 246-248, [261](#), 348-349 tạo,
[348](#) xóa, 354-355 đăng thức, 240-241, [281](#) tính bất
biến, 340-347 chèn phần tử vào, [355](#) điểm chính,
[398](#) độ dài, [350](#) chữ, [182](#), [347](#) chữ thư ờng, [351](#) bộ
nhớ, [347](#) phư ơng thức, 348-352 thay thế phần tử
trong, [350](#) đảo ngược, [355](#) chuỗi con, 350 -351 cắt
xén, [352](#) chữ hoa, [351](#)

nhập mạnh, 5
lớp con cụ thể,
kết thừa 46-47,
phương thức 3
chuỗi (), phép gán ghép 350-
351 phép trừ, toán tử 236 ,
244 mức độ ưu tiên, 257
kiểu con cho biến tham
chiếu, 101 lệnh gọi super
() cho hàm tạo, 144 đối số hàm tạo
siêu, 137 lớp cha, 3 hàm tạo, 135
phương thức bị ghi đè, 109-110 câu
lệnh chuyển đổi, 278 ngắt và bỏ qua,
281-283 trường hợp mặc định,
283-284 bài tập, 285 điểm chính,
325-326 biểu thức pháp lý, 278-280
đảng thức chuỗi, 281 phương
thức đồng bộ hóa, 48-49 vòng lặp
phương thức System.exit (),
290 thử và bắt, 324 phương thức
System.gc (), 215-216

T

Giao diện TemporalAmount, 358
tính toán tử bậc ba có điều kiện,
250-251 điểm chính, 261
phương thức test (), 393,
396 this () gọi hàm tạo, 135 ,
144

từ khóa này, [57](#)
luồng trong trình thu gom rác, [212](#)
mảng ba chiều, [59](#), [364](#) lớp có thẻ ném,
[307](#) mô tả ngoại lệ đư ợc ném, [299](#) JVM,
319-320 theo lập trình, dữ liệu thời
gian [320](#)-[321](#) sử dụng và thao tác,
định dạng [360](#)-[361](#), [362](#) toLowerCase
() phư ơ ng thức, [351](#) toString ()
phư ơ ng thức ArrayLists, [381](#) String,
[351](#) StringBuilder, [356](#) toUpperCase ()
phư ơ ng thức, [351](#) biến thoáng qua, [60](#)-
[61](#) phư ơ ng thức trim (), [352](#) giá trị đúng,
[181](#) cắt bớt từ các phôi, [189](#) tính năng thử
và bắt cuối cùng , Tổng quan [301](#)-[303](#),
mảng hai chiều [299](#)-[300](#) , ký hiệu bỏ
sung và ép kiểu [59](#), [364](#) hai, khai báo
mảng [189](#) kiểu, [363](#) phép gán, [235](#) ép kiểu .
Xem phôi trở lại. Xem các biến kiểu trả về,
[183](#)

U

tiền tố \ u, [181](#)
UML (Ngôn ngữ tạo mô hình hợp nhất), [100](#) toán tử
đơn phân, mức độ ưu tiên, [257](#) biến chư a đư ợc
gán

các điểm chính của
biến chư a đư ợc
gán, [221](#) làm việc với,
195-199 ngoại lệ chư a đư ợc kiểm
tra, 303-306 mô tả ngoại lệ chư a
đư ợc kiểm tra, 314-315
phư ơng thức bị ghi đè, [108](#)
dấu gạch dưới (_) trong các ký tự số, 178-179
Loại ký tự Unicode,
[53](#) mã định danh,
[6](#) chữ, [181](#) chuỗi,
[341](#)

Ngôn ngữ tạo mô hình hợp nhất (UML), [100](#) điểm
chính của biến chư a đư ợc khởi tạo, [221](#) hoạt
động với, 195-199 câu lệnh không đư ợc
gắn nhãn, [295](#) phư ơng thức cho đến (),
[361](#) giải nén ngăn xếp, [308](#) dự báo , [120](#) chữ
hoa cho chuỗi, [351](#)

V

phư ơng thức giá trị
(), [66](#) giá trị của biến,
[182](#) điểm chính var-args,
[75](#) phư ơng thức, 49-
50 danh sách đối số
biến, 49-50 biến

truy cập. Xem các bài tập bổ trợ truy cập và truy cập. Xem khai báo bài tập,
[51-53](#), [75-76](#) mô tả, [2](#) enums, [65-66](#)

cuối
cùng, [59](#) in cho
vòng lặp, [291](#) đồng và
ngăn xếp, 176-177 khởi
tạo, [185](#) phiên bản , 54-
55 cục bộ. Xem tên biến cục bộ, [9](#)
nguyên thủy, phạm vi 51-53, 192-195
bóng, 208-209 tĩnh, [61](#), 148-150 tạm
thời, 60-61 chư a khởi tạo và chư a
đư ợc gán, 195-199, [221](#) giá trị, [182](#)
dễ bay hơi i, [61](#) thanh dọc (|) toán
tử bitwise, toán tử OR lôgic 251-252, ưu tiên
253-255, [257](#) khả năng hiển thị, truy cập, [18](#),
[43](#) kiểu trả về void, [131](#) biến dễ bay hơi i, [61](#)

W

Walraven, Fritz, [258](#)
vòng lặp trong khi
đư ợc gắn nhãn,
[297](#) làm việc với, 285-
286 khoảng trắng, cắt từ chuỗi, [352](#)
chuyển đổi mở rộng, [186](#) ký tự đại diện
trong câu lệnh nhập, [14](#), [16](#) lớp bao bọc
nguyên thủy, [192](#) chuỗi, [320](#) trình bao
bọc, ArrayLists, 384-388

X

Toán tử XOR (độc quyền HỌC), [255](#)

Z

chỉ mục dựa trên 0,

tiền tố [12 0x](#), [179](#)

Machine Translated by Google

Join the Largest Tech Community in the World



Download the latest software, tools, and developer templates



Get exclusive access to hands-on trainings and workshops



Grow your professional network through the Oracle ACE Program



Publish your technical articles – and get paid to share your expertise

Join the Oracle Technology Network
Membership is free. Visit community.oracle.com

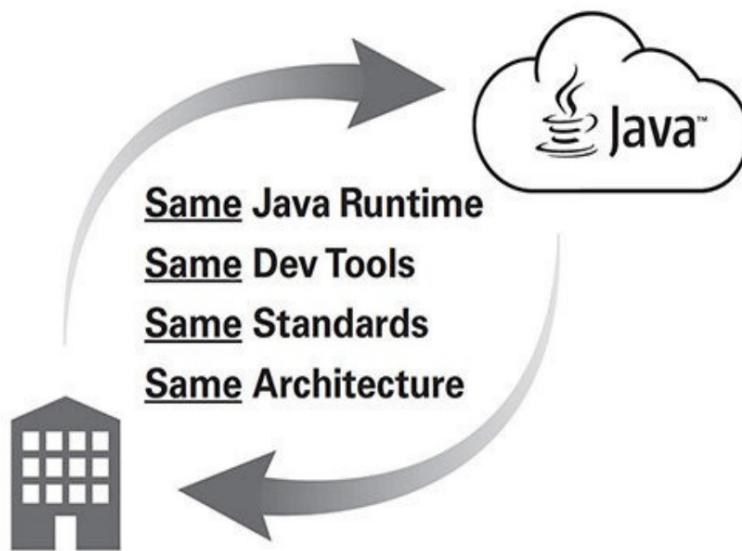
@OracleOTN

facebook.com/OracleTechnologyNetwork

ORACLE®

Machine Translated by Google

Push a Button Move Your Java Apps to the Oracle Cloud



... or Back to Your Data Center

ORACLE®

cloud.oracle.com/java

Machine Translated by Google



Reach More than 640,000 Oracle Customers with Oracle Publishing Group

Connect with the Audience that Matters Most to Your Business

ORACLE
MAGAZINE

Oracle Magazine

The Largest IT Publication in the World

Circulation: 325,000

Audience: IT Managers, DBAs, Programmers, and Developers

PROFIT ORACLE

Profit

Business Insight for Enterprise-Class Business Leaders to Help Them Build
a Better Business Using Oracle Technology

Circulation: 90,000

Audience: Top Executives and Line of Business Managers



Java Magazine

The Essential Source on Java Technology, the Java Programming Language,
and Java-Based Applications

Circulation: 225,00 and Growing Steady

Audience: Corporate and Independent Java Developers, Programmers,
and Architects



For more information
or to sign up for a FREE
subscription: Scan the
QR code to visit Oracle
Publishing online.

Copyright © 2016, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

ORACLE®

Machine Translated by Google

Beta Test Oracle Software

Get a first look at our newest products—and help perfect them. You must meet the following criteria:

- ✓ Licensed Oracle customer or Oracle PartnerNetwork member
- ✓ Oracle software expert
- ✓ Early adopter of Oracle products

Please apply at: pdpm.oracle.com/BPO/userprofile

ORACLE®

If your interests match upcoming activities, we'll contact you. Profiles are kept on file for 12 months.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Machine Translated by Google

Climb the Career Ladder

Think about it—97 percent of the Fortune 500 companies run Oracle solutions. Why wouldn't you choose Oracle certification to secure your future? With certification through Oracle, your resume gets noticed, your chances of landing your dream job improve, you become more marketable, and you earn more money. It's simple. Oracle certification helps you get hired and get paid for your skills.



93 %

Hiring managers who say IT certifications are beneficial and provide value to the company¹

7 %

Salary growth for Oracle Certified professionals⁵

70 %

Believe that Oracle certification improved their earning power²

90 %

Say that Oracle certification gives them credibility when looking for a new job²

68 %

Think that certification has made them more in demand³

6 x

Increased LinkedIn profile views for people with certifications, boosting their visibility and career opportunities⁴

Take the next step

<http://education.oracle.com/certification/press>



ORACLE®

[1] "Value of IT Certifications," CompTIA, October 14, 2014, [2] Oracle Certification Survey, [3] "Certification: It's a Journey Not a Destination," Certification Magazine 2015 Salary Edition, [4] "The Future Value of Certifications: Insights from LinkedIn's Data Trove," ATP 2015 Innovations in Testing, [5] Certification Magazine 2015 Annual Salary Survey Copyright © 2015, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Machine Translated by Google

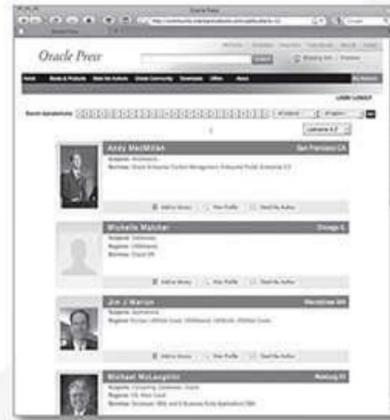
Join the Oracle Press Community at OraclePressBooks.com



Find the latest information on Oracle products and technologies. Get exclusive discounts on Oracle Press books. Interact with expert Oracle Press authors and other Oracle Press Community members. Read blog posts, download content and multimedia, and so much more. Join today!

Join the Oracle Press Community today and get these benefits:

- Exclusive members-only discounts and offers
- Full access to all the features on the site: sample chapters, free code and downloads, author blogs, podcasts, videos, and more
- Interact with authors and Oracle enthusiasts
- Follow your favorite authors and topics and receive updates
- Newsletter packed with exclusive offers and discounts, sneak previews, and author podcasts and interviews



**Oracle
Press™**

@OraclePress