Think Java: Cách suy nghĩ như nhà khoa học máy tính

Phiên bản 5.1.2

Xem thêm ở Facebook.

Think Java là cuốn sách giới thiệu về lập trình Java cho người mới học. Nó được soạn riêng cho học viên chuẩn bị thi Computer Science Advanced Placement (AP) Exam, nhưng cũng dành cho bất kì ai muốn học Java.

- *Think Java* rất ngắn gọn. Sách chỉ dùng một bộ phận nhỏ của ngôn ngữ Java đủ để giúp học viên thực hiện những bài tập lớn mà không bị sa đà vào những tiểu tiết của ngôn ngữ lập trình.
- *Think Java* dạy cách phát triển và gỡ lỗi chương trình; những chủ đề này được thảo luận trong suốt cuốn sách và được tóm tắt trong hai phụ lục.
- *Think Java* bao gồm cả nghiên cứu cụ thể GridWorld vốn là một phần của đề thi AP. Cuốn sách này cung cấp kiến thức cơ sở cần để làm quen với GridWorld, cùng với những bài tập phụ thêm để thực hành.
- *Think Java* được viết theo cuốn sách gốc How to Think Like a Computer Scientist, một cuốn sách trực tuyến quen thuộc với những phiên bản cho lập trình Python, C++ và OCaml, cùng với những bản dịch sang tiếng Tây Ban Nha, tiếng Pháp và những thứ tiếng khác.

Mục lục

Chương 0. Lời nói đầu

> ""“Khi được hưởng những Thành quả lớn từ Phát minh của người khác, ta nên vui vẻ đó nhận Cơ hội để giúp đỡ người khác bằng Phát minh của ta, và việc này nên làm một cách tự ý và hào phóng.”
> —Benjamin Franklin, trích từ cuốn *Benjamin Franklin* của tác giả Edmund S. Morgan.

Lý do mà tôi viết cuốn sách này

Đây là ấn bản thứ năm của cuốn sách mà tôi đã bắt đầu viết từ năm 1999, khi còn dạy ở trường Colby College. Tôi dã dạy một lớp học nhập môn khoa học máy tính bằng ngôn ngữ Java, nhưng chưa tìm được một cuốn giáo trình vừa ý. Một trong những lý do là, chúng quá dày! Không có cách nào mà sinh viên có thể đọc hết cuốn sách dày cỡ 800 trang, đầy những kiến thức kĩ thuật, kể cả tôi có yêu cầu họ thực hiện. Mà tôi chẳng muốn thế. Phần lớn những gì viết trong sách đều quá cụ thể—những chi tiết về Java cùng các thư viện của nó mà sẽ lạc hậu ngay khi học kì kết thúc, đồng thời những thứ đó sẽ làm lu mờ những kiến thức mà tôi thực sự muốn sinh viên học.

Một vấn đề khác mà tôi thấy, đó là phần giới thiệu lập trình hướng đối tượng là quá đột ngột. Nhiều sinh viên đáng ra đã học suôn sẻ rồi nhưng khi bắt đầu vào phần “đối tượng” là bị vấp, bất kể phần này được đưa vào đâu trong giáo trình.

Vì vậy tôi bắt tay vào viết. Mỗi ngày tôi viết một chương, liên tục trong 13 ngày đầu. Rồi ngày thứ 14 tôi biên tập lại. Sau đó tôi đi phô-tô và đóng bìa. Buổi học đầu tiên khi phân phát giáo trình, tôi dặn sinh viên mỗi tuần phải đọc một chương. Nói cách khác, họ cần đọc chậm hơn 7 lần so với tốc độ viết của tôi.

Triết lý ẩn sau cuốn sách

Sau đây là một số ý tưởng định hình cho cuốn sách:

- Thuật ngữ là quan trọng. Sinh viên cần phải trao đổi được về chương trình máy tính và hiểu được điều tôi giảng. Tôi cố gắng giới thiệu một số tối thiểu các thuật ngữ, để định nghĩa được rõ ràng khi dùng lần đầu, và để tổ chức lại thành mục “Thuật ngữ” cuối từng chương. Trên lớp, tôi có đưa những câu hỏi liên quan đến thuật ngữ vào đề kiểm tra, đề thi, và yêu cầu sinh viên phải dùng thuật ngữ thích hợp để viết vào đáp án.
- Để viết một chương trình, sinh viên cần phải hiểu được thuật toán, biết ngôn ngữ lập trình, và có khả năng gỡ lỗi. Tôi nghĩ rằng quá nhiều quyển sách bỏ qua khâu gỡ lỗi. Cuốn sách này có một phụ lục viết về gỡ lỗi và một phụ lục về phát triển chương trình (giúp tránh được gỡ lỗi). Tôi khuyến khích sinh viên sớm đọc ngay những phần này và thường xuyên tham khảo đến chúng.
- Một số khái niệm phải mất thời gian mới lắng đọng lại được. Một số chỗ khó trong sách, như đệ quy, sẽ xuất hiện vài lần. Bằng cách nêu lại những điểm này, tôi cố gắng tạo cho sinh viên cơ hội để ôn lại và củng cố, hoặc nếu lần đầu họ không nắm được, thì đó là cơ hội để theo kịp.
- Tôi cố gắng dùng càng ít Java càng tốt để đạt được công hiệu lập trình tối đa. Mục đích của cuốn sách này là dạy lập trình và một số ý tưởng cơ bản về khoa học máy tính, chứ không phải dạy Java. Tôi bỏ qua một số đặc điểm của ngôn ngữ này, như lệnh switch, vốn không cần thiết, và tránh hầu hết các thư

viện chương trình, đặc biệt những thư viện như AWT vốn đã thay đổi quá nhanh hoặc có xu hướng lỗi thời, phải thay thế.
Phương pháp tiếp cận theo xu hướng "tối thiểu" như vậy có một số ưu điểm. Từng chương chỉ dài khoảng 10 trang, không kể bài tập. Trên lớp, tôi yêu cầu sinh viên đọc mỗi chương trước khi thảo luận, và thấy được rằng họ sẵn sàng thực hiện và nắm bắt được lượng kiến thức. Sự chuẩn bị trước của sinh viên đã giúp dành khoảng thời gian trên lớp để thảo luận những nội dung trừu tượng hơn, để làm bài tập trên lớp, và những chủ đề thêm không có trong sách.

Nhưng xu hướng "tối thiểu" cũng có những nhược điểm. Không có nhiều chỗ thú vị về bản chất. Đa số các ví dụ trong sách nhằm minh họa cho cách sử dụng cơ bản nhất của ngôn ngữ, và nhiều bài tập có liên quan đến thao tác chuỗi kí tự và khái niệm toán học. Tôi nghĩ một số bài thì thú vị, song những thứ làm sinh viên thích ngành khoa học máy tính, như đồ họa, âm thanh và ứng dụng mạng, lại chỉ được giới thiệu qua loa.

Vấn đề nằm ở chỗ phần lớn các đặc điểm thú vị như vậy thì liên quan tới chi tiết vặt mà ít liên quan đến khái niệm. Xét trên khía cạnh giáo dục, điều này có nghĩa là nhiều công sức bỏ ra để thu được ít. Như vậy có một sự tráo đổi giữa nội dung mà sinh viên ưa thích và nội dung mang đậm tri thức. Việc giữ cân bằng hợp lý, tôi nhường lại cho giáo viên đứng lớp. Để giúp phần nào, cuốn sách này có phụ lục đề cập đến đồ họa, nhập liệu từ bàn phím và từ tập tin.

Lập trình hướng đối tượng

Một số quyển sách giới thiệu ngay khái niệm đối tượng; lại có quyển dạo đầu bằng phong cách lập trình thủ tục và dần dần xây dựng phong cách hướng đối tượng. Cuốn sách này thì theo lối "giới thiệu đối tượng sau".

Nhiều đặc điểm hướng đối tượng của Java khởi nguồn từ các vấn đề đặt ra cho ngôn ngữ đi trước, và cách thực hiện những đặc điểm này chịu ảnh hưởng bởi quá trình lịch sử. Một số đặc điểm rất khó giải thích nếu người học không thạo những bài toán cần giải.

Việc hoãn lại kĩ thuật lập trình hướng đối tượng không phải là chủ ý của tôi. Trái lại, tôi cố gắng tới đó càng nhanh càng tốt, song bị hạn chế bởi ý muốn giới thiệu lần lượt từng khái niệm một, thật rõ ràng, theo cách mà sinh viên có thể thực hành riêng từng khái niệm trước khi chuyển tiếp. Nhưng cũng phải thừa nhận rằng phải mất một thời gian học mới đến được phần hướng đối tượng.

Kì thi Computer Science AP

Theo lẽ thường, khi được biết rằng Hội đồng tuyển sinh (College Board) công bố rằng nội dung thi AP sẽ chuyển sang dùng Java, tôi đã có kế hoạch cập nhật phiên bản Java của cuốn sách này. Đối chiếu với đề cương AP được đưa ra, tôi thấy rằng bộ phận nhỏ của Java dùng để thi rất giống với bộ phận mà tôi đã chọn.

Trong tháng 1 năm 2003, tôi đã soạn ấn bản thứ 4 của cuốn sách, với những sửa đổi sau:

- Tôi đã thêm vào các mục nhằm bao quát được nội dung trong đề cương thi AP.
- Tôi hoàn thiện các phụ lục về gỡ lỗi và phát triển chương trình.

- Tôi đi tập hợp lại những bài tập, câu đố, và câu hỏi thi đã ra trên lớp rồi đưa vào cuối các chương, ngoài ra còn soạn thêm một số câu hỏi giúp chuẩn bị kì thi AP.
  Cuối cùng, vào tháng 8-2011, tôi viết xong ấn bản thứ 5, bao quát được phần nghiên cứu cụ thể GridWorld là nội dung trong kì thi AP.

Sách phát hành tự do

Ngay từ đầu, cuốn sách này đã theo giấy phép mà bạn đọc được quyền sao chép, phân phối và sửa chữa nội dung. Độc giả có thể tải sách về với nhiều định dạng khác nhau và có thể đọc trên màn hình hoặc in ra giấy. Giáo viên có thể in bao nhiêu bản tùy ý. Và mọi người đều có thể sửa đổi sách theo nhu cầu.

Đã có người chuyển nội dung cuốn sách sang cho những ngôn ngữ lập trình khác (như Python và Eiffel), và những thứ tiếng khác (như Tây Ban Nha, Pháp, và Đức). Trong số đó, nhiều phiên bản được đăng theo hình thức tự do.

Với động lực từ Phần mềm nguồn mở, tôi đã đón nhận triết lý phát hành sách thật sớm và cập nhật thường xuyên. Tôi đã cố gắng hết sức để giảm thiểu các lỗi, những cũng nhờ bạn đọc giúp sức.

Tình hình phản hồi thật tuyệt. Gần như ngày nào tôi cũng nhận được thông tin từ bạn đọc, với sự ưa thích cuốn sách đến nỗi họ gửi hẳn một "danh sách liệt kê lỗi". Thông thường tôi chữa một lỗi mất vài phút và sau đó cập nhật ngay bản thảo qua sửa đổi. Tôi coi cuốn sách như một tác phẩm đang trong quá trình hoàn thiện, sẽ được cải tiến ít một mỗi khi tôi có thời gian soạn lại, hoặc khi bạn đọc gửi phản hồi.

À, còn về tiêu đề

Tôi đã thật buồn phiền về tiêu đề cuốn sách Không phải ai cũng hiểu được rằng chủ yếu đó chỉ là cách nói đùa. Có thể sau khi đọc cuốn sách này, bạn chưa tư duy được như nhà khoa học máy tính. Điều đó cần thời gian, kinh nghiệm, và có thể phải qua mấy lớp học nữa.

Nhưng có một điểm cốt lõi có thật ở tiêu đề này: cuốn sách này không phải viết về Java, và nó chỉ một phần là về lập trình. Nếu có chăng, sự thành công ở cuốn sách là nằm chỗ một cách nghĩ mới. Nhà khoa học máy tính luôn có một cách tiếp cận để giải quyết vấn đề, và một cách định hình lời giản, rất độc đáo, linh hoạt và mạnh mẽ. Tôi hi vọng rằng cuốn sách này giúp bạn hình dung được phương pháp đó là gì, và ở những lúc nào đó bạn sẽ tự thấy mình có tư duy như nhà khoa học máy tính.

Allen B. Downey
Needham Massachusett, Hoa Kì
13-7-2011

Danh sách bạn đọc đã đóng góp nội dung

Khi bắt đầu viết sách thể loại tự do, tôi vẫn chưa có ý định lập danh sách đóng góp từ phía bạn đọc. Rồi Jeff Elkner đề xuất, và rõ ràng tôi đã tỏ ra lúng túng vì thiếu sót này. Danh sách dưới đây tính từ ấn bản thứ 4, vì vậy nó không có tên nhiều người đã đóng góp, sửa đổi từ trước đó.

Nếu bạn có bất kì nhận xét nào thêm, hãy gửi thư về đại chỉ feedback@greenteapress.com

- Ellen Hildreth đã dùng sách này để dạy môn học Cấu trúc dữ liệu ở trường Wellesley College, và cô đã

gửi một loạt những chỗ cần đính chính, kèm theo một số đề xuất hay.

- Tania Passfield chỉ ra rằng phần Thuật ngữ cuối Chương 4 đã ghi thừa một số mục không có trong sách.
- Elizabeth Wiethoff nhận thấy cách tôi khai triển $\exp(-x_2)$ là sai. Cô cũng đã soạn ra một phiên bản sách dùng ngôn ngữ lập trình Ruby!
- Matt Crawford đã gửi một file "bản vá" đầy những chỗ cần sửa!
- Chi-Yu Li chỉ ra một lỗi typo và một lỗi trong mã lệnh ví dụ.
- Doan Thanh Nam chữa lại một ví dụ ở Chương 3.
- Stijn Debrouwere phát hiện một typo trong biểu thức toán.
- Muhammad Saied dịch cuốn sách sang tiếng A-rập, và phát hiện một số lỗi.
- Marius Margowski phát hiện một điểm không nhất quán trong mã lệnh ví dụ.
- Guy Driesen phát hiện một số lỗi typo.
- Leslie Klein phát hiện một chỗ sai khác trong cách khai triển $\exp(-x_2)$, phát hiện các typo trong hình vẽ biểu diễn mảng chứa các quân bài, và có đề xuất hay giúp cho bài tập được rõ ràng hơn.

Sau cùng, tôi xin được cám ơn Chris Mayfield đã đóng góp đáng kể cho phiên bản 5.1 của sách. Qua việc phản biện cẩn thận, ông đã chỉ ra hơn một trăm chỗ cần sửa và bổ sung. Một số đặc điểm mới gồm có liên kết đến các trang web và liên kết chéo giữa các mục trong sách, sự trình bày nhất quán về hình thức cho các bài tập, và tô màu mã lệnh Java [chỉ có ở sách gốc].

Chương 1: Lối đi của chương trình máy tính

Trở về Mục lục cuốn sách

Mục đích của cuốn sách này là hướng dẫn bạn suy nghĩ như là một nhà khoa học máy tính. Tôi thích lối suy nghĩ của những nhà khoa học máy tính vì ở đó có sự kết hợp những đặc điểm hay nhất của toán học, kĩ thuật, và khoa học tự nhiên. Cũng như những nhà toán học, những nhà khoa học máy tính dùng những ngôn ngữ có quy cách để diễn đạt ý tưởng (đặc biệt là tính toán). Giống như những kĩ sư, họ cũng làm công việc thiết kế, gắn kết các thành phần tạo nên một hệ thống và đánh giá những ưu khuyết giữa các phương án khác nhau. Giống như những nhà khoa học, họ khảo sát các động thái của hệ thống phức tạp, đề ra các giả thiết, và kiểm định những tính toán.

Kĩ năng quan trọng nhất của nhà khoa học máy tính là **giải quyết vấn đề**. Giải quyết vấn đề chính là cách tạo lập vấn đề, suy nghĩ giải pháp một cách sáng tạo, và trình bày giải pháp một cách rõ ràng và chính xác. Như bạn sẽ thấy, việc học lập trình chính là một cơ hội tuyệt vời để bạn luyện tập những kĩ năng giải quyết vấn đề. Đó là lí do tại sao chương này lại có tên là "Lối đi của chương trình máy tính". Một mặt, bạn sẽ được học cách lập trình, vốn bản thân nó là một kĩ năng hữu dụng. Mặt khác, bạn sẽ dùng lập trình như một phương tiện để giải quyết vấn đề. Điều này bạn sẽ dần dần làm được trong quá trình học.

1.1 Ngôn ngữ lập trình là gì?

Ngôn ngữ lập trình mà bạn sẽ học là Java, vốn là một ngôn ngữ tương đối mới (phiên bản đầu tiên do Sun phát hành vào tháng 5-1995). Java là một ví dụ trong số các **ngôn ngữ lập trình bậc cao**; một số ngôn ngữ lập trình bậc cao khác mà bạn có thể biết đến gồm có Python, C, C++, và Perl.

Nhắc đến "ngôn ngữ lập trình bậc cao", có lẽ bạn cũng suy đoán được rằng còn những **ngôn ngữ lập trình bậc thấp**, đôi khi mà ta gọi là "ngôn ngữ máy" hoặc "hợp ngữ". Nói nôm na, máy tính chỉ có thể thực hiện các chương trình được viết bằng ngôn ngữ bậc thấp. Vì vậy những chương trình được viết bằng một ngôn ngữ bậc cao cần được xử lý trước khi chúng có thể chạy được. Bước phụ trợ này sẽ tốn thêm thời gian, đây là một nhược điểm nhỏ của các ngôn ngữ bậc cao.

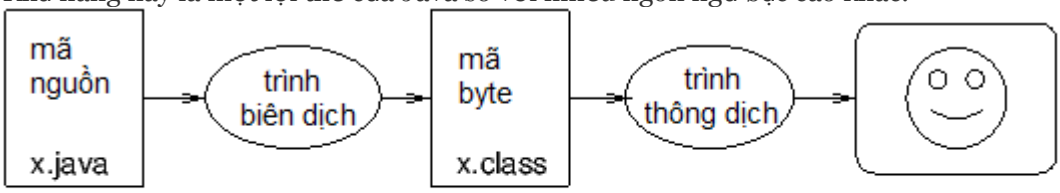
Tuy vậy, các ưu điểm là rất lớn. Thứ nhất, việc lập trình bằng ngôn ngữ bậc cao dễ hơn *nhiều*. Chương trình được viết bằng ngôn ngữ bậc cao được viết nhanh hơn, nội dung chương trình ngắn hơn, dễ đọc hơn, và nhiều khả năng là chúng chính xác. Thứ hai, các ngôn ngữ bậc cao có tính **khả chuyển** theo nghĩa chạy được trên nhiều hệ máy tính khác nhau mà ít hoặc không cần phải sửa đổi. Các chương trình bậc thấp chỉ có thể chạy trên một loại máy tính và phải được viết lại nếu muốn chạy trên các hệ máy khác.

Bởi các ưu điểm nêu trên, hầu hết các chương trình đều được lập trình bằng ngôn ngữ bậc cao. Các ngôn ngữ bậc thấp chỉ được dùng cho một số ít những ứng dụng đặc biệt.

Hai loại chương trình có nhiệm vụ chuyển đổi các ngôn ngữ bậc cao về dạng ngôn ngữ bậc thấp: **trình thông dịch** và **trình biên dịch**. Trình thông dịch là một chương trình máy tính, có nhiệm vụ đọc một chương trình bậc cao và thực hiện nó theo đúng những gì mà chương trình chỉ định. Nó xử lý chương trình một cách dần dần, nghĩa là đọc câu lệnh đến đâu thì thực hiện tính toán tới đó.

Còn trình biên dịch thì có nhiệm vụ đọc chương trình và dịch nó hoàn toàn trước khi thực hiện bất kì

một câu lệnh nào trong chương trình. Thường thì bạn thực hiện bước biên dịch chương trình trước, sau đó mới chạy mã lệnh đã biên dịch. Khi đó, chương trình bậc cao được gọi là **mã nguồn**, và chương trình sau khi được dịch gọi là **mã đối tượng**, hoặc **chương trình chạy**.
Chương trình Java vừa được biên dịch lẫn thông dịch. Thay vì việc chuyển chương trình sang ngôn ngữ máy, trình biên dịch Java phát sinh ra **mã byte**. Mã byte dễ thông dịch (và thông dịch cũng nhanh), giống như mã máy; song nó còn khả chuyển, như một ngôn ngữ bậc cao. Vì vậy, ta có thể biên dịch một chương trình trên máy này, đưa mã byte sang máy khác, sau đó thông dịch mã byte này trên máy mới. Khả năng này là một lợi thế của Java so với nhiều ngôn ngữ bậc cao khác.



Trình biên dịch đọc mã lệnh... ... rồi phát sinh byte code Java. Một trình thông dịch Java đọc byte code này... ... và kết quả sẽ xuất hiện trên màn hình.

Mặc dù quá trình này có vẻ phức tạp, nhưng đa số các môi trường phát triển chương trình đều giúp bạn tự động thực hiện các bước kể trên. Thông thường bạn sẽ chỉ phải viết một chương trình rồi ấn một nút hoặc gõ vào một câu lệnh để biên dịch và chạy. Mặt khác, ta vẫn cần biết những bước nào đang được máy thực hiện ngầm, để nhỡ có trục trặc thì có thể hình dung ra sai ở khâu nào.

1.2 Chương trình là gì?

Chương trình là một danh sách các chỉ dẫn cách thực hiện tính toán.[1] Việc tính toán có thể là phép thao tác toán học, chẳng hạn giải hệ phương trình hoặc tìm nghiệm đa thức, nhưng cũng có thể là những phép tính trên các kí hiệu, chẳng hạn tìm kiếm và thay thế chữ trong một văn bản, hoặc (kì lạ hơn) là biên dịch một chương trình.
Những chỉ dẫn, mà ta gọi là những **câu lệnh**, sẽ khác nhau tùy loại ngôn ngữ lập trình, nhưng chung quy lại có một số ít các phép thao tác mà nhiều ngôn ngữ thực hiện:

nhập số liệu:
Là việc lấy số liệu từ bàn phím, file, hoặc một thiết bị khác.
xuất kết quả:
Hiển thị kết quả trên màn hình hoặc gửi kết quả ra file hoặc một thiết bị khác.
tính toán:
Thực hiện các phép toán cơ bản như cộng và nhân.
kiểm tra:
Kiểm tra một điều kiện cụ thể và thực hiện danh sách câu lệnh tương ứng với điều kiện đó.
tính lặp:
Thực hiện lặp lại công việc nhiều lần, thường là với một số thay đổi giữa các lần lặp.
Như vậy đã tương đối đầy đủ. Mỗi chương trình mà bạn đã từng dùng qua, bất kể nó phức tạp đến đâu, đều được hợp thành từ những câu lệnh thực hiện tính toán. Vì vậy, một cách mô tả lập trình, đó là quá trình chia một bài toán lớn, phức tạp thành nhiều bài toán nhỏ hơn cho đến khi từng bài toán nhỏ này đơn giản đến mức có thể được thực hiện theo một trong các chỉ dẫn trên đây.

1.3 Gỡ lỗi là gì?

Việc lập trình rất hay mắc phải lỗi. Việc theo dõi, phân tích nguyên nhân gây ra lỗi được gọi là **gỡ lỗi**. Có ba loại lỗi có thể xuất hiện trong chương trình: lỗi cú pháp, lỗi chạy và lỗi ngữ nghĩa. Để nhanh chóng tìm ra lỗi ta cần phân biệt được chúng.

1.3.1 LỖI CÚ PHÁP

Trình biên dịch chỉ có thể chuyển đổi được chương trình nếu như nó đúng đắn về cú pháp; còn nếu không, việc biên dịch sẽ thất bại và bạn sẽ không chạy được chương trình. **Cú pháp** nghĩa là cấu trúc của chương trình và các quy tắc về cấu trúc đó.
Chẳng hạn, trong tiếng Anh, một câu viết phải bắt đầu bằng chữ in hoa và kết thúc bằng dấu chấm. câu này có một lỗi cú pháp. Và câu này cũng vậy

Đa phần ban đọc thường không để tâm đến một số ít lỗi cú pháp, vì vậy ta có thể đọc thơ của tác giả e e cummings mà không thốt ra lời thông báo lỗi nào.

Các trình biên dịch thì không như vậy. Chỉ cần trong chương trình có lỗi cú pháp ở bất cứ đâu, trình biên dịch sẽ hiển thị thông báo lỗi và kết thúc, và bạn sẽ không thể chạy chương trình.

Tệ hơn nữa là trong Java có nhiều quy tắc cú pháp hơn là trong tiếng Anh, và thường thì những thông báo lỗi mà bạn nhận được từ trình biên dịch đều không giúp ích gì nhiều. Nếu bạn mới nhập môn lập trình được vài tuần, rất có thể bạn phải dành nhiều thời gian dò tìm lỗi. Khi kinh nghiệm tăng dần lên, bạn sẽ tránh được lỗi tốt hơn và nếu mắc thì cũng phát hiện ra lỗi nhanh hơn.

1.3.2 LỖI THỰC THI

Loại lỗi thứ hai là lỗi thực thi; chúng có tên như vậy bởi vì chỉ xuất hiện khi chương trình đã bắt đầu chạy. Trong Java, lỗi thực thi xảy ra khi trình thông dịch đang chạy mã byte và có điều gì đó trục trặc.

Java có xu hướng là ngôn ngữ **an toàn**, theo nghĩa trình biên dịch sẽ bắt rất nhiều lỗi. Do vậy lỗi thực thi sẽ hiếm, đặc biệt là ở những chương trình đơn giản.
Trong Java, lỗi thực thi được gọi là **biệt lệ**, và ở hầu hết các môi trường lập trình, chúng xuất hiện dưới hình thức của sổ hoặc hộp thoại ghi rõ những thông tin về tình trạng đã diễn ra và lúc đó thì chương trình đang thực hiện những gì. Thông tin này rất có ích đối với việc gỡ lỗi.

1.3.3 LỖI LOGIC VÀ NGỮ NGHĨA

Loại lỗi thứ ba là **lỗi logic** hay **lỗi ngữ nghĩa**. Trong trường hợp có lỗi kiểu này, chương trình sẽ vẫn được biên dịch và chạy mà không phát ra thông báo lỗi nào, nhưng sẽ không thực hiện đúng yêu cầu mong muốn, mà sẽ cho kết quả khác. Cụ thể là thực hiện theo đúng những câu lệnh mà bạn đã chỉ dẫn. Vấn đề ở đây là chương trình bạn viết sẽ không đúng theo ý muốn của bạn. Ý nghĩa của chương trình bị sai lệch. Việc phát hiện các lỗi ngữ nghĩa đôi lúc rất khó vì bạn cần phải quay ngược lại và nhìn vào kết quả của chương trình để phán đoán xem bản thân chương trình đã thực hiện những gì.

1.3.4 GỠ LỖI THỬ NGHIỆM

Một trong những kĩ năng quan trọng nhất mà bạn sẽ học được, đó là gỡ lỗi. Mặc dù đôi khi bị vấp váp, nhưng việc gỡ lỗi rất thú vị, chứa đầy thử thách và là một phần có giá trị trong lập trình.

Gỡ lỗi giống như việc điều tra tội phạm. Bạn có trong tay các manh mối, phải suy luận ra các quá trình và sự kiện dẫn đến những hậu quả đang chứng kiến.

Việc gỡ lỗi cũng giống như khoa học thực nghiệm. Mỗi khi có ý kiến về nguyên nhân dẫn đến lỗi sai, bạn sửa chữa chương trình và thực hiện lại. Nếu giả thiết của bạn là đúng thì bạn thu được kết quả của công việc sửa chữa, đồng thời tiến một bước gần hơn tới chương trình đúng. Còn nếu giả thiết là sai thì bạn cần đề ra một giả thiết mới. Sherlock Holmes đã chỉ ra, "Khi bạn đã loại trừ tất cả những điều không thể thì những gì còn lại, dù có mập mờ đến đâu, chính là sự thật". (A. Conan Doyle, *Dấu của bộ tứ*)

Đối với một số người, việc lập trình và gỡ lỗi là giống nhau. Đó là vì lập trình chính là quá trình gỡ lỗi dần dần đến khi bạn có được chương trình mong muốn. Ý tưởng ở đây là bạn nên bắt đầu với một chương trình thực hiện được *một điều gì đó*, rồi thực hiện các chỉnh sửa nhỏ, gỡ lỗi trong quá trình phát triển, đến khi bạn có được một chương trình hoàn thiện.

Chẳng hạn, Linux là một hệ điều hành bao gồm hàng nghìn dòng lệnh, nhưng nó chỉ bắt đầu từ một chương trình đơn giản do Linus Torvalds dùng để khám phá chip Intel 80386. Theo Larry Greenfield thì "Một trong những dự án trước đó của Linus là một chương trình có nhiệm vụ chuyển từ việc in AAAA thành BBBB. Sau đó nó dần trở thành Linux". (*The Linux Users' Guide* Beta Version 1 / *Hướng dẫn sử dụng Linux*, phiên bản Beta 1).

Các chương tiếp sau đây sẽ nói thêm về việc gỡ lỗi và các vấn đề thực tế trong lập trình.

1.4 Ngôn ngữ hình thức và ngôn ngữ tự nhiên

Ngôn ngữ tự nhiên được mọi người dùng để giao tiếp, ví dụ Tiếng Anh, Tiếng Tây Ban Nha, Tiếng Pháp. Chúng tự do phát triển mà không định theo khuôn mẫu với bất kì mục đích nào (mặc dù có một số trật tự chẳng hạn như ngữ pháp);

Ngôn ngữ hình thức được con người thiết kế để ứng dụng trong những lĩnh vực riêng. Chẳng hạn, kí hiệu toán học chính là một ngôn ngữ hình thức rất hữu dụng để biểu diễn mối quan hệ giữa những biến lượng và con số. Trong hoá học, một loại ngôn ngữ hình thức khác được dùng để biểu diễn cấu trúc hoá học của các phân tử. Và quan trọng nhất:

> **Ngôn ngữ lập trình là những ngôn ngữ hình thức được thiết kế phục vụ mục đích diễn tả quá trình tính toán.**

Các ngôn ngữ hình thức thường có quy định rất chặt chẽ về cú pháp. Chẳng hạn, 3 + 3 = 6 là một biểu thức toán học đúng, nhưng 3 $ = thì không. H_2O là một công thức hoá học đúng về cú pháp, còn $_2Zz$ thì không.

Các quy tắc cú pháp có hai dạng, thuộc về các nguyên tố và cấu trúc. Nguyên tố là các thành phần cơ sở của ngôn ngữ, chẳng hạn, các từ, các con số, và các nguyên tố hoá học. Trong ví dụ nêu trên, 3 $ = có lỗi sai vì `$` không phải là một nguyên tố hợp lệ trong toán học (theo như tôi được biết). Tương tự như vậy, $_2Zz$ không hợp lệ vì không có nguyên tố hoá học nào có kí hiệu là *Zz*.

Loại lỗi cú pháp thứ hai thuộc về dạng cấu trúc của một mệnh đề; nghĩa là cách sắp xếp các nguyên tố. Mệnh đề 3 $ = không hợp lệ về cấu trúc là vì bạn không thể để dấu bằng ở cuối phương trình được. Tương tự như vậy, trong một công thức hoá học thì chỉ số phải được đặt sau tên nguyên tố chứ không phải đặt trước.

Mỗi khi đọc một câu trong ngôn ngữ tự nhiên, hoặc trong ngôn ngữ hình thức, bạn cần hình dung được cấu trúc của câu đó là gì (mặc dù với ngôn ngữ tự nhiên thì việc làm này được thực hiện một cách vô thức). Quá trình này được gọi là **phân tách**.

Mặc dù ngôn ngữ hình thức và ngôn ngữ tự nhiên có nhiều đặc điểm chung—nguyên tố, cấu trúc, cú pháp, và ngữ nghĩa—nhưng chúng có một số khác biệt:

về sự mập mờ:
Ngôn ngữ tự nhiên chứa đựng sự mập mờ theo nghĩa con người muốn hiểu đúng phải có suy luận tuỳ từng ngữ cảnh. và có thêm các thông tin khác để bổ sung. Các ngôn ngữ hình thức được thiết kế gần như rõ ràng tuyệt đối, tức là mỗi mệnh để chỉ có đúng một nghĩa, bất kể ngữ cảnh như thế nào.
về sự dư thừa:
Để loại trừ sự mập mờ và tránh gây hiểu nhầm, ngôn ngữ tự nhiên cần dùng đến nhiều nội dung bổ trợ làm dài thêm nội dung. Các ngôn ngữ hình thì gọn gàng hơn.
về văn phong:
Các ngôn ngữ tự nhiên có chứa nhiều thành ngữ và ẩn dụ. Các ngôn ngữ hình thức luôn luôn có nghĩa đúng theo những gì được viết ra.
Chúng ta dùng ngôn ngữ tự nhiên ngay từ thuở nhỏ, nên thường có một thời gian khó khăn ban đầu khi làm quen với ngôn ngữ hình thức. Về phương diện nào đó, sự khác biệt giữa ngôn ngữ hình thức và ngôn ngữ tự nhiên cung như khác biệt giữa thơ ca và văn xuôi, dù hơn thế nữa.

Thơ ca:
Các từ được dùng với cả chức năng âm điệu bên cạnh chức năng ý nghĩa, và toàn bộ bài thơ/ca tạo ra hiệu quả cảm xúc. Luôn mang tính không rõ ràng, thậm chí còn là chủ định của tác giả.
Văn xuôi:
Coi trọng ý nghĩa của câu chữ hơn, và cấu trúc giúp cho việc diễn đạt ý nghĩa.
Chương trình:
Ý nghĩa của một chương trình máy tính là rõ ràng và được diễn đạt hoàn toàn thông qua câu chữ, theo đó ta có thể hiểu được trọn ven bằng cách phân tích các nguyên tố và cấu trúc.
Khi đọc chương trình (hoặc một ngôn ngữ hình thức nào khác) bạn nên làm như sau. Trước hết, hãy nhớ rằng ngôn ngữ hình thức cô đọng hơn ngôn ngữ tự nhiên, nên phải mất nhiều thời gian để đọc hơn. Mặt khác, cấu trúc cũng rất quan trọng, do đó không nên chỉ đọc qua một lượt từ trên xuống dưới. Thay vì vậy, bạn nên học cách phân tách ngôn ngữ trong trí óc, nhận diện các nguyên tố và diễn giải cấu trúc. Cuối cùng, những chi tiết đóng vai trò quan trọng. Các lỗi dù là nhỏ nhất trong cách viết các từ hoặc dấu câu trong ngôn ngữ hình thức sẽ có thể gây ra khác biệt lớn về ý nghĩa.

1.5 Chương trình đầu tiên

Theo thông lệ, chương trình đầu tiên mà bạn viết theo một ngôn ngữ lập trình mới có tên gọi là “Hello, World!” vì tất cả những gì nó thực hiện chỉ là làm hiện ra dòng chữ “Hello, World!” Một chương trình như vậy trong Java được viết như sau:

```
class Hello {
    // main: xuất ra một thông tin đơn giản
    public static void main(String[] args) {
        System.out.println("Hello, world.");
    }
}
```

Chương trình này có những đặc điểm hơi khó giải thích cho người mới bắt đầu, song nó giúp ta có cái nhìn bao quát về những chủ đề sau này sẽ được học.

Một chương trình Java được hợp thành từ những lời **khai báo lớp**, vốn có dạng sau:

```
class TENLOP {

    public static void main (String[] args) {


        CAC_CAU_LENH

    }


}
```

Ở đây TENLOP là một tên gọi do người lập trình đặt. Trong ví dụ trên, tên lớp là `Hello`.
`main` là một **phương thức**, tức là một tập hợp được đặt tên, bao gồm các câu lệnh. Tên gọi main này rất đặc biệt; nó đánh dấu điểm khởi đầu của chương trình. Khi chạy chương trình, câu lệnh đầu tiên trong main sẽ là điểm bắt đầu và kết thúc ở câu lệnh cuối cùng trong đó.
`main` có thể gồm nhiều câu lệnh, nhưng ở ví dụ trên thì chỉ có một. Đó là câu lệnh in, nghĩa là nó hiển thị một giá trị trên màn hình. Chỗ này dễ gây lẫn, "print" có thể mang ý nghĩa "hiện ra trên màn hình" hay "gửi nội dung đến máy in". Trong cuốn sách này, tôi không nói về việc gửi đến máy in; tất cả việc in của chúng ta là hiển thị lên màn hình. Lệnh in kết thúc bằng một dấu chấm phẩy (;).
`System.out.println` là một phương thức do thư viện của Java cung cấp. Một **thư viện** là tập hợp gồm những lời định nghĩa lớp và phương thức.
Java dùng những cặp ngoặc nhọn ({ và }) để nhóm thông tin lại với nhau. Cặp ngoặc nhọn ở ngoài cùng (các dòng 1 và 8) chứa lời định nghĩa lớp, còn cặp ngoặc nhọn phía trong thì chứa lời định nghĩa cho main.

Dòng 3 bắt đầu bằng //. Như vậy dòng này là một **lời chú thích**, tức là một đoạn chữ mà bạn có thể viết vào chương trình, thường để giải thích công dụng của chương trình. Khi trình biên dịch thấy //, nó sẽ phớt lờ những gì kể từ đó đến cuối dòng.

1.6 Thuật ngữ

giải quyết vấn đề:
Quá trình thiết lập bài toán, tìm lời giải, và biểu diễn lời giải.
ngôn ngữ bậc cao:
Ngôn ngữ lập trình như Python được thiết kế nhằm mục đích để con người dễ đọc và viết.
ngôn ngữ bậc thấp:
Ngôn ngữ lập trình được thiết kế nhằm mục đích để máy tính dễ thực hiện; còn gọi là "ngôn ngữ máy" hoặc "hợp ngữ".
tính khả chuyển:
Đặc tính của chương trình mà có thể chạy trên nhiều loại máy tính khác nhau.
thông dịch:
Thực hiện chương trình được viết bằng ngôn ngữ bậc cao bằng cách dịch nó theo từng dòng một.
biên dịch:

Dịch một lượt toàn bộ chương trình viết bằng ngôn ngữ bậc cao sang ngôn ngữ bậc thấp, để chuẩn bị thực hiện sau này.

mã nguồn:

Chương trình ở dạng ngôn ngữ bậc cao trước khi được biên dịch.

mã đối tượng:

Sản phẩm đầu ra của trình biên dịch sau khi nó đã dịch chương trình.

chương trình chạy:

Tên khác đặt cho mã đối tượng đã sẵn sàng được thực hiện.

dấu nhắc:

Các kí tự được hiển thị bởi trình thông dịch nhằm thể hiện rằng nó đã sẵn sàng nhận đầu vào từ phía người dùng.

văn lệnh:

Chương trình được lưu trong file (thường chính là chương trình sẽ được thông dịch).

chế độ tương tác:

Cách dùng trình thông dịch Python thông qua việc gõ các câu lệnh và biểu thức vào chỗ dấu nhắc.

chế độ văn lệnh:

Cách dùng trình thông dịch Python để đọc và thực hiện các câu lệnh có trong một văn lệnh.

chương trình:

Danh sách những chỉ dẫn thực hiện tính toán.

thuật toán:

Quá trình tổng quát để giải một lớp các bài toán.

lỗi:

Lỗi trong chương trình.

gỡ lỗi:

Quá trình dò tìm và gỡ bỏ cả ba kiểu lỗi trong lập trình.

cú pháp:

Cấu trúc của một chương trình.

lỗi cú pháp:

Lỗi trong chương trình mà làm cho quá trình phân tách không thể thực hiện được (và hệ quả là không thể biên dịch được).

biệt lệ:

Lỗi được phát hiện khi chương trình đang chạy.

ngữ nghĩa:

Ý nghĩa của chương trình.

lỗi ngữ nghĩa:

Lỗi có trong chương trình mà khiến cho chương trình thực hiện công việc ngoài ý định của người viết.

ngôn ngữ tự nhiên:

Ngôn ngữ bất kì được con người dùng, được trải qua sự tiến hóa tự nhiên.

ngôn ngữ hình thức:

Ngôn ngữ bất kì được con người thiết kế nhằm mục đích cụ thể, như việc biểu diễn các ý tưởng toán học hoặc các chương trình máy tính; tất cả các ngôn ngữ lập trình đều là ngôn ngữ hình thức.

nguyên tố:

Một trong những thành phần cơ bản trong cấu trúc cú pháp của một chương trình, tương đương với một từ trong ngôn ngữ tự nhiên.

phân tách:

Việc kiểm tra một chương trình và phân tích cấu trúc cú pháp.

lệnh print:

Câu lệnh khiến cho kết quả được hiển thị lên màn hình.

1.7 Bài tập

BÀI TẬP 1

Các nhà khoa học máy tính thường có thói quen dùng những từ tiếng Anh thông thường để chỉ những thứ khác với nghĩa tiếng Anh thông dụng của từ đó. Chẳng hạn, trong tiếng Anh, “statement” và “comment” đồng nghĩa với nhau, nhưng trong chương trình thì chúng khác hẳn.

Phần thuật ngữ ở cuối mỗi chương nhằm điểm lại những từ và cụm từ có ý nghĩa riêng trong ngành khoa học máy tính. Khi bạn thấy những từ quen thuộc, thì đừng lờ đi coi như đã biết nghĩa của chúng nhé!

1. Theo thuật ngữ máy tính, sự khác biệt giữa câu lệnh và chú thích như thế nào?
2. Nói một chương trình có tính khả chuyển nghĩa là gì?
3. Một chương trình chạy có nghĩa là gì?

BÀI TẬP 2

Trước khi tiếp tục, bạn hãy tìm hiểu cách biên dịch và chạy chương trình Java trong môi trường lập trình của mình. Một số loại môi trường cung cấp sẵn những chương trình mẫu tựa như ví dụ ở Mục 1.5.

1. Gõ vào chương trình “Hello, World”, rồi biên dịch và chạy nó.
2. Thêm một câu lệnh để in ra một dòng chữ thứ hai theo sau “Hello, World!”. Có thể là câu đùa vui “How are you?” Hãy biên dịch và chạy lại chương trình.
3. Thêm một chú thích vào (bất kì đâu) trong chương trình, biên dịch lại, và chạy lại lần nữa. Lời chú thích mới phải không làm ảnh hưởng đến kết quả.

Bài tập này có vẻ lặt vặt, song đây chính là điểm khởi đầu cho nhiều chương trình mà ta sẽ làm việc với. Để chắc tay gỡ lỗi, bạn phải dùng thạo môi trường lập trình của mình. Trong một số môi trường, rất dễ bị mất dấu chương trình đang chạy, và bạn có thể rơi vào trường hợp đi gỡ lỗi một chương trình trong khi vô ý chạy chương trình khác. Việc thêm vào (và thay đổi) câu lệnh in là một cách đơn giản để đảm bảo chắc chương trình đang làm việc là chương trình mà bạn chạy.

BÀI TẬP 3

Một ý tưởng hay là hãy phạm càng nhiều lỗi trong lập trình mà bạn có thể hình dung được, để thấy những thông báo lỗi nào mà trình biên dịch đưa ra. Đôi khi trình biên dịch cho bạn biết chính xác sai ở chỗ nào, và bạn chỉ việc sửa nó. Nhưng đôi khi các thông báo lỗi lại đánh lại hướng. Bạn sẽ hình thành nên một trực giác để phân biệt lúc nào thì tin cậy trình biên dịch và lúc nào phải tự hình dung ra lỗi.

1. Xóa bớt một trong các dấu mở ngoặc nhọn.
2. Xóa bớt một trong các dấu đóng ngoặc nhọn.
3. Thay vì `main`, hãy viết `mian`.
4. Xóa từ `static`.
5. Xóa từ `public`.
6. Xóa từ `System`.
7. Thay thế `println` bằng `Println`.
8. Thay thế `println` bằng `print`. Câu hỏi này đánh đố ở chỗ, đây là lỗi logic chứ không phải lỗi cú pháp. Câu lệnh`System.out.print` hoàn toàn hợp lệ, nhưng nó có thể có hoặc không làm theo điều bạn dự kiến.
9. Xóa một trong số các ngoặc tròn. Thêm vào một ngoặc tròn.

1.Định nghĩa này không đúng với mọi ngôn ngữ lập trình. Một ví dụ là các ngôn ngữ đặc tả, xemhttp://en.wikipedia.org/wiki/Declarative_programming. ↩

Chương 2: Biến và kiểu

Trở về Mục lục cuốn sách

2.1 Nói thêm về lệnh in

Bạn có thể tùy ý đặt bao nhiêu câu lệnh vào trong main cũng được; chẳng hạn, để in nhiều dòng:

```
class Hello {
  // Generates some simple output.
  public static void main(String[] args) {
    System.out.println("Hello, world."); // in một dòng
    System.out.println("How are you?"); // in dòng nữa
  }
}
```

Như ví dụ này đã cho thấy, bạn có thể đặt lời chú thích ở cuối dòng lệnh, hoặc đặt nó ở riêng một dòng.

Những cụm từ đặt giữa hai dấu nháy kép được gọi là **chuỗi**, vì chúng được hợp thành từ một dãy (chuỗi) các kí tự. Chuỗi có thể gồm bất kì tổ hợp nào từ các chữ cái, chữ số, dấu câu, và các kí tự đặc biệt khác.

println là tên gọi tắt của "print line," vì sau mỗi dòng nó thêm vào một kí tự đặc biệt, gọi là **newline**, để đẩy con trỏ xuống dòng tiếp theo trên màn hình. Lần tới, khi println được gọi, các chữ mới sẽ xuất hiện ở dòng kế tiếp.
Để hiển thị kết quả từ nhiều lệnh in trên cùng một dòng, hãy dùng print:

```
class Hello {
  // Phát sinh một số kết quả đơn giản.
  public static void main(String[] args) {
    System.out.print("Goodbye, ");
    System.out.println("cruel world!");
  }
}
```

Kết quả xuất hiện trên cùng một dòng là Goodbye, cruel world!. Có một dấu cách giữa từ "Goodbye" và dấu nháy kép tiếp theo. Dấu cách này xuất hiện ở kết quả, vì vậy nó ảnh hưởng đến hành vi của chương trình.
Những dấu cách xuất hiện ngoài cặp dấu nháy kép thì nói chung không ảnh hưởng gì đến hành vi của chương trình. Chẳng hạn, tôi đã có thể viết:

```
class Hello {
public static void main(String[] args) {
System.out.print("Goodbye, ");
System.out.println("cruel world!");
}
}
```

Chương trình này sẽ biên dịch và chạy được thông suốt như chương trình ban đầu. Dấu ngắt ở cuối dòng (dấu newline) cũng không ảnh hưởng tới hành vi của chương trình, vì vậy tôi cũng có thể đã viết thành:

```
class Hello { public static void main(String[] args) {
System.out.print("Goodbye, "); System.out.println
("cruel world!");}}
```

Chương trình này cũng hoạt động được, nhưng nó trở nên ngày càng khó đọc. Các dấu ngắt dòng và dấu cách rất có ích trong việc bố trí hình thức của chương trình, làm chương trình dễ đọc và dễ định vị lỗi hơn.

2.2 Biến

Một trong những tính năng mạnh nhất của một ngôn ngữ lập trình là khả năng thao tác với các **biến**. Biến là một tên gọi tham chiếu đến một**giá trị**. Giá trị là những thứ có thể in ra, lưu giữ, và (như ta sẽ thấy sau này) thao tác tính toán được. Các chuỗi mà ta đã in đến giờ ("Hello, World.", "Goodbye, ", v.v.) đều là những giá trị.
Để lưu một giá trị, bạn phải tạo ra một biến. Vì những giá trị ta muốn lưu trữ ở đây là các chuỗi, nên ta khai báo biến mới là một chuỗi

```
    String bob;
```

Đây là **câu lệnh khai báo**, vì nó khai báo rằng biến mang tên bob có kiểu String. Mỗi biến có một kiểu với tác dụng quyết định loại giá trị nào mà biến đó có thể lưu trữ được. Chẳng hạn, kiểu int có thể lưu trữ các số nguyên, còn kiểu String lưu trữ chuỗi.
Một số kiểu có tên gọi bắt đầu bằng chữ in và một số kiểu thì bắt đầu bằng chữ thường. Ta sẽ học ý nghĩa của sự phân biệt này về sau, còn bây giờ chỉ cần lưu ý để viết đúng. Không có kiểu nào gọi là Int hay string, và trình biên dịch sẽ bác bỏ nếu bạn cố gắng dựng nên một cái tên như vậy.
Để tạo nên một biến nguyên, cú pháp là int bob;, trong đó bob là tên gọi tùy ý mà bạn đặt cho biến. Nói chung, bạn sẽ muốn đặt tên biến để chỉ rõ mục tiêu dùng biến đó. Chẳng hạn, nếu thấy các lệnh khai báo biến sau đây:

```
    String firstName;
    String lastName;
    int hour, minute;
```

thì bạn có thể đoán được rằng những giá trị nào sẽ được lưu vào chúng. Ví dụ này cũng giới thiệu cú pháp để khai báo nhiều biến với cùng kiểu: hour và second đều là số nguyên (kiểu int).

2.3 Lệnh gán

Bây giờ khi đã tạo nên các biến, ta muốn lưu giữ những giá trị. Ta làm điều này với **lệnh gán.**

```
    bob = "Hello."; // cho bob giá trị "Hello."
    hour = 11; // gán giá trị 11 vào hour
    minute = 59; // đặt minute là 59
```

Ví dụ này có ba lệnh gán, và các lời chú thích đi kèm cho thấy ba cách khác nhau mà chúng ta đôi khi nói về câu lệnh gán. Cách dùng từ có thể gây nhầm lẫn, song ý tưởng rất đơn giản:

- Khi khai báo một biến, bạn tạo nên một chỗ lưu dữ liệu được đặt tên.
- Khi gán cho một biến, bạn cho nó một giá trị.
  Một cách thông dụng để biểu diễn biến trên giấy là vẽ một hộp với tên biến ghi bên ngoài và giá trị biến ở beent rong. Hình dưới đây cho thấy hiệu ứng của ba câu lệnh gán này:

Một quy tắc chung là biến phải có cùng kiểu với giá trị mà bạn gán cho nó. Bạn không thể lưu trữ một String vào trong minute hay một số nguyên vào bob.
Mặt khác, quy tắc này cũng có thể gây nhầm lẫn, vì có rất nhiều cách để bạn chuyển giá trị từ kiểu này sang kiểu khác, và đôi khi Java cũng tự động chuyển đổi. Riêng bây giờ thì bạn nên nhớ quy tắc chung, và sau này ta sẽ nói về những ngoại lệ.

Một điều nữa dễ gây nhầm đó là có những chuỗi *trông* giống như số nguyên nhưng thực ra lại không phải. Chẳng hạn, bob có thể chứa chuỗi "123", vốn được tạo thành từ các kí tự 1, 2 và 3, nhưng nó không phải là *số* 123.

```
    bob = "123"; // hợp lệ

    bob = 123; // không hợp lệ
```

2.4 In các biến

Để hiển thị giá trị của một biến, bạn có thể dùng println hoặc print:

```
class Hello {

  public static void main(String[] args) {

    String firstLine;

    firstLine = "Hello, again!";

    System.out.println(firstLine);

  }

}
```

Chương trình này tạo ra ba biến firstLine, gán nó với giá trị "Hello, again!" rồi in giá trị đó ra. Khi ta nói “in một giá trị,” điều này nghĩa là in *giá trị* của biến đó. Để in *tên* của một biến, bạn phải đặt cái tên này trong cặp dấu nháy kép. Chẳng hạn:System.out.println("firstLine");
Ví dụ như, bạn có thể viết

```
    String firstLine; firstLine = "Hello, again!";

    System.out.print("The value of firstLine is ");

    System.out.println(firstLine);
```

Kết quả của chương trình này là

```
The value of firstLine is Hello, again!
```

Tôi vui mừng thông báo với bạn rằng cú pháp của lệnh in một biến thì giống nhau bất kể kiểu của biến đó là gì.

```
    int hour, minute;
```

```
    hour = 11;
    minute = 59;
    System.out.print("The current time is ");
    System.out.print(hour);
    System.out.print(":");
    System.out.print(minute);
    System.out.println(".");
```

Kết quả của chương trình này là The current time is 11:59.
CẢNH BÁO: Để đặt nhiều giá trị trên cùng một dòng, cách thông dụng là dùng nhiều lệnh print và tiếp theo là println. Nhưng bạn phải nhớ viết println ở cuối. Trong nhiều môi trường lập trình, kết quả của print chỉ lưu giữ mà không được hiển thị đến tận lúc println được gọi, khi đó cả dòng sẽ xuất hiện cùng lúc. Nếu bạn bỏ mất println, chương trình có thể kết thúc mà không hiển thị kết quả đã được lưu trữ!

2.5 Từ khoá

Cách đây vài mục, tôi đã nói rằng bạn có thể đặt một tên tùy ý cho biến, nhưng điều này không hẳn là đúng. Có những từ nhất định được dành riêng trong Java vì chúng được trình biên dịch sử dụng để phân tách cấu trúc của chương trình mà bạn viết; và nếu bạn dùng những từ này đặt cho tên biến thì trình biên dịch sẽ bị lẫn. Các từ như vậy, gọi là **từ khóa**, bao gồm có public, class, void, int, và nhiều từ khác. Bạn có thể xem danh sách đầy đủ tại http://download.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html. Trang này, được Oracle cung cấp, có đăng tài liệu về Java mà trong sách này tôi thường xuyên tham khảo đến.

2.6 Toán tử

Toán tử là các kí hiệu đặc biệt để biểu diễn các phép tính như cộng và nhân. Hầu hết các toán tử của Java đều thực hiện theo đúng dự định của bạn vì chúng là những kí hiệu toán học thông dụng. Chẳng hạn, toán tử của phép cộng là `+`. Phép trừ là `-`, phép nhân là `*`, và phép chia là `/`.

```
1+1   hour-1   hour*60 + minute   minute/60
```

Các biểu thức có thể chứa cả tên biến và con số. Các biến đều được thay bằng giá trị của chúng trước khi phép tính được thực hiện.

Hơn nữa, dù phép trừ và phép nhân làm đúng điều bạn muốn, song phép chia có thể làm bạn ngạc nhiên. Chẳng hạn, chương trình này:

```
    int hour, minute;
    hour = 11;
    minute = 59;
    System.out.print("Number of minutes since midnight: ");
    System.out.println(hour*60 + minute);
    System.out.print("Fraction of the hour that has passed: ");
    System.out.println(minute/60);
```

phát sinh ra kết quả này:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

Dòng đầu tiên thì đúng như mong đợi, nhưng dòng thứ hai thật kì quặc. Giá trị của `minute` là 59, và 59 chia cho 60 bằng 0.98333, chứ không phải là 0. Vấn đề ở đây là Java đã thực hiện phép **chia nguyên**. Khi cả hai **toán hạng** đều là số nguyên (toán hạng là những đại lượng mà toán tử thực hiện tính toán), thì kết quả cũng sẽ là một số nguyên, và theo quy định chung, phép chia nguyên làm tròn *xuống*, ngay cả trong trường hợp này khi giá trị sát với số nguyên phía trên hơn.
Một cách làm khác là đi tính phần trăm thay vì một phân số:

```
    System.out.print("Percentage of the hour that has passed: ");
    System.out.println(minute*100/60);
```

Kết quả là:

```
Percentage of the hour that has passed: 98
```

Một lần nữa kết quả lại được làm tròn xuống, nhưng lần này đáp số đã gần đúng hơn. Để được kết quả chính xác hơn, ta có thể dùng một kiểu biến khác, gọi là dấu phẩy động, để lưu trữ những giá trị có phần thập phân. Ta sẽ tiếp tục vấn đề này trong chương sau.

2.7 Thứ tự thực hiện

Khi trong biểu thức có nhiều hơn một toán tử, thứ tự định lượng sẽ tuân theo **quy tắc ưu tiên**. Giải thích đầy đủ quy tắc ưu tiên này có thể phức tạp, nhưng để bắt đầu, bạn chỉ cần nhớ:

- Các phép nhân và chia phải được thực hiện trước cộng và trừ. Vì vậy `2*3-1` bằng 5 chứ không phải 4, và 2/3-1 được -1, chứ không phải 1 (hãy nhớ rằng phép chia nguyên 2/3 bằng 0).
- Nếu các toán tử có cùng độ ưu tiên thì chúng được định lượng từ trái sang phải. Vì vậy, trong biểu thức minute*100/60, phép nhân được thực hiện trước, cho ra 5900/60, và cuối cùng là 98. Nếu các phép toán chạy từ phải qua trái, kết quả đã thành 59*1 tức là 59, điều này là sai.
- Bất cứ khi nào muốn vượt quy tắc ưu tiên (hay khi bạn không chắc quy tắc này như thế nào), bạn có thể dùng cặp ngoặc đơn. Biểu thức trong cặp ngoặc đơn được thực hiện trước, bởi vậy 2 *(3-1) là 4. Bạn cũng có thể dùng cặp ngoặc đơn để biểu thức trở nên dễ đọc, như với `(minute * 100) / 60`, ngay cả khi không có nó thì kết quả cũng không đổi.

2.8 Các thao tác với chuỗi

Nói chung, bạn không thể thực hiện các phép toán đối với chuỗi, ngay cả khi chuỗi trông giống như những con số. Vì vậy các biểu thức sau đây đều không hợp lệ:

```
bob - 1    "Hello"/123    bob * "Hello"
```

Tiện thể, qua những biểu thức trên bạn có phân biệt được liệu bob là số nguyên hay chuỗi không? Không. Cách duy nhất để biết được kiểu của biển là nhìn vào nơi nó được khai báo.
Điều thú vị là toán tử `+` *có* tác dụng với chuỗi, nhưng có lẽ không hoạt động theo cách bạn mong đợi. Với các String, toán tử + có nhiệm vụ**nối**, nghĩa là ghép nối tiếp hai toán hạng với nhau. Bởi vậy "Hello, " + "world." sẽ cho ra chuỗi "Hello, world." còn bob + "ism" thì thêm đuôi *ism* vào bất kì chữ gì mà bob lưu trữ, cách đặt tên này này thật tiện trong tay những người quen tính bài bác.

2.9 Kết hợp

Đến giờ ta đã xem xét những thành phần của ngôn ngữ lập trình—biến, biểu thức, và câu lệnh—một cách biệt lập, mà chưa nói về cách kết hợp chúng.

Một trong những đặc điểm có ích nhất của ngôn ngữ lập trình là khả năng tập hợp những thành phần nhỏ rồi **kết hợp** chúng lại. Chẳng hạn, ta biết cách tính nhân và biết dùng lệnh in; như vậy hóa ra là có thể kết hợp chúng lại thành một câu lệnh:

```
    System.out.println(17 * 3);
```

Bất kì biểu thức nào có số, chuỗi, và biến đều có thể dùng trong lệnh in. Ta đã thấy một ví dụ:

```
    System.out.println(hour*60 + minute);
```

Nhưng bạn cũng có thể đặt biểu thức bất kì ở vế phải của một lệnh gán:

```
    int percentage;
    percentage = (minute * 100) / 60;
```

Ngay bây giờ thì tính năng này xem ra chưa có gì ấn tượng, nhưng ta sẽ thấy những ví dụ mà cách kết hợp này biểu diễn những phép tính phức tạp một cách gọn gàng, ngăn nắp.

CẢNH BÁO: Vế trái của một lệnh gán phải là một tên *biến*, chứ không phải một biểu thức. Đó là vì vế trái dùng để chỉ định vị trí lưu giữ kết quả. Các biểu thức thì không thể hiện vị trí lưu giữ này, mà chỉ thể hiện giá trị. Vì vậy cách viết sau không hợp lệ: minute+1 = hour;.

2.10 Thuật ngữ

biến:
Tên được tham chiếu đến một giá trị.

giá trị:
Một con số hoặc chuỗi kí tự (hoặc những thứ khác sau này được đặt tên) mà lưu trữ được vào trong một biến. Mỗi giá trị thuộc về một kiểu.

kiểu:
Một tập hợp gồm các giá trị. Kiểu của biến quyết định những giá trị nào có thể lưu trữ trong biến đó. Những kiểu mà ta đã gặp bao gồm kiểu số nguyên (`int` trong Java) và chuỗi (`String` trong Java).

từ khoá:
Từ dành riêng cho trình biên dịch để phân tách một chương trình. Bạn không thể dùng những từ khoá nhưpublic, class và void để đặt tên biến.

lệnh khai báo:
Câu lệnh nhằm tạo ra một biến mới và quy định kiểu cho nó.

lệnh gán:
Lệnh để gán một giá trị cho một biến.

biểu thức:
Tổ hợp của các biến, toán tử, và giá trị nhằm biểu diễn một giá trị kết quả duy nhất. Biểu thức cũng có kiểu; kiểu này được quyết định bởi các toán tử và toán hạng.

toán tử:

Kí hiệu dùng để biểu diễn một phép tính đơn nhất như cộng, nhân, hoặc nối chuỗi.

toán hạng:

Một trong những giá trị mà toán tử thực hiện với.

ưu tiên:

Thứ tự mà những biểu thức bao gồm nhiều toán tử và toán hạng được định lượng.

nối:

Ghép nối tiếp hai toán hạng.

kết hợp:

Khả năng ghép những biểu thức và câu lệnh đơn giản thành những biểu thức và câu lệnh phức hợp để biểu diễn gọn gàng các thao tác tính toán.

2.11 Bài tập

BÀI TẬP 1

Nếu đang đọc quyển sách này trên lớp, bạn có thể thích bài tập này: hãy tìm một người bạn để chơi trò "Stump the Chump":

Bắt đầu từ một chương trình biên dịch và chạy được trơn tru. Từng người một quay mặt đi trong lúc người kia gài một lỗi vào chương trình. Sau đó người thứ nhất quay lại rồi cố tìm và sửa lỗi. Nếu tìm được lỗi mà không cần biên dịch, sẽ được hai điểm; tìm được sau khi biên dịch thì được 1 điểm; và nếu không tìm được thì người kia sẽ được một điểm.

BÀI TẬP 2

1. Hãy tạo ra một chương trình có tên Date.java. Sao chép hoặc gõ vào một chương trình kiểu như "Hello, World" rồi đảm bảo chắc rằng bạn có thể biên dịch và chạy được chương trình.
2. Làm theo ví dụ ở Mục 2.4, hãy viết một chương trình để tạo ra các biến day, date, month và year. day sẽ chứa ngày trong tuần còn date thì chứa ngày trong tháng. Từng biến sẽ có kiểu gì? Hãy gán giá trị vào những biến này để biểu diễn ngày hôm nay.
3. In giá trị từng biến trên mỗi dòng riêng. Đây là một bước trung gian và rất cần để kiểm tra rằng chương trình còn hoạt động ổn thỏa.
4. Sửa chương trình để in ra ngày theo dạng chuẩn Hoa Kỳ: Saturday, July 16, 2011.
5. Sửa lại chương trình lần nữa để kết quả thu được là:

```
American format:

Saturday, July 16, 2011

European format:

Saturday 16 July, 2011
```

Mục đích của bài tập này là dùng phép nối chuỗi để hiển thị các giá trị có kiểu khác nhau (int và String), đồng thời thực hành kĩ năng phát triển dần chương trình qua việc mỗi lần chỉ thêm vào một vài câu lệnh.

BÀI TẬP 3

1. Hãy tạo ra một chương trình mới có tên là Time.java. Từ giờ trở đi, tôi không nhắc bạn bắt đầu bằng việc tạo một chương trình nhỏ nhưng chạy được; song bạn nên làm điều này.
2. Từ ví dụ ở Mục 2.6, hãy tạo ra các biến có tên hour, minute và second, rồi gán cho chúng giá

trị biểu diễn gần đúng giờ hiện tại. Dùng cách đếm số 24 giờ, theo đó 2 giờ chiều sẽ ứng với giá trị của hour bằng 14.

3. Làm cho chương trình tính toán rồi in ra số giây kể từ nửa đêm.
4. Làm cho chương trình tính toán rồi in ra số giây từ giờ đến hết ngày hôm nay.
5. Làm cho chương trình tính toán rồi in ra số phần trăm thời gian đã trôi qua trong ngày hôm nay.
6. Thay đổi các giá trị của hour, minute và second để phản ánh thời gian hiện tại (coi như có thời gian trôi qua kể từ lần chạy trước), và kiểm tra để đảm bảo rằng chương trình hoạt động được với nhiều giá trị khác nhau.

   Mục đích của bài tập này là vận dụng một số phép toán, và bắt đầu suy nghĩ về những dữ liệu phức hợp như thời gian trong ngày, vốn được biểu diễn bởi nhiều giá trị. Đồng thời, bạn cũng có thể vấp phải nhiều vấn đề khi tính phần trăm với các số int; đây chính là lí do dẫn đến việc dùng số có dấu phẩy động trong chương sau.

   GỢI Ý: có thể bạn sẽ cần dùng thêm các biến để lưu giữ tạm thời những giá trị trong quá trình tính toán. Các biến kiểu này, vốn được dùng trong tính toán nhưng không bao giờ được in ra, đôi khi được gọi là biến trung gian hoặc biến tạm.

Chương 3: Phương thức rỗng

Trở về Mục lục cuốn sách

3.1 Dấu phẩy động

Ở chương trước ta đã gặp trục trặc khi tính toán những số không nguyên. Ta đã sửa một cách tạm bợ bằng việc tính số phần trăm thay vì số thập phân, nhưng một giải pháp tổng quát hơn sẽ là dùng số có dấu phẩy động, để biểu diễn được cả số nguyên lẫn số có phần thập phân. Trong Java, kiểu dấu phẩy động có tên double, là chữ gọi tắt của "double-precision" (độ chuẩn xác kép).
Bạn có thể tạo nên các biến phẩy động rồi gán giá trị cho chúng theo cú pháp giống như ta đã làm với những kiểu dữ liệu khác. Chẳng hạn:

```
double pi;
pi = 3.14159;
```

Việc khai báo một biến đồng thời gán giá trị cho nó cũng hợp lệ:

```
int x = 1;
String empty = "";
double pi = 3.14159;
```

Cú pháp này rất thông dụng; việc kết hợp giữa khai báo và gán đôi khi còn được gọi là phép **khởi tạo**.

Mặc dù các số phẩy động rất hữu ích nhưng chúng cũng là nguồn gây nên rắc rối, vì dường như có phần trùng nhau giữa các số nguyên và số phẩy động. Chẳng hạn, nếu bạn có giá trị 1, thì đó là số nguyên, số phẩy động, hay cả hai?
Java phân biệt giá trị số nguyên 1 với giá trị phẩy động 1.0, dù rằng chúng có vẻ cùng là một số. Chúng thuộc về hai kiểu dữ liệu khác nhau, và nói chặt chẽ thì bạn không được phép gán giá trị kiểu này cho một biến kiểu khác. Chẳng hạn, câu lệnh sau là hợp lệ:

```
int x = 1.1;
```

vì biến ở vế trái là int còn giá trị ở vế phải là double. Nhưng rất dễ quên mất quy tắc này, đặc biệt là vì có những nơi mà Java sẽ tự động chuyển từ một kiểu này sang kiểu khác. Chẳng hạn:

```
double y = 1;
```

về lý thì không hợp lệ, nhưng Java vẫn cho phép điều này nhờ cách tự động chuyển đổi
từ int sang double. Sự dễ dãi này khá tiện lợi, song có thể gặp vấn đề, chẳng hạn:

```
double y = 1 / 3;
```

Bạn có thể trông đợi rằng biến y nhận giá trị 0.333333, vốn là một giá trị phẩy động hoàn toàn hợp lệ, song thực ra nó nhận được 0.0. Lý do là biểu thức vế phải là tỉ số giữa hai số nguyên, vì vậy Java thực hiện phép *chia nguyên*, và cho giá trị số nguyên bằng 0. Chuyển thành dạng số phẩy động, kết quả là 0.0.
Một cách giải quyết vấn đề này (khi bạn đã hình dung ra) là làm cho vế phải trở thành một biểu thức chứa số phẩy động:

```
double y = 1.0 / 3.0;
```

Bằng cách này đã đặt y bằng 0.333333, như dự kiến.
Các toán tử mà ta đã gặp đến giờ—cộng, trừ, nhân, và chia—cũng làm việc được với các giá trị dấu phẩy động, mặc dù bạn có thể thấy thú vị khi biết được rằng cơ chế bên trong thì khác hẳn. Thực ra, đa số các bộ vi xử lý đều có phần mềm chuyên dụng để thực hiện các phép tính có dấu phẩy động.

3.2 Chuyển đổi từ double sang int

Như tôi đã nói, Java quy đổi các số int thành double một cách tự động nếu thấy cần thiết, vì trong quá trình chuyển đổi không bị mất thông tin. Ngược lại, chuyển từ double sang int lại cần phải làm tròn số. Java không tự động làm việc này, để đảm bảo rằng bạn, người lập trình, cũng biết được rằng phần thập phân của số sẽ bị mất đi.
Cách đơn giản nhất để chuyển một giá trị số phẩy động sang số nguyên là thực hiện việc **định kiểu** (typecast). Sở dĩ gọi là định kiểu vì bằng cách đó ta có thể lấy một giá trị thuộc kiểu này rồi "ấn định" nó thành kiểu khác (như việc định hình bằng khuôn đúc kim loại).
Cú pháp của định kiểu là đặt tên kiểu giữa cặp ngoặc tròn rồi dùng nó như một toán tử. Chẳng hạn,

```
double pi = 3.14159;

int x = (int) pi;
```

Toán tử (int) có tác dụng chuyển bất kì thứ gì đi sau nó thành một số nguyên, bởi vậy x nhận giá trị bằng 3.
Định kiểu có quyền ưu tiên cao hơn so với các toán tử số học, bởi vậy ở ví dụ sau, trước hết giá trị của pi được chuyển thành số nguyên, và kết quả sẽ là 60.0, chứ không phải 62.

```
double pi = 3.14159;

double x = (int) pi * 20.0;
```

Việc chuyển thành số nguyên sẽ luôn làm tròn xuống, ngay cả khi phần thập phân là 0.99999999. Cách hoạt động này (quyền ưu tiên và việc làm tròn) có thể khiến cho việc định kiểu dễ gây nên lỗi.

3.3 Các phương thức Math

Khi làm toán, có lẽ bạn đã thấy các hàm như sin và log, đồng thời cũng biết cách tính các biểu thức như $\sin(\pi/2)$ và $\log(1/x)$. Đầu tiên, bạn lượng giá biểu thức trong cặp ngoặc tròn, vốn được gọi là **đối số** của hàm. Tiếp theo bạn lượng giá bản thân hàm đó, bằng cách tra bảng hoặc tính toán.
Công đoạn này có thể được áp dụng lặp lại để lượng giá những biểu thức phức tạp hơn như $\log(1/\sin(\pi/2))$. Đầu tiên, bạn lượng giá đối số của hàm đứng trong cùng, rồi lượng giá bản thân hàm đó, và cứ như vậy.

Java cung cấp cho ta các hàm để thực hiện những phép toán thông dụng nhất. Những hàm này được gọi là **phương thức**. Các phương thức toán học được kích hoạt bằng cách dùng cú pháp tương tự như câu lệnh print mà ta đã gặp:

```
double root = Math.sqrt(17.0);

double angle = 1.5;

double height = Math.sin(angle);
```

Ví dụ đầu tiên đặt root bằng căn bậc hai của 17. Ví dụ thứ hai đi tìm sin của giá trị angle, vốn là 1.5. Java giả thiết rằng những giá trị bạn dùng với sin và các hàm lượng giác khác (cos, tan) đều tính theo *radian*.

Để chuyển từ độ sang radian, bạn có thể chia cho 360 đồng thời nhân với 2π. Thật tiện là Java có cung cấp Math.PI:

```
    double degrees = 90;

    double angle = degrees * 2 * Math.PI / 360.0;
```

Lưu ý rằng chữ PI đều viết in toàn bộ. Java không nhận ra Pi, pi, hay pie.
Một phương thức hữu dụng khác có trong lớp Math là round, để làm tròn một giá trị số phẩy động về số nguyên gần đó nhất rồi trả lại một int.

```
int x = Math.round(Math.PI * 20.0);
```

Trong trường hợp này phép nhân xảy ra đầu tiên, trước khi phương thức được kích hoạt. Kết quả là 63 (được làm tròn lên từ 62.8319).

3.4 Kết hợp

Cũng như với các hàm toán học, những phương thức trong Java có thể được **kết hợp** lại, nghĩa là bạn có thể dùng một biểu thức làm thành phần trong biểu thức khác. Chẳng hạn, bạn có thể dùng bất kì biểu thức nào làm đối số cho một phương thức:

```
    double x = Math.cos(angle + Math.PI/2);
```

Câu lệnh này lấy giá trị Math.PI, đem chia cho hai rồi cộng kết quả thu được vào giá trị của biến angle. Tiếp theo, tổng này được truyền làm tham số cho cos. (PI là tên của một biến, chứ không phải một phương thức; bởi vậy mà không có đối số nào, thậm chí không có cả đối số rỗng ()).
Bạn cũng có thể lấy kết quả của một phương thức để truyền làm đối số cho phương thức khác:

```
    double x = Math.exp(Math.log(10.0));
```

Trong Java, phương thức log luôn dùng cơ số bằng *e*, bởi vậy câu lệnh này tìm loga cơ số *e* của 10 rồi nâng *e* lên số mũ đó. Kết quả được gán cho x; hi vọng rằng bạn biết phép tính này để làm gì.

3.5 Bổ sung những phương thức mới

Đến bây giờ, chúng ta mới chỉ dùng những phương thức có sẵn trong Java, song thật ra có thể tạo ra những phương thức mới. Ta đã thấy một lời định nghĩa cho phương thức main. Phương thức tên là main có ý nghĩa đặc biệt, song cú pháp của nó cũng giống như các phương thức khác:

```
    public static void TÊN( DANH SÁCH THAM SỐ ) { CÁC CÂU LỆNH }
```

Bạn có thể lấy tên bất kì để đặt cho phương thức mới, miễn là không phải main hay một từ khóa Java nào đó. Theo quy ước, các phương thức Java bắt đầu bằng chữ thường và dùng cách viết in từng chữ đầu của từ (còn gọi là "camel caps"), một tên gọi thú vị để chỉ những cái tên kiểu như jammingWordsTogetherLikeThis.
Danh sách các tham số thì quy định những thông tin (nếu có) mà bạn phải cung cấp khi dùng (hay **kích hoạt**) phương thức mới này.

Tham số của phương thức main là String[] args; điều này nghĩa là ai muốn kích hoạt main thì phải cung cấp một mảng các chuỗi (String) (ta sẽ bàn đến mảng ở Chương 12). Một số phương thức ta tập viết đầu tay thì không có tham số nào, vì vậy cú pháp sẽ có dạng như sau:

```
  public static void newLine() {

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

```
  }
```

Phương thức này có tên newLine, và cặp ngoặc tròn không chứa gì đồng nghĩa với việc phương thức này không nhận tham số. Nó chỉ có một câu lệnh, nhằm in một String rỗng, được biểu thị bởi "". Việc in một String mà không có chữ nào trong đó dường như là việc vô ích, nhưng vì println sẽ nhảy xuống dòng dưới sau khi in, nên câu lệnh này có tác dụng xuống dòng.
Trong main ta kích hoạt phương thức mới này cũng giống như cách ta kích hoạt các phương thức của Java:

```
  public static void main(String[] args) {
    System.out.println("First line.");
    newLine();
    System.out.println("Second line.");
  }
```

Kết quả của chương trình này là

```
First line.


Second line.
```

Lưu ý đến dòng trống giữa hai dòng chữ trên. Ta phải làm gì nếu muốn hai dòng này cách nhau xa hơn? Ta có thể liên tiếp kích hoạt phương thức mới này:

```
  public static void main(String[] args) {
    System.out.println("First line.");
    newLine();
    newLine();
    newLine();
    System.out.println("Second line.");
  }
```

Hoặc ta cũng có thể viết một phương thức mới khác, có tên threeLine, để in ra ba dòng trống:

```
  public static void threeLine() {
    newLine(); newLine(); newLine();
  }
  public static void main(String[] args) {
    System.out.println("First line.");
    threeLine();
    System.out.println("Second line.");
  }
```

Bạn có thể nhận thấy vài điều sau từ chương trình trên:

•Có thể kích hoạt cùng một phương thức nhiều lần.
•Trong một phương thức, bạn có thể kích hoạt một phương thức khác. Ở trường hợp này, main kích

hoạt threeLine còn threeLine thì kích hoạt newLine.
•Trong threeLine tôi đã viết ba câu lệnh trên cùng một dòng; đây là điều hoàn toàn hợp lệ (hãy nhớ lại rằng các dấu trống và dấu xuống dòng thường không làm thay đổi ý nghĩa của chương trình). Mặc dù ta nên đặt mỗi câu lệnh trên một dòng riêng, song đôi khi tôi vẫn phá vỡ nguyên tắc này.
Có thể bạn sẽ tự hỏi tại sao lại phiền phức tạo ra những phương thức mới như vậy. Có một vài lí do, mà hai lí do trong số đó thể hiện qua ví dụ trên là:

1.Việc tạo phương thức mới cho ta cơ hội đặt tên cho một nhóm các câu lệnh. Những phương thức có thể làm đơn giản chương trình qua việc ẩn giấu những thao tác tính toán phức tạp phía sau một câu lệnh đơn giản, và qua việc dùng những từ tiếng Anh thay cho mã lệnh bí hiểm. Theo bạn, cách viết nào rõ ràng hơn, newLine hay System.out.println("")?
2.Việc tạo phương thức mới có thể rút ngắn chương trình bằng cách loại bỏ những đoạn mã lệnh lặp đi lặp lại. Chẳng hạn, để in chín dòng trống liên tiếp, bạn chỉ cần kích hoạt threeLine đúng ba lần.
Ở mục 7.6 ta sẽ quay trở lại câu hỏi này đồng thời kể thêm một số lợi ích khác của việc chia nhỏ chương trình thành các phương thức.

3.6 Lớp và phương thức

Chắp nối lại những đoạn mã từ mục trước, ta có lời định nghĩa lớp như sau:

```
class NewLine {
  public static void newLine() {
    System.out.println("");
  }
  public static void threeLine() {
    newLine(); newLine(); newLine();
  }
  public static void main(String[] args) {
    System.out.println("First line.");
    threeLine();
    System.out.println("Second line.");
  }
}
```

Dòng thứ nhất cho biết rằng đó là lời định nghĩa một lớp mới có tên NewLine. **Lớp** là tập hợp các phương thức có liên quan đến nhau. Trong trường hợp này, lớp với tên gọi NewLine có chứa 3 phương thức tên là newLine, threeLine, và main.
Một lớp khác mà ta đã gặp là lớp Math. Nó gồm các phương thức có tên sqrt, sin, v.v. Khi kích hoạt một phương thức toán học, ta phải nêu tên của lớp (Math) và tên của phương thức. Đó là lý do mà về cú pháp, có điểm khác biệt nhỏ giữa các phương thức Java và các phương thức mà ta viết:

```
Math.pow(2.0, 10.0);

newLine();
```

Câu lệnh thứ nhất kích hoạt phương thức pow trong lớp Math (để đưa đối số thứ nhất lên lũy thừa cấp

của đối số thứ hai). Câu lệnh tiếp theo kích hoạt phương thức newLine, mà Java giả sử rằng nó có ở trong lớp mà ta đang (tức là lớp NewLine).
Nếu bạn thử kích hoạt nhầm một phương thức từ lớp khác, trình biên dịch sẽ phát sinh một lỗi. Chẳng hạn, nếu bạn gõ vào:

```
pow(2.0, 10.0);
```

Trình biên dịch sẽ nói kiểu như, "Không thể tìm thấy phương thức có tên pow trong lớp NewLine." Nếu bạn từng thấy lời thông báo này và có lẽ đã tự hỏi rằng tại sao nó phải tìm pow trong lời định nghĩa lớp của bạn, thì bây giờ bạn đã biết rồi đó.

3.7 Chương trình có nhiều phương thức

Khi bạn nhìn vào lời định nghĩa một lớp có chứa nhiều phương thức, tất sẽ có xu hướng muốn đọc từ trên xuống dưới, nhưng điều này dễ gây nhầm lẫn, bởi đó không phải là **thứ tự thực hiện** chương trình.

Việc thực hiện (thực thi) luôn bắt đầu từ câu lệnh thứ nhất của main, bất kể nó nằm đâu trong chương trình (ở ví dụ này thì tôi đã cố ý đặt ở cuối cùng). Những câu lệnh được thực hiện lần lượt, theo thứ tự, đến khi bạn gặp một lời gọi (kích hoạt) phương thức. Việc kích hoạt phương thức cũng giống như lối rẽ khỏi luồng thực thi chương trình. Thay vì đi tiếp đến câu lệnh liền kề, bạn chuyển đến dòng lệnh đầu tiên được kích hoạt, thực hiện tất cả những câu lệnh ở đó, rồi quay lại và tiếp tục tại điểm đã rẽ ngang. Điều này nghe thật đơn giản, song bạn vẫn cần nhớ rằng một phương thức có thể kích hoạt phương thức khác. Bởi vậy, khi ta đang ở đoạn giữa của main, ta có thể buộc phải dời đi để thực hiện những câu lệnh trong threeLine. Như trong khi thực thi threeLine, có ba lần ta bị gián đoạn và phải dời đi và thực hiện newLine.
Về phần mình, newLine kích hoạt println, và tạo thêm một lối rẽ nữa. Thật may là Java rất khéo theo dõi vị trí đang thực thi, nên khi println hoàn thành, công việc lại được trả về đúng chỗ mà vừa rời khỏi newLine, và sau đó thì trở lại threeLine, rồi sau cùng trở lạimain để chương trình có thể kết thúc.
Xét về khía cạnh kĩ thuật, chương trình chưa kết thúc sau main. Thay vì vậy, luồng thực thi tìm đến chỗ mà nó dời khỏi chương trình đã kích hoạt main, tức là trình thông dịch Java. Trình thông dịch này đảm nhiệm các việc như xóa cửa sổ và dọn dẹp nói chung, rồi *sau đó* chương trình mới kết thúc.
Vậy nghĩa lí của toàn bộ những thứ lằng nhằng này là gì? Khi đọc một chương trình, bạn đừng đọc từ trên xuống dưới, mà phải đọc theo luồng thực thi.

3.8 Tham số và đối số

Có những phương thức ta đã dùng yêu cầu phải có **đối số**, vốn là những giá trị mà bạn cần cung cấp để có thể kích hoạt được chúng. Chẳng hạn, để tìm sin của một số, bạn phải cung cấp số đó. Như vậy, sin đã nhận đối số là một double. Để in ra một chuỗi, bạn phải cung cấp chuỗi đó, vì vậy println nhận đối số là một String.
Lại có những phương thức nhận nhiều đối số; chẳng hạn, pow nhận hai double, đó là cơ số và số mũ. Khi bạn dùng một phương thức, bạn phải cung cấp đối số. Khi bạn viết một phương thức, bạn cung cấp một danh sách các tham số (hay tham biến). Một **tham số** là một biến để chứa một đối số. Danh sách các tham biến chỉ định rằng cần phải có những đối số nào.

Chẳng hạn, printTwice chỉ định một tham số duy nhất, s, vốn có kiểu String. Tôi đặt tên nó là s để gợi

nhớ rằng đó là một String, song tôi cũng có thể đặt bất kì tên biến hợp lệ nào cho nó.

```
public static void printTwice(String s) {

  System.out.println(s);

  System.out.println(s);

}
```

Khi kích hoạt printTwice, ta phải cung cấp một đối số duy nhất có kiểu String.

```
printTwice("Don't make me say this twice!");
```

Khi bạn kích hoạt một phương thức, đối số mà bạn cung cấp được dùng để gán cho các tham số. Trong trường hợp này, đối số "Don't make me say this twice!" được gán cho tham số s. Quá trình này được gọi là **truyền tham số** vì giá trị được truyền từ bên ngoài phương thức vào bên trong.

Một đối số có thể là biểu thức bất kì, vì vậy nếu bạn có một biến String thì có thể dùng chính biến này làm đối số:

```
String argument = "Never say never.";

printTwice(argument);
```

Giá trị mà bạn cung cấp làm đối số sẽ phải có cùng kiểu với tham số. Chẳng hạn, nếu bạn thử dòng lệnh sau:

```
printTwice(17);
```

Bạn sẽ nhận được thông báo lỗi kiểu như "cannot find symbol" (không tìm thấy kí hiệu); thông báo này không mấy hữu ích. Lí do là Java đang tìm một phương thức có tên printTwice mà có thể nhận đối số là số nguyên. Vì chẳng có phương thức nào như vậy nên nó không thể tìm thấy "kí hiệu" đó.

System.out.println chấp nhận được tham số thuộc kiểu dữ liệu bất kì. Nhưng phương thức này chỉ là một ngoại lệ; đại đa số các phương thức thì không dễ tính như vậy.

3.9 Biểu đồ ngăn xếp

Các tham số và những biến khác chỉ tồn tại trong phương thức riêng của chúng. Trong phạm vi của main, không có cái gì gọi là s. Nếu bạn thử dùng biến này, trình biên dịch sẽ phản đối. Tương tự, trong printTwice không có thứ gì gọi là argument cả.

Một cách theo dõi xem những biến nào được sử dụng ở đâu là dùng một **biểu đồ ngăn xếp**. Với ví dụ trên, biểu đồ ngăn xếp sẽ như sau:

Mỗi phương thức đều có một hộp màu xám gọi là **khung**., trong đó chứa các tham số và biến của phương thức. Tên của phương thức được ghi bên ngoài khung. Như thường lệ, giá trị của mỗi biến lại được viết trong một hộp cùng với tên biến ghi bên cạnh.

3.10 Phương thức có nhiều tham số

Có một lý do thường gây ra lỗi khi lập trình: đó chính là cú pháp để miêu tả và kích hoạt phương thức gồm nhiều tham số. Trước hết, hãy nhớ rằng bạn phải khai báo kiểu của từng tham số. Chẳng hạn

```
public static void printTime(int hour, int minute) {

  System.out.print(hour);

  System.out.print(":");
```

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

Rất dễ bị lôi cuốn theo cách viết int hour, minute, nhưng cách này chỉ đúng với việc khai báo biến, chứ không phải với danh sách tham số.
Một lý do khác gây nhầm lẫn là bạn không cần phải khai báo kiểu của đối số. Viết như dưới đây là sai!

```
    int hour = 11;
    int minute = 59;
    printTime(int hour, int minute); // SAI!
```

Trong trường hợp này, Java có thể tự biết kiểu của hour và minute khi nhìn vào đoạn khai báo của chúng. Ta không cần phải kèm thêm kiểu của biến khi truyền chúng làm đối số. Cú pháp đúng phải là printTime(hour, minute).

3.11 Các phương thức trả lại kết quả

Một số phương thức ta đang dùng, như các phương thức của lớp Math, đều trả lại kết quả. Những phương thức khác, như println và newLine, đều thực hiện một thao tác nhưng không trả lại kết quả nào. Điều này nảy sinh một số câu hỏi sau:
•Điều gì sẽ xảy ra nếu nếu bạn kích hoạt một phương thức mà không làm gì với kết quả (nghĩa là bạn không gán nó vào một biến hay không dùng kết quả này làm bộ phận trong một biểu thức lớn hơn)?
•Điều gì sẽ xảy ra nếu bạn dùng một phương thức print như một phần của biểu thức lớn hơn, chẳng hạn System.out.println("boo!") + 7?
•Ta có thể viết những phương thức để trả lại giá trị không, hay chỉ loanh quanh với những phương thức kiểu như newLine và printTwice?
Lời giải đáp đối với câu hỏi thứ ba là "Có, bạn có thể viết những phương thức để trả lại giá trị," mà ta sẽ thấy cách làm sau một vài chương nữa. Tôi sẽ để cho bạn tự trả lời hai câu hỏi còn lại bằng cách thực hành trực tiếp. Thật ra, bất kì lúc nào bạn đặt ra câu hỏi về sự hợp lệ hay không hợp lệ của thao tác trong Java, thì một cách hay để tìm hiểu là đi hỏi trình biên dịch.

3.12 Thuật ngữ

khởi tạo:

Câu lệnh nhằm khai báo một biến đồng thời gán giá trị cho nó.

dấu phẩy động:

Một kiểu của biến (hoặc giá trị) có thể chứa cả số có phần thập phân lẫn số nguyên. Kiểu dấu phẩy động mà ta sẽ dùng là double.

lớp:

Một tập hợp được đặt tên, có chứa các phương thức. Đến giờ ta đã dùng lớp Math và lớp System, và cũng viết được các lớp có tên Hello và NewLine.

phương thức:

Một loạt những câu lệnh nhằm thực hiện một chức năng có ích. Phương thức được đặt tên. Nó có thể nhận hoặc không nhận tham số, đồng thời có thể trả lại hoặc không trả một giá trị.

tham số:

Một đơn vị thông tin mà phương thức yêu cầu trước khi nó có thể được thực hiện. Tham số là các biến:

chúng chứa những giá trị và có kiểu riêng.

đối số:

Giá trị mà bạn cung cấp khi bạn kích hoạt một phương thức. Giá trị này phải có cùng kiểu với tham số tương ứng.

khung:

Một cấu trúc (biểu diễn bởi khối chữ nhật màu xám trong biểu đồ ngăn xếp) có chứa các tham số và biến của một phương thức.

kích hoạt:

Làm cho phương thức được thực thi.

3.13 Bài tập

BÀI TẬP 1

Hãy vẽ một khung ngăn xếp để biểu diễn trạng thái chương trình ở Mục 3.10 khi main kích hoạt printTime với các đối số 11 và 59.

BÀI TẬP 2

Mục đích của bài tập này là luyện đọc mã lệnh để đảm bảo rằng bạn hiểu được luồng thực thi của chương trình gồm nhiều phương thức khác nhau.

1. Kết quả của chương trình sau là gì? Hãy nói chính xác vị trí các dấu trống và các chỗ xuống dòng. GỢI Ý: Bắt đầu bằng việc diễn tả bằng lời xem ping và baffle làm những gì khi chúng được kích hoạt.
2. Hãy vẽ một biểu đồ ngăn xếp biểu diễn trạng thái của chương trình khi ping được kích hoạt lần đầu.

```
public static void zoop() {
  baffle();
  System.out.print("You wugga ");
  baffle();
}
public static void main(String[] args) {
  System.out.print("No, I ");
  zoop();
  System.out.print("I ");
  baffle();
}
public static void baffle() {
  System.out.print("wug");
  ping();
}
public static void ping() {
  System.out.println(".");
}
```

BÀI TẬP 3

Mục đích của bài tập này là đảm bảo hiểu được cách viết và cách kích hoạt phương thức nhận tham số.

1. Hãy viết dòng đầu tiên của một phương thức có tên zool nhận vào ba tham số: một int và hai String.
2. Hãy viết một dòng lệnh zool, truyền làm tham số các giá trị sau: 11, tên của con thú cưng lần đầu bạn nuôi, và tên của dãy phố mà bạn sống thời thơ ấu.

BÀI TẬP 4

Mục đích của bài tập này là lấy đoạn mã từ một bài tập trước rồi bao gói nó vào trong một phương thức có nhận tham số. Bạn nên tìm một lời giải hoàn chỉnh cho Bài tập 2 để bắt đầu.

1. Hãy viết một phương thức có tên printAmerican để nhận ngày, tháng, năm làm các tham số rồi in chúng ra dưới dạng quy định của Mỹ.
2. Kiểm tra phương thức của bạn bằng cách kích hoạt nó từ main rồi truyền các đối số phù hợp. Kết quả phải trông giống như sau (chỉ trừ số ngày có thể khác đi):

```
Saturday, July 16, 2011
```

3. Một khi bạn đã gỡ lỗi xong cho printAmerican, hãy viết một phương thức khác có tên printEuropean để in ra ngày tháng theo quy chuẩn châu Âu.

Chương 4: Câu lệnh điều kiện và đệ quy Java

Trở về Mục lục cuốn sách

4.1 Toán tử chia dư

Toán tử chia dư tính với các số nguyên (cùng các biểu thức số nguyên) và cho kết quả là *phần dư* của phép chia số thứ nhất cho số thứ hai. Trong Java, toán tử chia dư có kí hiệu là dấu phần trăm, `%`. Cú pháp cũng giống như các toán tử khác:

```
int quotient = 7 / 3;

int remainder = 7 % 3;
```

Với toán tử thứ nhất, chia nguyên, kết quả là 2. Với toán tử thứ hai ta được kết quả bằng 1. Như vậy 7 chia cho 3 bằng 2 dư 1.

Toán tử số dư bất ngờ trở nên có ích. Chẳng hạn, bạn có thể kiểm tra xem một số có chia hết cho số khác không: nếu `x % y` bằng không thì `x` chia hết cho `y`.

Hơn nữa, bạn còn có thể lọc ra những chữ số cuối cùng bên phải từ số ban đầu. Chẳng hạn, `x % 10` cho ta số hàng đơn vị của `x` (trong hệ thập phân). Tương tự, `x % 100` cho ta hai chữ số hàng chục và đơn vị.

4.2 Thực hiện lệnh theo điều kiện

Để viết được những chương trình hữu ích, chúng ta thường luôn phải kiểm tra những điều kiện và thay đổi biểu hiện tương ứng của chương trình. Các **câu lệnh điều kiện** cung cấp cho ta khả năng này.
Dạng đơn giản nhất là lệnh `if`:

```
if (x > 0) {

  System.out.println("x la so duong");

}
```

Biểu thức ở trong cặp ngoặc tròn được gọi là điều kiện. Nếu nó được thoả mãn thì đoạn lệnh bên trong cặp ngoặc nhọn được thực thi. Nếu không, sẽ chẳng có điều gì xảy ra.

Điều kiện có thể chứa bất kì toán tử so sánh nào, vốn đôi khi còn được gọi là **toán tử quan hệ**:

```
x == y // x bằng y

x != y // x không bằng y

x > y // x is lớn hơn y

x < y // x nhỏ hơn y

x >= y // x lớn hơn hoặc bằng y

x <= y // x nhỏ hơn hoặc bằng y
```

Mặc dù có thể bạn đã quen thuộc với những phép toán này, cú pháp dùng trong Java vẫn hơi khác những biểu thức như =, ≠ và ≤. Một lỗi thường mắc phải là dùng một dấu = thay vì hai ==. Hãy nhớ rằng = là toán tử gán, còn == là toán tử so sánh. Ngoài ra không có toán tử nào được viết là `=<` hoặc `=>`.
Hai vế trong một biểu thức điều kiện phải có cùng kiểu dữ liệu. Bạn chỉ được phép so sánh int với ints hoặc double với double.
Hai toán tử == và != cũng làm việc với các chuỗi kí tự, nhưng cách hoạt động của chúng không giống như bạn đã dự kiến. Còn tất cả những toán tử quan hệ khác thì không có tác dụng gì đối với chuỗi. Ta sẽ

xem cách so sánh chuỗi ở Mục 8.10.

4.3 Thực hiện chọn lựa

Dạng thứ hai của thực hiện teho điều kiện là thực hiện lệnh theo lựa chọn, trong đó có hai khả năng và điều kiện được đặt ra để căn cứ vào đó mà lựa chọn thực hiện một trong hai. Cú pháp có dạng như sau:

```
if (x%2 == 0) {
  System.out.println("x la so chan");
} else {
  System.out.println("x la so le");
}
```

Nếu phần dư của phép chia `x` cho 2 là 0, thì chúng ta biết rằng `x` là số chẵn, và chương trình sẽ hiển thị thông báo điều này. Nếu điều kiện không được thoả mã thì lệnh thứ hai sẽ được thực hiện. Vì điều kiện hoặc là được thoả mãn, hoặc không; nên luôn chỉ có một trong hai phương án được thực hiện.

Nhân tiện nói thêm, nếu bạn có ý định thường xuyên kiểm tra tính chẵn lẻ, có thể bạn sẽ muốn "gói" đoạn mã lệnh này vào trong một phương thức, như sau:

```
public static void printParity(int x) {
  if (x%2 == 0) {
    System.out.println("x la so chan");
  } else {
    System.out.println("x la so le");
  }
}
```

Bây giờ bạn có một phương thức tên là printParity để in ra thông báo thích hợp cho mỗi số nguyên bạn cung cấp cho nó. Trong main bạn sẽ kích hoạt phương thức này như sau:

```
printParity(17);
```

Hãy luôn nhớ rằng khi bạn *kích hoạt* một phương thức, thì không nhất thiết phải khai báo các kiểu của đối số được cung cấp. Java có thể hình dung ra kiểu dữ liệu là gì. Bạn phải kiềm chế để tránh viết những lệnh kiểu như:

```
int number = 17;
printParity(int number); // SAI!!!
```

4.4 Các điều kiện xâu chuỗi

Đôi khi bạn cần phải kiểm tra một số các điều kiện có liên quan và chọn trong một số những hành động. Một cách thực hiện việc này là **xâu chuỗi** một loạt các if và else:

```
if (x > 0) {
  System.out.println("x la so duong");
} else if (x < 0) {
  System.out.println("x la so am");
} else {
  System.out.println("x bang khong");
```

```
    }
```

Việc xâu chuỗi như vậy có thể dài tùy ý, mặc dù chúng có thể khó đọc nếu đi quá đà. Một cách làm để dễ đọc hơn là sử dụng quy tắc thụt đầu dòng tiêu chuẩn, như đã trình bày trong các ví dụ trên. nếu bạn giữ cho các câu lệnh và các ngoặc nhọn được thẳng hàng với nhua thì ít có khả năng gây lỗi cú pháp hơn, và nếu có thì cũng dễ tìm thấy hơn.

4.5 Các điều kiện lồng ghép

Ngoài việc xâu chuỗi, bạn còn có thể lồng ghép một điều kiện bên trong điều kiện khác. Ta có thể viết lại ví dụ trên như sau:

```
    if (x == 0) {
      System.out.println("x bang khong");
    } else {
      if (x > 0) {
        System.out.println("x la so duong");
      } else {
        System.out.println("x la so am");
      }
    }
```

Bây giờ thì câu lệnh điều kiện bên ngoài có hai nhánh. Nhánh thứ nhất chỉ chứa một lệnh print, nhánh thứ hai lại chứa một câu lệnh điều kiện khác, mà bản thân nó lại có hai nhánh. Hai nhánh này đều chứa những câu lệnh print đơn giản, mặc dù dĩ nhiên chúng có thể là những câu lệnh điều kiện khác.
Tuy cách viết thụt vào trong làm cho cấu trúc rõ ý, nhưng các lệnh điều kiện lồng ghép trở nên rất khó để người đọc nhanh. Ta nên cố gắng tránh dùng chúng.

Mặt khác, dạng **cấu trúc lồng ghép** này cũng thường thấy, và sau này ta còn gặp chúng, do vậy bạn cũng làm quen với nó.

4.6 Câu lệnh return

Câu lệnh return cho phép bạn kết thúc việc thực thi của một phương thức trước khi đến cuối phương thức đó. Một lí do dùng câu lệnh này là nếu bạn phát hiện ra điều kiện gây lỗi:

```
  public static void printLogarithm(double x) {
    if (x <= 0.0) {
      System.out.println("Yêu cau nhap vao so duong.");
      return;
    }
    double result = Math.log(x);
    System.out.println("Gia tri log cua x bang " + result);
  }
```

Mã lệnh này định nghĩa một phương thức có tên printLogarithm; nó nhận tham số là một double có tên x. Phương thức này kiểm tra xem liệu x có nhỏ hơn hoặc bằng 0 hay không, và trong trường hợp

như vậy thì in ra một thông báo lỗi rồi dùng return để thoát khỏi phương thức. Luồng thực thi sẽ lập tức trở lại chỗ gọi phương thức đó và những dòng còn lại của phương thức sẽ không được thực hiện. Tôi đã dùng một giá trị dấu phẩy động ở bên vế phải của điều kiện vì vế trái biểu thức này là một biến phẩy động.

4.7 Chuyển đổi kiểu

Bạn có thể tự hỏi rằng làm sao chương trình của ta có thể êm xuôi với biểu thức kiểu như "Gia tri log cua x bang " + result, bởi một toán hạng là String còn toán hạng kia là double. Truong trường hợp này Java đã thông minh để thay ta chuyển giá trị double thành String trước khi thực hiện việc ghép chuỗi. Mỗi khi bạn thử "cộng" hai biểu thức, mà một trong số đó là String, Java sẽ chuyển đổi cái còn lại thành String rồi mới thực hiện ghép chuỗi. Bạn nghĩ điều gì sẽ xảy ra nếu thực hiện phép cộng giữa một số nguyên với một giá trị phẩy động?

4.8 Đệ quy

Ở chương trước tôi đã nói rằng việc một phương thức kích hoạt phương thức khác là hợp lệ, và a đã xét vài ví dụ. Tôi chưa đề cập rằng một phương thức kích hoạt chính nó cũng hợp lệ. Mặc dù bề ngoài thì có thể điều này không rõ hay dở ra sao, nhưng thực ra đó chính là một trong những đặc điểm hay nhất trong lập trình.

Chẳng hạn, hãy xét phương thức sau:

```
public static void countdown(int n) {
  if (n == 0) {
    System.out.println("Bum!");
  } else {
    System.out.println(n);
    countdown(n-1);
  }
}
```

Phương thức có tên là countdown và nó nhận tham số là một số nguyên. Nếu tham số bằng 0 hoặc âm, chương trình sẽ in ra chữ, "Bùm!" Còn nếu không, nó sẽ in ra giá trị tham số và sau đó kích hoạt một phương thức có tên `countdown`—nghĩa là chính nó—nhưng truyền vào đối số `n-1`.

Điều gì sẽ xảy ra khi ta kích hoạt một phương thức kiểu như thế này?

```
countdown(3);
```

Việc thực hiện `countdown` bắt đầu với `n=3`, và do `n` lớn hơn 0, nó đưa ra giá trị 3, và rồi gọi chính nó...
Việc thực hiện `countdown` bắt đầu với `n=2`, và do `n` lớn hơn 0, nó đưa ra giá trị 2, và rồi gọi chính nó...
Việc thực hiện `countdown` bắt đầu với `n=1`, và do `n` lớn hơn 0, nó đưa ra giá trị 1, và rồi gọi chính nó...
Việc thực hiện `countdown` bắt đầu với `n=0`, và do `n` không còn lớn hơn 0, nó đưa ra dòng chữ "Bùm!" và rồi quay về.
Phương thức `countdown` ứng với `n=1` quay về.
Phương thức `countdown` ứng với `n=2` quay về.

Phương thức `countdown` ứng với `n=3` quay về.

Và rồi bạn trở về với `main`. Như vậy, toàn bộ kết quả đầu ra như sau:

```
3

2

1

Bum!
```

Ví dụ thứ hai là hãy xem lại các phương thức newLine và threeLine.

```
  public static void newLine() {
    System.out.println("");
  }
  public static void threeLine() {
    newLine(); newLine(); newLine();
  }
```

Mặc dù cách này có tác dụng, nhưng sẽ không giúp ích được nhiều trong trường hợp ta cần in 2, hoặc 106 dòng mới. Một cách làm hay hơn là

```
  public static void nLines(int n) {
    if (n > 0) {
      System.out.println("");
      nLines(n-1);
    }
  }
```

Chương trình này tương tự như countdown; khi `n` còn lớn hơn 0, nó sẽ in ra một dòng mới và sau đó sẽ kích hoạt chính nó để in thêm *n* - 1 dòng mới nữa. Như vậy số dòng kết quả sẽ là `1 + (n-1)`, tức là bằng `n`.

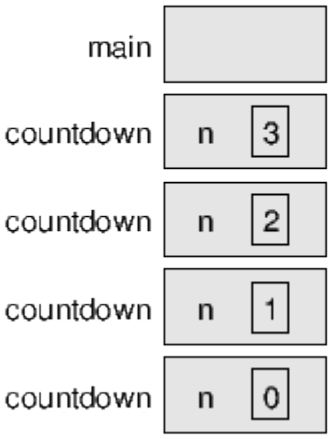
Khi một phương thức kích hoạt chính nó, điều này gọi là **đệ quy**, và những phương thức đó có **tính đệ quy**.

4.9 Biểu đồ ngăn xếp cho các phương thức đệ quy

Trong chương trước, chúng ta đã dùng một biểu đồ ngăn xếp để biểu thị trạng thái của một chương trình trong quá trình phương thức được kích hoạt. Loại biểu đồ này cũng tiện dùng cho việc diễn giải một phương thức đệ quy.

Hãy nhớ rằng mỗi khi phương thức được kích hoạt, Java tạo ra một "khung" mới trong đó có chứa phiên bản mới của các biến cục bộ và tham số trong phương thức.

Hình vẽ này minh hoạ một sơ đồ ngăn xếp cho phương thức `countdown` khi gọi với `n = 3`:



Có một khung dành cho `main` và bốn khung `countdown`, mỗi khung có một giá trị riêng cho tham biến `n`. Đáy của ngăn xếp, countdown với `n=0`, được gọi là **trường hợp cơ sở**. Nó không thực hiện lời gọi đệ quy, do đó không có thêm khung countdown nào.

Khung chứa main thì rỗng vì main không chứa bất kì tham số hay biến nào.

4.10 Thuật ngữ

toán tử module:

Toán tử dùng với hai số nguyên và trả lại phần dư trong phép chia giữa hai số đó. Trong Java, toán tử này được kí hiệu bởi dấu phần trăm (%).

lệnh điều kiện:

Một khối lệnh có thể được thực thi hay không tùy theo một điều kiện nào đó.

xâu chuỗi:

Cách nối nhiều lệnh điều kiện thành dãy liên tục.

lồng ghép:

Cách đặt một lệnh điều kiện này vò trong một hoặc cả hai nhánh của một lệnh điều kiện khác.

định kiểu:

Một toán tử giúp chuyển đổi từ kiểu dữ liệu này sang kiểu khác. Trong Java nó có dạng tên một kiểu dữ liệu viết giữa cặp ngoặc tròn, như (int).

đệ quy:

Quá trình kích hoạt chính phương thức đang được thực thi.

trường hợp cơ sở:

Một điều kiện để cho phương thức đệ quy không kích hoạt đệ quy nữa.

4.11 Bài tập

Bài tập 1 Hãy vẽ một biểu đồ ngăn xếp biểu diễn trạng thái chương trình ở Mục 4.8 sau khi main kích hoạt nLines với tham số n=4, ngay trước khi lần kích hoạt cuối cùng của nLines trả về.

Bài tập 2 Bài tập này ôn lại luồng thực thi, bằng một chương trình với nhiều phương thức. Hãy đọc mã lệnh dưới đây rồi trả lời những câu hỏi đi theo.

```
public class Buzz {


  public static void baffle(String blimp) {
```

```
    System.out.println(blimp);
    zippo("ping", -5);
  }

  public static void zippo(String quince, int flag) {
    if (flag < 0) {
      System.out.println(quince + " zoop");
    } else {
      System.out.println("ik");
      baffle(quince);
      System.out.println("boo-wa-ha-ha");
    }
  }

  public static void main(String[] args) {
    zippo("rattle", 13);
  }
}
```

1. Hãy viết số 1 kế bên câu lệnh đầu tiên được thực thi của chương trình này. Hãy cẩn thận để tách biệt những thứ thuộc về câu lệnh với những thứ khác.
2. Viết số 2 kế bệnh câu lệnh thứ hai, và cứ như vậy đến cuối chương trình. Nếu một câu lệnh được thực hiện nhiều lần thì cuối cùng ta có thể sẽ thấy kết quả in ra chứa nhiều con số ghi bên cạnh nó.
3. Giá trị của tham số blimp khi baffle bị kích hoạt là gì?
4. Kết quả của chương trình này là gì?

Bài tập 3 Câu đầu trong lời bài hát "99 Bottles of Beer" là:

> 99 bottles of beer on the wall, 99 bottles of beer, ya' take one down, ya' pass it around, 98 bottles of beer on the wall.

Những câu tiếp theo cũng như vậy chỉ khác là số chai bia cứ giảm dần đi một, đến câu cuối cùng:

> No bottles of beer on the wall, no bottles of beer, ya' can't take one down, ya' can't pass it around, 'cause there are no more bottles of beer on the wall!

Và sau đó thì cuối cùng bài hát cũng kết thúc.

Hãy viết chương trình in ra toàn bộ lời bài hát "99 Bottles of Beer." Chương trình này cần có một phương thức đệ quy để giải quyết phần khó khăn, nhưng có thể bạn còn muốn viết thêm những phương thức phụ trợ việc phân chia những tính năng cơ bản của chương trình.

Trong quá trình phát triển mã lệnh, hãy thử chạy với một số ít các câu hát, như "3 Bottles of Beer."

Mục đích của bài tập này là tiếp nhận bài toán rồi chia nhỏ nó thành những bài toán con, và giải bài

toán con bằng cách viết những phương thức đơn giản.

Bài tập 4 Kết quả của chương trình sau đây là gì?

```
public class Narf {


  public static void zoop(String fred, int bob) {
    System.out.println(fred);
    if (bob == 5) {
      ping("not ");
    } else {
      System.out.println("!");
    }
  }


  public static void main(String[] args) {
    int bizz = 5;
    int buzz = 2;
    zoop("just for", bizz);
    clink(2*buzz);
  }


  public static void clink(int fork) {
    System.out.print("It's ");
    zoop("breakfast ", fork) ;
  }


  public static void ping(String strangStrung) {
    System.out.println("any " + strangStrung + "more ");
  }
}
```

Bài tập 5 Định lý cuối cùng của Fermat phát biểu rằng không có các số nguyên a, b, và c nào thoả mãn $a_n + b_n = c_n$
trừ trường hợp $n = 2$.Viết một phương thức có tên là `check_fermat` nhận vào bốn tham số—`a`, `b`, `c` và `n`—rồi kiểm tra xem có thoả mãn định lý Fermat không. Nếu n lớn hơn 2 và hoá ra $a_n + b_n = c_n$, thì chương trình sẽ in ra "Trời, Fermat đã lầm!" Còn nếu không thì chương trình sẽ in ra, "Không, vẫn không đúng".

Bạn cần phải giả sử rằng có một phương thức tên là raiseToPow ; phương thức này nhận đối số là hai số nguyên rồi nâng đối số thứ nhất lên lũy thừa số thứ hai. Chẳng hạn:

```
    int x = raiseToPow(2, 3);
```

sẽ gán giá trị 8 cho x, bởi $2_3 = 8$.

Chương 5: Grid World, phần 1

Trở về Mục lục cuốn sách

5.1 Khởi động

Bây giờ đã đến lúc ta bắt đầu làm Nghiên cứu cụ thể về kì thi Khoa học máy tính AP; nghiên cứu này xoay quanh một chương trình có tên GridWorld. Đầu tiên, hãy cài đặt GridWorld; bạn có thể tải chương trình này về từ Hội đồng tuyển sinh Hoa

Kì: http://www.collegeboard.com/student/testing/ap/compsci_a/case.html.

Khi giải nén mã nguồn này, bạn sẽ thu được một thư mục mang tên GridWorldCode trong đó chứa projects/firstProject, và bản thân thư mục này lại chứa BugRunner.java.
Hãy sao chép tập tin BugRunner.java vào một thư mục khác rồi nhập nó từ môi trường phát triển mà bạn đang dùng. Bạn có thể tham khảo hướng
dẫn: http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/ap07_gridworld_installation_guide.pdf.
Một khi chạy BugRunner.java, bạn hãy tải Bản hướng dẫn thực hành GridWorld
từ http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/ap07_gridworld_studmanual_appends_v3.pdf.
Bản hướng dẫn thực hành này có dùng những thuật ngữ mà tôi chưa trình bày. Bởi vậy để bạn quen được, sau đây là một danh sách giới thiệu tóm tắt:

- Các thành phần của GridWorld, bao gồm Bugs, Rocks và bản thân Grid đều là những **đối tượng**.
- **Constructor** là một phương thức đặc biệt để tạo nên những đối tượng mới.
- **Lớp** là một tập hợp các đối tượng; mỗi đối tượng đều thuộc một lớp nhất định.
- Đối tượng còn được gọi là **thực thể**, vì nó thuộc về một lớp.
- **Thuộc tính** là một đơn vị thông tin về một đối tượng, chẳng hạn màu sắc hay tọa độ (vị trí) của đối tượng đó.
- **Phương thức truy cập** là một phương thức nhằm trả lại thuộc tính của một đối tượng.
- **Phương thức sửa đổi** nhằm thay đổi thuộc tính của một đối tượng.

Bây giờ bạn đã có thể đọc được Phần 1 của cuốn Hướng dẫn thực hành và làm các bài tập.

5.2 BugRunner

BugRunner.java chứa mã lệnh sau:

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;
public class BugRunner {
  public static void main(String[] args) {
    ActorWorld world = new ActorWorld();
    world.add(new Bug());
    world.add(new Rock());
    world.show();
```

```
    }
}
```

Ba dòng đầu tiên là các câu lệnh import; chúng liệt kê các lớp trong GridWorld được dùng đến ở chương trình này. Bạn có thể tìm tài liệu cho những lớp này tại http://www.greenteapress.com/thinkapjava/javadoc/gridworld/.

Cũng như những chương trình khác ta đã gặp, BugRunner định nghĩa lớp có nhiệm vụ cung cấp phương thức main. Dòng đầu tiên trong main tạo ra một đối tượng ActorWorld. Ở đây, new là từ khóa Java để tạo nên đối tượng mới.

Hai dòng kết tiếp tạo ra một Bug (con bọ) và một Rock (tảng đá), rồi bổ sung chúng vào world (môi trường). Dòng cuối cùng hiển thị môi trường lên màn hình.

Hãy mở tập tin BugRunner.java để chỉnh sửa và thay dòng này:

```
    world.add(new Bug());
```

bằng các dòng này:

```
    Bug redBug = new Bug();

    world.add(redBug);
```

Dòng đầu tiên gán Bug cho một biến có tên redBug; ta có thể dùng redBug để kích hoạt những phương thức của Bug. hãy thử lệnh này:

```
    System.out.println(redBug.getLocation());
```

Chú ý: Nếu bạn chạy lệnh này trước khi bổ sung Bug vào world, thì kết quả sẽ là null, bởi đối tượng Bug này chưa có một vị trí cụ thể.

Hãy kích hoạt những phương thức truy cập khác rồi in ra các thuộc tính của con bọ vừa tạo ra. Kích hoạt các phương thức canMove, move và turn đồng thời đảm bảo rằng bạn nắm được tác dụng của chúng.

5.3 Bài tập

Bài tập 1

1. Hãy viết một phương thức có tên moveBug để nhận vào tham số là con bọ rồi kích hoạt move. Kiểm tra phương thức vừa viết ra bằng cách gọi nó từ main.
2. Sửa chữa moveBug để nó kích hoạt canMove rồi di chuyển con bọ chỉ khi nó chuyển động được.
3. Sửa chữa moveBug để nó nhận một tham số là số nguyên n, rồi di chuyển con bọ n lần (nếu có thể).
4. Sửa chữa moveBug sao cho nếu con bọ không chuyển động được thì phương thức này sẽ kích hoạt turn.

Bài tập 2

1. Lớp Math cung cấp một phương thức mang tên random để trả lại một số phẩy động giữa 0.0 và 1.0 (không bao gồm 1.0).
2. Hãy viết một phương thức mang tên randomBug để nhận tham số là một Bug rồi đặt hướng của con bọ này là một trong những giá trị 0, 90, 180 hoặc 270 theo xác suất bằng nhau, rồi cho con bọ chuyển động nếu nó có thể.
3. Sửa chữa randomBug để nhận vào số nguyên n rồi thực hiện lặp lại n lần thao tác trên.Kết quả sẽ là một quá trình "bước ngẫu nhiên", mà bạn có thể xem thêm ở http://en.wikipedia.org/wiki/Random_walk.
4. Để quan sát quá trình bước ngẫu nhiên dài hơn, bạn có thể cho ActorWorld một không gian rộng hơn.

Ở trên đầu file BugRunner.java, hãy bổ sung câu lệnh import sau:

```
import info.gridworld.grid.UnboundedGrid;
```

Bây giờ, hãy thay dòng lệnh tạo nên ActorWorld với dòng lệnh sau:

```
    ActorWorld world = new ActorWorld(new UnboundedGrid());
```

Bạn có thể trình diễn bước ngẫu nhiên với vài nghìn bước di chuyển (có thể phải kéo thanh trượt để tìm con bọ).

Bài tập 3 GridWorld dùng các đối tượng Color, vốn được định nghĩa trong một thư viện Java. Bạn có thể đọc tài liệu ở http://download.oracle.com/javase/6/docs/api/java/awt/Color.html.Để tạo nên nhiều con bọ với các màu sắc khác nhau, bạn phải nhập Color:

```
import java.awt.Color;
```

Khi đó bạn sẽ truy cập được các màu đã định sẵn, như Color.blue, hay một màu mới như sau:

```
    Color purple = new Color(148, 0, 211);
```

Hãy tạo ra một vài con bọ với màu sắc khác nhau. Tiếp theo, hãy viết một phương thức có tên colorBug để nhận tham số là một con bọ, đọc vào tọa độ của nó, rồi đặt màu.
Đối tượng Location mà bạn đã lấy từ getLocation có chứa những phương thức mang tên getRow và getCol vốn trả lại những số nguyên. VÌ vậy bạn có thể lấy tọa độ x của con bọ như sau:

```
    int x = bug.getLocation().getCol();
```

Hãy viết một phương thức có tên makeBugs để nhận vào một ActorWorld và một số nguyên n rồi tạo nên n con bọ có màu sắc tùy thuộc theo tọa độ của chúng. Hãy dùng số thứ tự dòng để điều khiển mức sắc đỏ và thứ tự cột để điều khiển mức sắc lam.

Chương 6: Phương thức trả lại giá trị

Trở về Mục lục cuốn sách

6.1 Những giá trị được trả lại

Một số phương thức mà ta đã dùng, như các hàm toán học, có trả lại kết quả. Nghĩa là, hiệu ứng từ việc kích hoạt phương thức là tạo ra một giá trị mới mà ta thường gán nó cho một biến hoặc dùng như một phần hợp nên một biểu thức lớn hơn. Chẳng hạn:

```
double e = Math.exp(1.0);
double height = radius * Math.sin(angle);
```

Nhưng cho đến giờ tất cả những phương thức mà tự tay viết đều là phương thức **rỗng**; theo nghĩa những phương thức này không trả lại giá trị nào. Khi bạn kích hoạt một phương thức rống, nó thường chỉ tự được đặt trên một dòng mà không có lệnh gán nào cả:

```
countdown(3);
nLines(3);
```

Trong chương này ta viết những phương thức trả lại thông tin, mà tôi gọi là phương thức **trả lại giá trị**. Ví dụ đầu tiên là area, một phương thức nhận vào tham số là một double, rồi trả lại diện tích của một hình tròn với bán kính cho trước:

```
public static double area(double radius) {
  double area = Math.PI * radius * radius;
  return area;
}
```

Điều đầu tiên mà ta nhận thấy là đoạn đầu của định nghĩa phương thức đã khác đi. Thay vì public static void, vốn để chỉ một phương thức rỗng, ta thấy public static double, có nghĩa là giá trị trả về từ phương thức này là một double. Tôi vẫn chưa giải thích ý nghĩa của public static, song bạn hãy kiên nhẫn.
Dòng cuối là một dạng mới của câu lệnh return trong đó bao gồm một giá trị trả lại. Câu lệnh này có nghĩa là "từ phương thức này hãy lập tức trở về và dùng biểu thức kèm theo đây làm giá trị trả lại." Biểu thức mà bạn đặt ra có thể phức tạp tùy ý, vì vậy ta có thể viết phương thức sau một cách gọn hơn:

```
public static double area(double radius) {
  return Math.PI * radius * radius;
}
```

Mặt khác, những **biến tạm thời** như area thường giúp cho việc gỡ lỗi được dễ dàng hơn. Trong cải hai trường hợp, kiểu của biểu thức trong lệnh return phải khớp với kiểu của phương thức. Nói cách khác, khi bạn khai báo rằng kiểu trả lại là double, bạn đã cam kết rằng phương thức này cuối cùng sẽ tạo ra một double. Nếu bạn thử return mà không kèm theo biểu thức nào, hoặc kèm theo biểu thức nhưng sai kiểu, thì trình biên dịch sẽ rầy la bạn.
Đôi khi cần phải có nhiều lệnh return, mỗi lệnh đặt ở một nhánh của lệnh điều kiện:

```
public static double absoluteValue(double x) {
  if (x < 0) {
```

```
    return -x;
  } else {
    return x;
  }
}
```

Vì những lệnh return này ở cấu trúc điều kiện lựa chọn, cho nên chỉ có một lệnh được thực thi. Dù rằng hoàn toàn hợp lệ nếu bạn có nhiều lệnh return trong cùng một phương thức, song bạn cần ghi nhớ rằng ngay khi một lệnh return được thực hiện, phương thức sẽ kết thúc mà không thực hiện bất cứ lệnh nào tiếp sau nó.

Mã lệnh xuất hiện sau dòng lệnh `return`, hay nói chung, trong bất cứ chỗ nào khác của chương trình mà không nằm trong luồng thực hiện thì được gọi là **mã lệnh chết**. Một số trình biên dịch sẽ cảnh báo nếu có đoạn lệnh chết trong mã lệnh bạn viết nên.
Nếu bạn đặt lệnh return trong cấu trúc điều kiện, thì phải đảm bảo được rằng *mỗi luồng thực hiện khả dĩ* đều dẫn tới một lệnh `return`. Chẳng hạn:

```
public static double absoluteValue(double x) {
  if (x < 0) {
    return -x;
  } else if (x > 0) {
    return x;
  } // SAI!!
}
```

Chương trình này không hợp lệ vì nếu x bằng 0, thì cả hai điều kiện không có điều kiện nào được thoả mãn, và hàm sẽ kết thúc mà không gặp phải lệnh `return` nào. Trình biên dịch thường sẽ đưa ra thông báo kiểu như “return statement required in absoluteValue” (yêu cầu phải có lệnh return trong absoluteValue); lời thông báo này dễ gây nhầm lẫn vì trong đó bạn đã viết hai lệnh return rồi.

6.2 Phát triển chương trình

Lúc này bạn đã có thể nhìn vào toàn bộ phương thức Java rồi cho biết chúng có nhiệm vụ gì. Nhưng chưa chắc bạn đã biết cách viết nên chúng. Tôi sẽ đề xuất một phương pháp gọi là **phát triển tăng dần**.
Ở ví dụ này, giả dụ bạn cần tìm khoảng cách giữa hai điểm cho bởi các toạ độ $(x1, y1)$ và $(x2, y2)$. Theo định nghĩa thông thường, khoảng cách (distance) sẽ là:

$$\text{distance} = \sqrt{(x_2 - x_1)_2 + (y_2 - y_1)_2}$$

Bước đầu tiên là cân nhắc xem một hàm `distance` trong Java sẽ trông như thế nào. Nói cách khác, các số liệu đầu vào (tham số) và kết quả (giá trị trả lại) là gì?

Trong trường hợp này, số liệu đầu vào mô tả hai điểm; ta có thể biểu thị chúng bằng bốn số double, dù rằng sau này ta sẽ thấy Java có kiểu đối tượng Point mà ta có thể tận dụng. Giá trị cần trả về là khoảng cách, tức là sẽ thuộc kiểu double.

Ta đã có thể phác thảo ngay ra hàm như sau:

```
public static double distance (double x1, double y1, double x2, double y2) {
  return 0.0;
}
```

Câu lệnh return 0.0; đóng vai trò giữ chỗ cần thiết cho việc biên dịch chương trình. Đương nhiên, vào lúc này nó chưa phát huy tác dụng, song vẫn đáng để ta thử biên dịch nhằm phát hiện ra lỗi cú pháp, nếu có, trước khi viết thêm mã lệnh.
Để kiểm tra phương thức mới viết này, ta phải kích hoạt nó bằng các giá trị mẫu. Đâu đó ở trong main, tôi sẽ phải viết lệnh:

```
  double dist = distance(1.0, 2.0, 4.0, 6.0);
```

Sở dĩ tôi chọn các tham số này vì khoảng cách ngang sẽ là 3 và khoảng cách dọc là 4, theo đó thì kết quả sẽ bằng 5 (cạnh huyền của một tam giác có các cạnh là 3-4-5). Khi thử nghiệm một hàm, bạn nên biết trước kết quả đúng.

Một khi đã kiểm tra xong cú pháp của lời định nghĩa hàm, ta có thể bắt tay vào thêm mã lệnh vào phần thân. Sau mỗi lần thay đổi tăng dần, ta biên dịch lại và chạy chương trình. Nếu có lỗi ở bất kì bước thay đổi nào, ta sẽ biết ngay rằng phải nhìn vào đâu: chính là vào dòng lệnh mà ta vừa mới bổ sung.

Một bước làm hợp lí tiếp theo là tính các hiệu số $x_2 - x_1$ và $y_2 - y_1$. Tôi lưu trữ các giá trị trên vào những biến tạm thời có tên dx và dy.

```
public static double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  System.out.println("dx is " + dx);
  System.out.println("dy is " + dy);
  return 0.0;
}
```

Tôi đã bổ sung hai lệnh in vào sau đó để ta kiểm tra được những giá trị trung gian trước khi tiếp tục. Những giá trị này phải bằng 3.0 và 4.0.

Một khi đã viết xong phương thức rồi thì ta cần phải bỏ những lệnh in này đi. Các câu lệnh như vậy còn có tên là **dàn giáo** vì nó có ích cho việc xây dựng chương trình nhưng lại không phải là một phần trong sản phẩm cuối cùng.

Tiếp theo chúng ta tính các bình phương của dx và dy. Ta đã có thể dùng phương thức Math.pow, nhưng đem nhân từng số với chính nó sẽ đơn giản hơn.

```
public static double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
```

```
    System.out.println("dsquared is " + dsquared);
    return 0.0;
  }
```

Một lần nữa, tôi biên dịch rồi chạy chương trình ở giai đoạn này và kiểm tra giá trị trung gian (vốn phải bằng 25.0).

Sau cùng, ta có thể dùng Math.sqrt để tính rồi trả lại kết quả.

```
public static double distance (double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    double result = Math.sqrt(dsquared);
    return result;
  }
```

Từ main, ta có thể in và kiểm tra giá trị của kết quả.
Sau này khi đã có kinh nghiệm, bạn sẽ viết và gỡ lỗi nhiều dòng lệnh cùng lúc. Song dù sao đi nữa, việc phát triển tăng dần sẽ giúp bạn tiết kiệm nhiều thời gian. Các điểm cơ bản của quy trình này là:

- Bắt đầu với một chương trình chạy được và thêm vào những thay đổi nhỏ. Bất cứ lúc nào khi gặp lỗi, bạn sẽ phát hiện được ngay lỗi đó ở đâu.
- Dùng các biến tạm để lưu giữ các giá trị trung gian, từ đó bạn có thể hiển thị và kiểm tra chúng.
- Một khi chương trình đã hoạt động, bạn có thể dỡ bỏ các đoạn mã "dàn giáo", hoặc rút gọn nhiều câu lệnh về một biểu thức phức hợp, nếu việc này không làm cho chương trình trở nên khó đọc hơn.

6.3 Kết hợp phương thức

Một khi đã định nghĩa một phương thức mới, bạn có thể dùng nó như một phần của biểu thức lớn, và bạn cũng có thể thiết lập những phương thức mới từ các phương thức sẵn có. Chẳng hạn, nếu ai đó cho bạn hai điểm: một là tâm đường tròn và một điểm trên đường tròn đó, rồi yêu cầu bạn tính diện tích hình tròn thì bạn sẽ làm thế nào?

Giả sử như toạ độ của tâm điểm được lưu trong các biến `xc` và `yc`, toạ độ điểm trên đường tròn là `xp` và `yp`. Bước đầu tiên sẽ là tìm bán kính của đường tròn, vốn là khoảng cách giữa hai điểm đó. Thật may là ta đã có một phương thức, `distance`, để làm việc này:

```
    double radius = distance(xc, yc, xp, yp);
```

Bước tiếp theo là tìm diện tích của một đường tròn có bán kính đó, rồi trả lại kết quả.

```
    double area = area(radius); return area;
```

Kết hợp hai bước này vào trong cùng một phương thức, ta thu được:

```
public static double circleArea (double xc, double yc, double xp, double yp) {
    double radius = distance(xc, yc, xp, yp);
    double area = area(radius);
    return area;
```

```
}
```

Các biến tạm thời `radius` và `area` có ích cho việc phát triển và gỡ lỗi chương trình, nhưng một khi chương trình đã hoạt động tốt, ta có thể rút gọn nó lại bằng cách kết hợp các lệnh kích hoạt phương thức:

```
public static double circleArea (double xc, double yc, double xp, double yp) {
  return area(distance(xc, yc, xp, yp));
}
```

6.4 Quá tải toán tử

Có thể bạn đã nhận thấy rằng cả circleArea lẫn area đều thực hiện những tính năng tương tự—tìm diện tích hình tròn—nhưng nhận các tham số khác nhau. Với area, chúng ta phải cung cấp bán kính; còn với circleArea ta cung cấp hai điểm.

Nếu hai phương thức cùng làm một việc, lẽ tự nhiên là ta đặt chung một tên cho cả hai. Việc có nhiều phương thức cùng tên, vốn được gọi là **quá tải** (overloading), là điều hợp lệ trong Java *miễn sao các dạng phương thức phải nhận những tham số khác nhau*. Như vậy ta có thể đổi tên circleArea:

```
public static double area (double x1, double y1, double x2, double y2) {
  return area(distance(xc, yc, xp, yp));
}
```

Khi bạn kích hoạt một phương thức quá tải, Java sẽ biết được rằng bạn muốn dùng dạng phương thức nào, qua việc xem xét các đối số mà bạn cung cấp. Nếu bạn viết:

```
double x = area(3.0);
```

thì Java sẽ đi tìm một phương thức mang tên area mà nhận đối số là một double; do đó nó sẽ dùng dạng thứ nhất, tức là hiểu đối số như một bán kính. Còn nếu bạn viết:

```
double x = area(1.0, 2.0, 4.0, 6.0);
```

thì Java sẽ dùng dạng thứ hai của area. Và lưu ý rằng thực ra dạng area thứ hai đã kích hoạt dạng thứ nhất.

Nhiều phương thức Java được qua tải, nghĩa là có nhiều dạng trong đó chấp nhận số lượng hoặc kiểu tham số khác nhau. Chẳng hạn, có những dạng print và println chấp nhận một tham số thuộc kiểu bất kì. Trong lớp Math, có một dạng abs làm việc với double, đồng thời có một dạng dành cho int.

Mặc dù quá tải là một đặc điểm hữu ích, so bạn hãy cẩn thận khi dùng. Bạn có thể thật sự cảm thấy lú lẫn nếu cố gắng gỡ lỗi một dạng phương thức trong khi bạn không chủ ý kích hoạt nó, mà là một phương thức khác cùng tên!

Và điều này làm tôi nhớ đến một quy tắc then chốt trong gỡ lỗi: **hãy đảm bảo chắc rằng phiên bản chương trình bạn cần gỡ lỗi chính là phiên bản chương trình bạn đang chạy!**

Một ngày nào đó có thể bạn sẽ thấy mình đang loay hoay sửa đi sửa lại chương trình, và cứ thấy kết quả vẫn y nguyên như vậy khi chạy lại. Đây là một tín hiệu cảnh báo rằng hiện bạn không chạy phiên bản chương trình như đang nghĩ. Để kiểm tra lại, bạn hãy thử thêm một câu lệnh print (chẳng quan trọng là in thứ gì) và xem chương trình có biểu hiện tương ứng hay không.

6.5 Biểu thức logic

Hầu hết các toán tử mà ta đã gặp đều tạo ra kết quả có cùng kiểu với các toán hạng trong đó. Lấy ví dụ, toán tử + nhận hai số int rồi cũng tạo ra một số int, hoặc hai số double rồi tạo thành một double, v.v. Những ngoại lệ mà ta gặp, đó là các **toán tử quan hệ**, vốn để so sánh các int hoặc float rồi trả lại true hoặc false. true và false là những giá trị đặc biệt trong Java; hai giá trị này hợp nên một kiểu gọi là **boolean**. Bạn có thể nhớ lại rằng khi tôi định nghĩa một kiểu, tôi có nói rằng đó là một tập các giá trị. Đối với các số int, double hay chuỗi String, những tập hợp như vậy đều rất lớn. Song với boolean, tập hợp này chỉ chứa hai giá trị.

Các biểu thức boolean, hay biểu thức logic, cùng các biến cũng hoạt động giống như các biểu thức và biến thuộc kiểu khác:

```
boolean flag;
flag = true;
boolean testResult = false;
```

Ví dụ thứ nhất là một lời khai báo biến đơn giản; ví dụ thứ hai là một lệnh gán, còn ví dụ thứ ba là một lệnh khởi tạo.

Các giá trị true và false là những từ khóa trong Java, vì vậy chúng có thể xuất hiện với màu chữ khác tùy theo môi trường phát triển tích hợp mà bạn đang dùng.

Kết quả của một toán tử điều kiện là một giá trị boolean, bởi vậy bạn có thể lưu trữ kết quả của phép so sánh vào một biến:

```
boolean evenFlag = (n%2 == 0); // đúng nếu n chẵn
boolean positiveFlag = (x > 0); // đúng nếu x dương
```

rồi lại dùng nó làm bộ phận của một câu lệnh điều kiện:

```
if (evenFlag) {
  System.out.println("Khi tôi kiểm tra, n là số chẵn");
}
```

Một biến được dùng theo cách này có thể gọi là một biến **dấu hiệu** vì nó đánh dấu cho sự có mặt hoặc vắng mặt của một điều kiện nào đó.

6.6 Toán tử logic

Có ba **toán tử logic** trong Java: AND, OR và NOT, vốn được kí hiệu bởi ba dấu &&, || và !. Ý nghĩa của các toán tử này giống như nghĩa các từ tương ứng trong tiếng Anh. Chẳng hạn, x > 0 && x < 10 chỉ đúng khi `x` lớn hơn 0 *và* nhỏ hơn 10.

evenFlag || n%3 == 0 chỉ đúng khi *một trong hai* điều kiện là đúng; nghĩa là evenFlag đúng *hoặc* số `n` chia hết cho 3.

Sau cùng, toán tử `not` phủ định một biểu thức Boole. Do vậy !evenFlag là đúng nếu như evenFlag là sai —tức là nếu số đã cho là lẻ.

Toán tử logic có thể làm đơn giản những câu lệnh điều kiện lồng ghép. Chẳng hạn, bạn có thể viết lại mã lệnh dưới đây bằng một câu lệnh điều kiện đơn lẻ được không?

```
  if (x > 0) {
    if (x < 10) {
      System.out.println("x là số dương gồm 1 chữ số.");
    }
  }
```

6.7 Phương thức logic

Các phương thức có thể trả lại giá trị boolean cũng như các kiểu dữ liệu khác; và điều này thường thuận tiện cho việc đem những thao tác kiểm tra cất giấu vào trong phương thức. Chẳng hạn:

```
public static boolean isSingleDigit(int x) {
  if (x >= 0 && x < 10) {
    return true;
  } else {
    return false;
  }
}
```

Phương thức này có tên là isSingleDigit. Thường thì người ta hay đặt tên phương thức logic theo kiểu như những câu hỏi đúng/sai. Kiểu dữ liệu trả lại là boolean, như vậy mỗi câu lệnh return đều phải đưa ra một biểu thức boolean.

Bản thân đoạn mã lệnh rất rõ nghĩa, mặc dù nó dài hơn mức cần thiết. Hãy nhớ rằng biểu thức x >= 0 && x < 10 có kiểu boolean, bởi vậy không có gì sai khi ta trực tiếp trả lại nó đồng thời tránh được câu lệnh if:

```
public static boolean isSingleDigit(int x) {
  return (x >= 0 && x < 10);
}
```

Từ main bạn có thể kích hoạt phương thức này theo cách thông thường:

```
boolean bigFlag = !isSingleDigit(17);
System.out.println(isSingleDigit(2));
```

Dòng đầu tiên đặt bigFlag là true chỉ khi 17 *không phải* số có một chữ số. Dòng lệnh thứ hai in ra true bởi 2 là chỉ có một chữ số.

Cách dùng hay gặp nhất đối với phương thức boole là trong các câu lệnh điều kiện

```
if (isSingleDigit(x)) {
  System.out.println("x nhỏ");
} else {
  System.out.println("x lớn");
}
```

6.8 Nói thêm về đệ quy

Bây giờ khi đã biết phương thức trả lại giá trị, ta có được một ngôn ngữ lập trình **Turing đầy đủ**; theo nghĩa là chúng ta sẽ tính được mọi thứ có thể tính toán, trong đó "có thể tính toán" được định nghĩa

theo cách bất kì, miễn là hợp lý. Ý tưởng này được Alonzo Church và Alan Turing phát triển, bởi vậy nó còn mang tên luận án Church-Turing. Bạn có thể đọc thêm thông tin ởhttp://en.wikipedia.org/wiki/Turing_thesis.

Để cụ thể hoá tác dụng của những kiến thức lập trình mà bạn vừa được học, chúng ta hãy cùng lập một số hàm toán học theo cách đệ quy. Một định nghĩa đệ quy giống như việc định nghĩa vòng quanh; điểm tương đồng là trong phần định nghĩa lại có tham chiếu đến sự vật được định nghĩa. Nhưng cách định nghĩa vòng quanh thực sự thì không mấy có tác dụng:

đệ quy:
một tính từ để chỉ một phương thức mang tính đệ quy.
Bạn hẳn sẽ bực mình khi thấy một định nghĩa kiểu như vậy trong cuốn từ điển. Ngược lại, khi bạn xem định nghĩa về hàm **giai thừa** trong toán học, có thể bạn sẽ thấy:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

(Giai thừa thường được kí hiệu bởi dấu !, xin đừng nhầm với toán tử logic! với ý nghĩa NOT.) Định nghĩa này phát biểu rằng giai thừa của 0 là 1, và giai thừa của bất kì một giá trị nào khác, *n*, thì bằng *n* nhân với giai thừa của *n* - 1. Theo đó, 3! bằng 3 nhân với 2!, vốn lại bằng 2 nhân với 1!, vốn bằng 1 nhân với 0!. Gộp tất cả lại, ta có 3! bằng 3 nhân 2 nhân 1 nhân 1, tức là bằng 6.
Nếu bạn có thể phát biểu một định nghĩa có tính đệ quy cho một hàm nào đó thì bạn cũng có thể viết một phương thức Java để tính nó. Bước đầu tiên là xác định các tham số và kiểu dữ liệu của giá trị trả lại. Vì giai thừa được định nghĩa cho các số nguyên, nên phương thức cần viết sẽ nhận tham số là số nguyên rồi trả lại cũng một số nguyên:

```
public static int factorial(int n) {
}
```

Nếu đối số bằng 0, chúng ta chỉ cần trả lại giá trị 1:

```
public static int factorial(int n) {
  if (n == 0) {
    return 1;
  }
}
```

Đó là trường hợp cơ sở.

Nếu điều đó không xảy ra (đây chính là phần hay nhất), chúng ta thực hiện lời gọi đệ quy để tính giai thừa của *n* - 1 và sau đó nhân nó với *n*.

```
public static int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    int recurse = factorial(n-1);
    int result = n * recurse;
```

```
        return result;
    }
}
```

Luồng thực hiện của chương trình này cũng giống với `countdown` trong Mục 4.8. Nếu ta kích hoạt `factorial` với giá trị 3:

Vì 3 khác 0 nên ta chọn nhánh thứ hai và tính giai thừa của `n-1`...

Vì 2 khác 0 nên ta chọn nhánh thứ hai và tính giai thừa của `n-1`...

Vì 1 khác 0 nên ta chọn nhánh thứ hai và tính giai thừa của `n-1`...

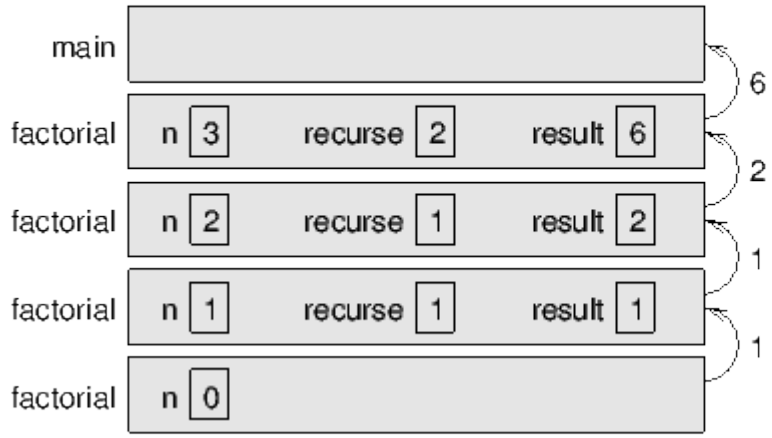
Vì 0 *bằng* 0 nên ta chọn nhánh thứ nhất và trả lại giá trị 1 và không gọi đệ quy thêm lần nào nữa.

Giá trị được trả về, 1, được nhân với *n*, vốn bằng 1, và kết quả được trả lại.

Giá trị được trả về (1) được nhân với *n*, vốn bằng 2, và kết quả được trả lại.

Giá trị được trả về (2) được nhân với *n*, vốn bằng 3, và kết quả, 6 trở thành giá trị trả về của hàm ứng với lúc bắt đầu gọi đệ quy.

Sau đây là nội dung của biểu đồ ngăn xếp khi một loạt các phương thức được kích hoạt:



Các giá trị trả lại như ở đây được chuyển về ngăn xếp.

Lưu ý rằng ở khung cuối cùng, các biến địa phương `recurse` và `result` đều không tồn tại, vì khi n=0, nhánh tạo ra chúng không được thực hiện.

6.9 Niềm tin

Việc dõi theo luồng thực hiện của chương trình là một cách đọc mã lệnh, nhưng bạn sẽ nhanh chóng lạc vào mê cung. Một cách làm khác mà tôi gọi là "niềm tin" như sau. Khi bạn dò đến chỗ kích hoạt phương thức, thay vì việc đi theo luồng thực hiện, hãy *coi như* là phương thức đó hoạt động tốt và trả lại kết quả đúng.

Thật ra, bạn đã từng có "niềm tin" này khi dùng các phương thức của Java. Mỗi lần kích hoạt `Math.cos` hay `System.out.println`, bạn không kiểm tra nội dung bên trong các phương thức này. Bạn chỉ việc giả sử rằng chúng hoạt động được.

Cũng với lý lẽ tương tự khi bạn kích hoạt các phương thức do mình viết nên. Chẳng hạn, trong Mục 6.7, chúng ta đã viết một hàm tên là `isSingleDigit` để xác định xem một số có nằm trong khoảng từ 0 đến 9 hay không. Một khi chúng ta tự thuyết phục rằng phương thức này đã viết đúng—bằng cách kiểm tra

và thử mã lệnh—chúng ta có thể sử dụng phương thức mà không cần phải xem lại phần mã lệnh nữa. Điều tương tự cũng đúng với các chương trình đệ quy. Khi bạn đến điểm kích hoạt đệ quy, thay vì đi theo luồng thực hiện, bạn cần *coi rằng* lời gọi đệ quy hoạt động tốt (tức là cho kết quả đúng) và sau đó tự hỏi mình "Giả dụ như ta đã tìm được giai thừa của $n-1$, liệu ta có tính được giai thừa của n không?" Trong trường hợp này, rõ ràng là ta sẽ tính được, bằng cách nhân với n.
Dĩ nhiên là sẽ có chút kì lạ trong việc ta giả sử rằng hàm hoạt động tốt khi chưa viết xong nó, nhưng chính vì vậy mà ta gọi đó là niềm tin!

6.10 Thêm một ví dụ

Ví dụ thông dụng thứ hai để minh họa cho một hàm toán toán học đệ quy là fibonacci, với cách định nghĩa hàm như sau:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);$$

Chuyển sang ngôn ngữ Java, ta viết được

```
public static int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Nếu bạn thử gắng theo luồng thực hiện ở đây, ngay cả với các giá trị nhỏ của n, bạn sẽ đau đầu ngay. Nhưng bằng niềm tin, nếu bạn coi rằng cả hai lời gọi đệ quy đều hoạt động tốt, thì rõ ràng bạn sẽ thu được kết quả đúng khi cộng chúng lại với nhau.

6.11 Thuật ngữ

kiểu trả lại:

Phần của lời khai báo phương thức, trong đó quy định kiểu của giá trị mà phương thức đó sẽ trả lại.

giá trị trả lại:

Giá trị được đưa ra làm kết quả của việc kích hoạt phương thức.

đoạn mã chết:

Phần chương trình không bao giờ được thực hiện, thường là do nó xuất hiện sau một câu lệnh `return`.

dàn giáo:

Mã lệnh được dùng trong giai đoạn phát triển chương trình nhưng bị bỏ đi ở phiên bản chương trình cuối.

rỗng (void):

Một kiểu trả lại đặc biệt có ở phương thức rỗng; nghĩa là phương thức không trả lại giá trị nào.

quá tải:

Việc có nhiều phương thức với cùng tên gọi nhưng có các tham số khác nhau. Khi bạn kích hoạt một

phương thức quá tải, Java sẽ biết được phải dùng dạng nào của phương thức, căn cứ vào những đối số mà bạn cung cấp. (Tiếng Anh: "Overloading")

boolean:

Một kiểu biến chỉ chứa hai giá trị true và false (đúng và sai).

dấu hiệu:

Một biến (thường với kiểu boolean) để ghi lại thông tin về một điều kiện hoặc trạng thái nào đó.

toán tử điều kiện:

Một toán tử dùng để so sánh hai giá trị rồi tạo ra một giá trị boolean để chỉ định quan hệ giữa hai toán hạng nêu trên.

toán tử logic:

Một toán tử nhằm kết hợp các giá trị boolean rồi trả lại cũng giá trị boolean.

6.12 Bài tập

Bài tập 1 Hãy viết một phương thức có tên isDivisible để nhận vào hai số nguyên, n và m rồi trả lại true nếu n chia hết cho m và trả lại false trong trường hợp còn lại.

Bài tập 2 Nhiều phép tính có thể được diễn đạt ngắn gọn bằng phép "multadd" (nhân-cộng), trong đó lấy ba toán hạng rồi đi tính a*b + c. Thậm chí có bộ vi xử lý còn tích hợp cả phép tính này đối với những số phẩy động.

1. Hãy lập một chương trình mới có tên gọi Multadd.java.
2. Viết một phương thức gọi là multadd để lấy tham số là ba số double rồi trả lại kết quả của phép nhân-cộng giữa chúng.
3. Viết một phương thức main để kiểm tra multadd bằng cách kích hoạt nó với một vài tham số đơn giản như 1.0, 2.0, 3.0.
4. Cũng trong main, hãy dùng multadd để tính các giá trị sau:

$$\sin\frac{\pi}{4} \quad \cos\frac{\pi}{4} + \frac{}{2}$$

$$\log 10 + \log 20$$

5. Hãy viết một phương thức có tên yikes để nhận tham số là một double rồi dùng multadd để tính

$$x\, e^{-x} + \sqrt{\frac{}{1 - e^{-x}}}$$

Gợi ý: để nâng e lên một số mũ, hãy dùng phương thức có tên Math.exp.

Trong câu hỏi sau cùng, bạn có cơ hội viết một phương thức để kích hoạt một phương thức mà bạn đã viết trước đó. Mỗi khi làm như vậy, bạn nên cẩn thận kiểm thử phương thức đầu trước khi viết sang phương thức thứ hai. Nếu không, có thể bạn sẽ rơi vào trường hợp phải gỡ lỗi hai phương thức cùng lúc, một công việc rất khó khăn.

Một mục đích của bài này là nhằm luyện tập cách khớp mẫu: đó là khi được cho một bài toán cụ thể, ta cần nhận dạng nó trong số một tập hợp các thể loại bài toán.

Bài tập 3 Nếu có trong tay ba que gỗ, có thể bạn sẽ có hoặc không xếp được thành hình tam giác.

Chẳng hạn, nếu một que dài 12 inch còn hai que kia, mỗi que chỉ dài 1 inch, thì bạn không thể kéo hai đầu que ngắn chạm nhau ở giữa được. Với ba đoạn thẳng có dài bất kì, có một cách kiểm tra đơn giản để xem liệu chúng có xếp thành hình tam giác được không:

> "Nếu có bất kì chiều dài nào trong số đó lớn hơn tổng hai chiều dài còn lại, thì bạn không thể dựng thành hình tam giác. Trường hợp còn lại, thì có thể được."

Hãy viết một phương thức với tên gọi isTriangle, nhận vào đối số là ba số nguyên, rồi trả lại true hoặc false, tùy theo khả năng xếp thành hình tam giác bằng những que có chiều dài đã cho. Mục đích của bài tập này là nhằm áp dụng những lệnh điều kiện để viết nên một phương thức trả lại giá trị.

Bài tập 4 Kết quả của chương trình dưới đây là gì? Mục đích của bài tập này nhằm đảm bảo rằng bạn hiểu rõ các toán tử logic và luồng thực thi thông qua các phương thức trả giá trị.

```
public static void main(String[] args) {
  boolean flag1 = isHoopy(202);
  boolean flag2 = isFrabjuous(202);
  System.out.println(flag1);
  System.out.println(flag2);
  if (flag1 && flag2) {
    System.out.println("ping!");
  }
  if (flag1 || flag2) {
    System.out.println("pong!");
  }
}
public static boolean isHoopy(int x) {
  boolean hoopyFlag;
  if (x%2 == 0) {
    hoopyFlag = true;
  } else {
    hoopyFlag = false;
  }
  return hoopyFlag;
}
public static boolean isFrabjuous(int x) {
  boolean frabjuousFlag;
  if (x > 0) {
    frabjuousFlag = true;
  } else {
    frabjuousFlag = false;
```

```
    }
    return frabjuousFlag;
  }
```

Bài tập 5 Khoảng cách giữa hai điểm (x_1, y_1) và (x_2, y_2) thì bằng

$$\text{Distance} = \sqrt{(x_2 - x_1)_2 + (y_2 - y_1)_2}$$

Hãy viết một phương thức có tên distance để nhận các tham số gồm bốn số phẩy động—x1, y1, x2 và y2—rồi in ra khoảng cách giữa hai điểm này. Bạn cần giả sử rằng đã có một phương thức sumSquares để tính và trả lại tổng các bình phương của đối số. Chẳng hạn dòng lệnh:

```
    double x = sumSquares(3.0, 4.0);
```

sẽ gán giá trị 25.0 cho x.
Mục đích của bài tập này là nhằm viết một phương thức mới có áp dụng phương thức sẵn có. Bạn chỉ cần viết một phương thức: distance. Bạn không được viết sumSquares hay main và cũng không kích hoạt distance.

Bài tập 6 Mục đích của bài tập này là dùng biểu đồ ngăn xếp để hiểu được trình tự thực hiện một chương trình đệ quy.

```
public class Prod {


  public static void main(String[] args) {
    System.out.println(prod(1, 4));
  }


  public static int prod(int m, int n) {
    if (m == n) {
      return n;
    } else {
      int recurse = prod(m, n-1);
      int result = n * recurse;
      return result;
    }
  }


}
```

1. Hãy vẽ một biểu đồ ngăn xếp cho thấy trạng thái của chương trình ngay trước khi thực thể cuối cùng của prod hoàn tất thực thi. Kết quả của chương trình này là gì?
2. Giải thích ngắn gọn xem prod làm việc gì.
3. Viết lại prod mà không dùng đến các biến tạm recurse và result.

Bài tập 7 Mục đích của bài tập này là chuyển từ một lời định nghĩa đệ quy sang một phương thức Java. Hàm Ackerman được định nghĩa cho số nguyên không âm như sau:

$$A(m, n) = \begin{cases} n+1 & \text{nếu } m = 0 \\ A(m-1, 1) & \text{nếu } m > 0 \text{ và } n = 0 \\ A(m-1, A(m, n-1)) & \text{nếu } m > 0 \text{ và } n > 0. \end{cases} \quad (1)$$

Hãy viết một phương thức tên là ack để nhận tham số là hai số int rồi tính và trả lại giá trị của hàm Ackerman. Hãy kiểm tra phương thức vừa viết bằng cách kích hoạt nó từ main rồi in ra giá trị vừa trả lại.

CẢNH BÁO: giá trị được trả lại sẽ rất nhanh chóng tăng cao. Bạn chỉ nên thử chạy với các giá trị m và n nhỏ (không lớn quá 2).

Bài tập 8

1. Hãy tạo nên một chương trình có tên Recurse.java rồi gõ vào các phương thức sau:

```
// first: trả lại kí tự đầu tiên của String cho trước
public static char first(String s) {
  return s.charAt(0);
}

// last: trả lại một String mới có chứa toàn bộ
// chỉ trừ kí tự đầu của String cho trước
public static String rest(String s) {
  return s.substring(1, s.length());
}

// length: trả lại chiều dài của String cho trước
public static int length(String s) {
  return s.length();
}
```

2. Hãy viết vài câu lệnh trong main để kiểm tra từng phương thức trên. Đảm bảo chắc là chúng hoạt động được, và chắc chắn là bạn đã hiểu công dụng của chúng là gì.
3. Viết một phương thức có tên printString để nhận tham số là một String đồng thời in các chữ cái trong String đó, mỗi chữ cái trên một dòng. Phương thức này phải là kiểu rỗng.
4. Viết một phương thức có tên printBackward có công dụng gần giống printString chỉ khác là in String theo chiều ngược lại (mỗi kí tự trên một dòng riêng).
5. Viết một phương thức có tên reverseString để nhận tham số là một String rồi trả lại giá trị là một String mới. String mới này phải có đầy đủ các chữ cái như String đã nhập làm tham số; nhưng lại xếp theo thứ tự ngược lại. Chẳng hạn, kết quả của đoạn mã lệnh sau

```
String backwards = reverseString("Allen Downey");
```

```
  System.out.println(backwards);
```

sẽ phải là

```
yenwoD nellA
```

Bài tập 9 Hãy viết một phương thức đệ quy có tên power để nhận vào x và một số nguyên n rồi trả lại x^n. Gợi ý: một định nghĩa đệ quy đối với phép tính này là $x^n = x \cdot x^{n-1}$. Đồng thời, cần nhớ rằng mọi số nâng lên lũy thừa bậc 0 đều bằng 1. Câu hỏi khó tự chọn: bạn có thể làm cho phương thức này hiệu quả hơn, trong trường hợp n chẵn, bằng cách dùng công thức $x^n = (x^{n/2})^2$.

Bài tập 10 (Bài tập này được dựa trên trang 44 cuốn sách Structure and Interpretation of Computer Programs của Abelson và Sussman.) Kĩ thuật sau đây có tên gọi Thuật toán Euclid vì nó xuất hiện trong tập Cơ bản của Euclid (Cuốn số 7, khoảng năm 300 TCN). Có lẽ đây là thuật toán đáng kể từ lâu đời nhất[1]. Quy trình tính toán được dựa theo quan sát thấy, nếu r là phần dư trong phép chia a cho b, thì các ước số chung của a và b cũng bằng ước số chung của b và r. Do vậy ta có thể dùng phương trình `gcd(a, b) = gcd(b, r)` để liên tiếp rút gọn bài toán tính ước số chung (GCD) về bài toán tính GCD của các cặp số nguyên ngày càng nhỏ hơn. Chẳng hạn,

```
gcd(36, 20) = gcd(20, 16) = gcd(16, 4) = gcd(4, 0) = 4
```

ngụ ý rằng GCD của 36 và 20 thì bằng 4. Có thể thấy rằng với bất kì hai số ban đầu nào, cách liên tiếp rút gọn này cuối cùng sẽ cho ta một cặp số mà số thứ hai bằng 0. Khi đó GCD sẽ bằng số còn lại trong cặp.

Hãy viết một phương thức có tên gcd để nhận tham số là hai số nguyên rồi dùng Thuật toán Euclid để tính và trả lại ước số chung lớn nhất của hai số.

Chương 7: Phép lặp và vòng lặp

7.1 Phép gán nhiều lần

Bạn có thể khiến cho nhiều lệnh gán chỉ tới cùng một biến; mà hiệu quả của nó là nhằm thay thế giá trị cũ bằng giá trị mới.

```
int liz = 5;
System.out.print(liz);
liz = 7;
System.out.println(liz);
```

Kết quả của chương trình này bằng 57, vì lần đầu tiên khi in liz biến này có giá trị bằng 5, còn lần thứ hai thì biến có giá trị bằng 7.
Hình thức **gán nhiều lần** như thế này là lí do mà tôi mô tả các biến như là *hộp chứa* giá trị. Khi bạn gán một giá trị vào cho biến, bạn thay đổi nội dung của hộp chứa, như ở hình vẽ sau:

Khi có nhiều phép gán đối với cùng một biến, thì rất chú trọng việc phân biệt giữa câu lệnh gán và đẳng thức. Vì Java dùng dấu = cho lệnh gán nên ta bị lôi cuốn vào việc diễn giải một câu lệnh như a = b là câu lệnh đẳng thức. Thật ra không phải vậy!
Trước hết, đẳng thức thì có tính giao hoán, còn lệnh gán thì không. Chẳng hạn trong toán học, nếu $a = 7$ thì $7 = a$. Nhưng trong Java a = 7; lại là một lệnh gán hợp lệ, còn 7 = a; thì không.
Hơn nữa, trong toán học, một đẳng thức thì luôn đúng. Nếu bây giờ $a = b$, thì a sẽ luôn bằng b. Trong Java, một lệnh gán có thể làm cho hai biến bằng nhau, nhưng không có gì bắt buộc chúng bằng nhau mãi!

```
int a = 5;
int b = a; // bây giờ thì a bằng b
a = 3; // a không còn bằng b nữa
```

Dòng lệnh thứ ba đã thay đổi giá trị của a mà không làm thay đổi giá trị của b, vì vậy chúng không còn bằng nhau. Một số ngôn ngữ lập trình có dùng kí hiệu khác cho phép gán, như <- hoặc :=, để tránh sự nhầm lẫn này.
Mặc dù phép gán nhiều lần thường có ích, so bạn nên cẩn thận khi dùng. Nếu giá trị của các biến thay đổi thường xuyên thì có có thể khiến cho mã lệnh khó đọc và gỡ lỗi.

7.2 Câu lệnh while

Máy tính thường được dùng để tự động hóa các thao tác có tính lặp lại. Thực hiện những thao tác lặp lại này mà không phạm lỗi là điều mà máy tính làm tốt còn chúng ta làm rất dở.

Ta đã thấy các phương thức như countdown và factorial trong đó dùng đệ quy để thực hiện lặp. Quá trình này được gọi là **phép lặp**. Java có những đặc điểm ngôn ngữ giúp cho việc viết các phương thức nêu trên một cách dễ dàng hơn. Ở chương này ta xem xét câu lệnh while. Về sau (ở Mục 12.4) ta xét đến

câu lệnh for.
Dùng câu lệnh while, ta có thể viết lại countdown:

```
public static void countdown(int n) {
  while (n > 0) {
    System.out.println(n);
    n = n-1;
  }
  System.out.println("Bum!");
}
```

Gần như là bạn có thể đọc được toàn bộ câu lệnh while bằng tiếng Anh. Lệnh này diễn tả là, “Khi n lớn hơn không, hãy in giá trị của n rồi giảm giá trị của n xuống 1. Khi bạn đạt đến không, hãy in ra từ ‘Bum!’”
Theo cách quy củ hơn, luồng thực thi của một lệnh while như sau:
1.Định giá điều kiện trong cặp ngoặc tròn, cho ra true hoặc false.
2.Nếu điều kiện là sai, thì thoát khỏi lệnh while rồi tiếp tục thực thi câu lệnh liền sau.
3.Nếu điều kiện là đúng, thì thực thi những câu lệnh trong phạm vi cặp ngoặc nhọn, rồi trở lại bước 1.
Kiểu luồng thực thi này được gọi là **vòng lặp** vì bước thứ ba vòng ngược trở lên đầu. Những câu lệnh bên trong vòng lặp được gọi là **thân** của vòng lặp. Nếu điều kiện là sai ngay lần đầu tiên qua vòng lặp thì những câu lệnh bên trong vòng lặp không bao giờ được thực thi.

Phần thân vòng lặp cần phải thay đổi giá trị của một vài biến sao cho cuối cùng thì điều kiện trở nên sai và vòng lặp chấm dứt. Nếu không, vòng sẽ được lặp lại mãi, và được gọi là vòng lặp **vô hạn**. Một câu chuyện đùa luôn được nhắc đến trong giới khoa học máy tính là qua việc nhận thấy chỉ dẫn trên gói dầu gội đầu, “Xát, xả nước, rồi lặp lại,” chính là một vòng lặp vô hạn.

Ở trường hợp countdown, ta có thể chứng minh rằng vòng lặp sẽ kết thúc nếu n là số dương. Còn trong những trường hợp khác thì không dễ nói trước:

```
public static void sequence(int n) {
  while (n != 1) {
    System.out.println(n);
    if (n%2 == 0) { // n chẵn
      n = n / 2;
    } else { // n lẻ
      n = n*3 + 1;
    }
  }
}
```

Điều kiện của vòng lặp này là n != 1, vì vậy vòng lặp sẽ tiếp diễn đến tận khi n bằng 1, và điều này khiến cho điều kiện bị sai đi.
Tại mỗi vòng lặp, chương trình in ra giá trị của n rồi kiểm tra xem liệu số này chẵn hay lẻ. Nếu chẵn, giá

trị của n được chia cho 2. Nếu lẻ, giá trị được thay thế bởi $3n+1$. Chẳng hạn, nếu giá trị ban đầu (tức đối số được truyền vào sequence) bằng 3, thì kết quả là ta có dãy 3, 10, 5, 16, 8, 4, 2, 1.
Vì đôi khi n tăng và đôi khi giảm, nên sẽ không có cách chứng minh nào dễ thấy ràng cuối cùng n sẽ đạt đến 1, hay chương trình sẽ kết thúc. Với một số giá trị đặc biệt của n, ta có thể chứng minh được sự kết thúc đó. Chẳng hạn, nếu giá trị khởi đầu là một số lũy thừa của hai, thì giá trị của n sẽ luôn chẵn qua mỗi lần lặp, cho đến khi ta thu được 1. Ví dụ trước sẽ kết thúc với một dãy như vậy với giá trị ban đầu bằng 16.
Ngoài những giá trị đặc biệt, thì một câu hỏi thú vị là liệu ta có thể chứng minh được rằng đoạn chương trình trên có kết thúc với *tất cả* những giá trị của n hay không. Cho đến giờ, chưa ai có thể chứng minh hoặc bác bỏ nó! Bạn hãy tìm thêm thông tin ở http://en.wikipedia.org/wiki/Collatz_conjecture.

7.3 Bảng số liệu

Một trong những công việc thích hợp với dùng vòng lặp, đó là phát sinh ra bảng số liệu. Trước khi máy tính trở nên phổ biến, mọi người đã phải tính tay các phép logarit, sin, cosin, và những hàm toán học khác.

Để đơn giản hóa việc này, sách toán thường in kèm những bảng dài liệt kê giá trị các hàm nói trên. Việc tạo ra các bảng như vậy rất chậm và nhàm chán, và dễ mắc phải nhiều lỗi.

Khi máy tính xuất hiện, đã có những phản ứng ban đầu kiểu như: “Điều này thật tuyệt! Giờ ta có thể dùng máy tính để tạo ra các bảng, vì vậy sẽ không có lỗi.” Điều này trở nên (gần như là) sự thật nhưng vẫn chứa đựng tầm nhìn hạn hẹp. Không lâu sau đó, máy tính và máy bỏ túi đã xuất hiện tràn lan và bảng số trở nên lỗi thời.

Ừ, gần như vậy. Có những phép tính mà máy tính lấy con số từ bảng để có giá trị gần đúng, rồi thực hiện tính toán nhằm cải thiện kết quả gần đúng này. Ở trường hợp khác, có những lỗi nằm ngay ở bảng số, được biết đến nhiều nhất là bảng mà máy Intel Pentium đã dùng để thực hiện phép chia với số có dấu phẩy động.

Mặc dù bảng loga không còn hữu dụng như xưa, song nó vẫn dùng được làm ví dụ về tính lặp. Chương trình sau in ra một dãy các số ở cột bên trái cùng với giá trị logarit của chúng ở cột phải:

```
    double x = 1.0;
    while (x < 10.0) {
      System.out.println(x + " " + Math.log(x));
      x = x + 1.0;
    }
```

Kết quả của chương trình này là:

```
1.0   0.0
2.0   0.6931471805599453
3.0   1.0986122886681098
4.0   1.3862943611198906
```

```
5.0   1.6094379124341003
6.0   1.791759469228055
7.0   1.9459101490553132
8.0   2.0794415416798357
9.0   2.1972245773362196
```

Nhìn vào những giá trị này, bạn có thể nói rằng phương thức log này dùng cơ số nào?
Vì các lũy thừa của 2 rất quan trọng trong ngành khoa học máy tính, nên ta thường muốn lấy loga theo cơ số 2. Để tính toán, ta có thể dùng biểu thức:

$$\log_2 x = \log_e x / \log_e 2$$

Hãy thay câu lệnh print bằng

```
    System.out.println(x + " " + Math.log(x) / Math.log(2.0));
```

để cho ra

```
1.0   0.0
2.0   1.0
3.0   1.5849625007211563
4.0   2.0
5.0   2.321928094887362
6.0   2.584962500721156
7.0   2.807354922057604
8.0   3.0
9.0   3.1699250014423126
```

Có thể thấy rằng 1, 2, 4, và 8 là các lũy thừa của 2 vì các giá trị logarit cơ số 2 của chúng đều là những số nguyên. Nếu muốn tìm logarit của những lũy thừa khác của 2, ta có thể sửa chương trình trên thành:

```
    double x = 1.0;
    while (x < 100.0) {
      System.out.println(x + " " + Math.log(x) / Math.log(2.0));
      x = x * 2.0;
    }
```

Bây giờ thay vì cộng thêm một số với x trong mỗi vòng lặp (điều này cho ra dãy cấp số cộng), ta đem nhân một giá trị với x (thu được **cấp số nhân**). Kết quả là:

```
1.0   0.0
2.0   1.0
4.0   2.0
8.0   3.0
16.0   4.0
32.0   5.0
64.0   6.0
```

Bảng logarit có thể không còn có ích nữa, nhưng với nhà khoa học máy tính, việc nhớ được các lũy thừa

của hai nhất thiết có ích! Khi nào rảnh rỗi, bạn hãy ghi nhớ các lũy thừa của hai đến tận 65536 (tức là 2^{16}).

7.4 Bảng hai chiều

Trong một bảng hai chiều, bạn đọc giá trị ở điểm giao cắt giữa một hàng với một cột. Bảng cửu chương là một ví dụ điển hình. Giả sử bạn muốn in ra một bảng tính nhân với các giá trị từ 1 đến 6.

Một cách bắt đầu ổn thỏa là viết một vòng lặp để in ra các bội số của 2 trên cùng một dòng.

```
int i = 1;
while (i <= 6) {
  System.out.print(2*i + " ");
  i = i + 1;
}
System.out.println("");
```

Dòng đầu tiên khởi tạo một biến có tên là i; nó đóng vai trò một biến đếm hoặc **biến vòng lặp**. Khi vòng lặp được thực thi, giá trị của i tăng từ 1 lên 6. Khi i bằng 7, vòng lặp kết thúc. Mỗi lần lặp, chương trình sẽ in ra giá trị của 2*i, theo sau là ba dấu cách.
Một lần nữa, dấu phẩy trong câu lệnh print ngăn không cho xuống dòng. Sau khi vòng lặp kết thúc, lệnh print thứ hai bắt đầu một dòng mới. Vì ta dùng System.out.print, nên toàn bộ kết quả được ghi trên một dòng.
Có những môi trường mà kết quả từ print được lưu lại mà chưa hiển thị đến khi kích hoạt println. Nếu chương trình kết thúc, mà bạn quên kích hoạt println, có thể bạn sẽ không bao giờ thấy được kết quả được lưu lại này.

Kết quả của chương trình là:

```
2   4   6   8   10   12
```

Mọi việc đến giờ tiến triển tốt. Bước tiếp theo là **bao bọc** và **khái quát hóa**.

7.5 Bao bọc và khái quát hóa

Bao bọc là quá trình đặt một đoạn mã lệnh vào trong một phương thức; việc này cho phép ta tận dụng được những ưu điểm của phương thức. Ta đã thấy hai ví dụ về bao bọc, khi ta viết printParity ở Mục 4.3 và isSingleDigit ở Mục 6.7.
Khái quát hóa nghĩa là chọn lấy một điều cụ thể, như công việc in ra các bội số của 2, rồi làm cho nó trở thành khái quát hơn, chẳng hạn như in ra các bội số của một số nguyên bất kì.

Phương thức sau đây bao bọc đoạn mã lệnh nói trên rồi khái quát hóa nó để in ra các bội số của n.

```
public static void printMultiples(int n) {
  int i = 1;
  while (i <= 6) {
    System.out.print(n*i + " ");
    i = i + 1;
```

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

Để bao bọc, ta chỉ cần viết thêm dòng thứ nhất, tức là khai báo tên, tham số, và kiểu trả lại. Để khái quát hóa, ta chỉ cần thay thế giá trị 2 bởi tham số n.
Nếu ta kích hoạt phương thức này với đối số bằng 2, ta sẽ nhận được kết quả giống như trước. Với đối số bằng 3, kết quả sẽ là:

```
3   6   9   12   15   18
```

Với đối số bằng 4, kết quả là:

```
4   8   12   16   20   24
```

Bây giờ có thể bạn đã đoán được cách in một bảng tính nhân — bằng cách kích hoạt printMultiples lặp lại với những đối số khác nhau. Thực ra, ta có thể dùng một vòng khác để lặp qua các hàng trong bảng:

```
    int i = 1;
    while (i <= 6) {
      printMultiples(i);
      i = i + 1;
    }
```

Trước hết, hãy lưu ý sự giống nhau của vòng lặp này với vòng lặp bên trong printMultiples. Tất cả những gì ta đã làm chỉ là việc thay lệnh print bằng một lời kích hoạt phương thức.
Kết quả của chương trình này là

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

vốn là một bảng tính nhân (hơi lôi thôi). Nếu bạn không thích lôi thôi, thì Java sẵn có những phương thức giúp bạn kiểm soát chặt chẽ hơi định dạng của kết quả; song bây giờ ta không đề cập đến điều này.

7.6 Phương thức và bao bọc

Ở Mục 3.5 tôi đã liệt kê vài lý do mà phương thức trở nên có ích. Sau đây còn thêm một số lý do khác:

•Bằng cách đặt tên cho một dãy các câu lệnh, bạn có thể làm cho chương trình mình viết trở nên dễ đọc và gỡ lỗi hơn.
•Việc chia một chương trình dài thành nhiều phương thức cho phép bạn phân chia các phần của chương trình, tiến hành gỡ lỗi chúng một cách độc lập, rồi ghép lại thành tổng thể.
•Phương thức cho phép cả đệ quy lẫn lặp lại.
•Các phương thức được thiết kế tốt thì thường hữu ích cho nhiều chương trình khác nhau. Một khi đã viết ra và gỡ lỗi xong một phương thức, bạn có thể tái sử dụng nó.

Để biểu diễn tiếp kĩ thuật bao bọc, ta hãy lấy đoạn mã lệnh ở cuối mục trước rồi bọc nó vào trong một phương thức:

```
public static void printMultTable() {
  int i = 1;
  while (i <= 6) {
    printMultiples(i); i = i + 1;
  }
}
```

Quá trình mà tôi hiện đang giới thiệu được gọi là **bao bọc và khái quát hóa**. Ta phát triển mã lệnh bằng cách viết riêng những dòng lệnh vào main hoặc vào phương thức khác. Khi mã lệnh này thực hiện được, ta lấy lại nó rồi bọc vào một phương thức. Rồi bạn khái quát hóa phương thức bằng cách bổ sung các tham số.

Lúc mới lập trình, đôi khi bạn không biết cách chia chương trình thành các phương thức. Quy trình trên giúp bạn thiết kế trong khi lập trình.

7.7 Các biến địa phương

Có thể bạn tự hỏi bằng cách nào mà ta dùng được cùng một biến, i, cả
trong printMultiples lẫn printMultTable. Chẳng phải nó sẽ gây rắc rối khi một trong hai phương thức thay đổi giá trị của biến sao?

Lời giải đáp cho cả hai câu hỏi trên đều là không,
vì i trong printMultiples và i trong printMultTable *không* phải cùng một biến. Chúng có cùng tên gọi, nhưng không tham chiếu đến cùng vị trí lưu trữ, và việc thay đổi một biến này sẽ không ảnh hưởng gì tới biến kia.

Những biến được tạo ra bên trong phần định nghĩa phương thức được gọi là **biến địa phương**, vì chúng chỉ tồn tại bên trong phương thức đó. Bạn không thể truy cập biến địa phương từ ngoài phương thức "chủ" của nó, và bạn có thể tùy ý đặt nhiều biến cùng tên, miễn là chúng không phải trong cùng một hàm.

Mặc dù điều này có thể gây nhầm lẫn, song có những lí do thích đáng để sử dụng lại các tên gọi. Chẳng hạn, các tên i, j và k thường được dùng làm biến lặp. Nếu bạn tránh dùng chúng trong một phương thức chỉ vì bạn đã dùng chúng ở nơi khác, thì chương trình viết ra sẽ khó đọc hơn.

7.8 Nói thêm về khái quát hóa

Xét một ví dụ khác về khái quát hóa. Hãy hình dung rằng bạn muốn có một chương trình để in ra bảng tính nhân với kích thước bất kì, chứ không chỉ 6 × 6. Bạn có thể thêm một tham số vào printMultTable:

```
public static void printMultTable(int high) {
  int i = 1;
  while (i <= high) {
    printMultiples(i);
    i = i + 1;
  }
}
```

Tôi đã thay giá trị 6 bởi tham số high. Nếu tôi kích hoạt printMultTable với đối số 7, tôi sẽ được:

```
1   2   3   4   5   6

2   4   6   8   10   12

3   6   9   12   15   18

4   8   12   16   20   24

5   10   15   20   25   30

6   12   18   24   30   36

7   14   21   28   35   42
```

Thế này tạm được, nhưng có lẽ ta muốn nhận được một bảng hình vuông hơn (số cột và số hàng phải bằng nhau). Để làm điều này, ta thêm một tham số nữa vào printMultiples để cụ thể hóa xem bảng có bao nhiêu cột.

Ta gọi tham số này là high, nhằm cho thấy các phương thức khác nhau hoàn toàn có thể chứa những tham biến có cùng tên (cũng như các biến địa phương):

```
  public static void printMultiples(int n, int high) {
    int i = 1;
    while (i <= high) {
      System.out.print(n*i + " ");
      i = i + 1;
    }
    System.out.println("");
  }
  public static void printMultTable(int high) {
    int i = 1;
    while (i <= high) {
      printMultiples(i, high);
      i = i + 1;
    }
  }
```

Lưu ý rằng khi thêm một tham số mới, ta phải sửa lại dòng đầu tiên, đồng thời ta cũng phải sửa chỗ phương thức được kích hoạt trong printMultTable. Đúng như dự kiến, chương trình này phát sinh ra bảng vuông 7 × 7:

```
1   2   3   4   5   6   7

2   4   6   8   10   12   14

3   6   9   12   15   18   21

4   8   12   16   20   24   28

5   10   15   20   25   30   35

6   12   18   24   30   36   42

7   14   21   28   35   42   49
```

Khi bạn khái quát quá một phương thức theo cách thích hợp, thường bạn sẽ thu được chương trình với

những tính năng mà bạn chưa lường trước. Chẳng hạn, có thể bạn nhận thấy rằng bảng nhân có tính đối xứng, vì ab = ba, nên tất cả những con số trong bảng đều xuất hiện lặp hai lần. Lẽ ra bạn có thể tiết kiệm mực bằng cách chỉ in ra nửa bảng thôi. Để làm điều này, chỉ cần thay đổi một dòng lệnh trong printMultTable. Hãy sửa lệnh

```
      printMultiples(i, high);
```

thành

```
      printMultiples(i, i);
```

và thu được

```
1
2   4
3   6   9
4   8   12   16
5   10   15   20   25
6   12   18   24   30   36
7   14   21   28   35   42   49
```

Tôi sẽ để bạn tự hình dung cơ chế của cách máy tính đã xử lí trong trường hợp này.

7.9 Thuật ngữ

vòng lặp:
Một câu lệnh được lặp đi lặp lại nhiều lần khi một điều kiện nào đó được thỏa mãn.
vòng lặp vô hạn:
Một vòng lặp có điều kiện luôn luôn đúng.
phần thân:
Những câu lệnh bên trong vòng lặp.
lặp:
Một lượt chạy (thực thi) qua phần thân vòng lặp, bao gồm cả việc định giá điều kiện.
bao bọc:
Việc phân chia một chương trình lớn, phức tạp thành nhiều thành phần (như phương thức) rồi cô lập riêng các thành phần (chẳng hạn, bằng cách dùng các biến địa phương).
biến địa phương:
Một biến được khai báo bên trong một phương thức; biến này chỉ tồn tại trong phương thức đó. Những biến địa phương đều không truy cập được từ ngoài phương thức của nó, và không can thiệp tới bất kì phương thức nào khác.
khái quát hóa:
Việc thay thế những thứ cụ thể một cách không cần thiết (như một giá trị không đổi) bằng những thứ có tính khái quát thích hợp (nhưng một biến hoặc một tham số). Việc khái quát hóa khiến cho mã lệnh linh hoạt hơn, dễ sử dụng lại hơn, và đôi khi dễ viết hơn.
phát triển chương trình:

> *Một quá trình để viết nên những chương trình máy tính. Cho đến bây giờ ta đã gặp "phát triển tăng dần" và "bao bọc và khái quát hóa".*

7.10 Bài tập

Bài tập 1 Xét đoạn mã lệnh sau:

```
public static void main(String[] args) {
  loop(10);
}
public static void loop(int n) {
  int i = n;
  while (i > 0) {
    System.out.println(i);
    if (i%2 == 0) {
      i = i/2;
    } else {
      i = i+1;
    }
  }
}
```

1. Hãy kẻ một bảng để chỉ ra giá trị của các biến i và n trong quá trình thực thi loop. Bảng chỉ được phép chứa một cột cho mỗi biến và một hàng cho mỗi vòng lặp.
2. Kết quả của chương trình này là gì?

Bài tập 2 Giả sử bạn có một số, a, và bạn muốn tính căn bậc hai của nó. Một cách làm điều này là khởi đầu bằng một phỏng đoán sơ lược về đáp số, x_0, và rồi cải thiện phỏng đoán này theo công thức sau:

$x_1 = (x_0 + a/x_0) / 2$

Chẳng hạn, nếu ta muốn tìm căn bậc hai của 9, và bắt đầu với $x_0 = 6$, thì $x_1 = (6 + 9/6) / 2 = 15/4 = 3.75$, giá trị này đã sát hơn. Ta có thể lặp lại quy trình này, dùng x_1 để tính ra x_2, và cứ như vậy. Trong trường hợp này, $x_2 = 3.075$ và $x_3 = 3.00091$. Như vậy nó hội tụ rất nhanh về đáp số đúng (vốn bằng 3).

Hãy viết một phương thức có tên squareRoot nhận vào tham số là một double và trả lại một giá trị xấp xỉ cho căn bậc hai của tham số đó, theo kĩ thuật tính nêu trên. Bạn không được phép dùng Math.sqrt. Với giá trị ban đầu, bạn nên lấy a/2. Phương thức bạn viết cần phải lặp lại đến khi nó đạt được hai giá trị ước tính liên tiếp chỉ sai khác nhau chưa đến 0.0001; nói cách khác, là đến khi giá trị tuyệt đối của $x_n - x_{n-1}$ nhỏ hơn 0.0001. Bạn có thể dùng Math.abs để tính giá trị tuyệt đối này.

Bài tập 3 Ở Bài tập 9 ta đã viết một dạng đệ quy của power, trong đó nhận một biến double có tên x cùng một biến nguyên n and rồi trả lại x^n. Bây giờ hãy viết một phương thức lặp để thực hiện tính toán như vậy.

Bài tập 4 Mục 6.8 có trình bày một phương thức đệ quy để tính hàm giai thừa. Hãy viết một dạng tính lặp cho factorial.

Bài tập 5 Một cách để tính e^x là dùng khai triển chuỗi vô hạn

$e^x = 1 + x + x^2 / 2! + x^3 / 3! + x^4 / 4! + \ldots$

Nếu biến vòng lặp có tên i, thì số hạng thứ i sẽ là $x^i / i!$.

1. Hãy viết một phương thức có tên myexp để tính tổng của n số hạng đầu tiên trong dãy này. Bạn có thể dùng phương thức factorial ở Mục 6.8 hoặc dùng phiên bản tính lặp như ở bài tập trước.
2. Bạn có thể khiến phương thức này hiệu quả hơn nhiều nếu nhận thấy rằng ở mỗi lần lặp, tử số của số hạng thì đúng bằng tử số của số hạng liền trước đó nhân với x còn mẫu số thì đúng bằng mẫu của số hạng trước đó nhân với i. Hãy tận dụng kết quả của quan sát này để tránh dùng cả Math.pow lẫn factorial, rồi kiểm tra rằng bạn vẫn có thể đạt được kết quả y hệt.

3. Hãy viết một phương thức có tên check nhận vào một tham số, x, để in ra giá trị của x, Math.exp(x) và myexp(x) cho các giá trị x khác nhau. Kết quả phải có dạng như sau:

```
1.0     2.708333333333333       2.718281828459045
```

   GỢI Ý: bạn có thể dùng String "\t" để in ra một dấu tab giữa các cột trong bảng.
4. Hãy thay đổi số các số hạng trong chuỗi (chính là đối số thứ hai mà check gửi đến myexp) rồi xem sự ảnh hưởng đến độ chính xác của kết quả. Điều chỉnh giá trị này đến khi giá trị ước tính phù hợp với đáp số "đúng" khi x bằng 1.
5. Hãy viết một vòng lặp trong main để kích hoạt check với những giá trị 0.1, 1.0, 10.0, và 100.0. Độ chính xác của kết quả sẽ thay đổi thế nào khi x biến đổi? So sánh số chữ số giống nhau thay vief hiệu số giữa các giá trị đúng và giá trị ước tính được.
6. Thêm vào một vòng lặp trong main nhằm kiểm tra myexp với các giá trị -0.1, -1.0, -10.0, và -100.0. Hãy nhận xét về độ chính xác.

Bài tập 6 Một cách để tính $\exp(-x^2)$ là dùng khai triển chuỗi vô hạn

$$\exp(-x^2) = 1 - x^2 + x^4/2 - x^6/6 + \ldots$$

Nói cách khác, ta cần phải cộng các số hạng lại, trong đó số hạng thứ i bằng $(-1)^i x^{2i}/i!$. Hãy viết một phương thức có tên gauss nhận vào các đối số x và n rồi trả lại tổng của n số hạng đầu tiên trong chuỗi này. Bạn không được dùng cả factorial lẫn pow.

Chương 8: Chuỗi kí tự

8.1 Kí tự

Trong Java cũng như các ngôn ngữ hướng đối tượng khác thì **đối tượng** là tập hợp những dữ liệu có liên quan, cùng với một tập các phương thức. Những phương thức nafyhoajt động trên đối tượng kể trên, thực hiện tính toán và đôi lúc thay đổi dữ liệu trong đối tượng đó.
String (chuỗi kí tự) là các đối tượng, bởi vậy bạn có thể hỏi "Có dữ liệu nào được chứa trong một đối tượng String?" và "Có những phương thức nào mà ta có thể kích hoạt được từ đối tượng String?" Những thành phần trong một đối tượng String là các chữ cái, hay tổng quát hơn, là những kí tự. Không phải mọi kí tự đều là chữ cái; còn những kí tự là chữ số, kí hiệu, và các thứ khác. Để đơn giản tôi sẽ gọi chúng đều là các chữ cái. Có nhiều phương thức khác nhau, nhưng trong sách này chỉ dùng một số ít. Các phương thức còn lại được chỉ dẫn ởhttp://download.oracle.com/javase/6/docs/api/java/lang/String.html.
Phương thức đầu tiên mà ta xét đến là charAt; phương thức này cho phép bạn kết xuất những chữ cái từ một String. char là kiểu biến dùng được để lưu trữ từng kí tự riêng lẻ (trái ngược lại với một chuỗi các kí tự).
char cũng hoạt động như các kiểu dữ liệu khác ta đã gặp:

```
char ltr = 'c';

if (ltr == 'c') {

  System.out.println(ltr);

}
```

Những giá trị của kí tự đều xuất hiện trong cặp dấu nháy đơn, như 'c'. Khác với giá trị của chuỗi (xuất hiện giữa cặp dấu nháy kép), các giá trị kí tự chỉ có thể chứa một chữ cái hoặc một kí hiệu.
Sau đây là cách dùng phương thức charAt:

```
String fruit = "banana";

char letter = fruit.charAt(1);

System.out.println(letter);
```

fruit.charAt() có nghĩa rằng tôi đang kích hoạt phương thức charAt lên đối tượng có tên fruit. Tôi đang truyền đối số 1 vào phương thức này, tức là tôi đang muốn biết chữ cái đầu tiên của chuỗi là gì. Kết quả là một kí tự, và được lưu vào trong một char có tên letter. Khi tôi in ra giá trị của letter, tôi bị bất ngờ:

```
a
```

a không phải là chữ cái đầu tiên của "banana". Trừ khi bạn nghiên cứu khoa học máy tính. Vì những lí do kĩ thuật mà giới khoa học máy tính đều đếm từ số không. Chữ cái thứ 0 của "banana" là chữ b. Chữ cái thứ 1 là a và thứ 2 là n.
Nếu bạn muốn biết chữ cái thứ 0 của một chuỗi, bạn phải truyền tham số là 0:

```
char letter = fruit.charAt(0);
```

8.2 Length

Phương thức tiếp theo đối với String mà ta xét đến là length, vốn trả lại số kí tự có trong chuỗi. Chẳng hạn:

```
int length = fruit.length();
```

length không nhận đối số truyền vào, và trả lại một số nguyên, trong trường hợp này bằng 6. Lưu ý rằng việc có một biến trùng tên với phương thức là hoàn toàn hợp lệ (mặc dù điều này có thể gây nhầm lẫn đối với người đọc mã lệnh).
Để tìm chữ cái cuối cùng trong chuỗi, bạn có thể bị xui khiến để thử theo cách làm sau:

```
int length = fruit.length();
char last = fruit.charAt(length); // SAI!!
```

Cách này không có tác dụng. Lý do là không có chữ cái thứ 6 nào trong "banana". Vì ta đã bắt đầu đếm từ 0, nên sáu chữ cái trong chuỗi được đếm từ 0 tới 5. Để lấy chữ cái cuối cùng, ta phải trừ length đi một.

```
int length = fruit.length();
char last = fruit.charAt(length-1);
```

8.3 Duyệt chuỗi

Một công việc thường làm với một chuỗi là bắt đầu từ điểm đầu của chuỗi, lần lượt chọn từng kí tự, thực hiện một số thao tác đối với chữ cái đó, và công việc được tiếp diễn cho các chữ cái còn lại đến hết chuỗi. Kiểu xử lý như thế này được gọi là **duyệt**. Một cách tự nhiên để thực hiện việc duyệt là dùng vòng lặp `while`:

```
int index = 0;
while (index < fruit.length()) {
  char letter = fruit.charAt(index);
  System.out.println(letter);
  index = index + 1;
}
```

Vòng lặp này để duyệt chuỗi và hiển thị từng chữ cái trên một dòng riêng. Lưu ý điều kiện lặp là `index < fruit.length()`, nghĩa là khi `index` bằng với chiều dài của chuỗi, thì điều kiện bị vi phạm, và phần thân của vòng lặp không được thực hiện. Kí tự cuối cùng được truy cập đến sẽ tương ứng với chỉ số `fruit.length()-1`.
Tên của biến vòng lặp là index (có nghĩa là “chỉ số”). Một **chỉ số** là một biến hay giá trị được dùng để chỉ định một thành viên của một tập hợp được xếp thứ tự, trong trường hợp này là chuỗi các kí tự. Chỉ số có nhiệm vụ chỉ định thành viên nào bạn cần biết (vì vậy mà nó có tên “chỉ số”).

8.4 Lỗi thực thi

Trở về Mục 1.3.2 tôi đã nói tới các lỗi thực thi, những lỗi không xuất hiện đến tận khi chương trình bắt đầu chạn. Trong Java, những lỗi thực thi được gọi là các **biệt lệ**.

Có thể bạn không thấy nhiều lỗi thực thi, song vì ta chưa thực hiện nhiều thao tác có khả năng gây nên những lỗi loại này. Và bây giờ ta sẽ gây lỗi. Nếu bạn dùng phương thức charAt rồi cung cấp một chỉ số là số âm hoặc lớn hơn length-1, Java sẽ **phát ra** một biệt lệ. Bạn có thể hình dung việc “phát” biệt lệ cũng như phát ra một cơn giận dữ.
Khi điều này xảy đến, Java in ra một thông báo lỗi có ghi kiểu biệt lệ và một **lần vết ngăn xếp**, trong đó có biểu thị những phương thức đang hoạt động khi có biệt lệ xảy ra. Sau đây là một ví dụ:

```
public class BadString {
```

```
  public static void main(String[] args) {
    processWord("banana");
  }
  public static void processWord(String s) {
    char c = getLastLetter(s);
    System.out.println(c);
  }
  public static char getLastLetter(String s) {
    int index = s.length();    // SAI!
    char c = s.charAt(index);
    return c;
  }
}
```

Lưu ý rằng lỗi nằm trong getLastLetter: chỉ số của kí tự cuối cùng đáng ra phải là s.length()-1. Sau đây là kết quả bạn thu được:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 6
        at java.lang.String.charAt(String.java:694)
        at BadString.getLastLetter(BadString.java:24)
        at BadString.processWord(BadString.java:18)
        at BadString.main(BadString.java:14)
```

Sau đó chương trình kết thúc. Lần vết ngăn xếp này có thể khó đọc, song nó chứa đựng rất nhiều thông tin.

8.5 Đọc tài liệu

Nếu bạn truy cập đến http://download.oracle.com/javase/6/docs/api/java/lang/String.html và kích chuột vào charAt, bạn sẽ xem được tài liệu sau đây (hoặc với nội dung tương tự):

```
public char charAt(int index)

Returns the char value at the specified index. An index ranges
from 0 to length() - 1. The first char value of the sequence is
at index 0, the next at index 1, and so on, as for array indexing.

Parameters: index - the index of the char value.

Returns: the char value at the specified index of this string.
  The first char value is at index 0.

Throws: IndexOutOfBoundsException - if the index argument is
```

```
  negative or not less than the length of this string.
```

Dòng đầu tiên là **nguyên mẫu** của phương thức, có nhiệm vụ quy định tên của phương thức, kiểu dữ liệu của các tham số cũng như kiểu trả lại.

Dòng tiếp theo miêu tả những công việc mà phương thức thực hiện. Các dòng sau đó giải thích các tham số và giá trị trả lại. Trong trường hợp này, việc giải thích là quá thừa, nhưng tài liệu luôn được thiết kế để phù hợp một dạng mẫu tiêu chuẩn. Còn dòng cuối cùng mô tả các biệt lệ mà phương thức này có thể phát ra.

Có lẽ bạn sẽ mất chút thời gian để làm quen với kiểu tài liệu thế này, nhưng thời gian công sức bỏ ra cũng đáng.

8.6 Phương thức indexOf

indexOf là phép nghịch đảo của charAt: charAt nhận vào một chỉ số rồi trả lại kí tự ở vị trí chỉ số đó; indexOf nhận một kí tự rồi tìm chỉ số mà kí tự đó xuất hiện.
charAt thất bại nếu chỉ số nằm ngoài phạm vi chuỗi, khi đó phương thức này sẽ phát biệt lệ. indexOf thất bại nếu kí tự không có mặt trong chuỗi, và trả lại giá trị -1.

```
    String fruit = "banana";

    int index = fruit.indexOf('a');
```

Đoạn mã lệnh này tìm chỉ số của chữ cái 'a' trong chuỗi. Với trường hợp này, chữ cái nêu trên xuất hiện ba lần, nên ta chưa thấy ngay rằng indexOf nên làm gì. Nhưng theo tài liệu, thì phương thức này sẽ trả lại chỉ số của lần xuất hiện *đầu tiên*.
Để tìm các lần xuất hiện tiếp theo, còn có một dạng khác của indexOf. Nó nhận vào một đối số thứ hai quy định xem cần bắt đầu tìm kiếm từ vị trí nào trong chuỗi. Đây là một dạng quá tải toán tử, để biết thêm chi tiết, bạn hãy xem Mục 6.4.
Nếu ta kích hoạt:

```
   int index = fruit.indexOf('a', 2);
```

nó sẽ bắt đầu ở chữ cái số hai (chữ n đầu tiên) rồi tìm chữ a thứ hai, vốn có chỉ số là 3. Nếu tình cờ chữ cái đó xuất hiện ngay ở chỉ số khởi đầu, thì câu trả lời chính là chỉ số đầu này. Bởi vậy

```
    int index = fruit.indexOf('a', 5);
```

sẽ trả lại 5.

8.7 Lặp quay vòng và đếm

Chương trình dưới đây đếm số lần xuất hiện của chữ 'a' trong một chuỗi:

```
    String fruit = "banana";

    int length = fruit.length();

    int count = 0;

    int index = 0;

    while (index < length) {

      if (fruit.charAt(index) == 'a') {

        count = count + 1;
```

```
    }
    index = index + 1;
  }
  System.out.println(count);
```

Chương trình này cho thấy một cách viết quen tay thông dụng, đó là một **biến đếm**. Biến count được khởi tạo bằng không và sau đó tăng thêm một ứng với mỗi lần ta tìm thấy một chữ 'a'. Việc **tăng** ở đây là chỉ tăng thêm một đơn vị; nó ngược lại với thao tác **giảm**. Khi ta thoát khỏi vòng lặp, count sẽ chứa kết quả, đó là tổng số các chữ a.

8.8 Các toán tử tăng và giảm

Tăng và giảm là những thao tác thông dụng đến nỗi Java có những toán tử riêng cho chúng. Toán tử ++ cộng thêm một vào giá trị hiện thời của một int hay char. -- thì trừ đi một. Hai toán tử trên đều không có tác dụng đối với double, boolean hay String.

Về khía cạnh kĩ thuật, sẽ hoàn toàn hợp lệ nếu ta tăng một biến rồi đồng thời sử dụng nó trong một biểu thức. Chẳng hạn, bạn có thể thấy lệnh kiểu như sau:

```
  System.out.println(i++);
```

Nhìn vào câu lệnh này, thật không rõ là liệu việc tăng sẽ tiến hành trước hay sau khi giá trị được in ra. Bởi vì những biểu thức thế này có xu hướng gây nhầm lẫn, tôi khuyên bạn nên hạn chế sử dụng chúng. Thậm chí, để hạn chế hơn nữa, tôi sẽ không nói cho bạn biết kết quả bằng bao nhiêu. Nếu thực sự muốn biết, bạn có thể thử xem.

Bằng cách dùng toán tử tăng, ta có thể viết lại mã lệnh đếm chữ:\

```
  int index = 0;
  while (index < length) {
    if (fruit.charAt(index) == 'a') {
      count++;
    }
    index++;
  }
```

Một lỗi sai thường gặp là viết lệnh kiểu như sau:

```
  index = index++; // SAI!!
```

Tuy nhiên, cách này lại hợp lệ về mặt cú pháp, nên trình biên dịch sẽ không cảnh báo bạn. Hiệu ứng của lệnh này là giữ nguyên giá trị của index. Đây thường là một lỗi khó tìm ra.

Hãy nhớ, bạn có thể viết index = index+1, hay index++, nhưng đừng trộn lẫn hai cách viết này.

8.9 String có tính không đổi

Như đã đọc tài liệu về các phương thức của String, có thể bạn phát hiện ra hai phương thức toUpperCase và toLowerCase. Hai phương thức này thường gây nhầm lẫn, vì chúng có tên gọi nghe như thể chúng có tác dụng thay đổi chuỗi hiện có. Song thực ra, chẳng có phương thức nào nói chung và hai phương thức này nói riêng, có thể thay đổi được chuỗi, vì chuỗi có **tính không đổi**.

Khi bạn kích hoạt toUpperCase đối với một String, bạn sẽ thu được một String *mới* làm kết quả trả lại. Chẳng hạn:

```
String name = "Alan Turing";

String upperName = name.toUpperCase();
```

Sau khi dòng lệnh thứ hai được thực thi, upperName sẽ chứa giá trị "ALAN TURING", còn name vẫn chứa "Alan Turing".

8.10 String có tính không so sánh được

Ta thường cần so sánh hai chuỗi để xem chúng có giống nhau không, hay chuỗi nào sẽ xếp trước theo thứ tự bảng chữ cái. Thật tuyệt nếu ta sử dụng được các toán tử so sánh như == và >, song ta không thể làm vậy.

Để so sánh các String, ta phải dùng các phương thức equals và compareTo. Chẳng hạn:

```
String name1 = "Alan Turing";

String name2 = "Ada Lovelace";

if (name1.equals (name2)) {

  System.out.println("hai tên này là một.");

}

int flag = name1.compareTo (name2);

if (flag == 0) {

  System.out.println("Hai tên gọi này là một.");

} else if (flag < 0) {

  System.out.println("tên 1 xếp trước tên 2.");

} else if (flag > 0) {

  System.out.println("tên 2 xếp trước tên 1.");

}
```

Cú pháp ở đây hơi kì quặc. Để so sánh hai String, bạn phải kích hoạt một phương thức lên một chuỗi rồi truyền chuỗi còn lại làm tham số.

Giá trị trả về từ equals thật dễ hiểu; true nếu hai chuỗi có chứa cùng các kí tự, và false trong trường hợp còn lại.

Giá trị trả về từ compareTo lại kì quặc. Đó là khoảng cách giữa hai chữ cái đầu tiên có sự khác biệt ở hai chuỗi. Nếu hai chuỗi bằng nhau thì khoảng cách này bằng 0. Nếu chuỗi thứ nhất (chuỗi mà ta kích hoạt phương thức lên) đứng trước theo thứ tự bảng chữ cái, thì khoảng cách này có giá trị âm. Ngược lại, khoảng cách có giá trị dương. Trong trường hợp này, giá trị trả lại bằng 8, vì chữ cái thứ hai của “Ada” đi trước chữ cái thứ hai của “Alan” là 8 vị trí.

Để trọn vẹn, tôi cũng nói thật rằng việc dùng toán tử == đối với các Strings là *hợp lệ* nhưng ít khi *đúng đắn*. Tôi sẽ giải thích lí do trong Mục 13.4; song bây giờ thì chưa.

8.11 Thuật ngữ

đối tượng:

> *Một tập hợp các dữ liệu có liên quan cùng với một tập các phương thức hoạt động với nó. Các đối tượng mà ta dùng cho đến giờ gồm có String, Bug, Rock, và những đối tượng khác trong GridWorld.*

chỉ số:
Một biến hay giá trị được dùng để chọn một trong các thành viên (phần tử) của một tập hợp được xếp thứ tự, như chọn kí tự từ một chuỗi.

biệt lệ:
Một lỗi khi thực thi chương trình.

phát:
Gây nên một biệt lệ.

lần vết ngăn xếp:
Một bản báo cáo cho thấy trạng thái chương trình khi có biệt lệ xảy ra.occurs.

nguyên mẫu:
Dòng đầu tiên của một phương thức, trong đó quy định tên, các tham số và kiểu trả lại.

duyệt:
Việc lặp qua tất cả mọi phần tử của một tập hợp nhằm thực hiện một công việc tương tự đối với từng phần tử.

biến đếm:
Một biến dùng để đếm thứ gì đó; biến này thường được khởi tạo bằng không sau đó tăng thêm.

tăng:
Việc tăng giá trị của biến thêm một đơn vị. Toán tử tăng trong Java là ++.

giảm:
Việc giảm giá trị của biến thêm đi đơn vị. Toán tử giảm trong Java là --.

8.12 Bài tập

Bài tập 1 Hãy viết một phương thức nhận vào một String làm đối số rồi in tất cả các chữ cái theo chiều ngược lại trên cùng một dòng.

Bài tập 2 Hãy đọc nội dung lần vết ngăn xếp ở Mục 8.4 rồi trả lời những câu hỏi sau:

- Những loại biệt lệ nào đã xảy ra, và những biệt lệ này được định nghĩa trong các gói (package) nào?
- Giá trị nào của chỉ số gây nên biệt lệ?
- Phương thức nào phát ra biệt lệ, và phương thức đó được định nghĩa ở đâu?
- Phương thức nào kích hoạt charAt?
- Trong BadString.java, charAt được kích hoạt tại dòng số mấy?

Bài tập 3 Hãy bao bọc đoạn mã ở Mục 8.7 vào một phương thức có tên countLetters, sau đó khái quát hoá sao cho nó chấp nhận các đối số là chuỗi và chữ cái cần đếm. Tiếp theo, viết lại phương thức sao cho nó sử dụng indexOf để định vị các chữ a, thay vì kiểm tra từng chữ cái một.

Bài tập 4 Mục đích của bài tập này là ôn lại phép bao bọc và khái quát hoá.

1. Hãy bao bọc đoạn mã lệnh sau, chuyển đổi nó thành một phương thức nhận vào đối số là một String rồi trả lại giá trị cuối cùng của count.
2. Mô tả ngắn gọn công dụng của phương thức vừa lập nên (mà không đi vào chi tiết các bước thực hiện như thế nào).
3. Bây giờ khi bạn đã khái quát hoá để mã lệnh hoạt động được với chuỗi bất kì rồi, bạn còn có thể khái quát hoá theo cách nào nữa?

```
String s = "((3 + 7) * 2)";

int len = s.length();

int i = 0;
```

```
    int count = 0;
    while (i < len) {
      char c = s.charAt(i);
      if (c == '(') {
        count = count + 1;
      } else if (c == ')') {
        count = count - 1;
      }
      i = i + 1;
    }
    System.out.println(count);
```

Bài tập 5 Mục đích của bài tập này là khám phá những kiểu dữ liệu trong Java và điền vào một số thông tin chi tiết chưa được đề cập đến trong chương này.

1. Hãy tạo nên một chương trình mới có tên Test.java rồi viết một phương thức main có chứa những biểu thức có kết hợp nhiều kiểu dữ liệu bằng toán tử +. Chẳng hạn, điều gì sẽ xảy ra nếu bạn "cộng" một String và một char? Liệu nó có thực hiện tính tổng hay kết nối? Kiểu của kết quả sẽ là gì? (Bạn xác định được kiểu của kết quả như thế nào?)
2. Hãy sao chép lại và mở rộng bảng dưới đây rồi điền vào nó. Trong từng ô giao cắt giữa hai kiểu dữ liệu, bạn cần phải xác định xem liệu có hợp lệ nếu dùng toán tử + với những kiểu này không, phép toán nào được thực hiện (cộng hay kết nối), và kiểu kết quả sẽ là gì.

	boolean	char	int	String
boolean				
char				
int				
String				

3. Hãy tưởng tượng xem các nhà thiết kế nên ngôn ngữ Java đã lựa chọn thế nào khi họ điền vào bảng trên. Trong số các ô điền, có bao nhiêu ô dường như là lựa chọn chắc chắn? Có bao nhiêu ô dường như là lựa chọn tuỳ ý mà có vài phương án tốt như nhau? Có bao nhiêu ô có vẻ còn chứa đựng vấn đề?
4. Sau đây là một câu đố: thông thường, câu lệnh x++ đúng bằng x = x + 1. Nhưng nếu x là một char, thì nó sẽ không còn đúng! Trong trường hợp này, x++ là hợp lệ, nhưng x = x + 1 sẽ gây ra lỗi. Hãy thử lại và xem thông báo lỗi là gì, và sau đó xem liệu bạn có thể hình dung được điều gì đang diễn ra không.

Bài tập 6 Kết quả của chương trình dưới đây là gì? Bằng một câu, hãy mô tả xem mystery làm gì (chứ không phải các bước thực hiện ra sao).

```
public class Mystery {
  public static String mystery(String s) {
    int i = s.length() - 1;
    String total = "";
    while (i >= 0 ) {
      char ch = s.charAt(i);
      System.out.println(i + " " + ch);
      total = total + ch;
```

```
      i--;
    }
    return total;
  }
  public static void main(String[] args) {
    System.out.println(mystery("Allen"));
  }
}
```

Bài tập 7 Một người bạn cho bạn xem phương thức sau đây và diễn giải rằng nếu number là số có hai chữ số bất kì, thì chương trình sẽ in các chữ số theo chiều ngược lại. Người ấy khẳng định rằng nếu number là 17, thì phương thức sẽ cho ra kết quả bằng 71. Liệu người đó có đúng không? Nếu không, hãy giải thích chương trình thực sự làm gì và sửa chữa để nó cho kết quả đúng.

```
    int number = 17;
    int lastDigit = number%10;
    int firstDigit = number/10;
    System.out.println(lastDigit + firstDigit);
```

Bài tập 8 Kết quả của chương trình sau là gì?

```
public class Enigma {
  public static void enigma(int x) {
    if (x == 0) {
      return;
    } else {
      enigma(x/2);
    }
    System.out.print(x%2);
  }
  public static void main(String[] args) {
    enigma(5);
    System.out.println("");
  }
}
```

Hãy giải thích ngắn gọn bằng 4-5 từ xem phương thức enigma thực sự làm điều gì.

Bài tập 9

1. Hãy lập một chương trình mới có tên Palindrome.java.
2. Viết một phương thức có tên first nhận vào một String rồi trả lại chữ cái đầu tiên, và một phương thức last để trả lại chữ cái cuối cùng.
3. Viết một phương thức có tên middle nhận vào một String rồi trả lại một chuỗi con có chứa mọi thứ trừ hai chữ cái đầu và cuối. Gợi ý: hãy đọc tài liệu về phương thức substring trong lớp String. Hãy chạy một vài phép thử để chắc rằng bạn hiểu rõ cách hoạt động của substring trước khi thử viết middle. Điều gì

sẽ xảy ra nếu bạn kích hoạt middle lên một chuỗi chỉ có hai chứ cái? Một chữ cái? Không có chữ cái nào?

4. Cách định nghĩa thông thường của một palindrome là một từ mà đọc xuôi ngược đều giống nhau, chẳng hạn "otto" và "palindromeemordnilap." Một cách khác để định nghĩa một thuộc tính như thế baft là quy định một cách kiểm tra thuộc tính đó. Chẳng hạn, ta có thể nói "một chữ cái là một palindrome, và một từ hai chữ là một palindrome nếu hai chữ cái của nó giống nhau, và một từ bất kì khác là một palindrome nếu chữ cái đầu giống chữ cái cuối và khúc giữa cũng là một palindrome." Hãy viết một phương thức đệ quy có tên isPalindrome nhận vào một String và trả lại một boolean cho biết từ đó có phải là palindrome hay không.
5. Một khi bạn đã có đoạn mã để kiểm tra palindrome, hãy tìm cách đơn giản hoá nó bằng cách giảm số điều kiện trong phép kiểm tra. Gợi ý: việc lấy định nghĩa chuỗi rỗng cũng là palindrome có thể giúp ích.
6. Hãy viết ra trên giấy một chiến lược có tính lặp để kiểm tra palindrome. Có một số phương án khả dĩ, bởi vậy bạn hãy đảm bảo chắc chắn một kế hoạch rõ ràng trước khi bắt đầu viết mã lệnh.
7. Hãy tạo lập chiến lược bạn chọn thành một phương thức có tên isPalindromeIter.
8. Câu hỏi phụ: Phụ lục B có mã lệnh để đọc một danh sách các từ vựng từ một file. Hãy đọc một danh sách các từ rồi in ra những palindrome.

Bài tập 10 Một từ được gọi là "abecedarian" nếu các chữ cái trong từ đó xuất hiện theo thứ tự bảng chữ cái. Chẳng hạn, sau đây là tất cả những từ abecedarian gồm 6 chữ cái trong tiếng Anh.

> abdest, acknow, acorsy, adempt, adipsy, agnosy, befist, behint, beknow, bijoux, biopsy, cestuy, chintz, deflux, dehors, dehort, deinos, diluvy, dimpsy

1. Hãy miêu tả một quy trình kiểm tra xem một từ (String) cho trước là abecedarian hay không, nếu coi rằng từ đó chỉ gồm các chữ cái thường. Quy trình này có thể mang tính lặp hay đệ quy.
2. Tạo dựng quy trình trên thành một phương thức mang tên isAbecedarian.

Bài tập 11 Một dupledrome là một từ chỉ chứa các chữ cái ghép đôi, chẳng hạn như "llaammaa" hay "ssaabb". Tôi đề ra giả thiết ràng trong tiếng Anh thông dụng không hề có dupledrome nào. Để kiểm chứng giả thiết đó, tôi muốn có chương trình đọc lần lượt các từ vựng từ một cuốn từ điển rồi kiểm tra xem từ đó có phải là dupledrome hay không. Hãy viết một phương thức mang tên isDupledrome nhận vào một String rồi trả lại một boolean để cho biết từ đó có phải là dupledrome không.

Bài tập 12

1. Vòng giải mã Captain Crunch hoạt động bằng cách lấy mỗi chữ cái trong một chuỗi rồi cộng 13 vào nó. Chẳng hạn, 'a' trở thành 'n' và 'b' trở thành 'o'. Đến cuối, các chữ cái "quay vòng lại", bởi vậy 'z' trở thành 'm'. Hãy viết một phương thức nhận vào một String rồi trả lại một String mới có chứa chuỗi sau mã hoá. Bạn cần coi ràng String ban đầu chỉ chứa các chữ in, chữ thường, dấu cách, mà không có dấu chấm phẩy gì khác. Các chữ thường thì được mã hoá thành chữ thường, chữ in thành chữ in. Bạn không được mã hoá các dấu cách.
2. Hãy khái quát hoá phương thức Captain Crunch sao cho thay vì cộng 13 vào từng chữ cái, nó có thể cộng thêm bất kì số nào. Bây giờ bạn có thể mã hoá bằn cách cộng 13 rồi giải mã bằng cách cộng -13. Hãy thử làm điều này.

Bài tập 13 Nếu bạn đã giải các bài tập GridWorld trong Chương 5, có thể bạn sẽ thích bài tập này. Mục đích là dùng toán lượng giác để khiến các con bọ (Bug) đuổi bắt lẫn nhau. Hãy sao chép file BugRunner.java thành ChaseRunner.java rồi nhập nó vào môi trường phát triển của bạn. Trước khi

thay đổi bất cứ điều gì, hãy kiểm tra đảm bảo rằng bạn biên dịch và chạy được chương trình.

- Tạo nên hai Bug, một con màu đỏ và một màu xanh lam.
- Viết một phương thức mang tên distance nhận vào hai Bug rồi tính khoảng cách giữa chúng. Hãy nhớ rằng bạn có thể lấy được toạ độ x của một Bug như sau:

```
int x = bug.getLocation().getCol();
```

- Viết một phương thức mang tên turnToward nhận vào hai Bug rồi quay mặt một con hướng đến con kia. GỢI Ý: dùng Math.atan2, nhưng hãy nhớ rằng kết quả theo đơn vị radian, bởi vậy bạn phải chuyển sang độ. Ngoài ra, đối với Bug, 0 độ là hướng Bắc chứ không phải hướng Đông.
- Viết một phương thức mang tên moveToward nhận vào hai Bug, quay mặt con thứ nhất về phía con thứ hai, rồi di chuyển con thứ nhất, nếu có thể.
- Viết một phương thức mang tên moveBugs nhận hai Bug và một số nguyên n, rồi di chuyển một con Bug về phía con kia n lần. Bạn có thể viết phương thức này theo cách đệ quy, hoặc dùng một vòng lặp while.
- Kiểm tra từng phương thức vừa viết ở trên ngay khi bạn phát triển chúng. Khi chúng đều hoạt động được, hãy tìm mọi cơ hội cải thiện. Chẳng hạn, nếu bạn có mã lệnh dư thừa trong distance và turnToward, thì bạn có thể bao bọc đoạn mã lệnh lặp lại vào trong một phương thức.

Chương 9: Đối tượng có thể biến đổi

Trở về Mục lục cuốn sách

String là các đối tượng, song chúng là đối tượng không điển hình bởi lẽ

- Chúng không thể biến đổi.
- Chúng không có thuộc tính.
- Bạn không bắt buộc phải dùng new để tạo nên một chuỗi mới.

Trong chương này, ta dùng hai đối tượng thuộc thư viện Java, là đối tượng Point và Rectangle (điểm và hình chữ nhật). Song trước hết, tôi muốn nói rõ rằng những điểm và hình chữ nhật này không phải là những đối tượng đồ hoạ xuất hiện trên mà hình. Chúng chỉ là những giá trị có chứa số liệu, cũng như các int và double. Giống những giá trị khác, chúng được sử dụng bên trong chương trình để thực hiện tính toán.

9.1 Các gói chương trình

Các thư viện Java được chia thành các **gói**, trong đó có java.lang là gói chứa hầu hết các lớp mà ta dùng cho đến giờ, và java.awt, tên đầy đủ **Abstract Window Toolkit** (AWT), là gói chứa các lớp gồm cửa sổ, nút bấm, đồ hoạ, v.v.

Để dùng một lớp được định nghĩa trong gói khác, bạn phải **nhập** nó. Point và Rectangle nằm trong gói java.awt, bởi vậy để nhập chúng ta làm như sau:

```
import java.awt.Point;

import java.awt.Rectangle;
```

Tất cả câu lệnh import đều xuất hiện ở điểm đầu chương trình, bên ngoài lời định nghĩa lớp.

Các lớp trong java.lang, như Math và String, được nhập một cách tự động, bởi vậy từ trước đến giờ ta chưa cần dùng đến câu lệnhimport nào.

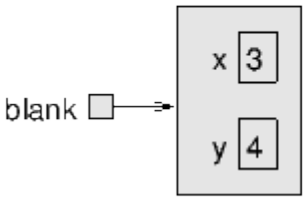
9.2 Đối tượng Point

Một điểm là hai con số (toạ độ) mà ta coi chúng hợp nhất như một đối tượng đơn lẻ. Theo kí hiệu toán học, điểm thường được viết trong cặp ngoặc tròn, với dấu phẩy phân cách giữa các toạ độ. Chẳng hạn, (0, 0) chỉ định gốc toạ độ, còn (x, y) chỉ định điểm cách điểm gốc x đơn vị về bên tay phải và y đơn vị lên trên.

Trong Java, một điểm được biểu diễn bởi một đối tượng Point. Để tạo nên một điểm mới, bạn phải dùng đến new:

```
Point blank;

blank = new Point(3, 4);
```

Dòng thứ nhất là một lời khai báo biến thông dụng: blank có kiểu Point. Dòng thứ hai kích hoạt new, quy định kiểu của đối tượng mới, và cung cấp các đối số. Ở đây các đối số là toạ độ của điểm mới, (3, 4). Kết quả của new là một **tham chiếu** đến điểm mới, vì vậy blank chứa một tham chiếu đến đối tượng mới tạo nên. Có một cách tiêu chuẩn để sơ đồ hoá phép gán này, xem trên hình vẽ.



Như thường lệ, tên biến, blank, được ghi bên ngoài ô và giá trị của nó ở trong ô. Với trường hợp này, giá trị là một tham chiếu, và được biểu diễn bởi một mũi tên. Mũi tên này chỉ đến đối tượng mà ta tham chiếu tới.
Ô lớn biểu diễn đối tượng mới tạo lập cùng với hai giá trị bên trong. Các tên gọi x và y là các tên của **biến thực thể**.

Xét tổng thể, tất cả các biến, giá trị, và đối tượng trong một chương trình thì được gọi là **trạng thái**. Những biểu đồ như thế này, dùng để biểu diễn trạng thái chương trình, được gọi là **biểu đồ trạng thái**. Khi chương trình chạy, trạng thái của nó thay đổi; bởi vậy bạn nên coi biểu đồ trạng thái như một ảnh chụp tại một thời điểm cụ thể trong quá trình thực thi.

9.3 Các biến thực thể

Những đơn vị thông tin hợp thành đối tượng được gọi là các biến thực thể vì từng đối tượng, vốn là một **thực thể** cho kiểu của nó, có một bản sao riêng của biến thực thể này.

Cũng giống như ngăn trước [nơi để CD, giấy tờ ở ghế ngồi phía trước] của một chiếc xe hơi. Mỗi cái xe là thực thể của kiểu "xe hơi," và từng chiếc xe có ngăn riêng của nó. Nếu bạn yêu cầu tôi lấy đồ từ ngăn trước của xe hơi bạn đang dùng, thì hãy cho tôi biết xe bạn đang dùng là xe nào.

Tương tự như vậy, nếu bạn muốn đọc một giá trị từ biến thực thể, bạn phải chỉ định đối tượng mà bạn cần lấy giá trị từ đó. Ở Java, điều này được thực hiện bằng cách dùng "kí pháp dấu chấm."

```
    int x = blank.x;
```

Biểu thức blank.x nghĩa là "đến đối tượng mà blank chỉ tới, rồi lấy giá trị của x." Trong trường hợp này ta gán giá trị đó vào một biến địa phương có tên là x. Không hề có xung đột gì giữa biến địa phương tên x này và biến thực thể mang tên x. Mục đích của kí pháp dấu chấm là để quy định rõ ràng xem biến *nào* mà bạn đang tham chiếu tới.
Bạn có thể dùng kí pháp dấu chấm làm một thành phần trong bất kì biểu thức Java nào, bởi vậy các biểu thức sau đều hợp lệ.

```
    System.out.println(blank.x + ", " + blank.y);

    int distance = blank.x * blank.x + blank.y * blank.y;
```

Dòng thứ nhất in ra 3, 4; dòng thứ hai tính giá trị của 25.

9.4 Đối tượng trong vai trò của tham số

Bạn có thể truyền đối tượng như những tham số theo cách thông thường. Chẳng hạn:

```
  public static void printPoint(Point p) {
```

```
    System.out.println("(" + p.x + ", " + p.y + ")");
  }
```

Phương thức này nhận một điểm làm đối số rồi in nó ra dưới định dạng tiêu chuẩn. Nếu bạn kích hoạt printPoint(blank), nó sẽ in ra (3, 4). Thực tế là Java đã có sẵn một phương thức để in ra các Point. Nếu kích hoạt System.out.println(blank), bạn sẽ nhận được

```
java.awt.Point[x=3,y=4]
```

Đây là định dạng tiêu chuẩn mà Java dùng để in các đối tượng. Nó in ra tên của kiểu dữ liệu, tiếp theo là các tên và giá trị của những biến thực thể.

Một ví dụ thứ hai là ta có thể viết lại phương thức distance ở Mục 6.2 để nó nhận hai Point làm tham số thay vì bốn double.

```
  public static double distance(Point p1, Point p2) {
    double dx = (double)(p2.x - p1.x);
    double dy = (double)(p2.y - p1.y);
    return Math.sqrt(dx*dx + dy*dy);
  }
```

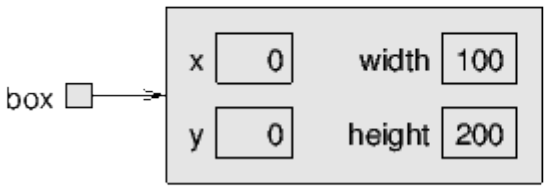
Các phép đổi kiểu dữ liệu ở đây đều không thật sự cần thiết. Tôi chỉ viết vào để tự nhắc rằng các biến thực thể trong một Point đều là các số nguyên.

9.5 Hình chữ nhật

Rectangle (hình chữ nhật) cũng giống như các điểm, chỉ khác rằng chúng có bốn biến thực thể: x, y, width (bề rộng) và height (chiều cao). Ngoài điều này ra thì những thứ còn lại vẫn y nguyên. Ví dụ sau đây tạo nên một đối tượng Rectangle rồi khiến box tham chiếu đến nó.

```
    Rectangle box = new Rectangle(0, 0, 100, 200);
```

Hình vẽ này mô tả hiệu ứng của lệnh gán nêu trên.



Nếu in box ra, bạn nhận được

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

Một lần nữa, đây là kết quả của một phương thức Java vốn biết cách in những đối tượng Rectangle.

9.6 Đối tượng với vai trò là kiểu được trả lại

Bạn có thể viết những phương thức trả lại đối tượng. Chẳng hạn, findCenter lấy một Rectangle làm đối số rồi trả lại một Point có chứa toạ độ của tâm Rectangle:

```
  public static Point findCenter(Rectangle box) {
    int x = box.x + box.width/2;
    int y = box.y + box.height/2;
```

```
    return new Point(x, y);
}
```

Lưu ý rằng bạn có thể dùng new để tạo nên một đối tượng mới, và rồi lập tức dùng kết quả này làm giá trị trả lại.

9.7 Đối tượng có tính thay đổi

Bạn có thể thay đổi nội dung của một đối tượng bằng cách viết lệnh gán cho một trong số những biến thực thể của nó. Chẳng hạn, để "dịch chuyển" một hình chữ nhật mà không làm thay đổi kích thước của nó, bạn có thể chỉnh sửa các giá trị x và y:

```
    box.x = box.x + 50;

    box.y = box.y + 100;
```

Kết quả được biểu diễn trên hình:

Ta có thể bao bọc đoạn mã lệnh trên vào một phương thức rồi khái quát hoá nó để dịch chuyển hình chữ nhật đi một khoảng cách bất kì:

```
  public static void moveRect(Rectangle box, int dx, int dy) {

    box.x = box.x + dx;

    box.y = box.y + dy;

  }
```

Các biến dx và dy chỉ định khoảng cách dịch chuyển hình theo từng hướng riêng. Việc kích hoạt phương thức này có ảnh hưởng làm thay đổi Rectangle được truyền vào dưới dạng tham số.

```
    Rectangle box = new Rectangle(0, 0, 100, 200);

    moveRect(box, 50, 100);

    System.out.println(box);
```

sẽ in ra java.awt.Rectangle[x=50,y=100,width=100,height=200].
Việc thay đổi các đối tượng bằng cách truyền chúng làm tham số cho các phương thức mặc dù có thể hữu ích, song nó cũng có thể gây khó khăn cho việc gỡ lỗi vì không phải lúc nào cũng dễ thấy là việc kích hoạt một phương thức có thay đổi các đối số của nó hay không. Về sau, tôi sẽ thảo luận những ưu nhược điểm của phong cách lập trình này.

Java có các phương thức thao tác với Point và Rectangle. Bạn có thể đọc tài liệu ở http://download.oracle.com/javase/6/docs/api/java/awt/Point.html và http://download.oracle.com/javase/6/docs/api/java/awt/Rectangle.html.
Chẳng hạn, translate có hiệu ứng tựa như moveRect, song thay vì phải truyền Rectangle làm đối số, bạn lại dùng kí pháp dấu chấm:

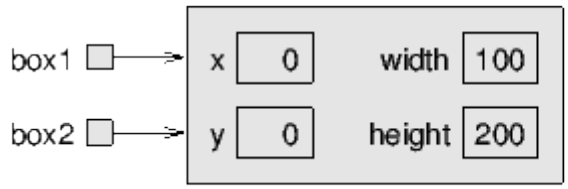
```
    box.translate(50, 100);
```

9.8 Aliasing

Hãy nhớ rằng khi bạn gán một đối tượng vào cho một biến, bạn đang gán một *tham chiếu* đến đối tượng. Hoàn toàn có thể có nhiều biến cùng tham chiếu tới một đối tượng. Chẳng hạn, đoạn mã sau:

```
    Rectangle box1 = new Rectangle(0, 0, 100, 200);
    Rectangle box2 = box1;
```

tạo nên một biểu đồ trạng thái trông như sau:



box1 và box2 cùng chỉ đến một đối tượng. Nói cách khác, đối tượng này có hai tên gọi, box1 và box2. Việc người nào đó dùng hai tên được gọi là **aliasing** (dùng bí danh). Với đối tượng cũng như vậy.
Khi hai biến được dùng bí danh, bất kì sự thay đổi nào ảnh hưởng tới biến này thì cũng ảnh hưởng tới biến kia. Chẳng hạn:

```
    System.out.println(box2.width);
    box1.grow(50, 50);
    System.out.println(box2.width);
```

Dòng lệnh thứ nhất in ra 100, vốn là bề rộng của Rectangle được tham chiếu qua biến box2. Dòng thứ hai kích hoạt phương thứcgrow lên box1, để mở rộng Rectangle thêm 50 điểm ảnh theo mỗi chiều (hãy đọc tài liệu để biết thêm thông tin). Hiệu ứng được cho thấy ở hình vẽ dưới đây:

Bất kể thay đổi nào thực hiện đối với box1 thì cũng ảnh hưởng đến box2. Do vậy, giá trị được in ra bởi dòng lệnh thứ ba là 200, bề rộng của hình chữ nhật sau khi mở rộng. (Nói thêm, việc các toạ độ của một Rectangle nhận giá trị âm là hoàn toàn hợp lệ.)
Từ ví dụ đơn giản này bạn có thể thấy rằng mã lệnh có chứa bí dạng nhanh chóng khiến ta nhầm lẫn và có thể khó gỡ lỗi. Nói chung, nên tránh dùng bí danh hoặc dùng thật cẩn thận.

9.9 null

Khi bạn tạo nên một biến đối tượng, hãy nhớ rằng bạn đang tạo nên một *tham chiếu* đến đối tượng. Trước khi bạn khiến cho một biến chỉ tới đối tượng, thì giá trị của biến vẫn là null. null là một giá trị đặc biệt (và cũng là một từ khoá trong Java) có nghĩa là "không có đối tượng."
Lời khai báo Point blank; thì tương đương với lệnh khởi tạo sau

```
    Point blank = null;
```

và được biểu diễn bởi biểu đồ trạng thái sau:

Giá trị null được biểu thị bằng một hình vuông nhỏ không kèm theo mũi tên.
Nếu bạn cố thử dùng một đối tượng null, qua việc truy cập một biến thực thể hay kích hoạt một

phương thức, thì Java sẽ phát một biệt lệ có tên NullPointerException, in một thông báo lỗi và kết thúc chương trình.

```
Point blank = null;

int x = blank.x; // NullPointerException

blank.translate(50, 50); // NullPointerException
```

Mặt khác, sẽ hoàn toàn hợp lệ nếu ta truyền một đối tượng null làm đối số hoặc nhận một null làm giá trị trả về. Thực ra, điều này rất thông dụng, với mục đích chẳng hạn là biểu diễn một tập hợp rỗng hay để chỉ một điều kiện có lỗi.

9.10 Thu dọn rác

Ở Mục 9.8 ta đã nói về những gì đã xảy ra khi nhiều biến cùng tham chiếu tới một đối tượng. Thế còn điều gì sẽ xảy ra khi *không có* biến nào tham chiếu đến đối tượng? Chẳng hạn:

```
Point blank = new Point(3, 4);

blank = null;
```

Dòng thứ nhất tạo ra một đối tượng Point mới rồi khiến cho blank tham chiếu đến nó. Dòng thứ hai sửa chữa blank để cho, thay vì tham chiếu đến đối tượng, nó không tham chiếu đến gì cả (hay tham chiếu đến đối tượng null).

Nếu không có ai tham chiếu đến đối tượng, thì cũng chẳng ai có thể đọc hay ghi giá trị bất kì nào từ nó, hay kích hoạt một phương thức lên nó. Hệ quả là, nó sẽ ngừng tồn tại. Ta có thể vẫn giữ đối tượng này trong bộ nhớ, song làm như vậy chỉ tốn dung lượng; bởi vậy khi chương trình chạy, theo định kì hệ thống sẽ tìm kiếm các đối tượng lang thang rồi thu hồi lại nó, theo một quá trình mang tên **thu dọn rác**. Sau này, dung lượng nhớ bị chiếm bởi đối tượng sẽ về tay người dùng để phục vụ đối tượng mới.

Bạn không cần phải làm bất cứ điều gì để tiến hành thu dọn rác, và nói chung bạn sẽ không nhận thức được quá trình này. Song bạn cần biết rằng quá trình luôn được ngầm chạy một cách định kì.

9.11 Các đối tượng và kiểu nguyên thủy

Trong Java có hai loại kiểu dữ liệu, kiểu nguyên thủy và kiểu đối tượng. Kiểu nguyên thủ, như int và boolean đều bắt đầu bằng chữ viết thường; kiểu đối tượng bắt đầu bằng chữ viết in. Sự phân biệt này rất có ích vì chúng nhắc ta một số điểm khác nhau giữa chúng:

- Khi khai báo một biến nguyên thủy, bạn được một dung lượng lưu trữ dành cho giá trị nguyên thủy. Khi bạn khai báo một biến đối tượng, bạn nhận được một dung lượng chứa tham chiếu tới đối tượng. Để giành được dung lượng cho bản thân đối tượng đó, bạn phải dùng đến new.
- Nếu bạn không khởi tạo một kiểu nguyên thủy, thì nó sẽ được điền giá trị mặc định tùy theo kiểu đó là gì. Chẳng hạn, 0 với trường hợp int và false với boolean. Giá trị mặc định của kiểu đối tượng là null, nghĩa là không có đối tượng nào.
- Các biến nguyên thủy tách biệt hoàn toàn, theo nghĩa bất cứ bạn làm gì trong một phương thức này sẽ không ảnh hưởng đến một biến ở phương thức khác. Các biến đối tượng thì lại cần phải khéo léo khi thao tác vì chúng không được biệt lập như vậy. Nếu bạn truyền một tham chiếu đến đối tượng để làm

đối số, thì phương thức mà bạn kích hoạt có thể sẽ thay đổi đối tượng, và trong trường hợp này bạn sẽ thấy hiệu ứng. Dĩ nhiên, đó có thể là điều hay, song bạn cần nhận thức được việc này.
Còn một điểm khác biệt giữa kiểu nguyên thủy và kiểu đối tượng. Bạn không thể bổ sung kiểu nguyên thủy mới nào vào Java (trứ khi bạn là thanh viên trong hội đồng tiêu chuẩn), nhưng bạn có thể tạo nên kiểu đối tượng mới! Bạn sẽ biết cách làm như vậy trong chương sau.

9.12 Thuật ngữ

gói:
Một tập hợp các lớp. Các lớp Java được tổ chức thành các gói.
AWT:
Abstract Window Toolkit, một trong các gói Java lớn nhất và thông dụng nhất.
thực thể:
Ví dụ lấy từ một thể loại nào đó. Con mèo nhà tôi là một thực thể thuộc thể loại "động vật họ miêu." Mỗi đối tượng đều là thực thể của một lớp nào đó.
biến thực thể:
Một trong số các đơn vị dữ liệu được đặt tên để cấu thành một đối tượng. Từng đối tượng (thực thể) đều có bản sao riêng các biến thực thể trong lớp mà nó thuộc vào.
tham chiếu:
Một giá trị để chỉ định một đối tượng. Trên sơ đồ trạng thái, một tham chiếu xuất hiện dưới dạng hình mũi tên.
aliasing (bí danh):
Tình trạng khi có nhiều biến cùng tham chiếu tới một đối tượng.
thu dọn rác:
Quá trình tìm các đối tượng không có tham chiếu và thu hồi dung lượng bộ nhớ mà chúng chiếm giữ.
trạng thái:
Một hình thức diễn tả đầy đủ tất cả các biến và đối tượng cùng những giá trị của chúng tại một thời điểm trong khi chương trình được thực thi.
sơ đồ trạng thái:
Một hình ảnh "chụp lại" trạng thái của chương trình.

9.13 Bài tập

Bài tập 1

1. Với chương trình sau đây, hãy vẽ một sơ đồ ngăn xếp cho thấy các biến địa phương và các đối số của main và riddle, rồi cho thấy mọi đối tượng mà hai biến này chỉ đến.
2. Kết quả của chương trình này là gì?

```
public static void main(String[] args) {
  int x = 5;
  Point blank = new Point(1, 2);
  System.out.println(riddle(x, blank));
  System.out.println(x); System.out.println(blank.x);
  System.out.println(blank.y);
}
```

```
public static int riddle(int x, Point p) {
  x = x + 7;
  return x + p.x + p.y;
}
```

Mục đích của bài tập này là để đảm bảo rằng bạn hiểu cơ chế truyền đối tượng làm tham số.

Bài tập 2

1. Với chương trình sau, hãy vẽ một biểu đồ ngăn xếp thể hiện trạng thái của chương trình ngay trước khi distance trả về. Hãy kèm theo tất cả các biến số và tham số cùng với những đối tượng mà các biến này tham chiếu tới.
2. Kết quả của chương trình này là gì?

```
public static double distance(Point p1, Point p2) {
  int dx = p1.x - p2.x;
  int dy = p1.y - p2.y;
  return Math.sqrt(dx*dx + dy*dy);
}
public static Point findCenter(Rectangle box) {
  int x = box.x + box.width/2;
  int y = box.y + box.height/2;
  return new Point(x, y);
}
public static void main(String[] args) {
  Point blank = new Point(5, 8);
  Rectangle rect = new Rectangle(0, 2, 4, 4);
  Point center = findCenter(rect);
  double dist = distance(center, blank);
  System.out.println(dist);
}
```

Bài tập 3 Phương thức grow thuộc về lớp Rectangle. Hãy đọc tài liệu ởhttp://download.oracle.com/javase/6/docs/api/java/awt/Rectangle.html#grow(int, int).

1. Kết quả của chương trình sau là gì?
2. Hãy vẽ một sơ đồ trạng thái chỉ ra trạng thái của chương trình ngay trước khi main kết thúc, trong đó bao gồm tất cả những biến địa phương cùng các đối tượng mà những biến này tham chiếu tới.
3. Ở điểm cuối của main, liệu p1 và p2 có cùng là bí danh không? Tại sao (không)?

```
public static void printPoint(Point p) {
  System.out.println("(" + p.x + ", " + p.y + ")");
}
public static Point findCenter(Rectangle box) {
  int x = box.x + box.width/2;
```

```
    int y = box.y + box.height/2;
    return new Point(x, y);
  }
  public static void main(String[] args) {
    Rectangle box1 = new Rectangle(2, 4, 7, 9);
    Point p1 = findCenter(box1);
    printPoint(p1);
    box1.grow(1, 1);
    Point p2 = findCenter(box1);
    printPoint(p2);
  }
```

Bài tập 4 Đến giờ có thể bạn đang tương tư về phương thức giai thừa, nhưng ta sẽ viết thêm một dạng mới.

1. Hãy tạo một chương trình mới có tên Big.java rồi viết một dạng lặp cho factorial.
2. In ra một bảng các số nguyên chạy từ 0 đến 30 cùng với giai thừa của chúng. Ở tầm khoảng 15, có thể bạn sẽ thấy kết quả không còn đúng nữa. Tại sao vậy?
3. BigIntegers là các đối tượng Java với khả năng biểu diễn những số nguyên lớn tùy ý. Không có giới hạn trên nào trừ giới hạn kích thước bộ nhớ và tốc độ xử lý. Hãy đọc tài liệu về BigIntegers athttp://download.oracle.com/javase/6/docs/api/java/math/BigInteger.html.
4. Để dùng được BigIntegers, bạn phải thêm dòng import java.math.BigInteger vào đầu chương trình của bạn.
5. Có vài cách tạo nên một BigInteger, nhưng tôi khuyên bạn cách dùng valueOf. Đoạn mã sau chuyển đổi một số nguyên thành BigInteger:
```
int x = 17;
BigInteger big = BigInteger.valueOf(x);
```
Hãy gõ đoạn mã lệnh này rồi chạy thử. Cố gắng in ra một BigInteger.

6. Vì BigIntegers không phải là kiểu nguyên thủy nên các toán tử toán học thông thường không thể thao tác với chúng. Thay vào đó, ta phải dùng những phương thức như add. Để cộng hai BigInteger, hãy kích hoạt add lên một số rồi truyền số kia làm đối số. Chẳng hạn:
```
BigInteger small = BigInteger.valueOf(17);
BigInteger big = BigInteger.valueOf(1700000000);
BigInteger total = small.add(big);
```
Hãy thử một số phương thức khác, như multiply và pow.
7. Chuyển đổi factorial sao cho nó tính toán với BigInteger rồi trả lại kết quả cũng là một BigInteger. Bạn có thể mặc kệ tham số—nó vẫn sẽ là một số nguyên.
8. Hãy thử in lại bảng bằng phương thức giai thừa mà bạn vừa sửa đổi. Liệu nó có đúng đến 30 không? Bạn có thể làm cho nó lớn đến bao nhiêu? Tôi đã tính giai thừa của tất cả các số từ 0 đến 999, nhưng vì máy tính của tôi khá chậm nên mất một lúc. Số cuối cùng, 999!, có tới 2565 chữ số.

Bài tập 5 Nhiều kĩ thuật mã hóa phụ thuộc vào khả năng nâng các số nguyên lớn lên những lũy thừa

nguyên. Sau đây là một phương thức thực hiện một kĩ thuật (tương đối) nhanh để tính lũy thừa số nguyên:

```
public static int pow(int x, int n) {
  if (n == 0) return 1;

  // tính x mũ n/2 bằng cách đệ quy
  int t = pow(x, n/2);

  // nếu n chẵn, kết quả là t bình phương
  // nếu n lẻ, kết quả là t bình phương nhân với x
  if (n%2 == 0) {
    return t*t;
  } else {
    return t*t*x;
  }
}
```

Vấn đề với phương thức này là ở chỗ nó chỉ hoạt động được nếu kết quả nhỏ hơn 2 tỷ. Hãy viết lại phương thức để kết quả là một BigInteger. Tuy vậy các tham số vẫn là số nguyên.
Bạn có thể dùng các phương thức cho BigInteger là add và multiply, song đừng dùng pow, như vậy sẽ chẳng còn gì để làm.

Bài tập 6 Nếu bạn thích đồ họa, bây giờ đúng là lúc đọc đến Phụ lục A rồi làm các bài tập ở đó.

Chương 10: Grid World, phần 2

Trở về Mục lục bài viết

Phần 2 của nghiên cứu cụ thể GridWorld có sử dụng một số đặc điểm mà ta chưa từng gặp, vì vậy bạn sẽ xem qua bây giờ và seau này sẽ xem xét kĩ hơn. Hãy nhớ lại rằng, bạn có thể tìm tài liệu cho các lớp GridWorld ở http://www.greenteapress.com/thinkapjava/javadoc/gridworld/.

Khi cài đặt GridWorld, bạn sẽ có một thư mục mang tên projects/boxBug, trong đó chứa BoxBug.java, BoxBugRunner.java và BoxBug.gif.

Hãy sao chép những file này vào thư mục hiện thời của bạn rồi nhập chúng vào môi trường phát triển. Có những chỉ dẫn trong tài liệu sau mà bạn có thể tham khảo: http://www.collegeboard.com/prod_downloads/student/testing/ap/compsci_a/ap07_gridworld_installation_guide.pdf.

Sau đây là mã lệnh lấy từ BoxBugRunner.java:

```
import info.gridworld.actor.ActorWorld;

import info.gridworld.grid.Location;

import java.awt.Color;

public class BoxBugRunner {

  public static void main(String[] args) {

    ActorWorld world = new ActorWorld();

    BoxBug alice = new BoxBug(6);

    alice.setColor(Color.ORANGE);

    BoxBug bob = new BoxBug(3);

    world.add(new Location(7, 8), alice);

    world.add(new Location(5, 5), bob);

    world.show();

  }

}
```

Ở đây mọi thứ có lẽ đều quen thuộc, ngoại trừ Location, thuộc về GridWorld, và đối tượng này tương đương với java.awt.Point.

BoxBug.java chứa lời định nghĩa lớp cho BoxBug.

```
public class BoxBug extends Bug {

  private int steps;

  private int sideLength;

  public BoxBug(int length) { steps = 0; sideLength = length; }

}
```

Dòng đầu tiên nói rằng lớp này mở rộng Bug, nghĩa là BoxBug là một dạng của Bug.

Hai dòng kế tiếp là những biến thực thể. Từng con Bug có một biến tên là sideLength, để quy định kích thước ô mà nó vẽ nên, và steps, để theo dõi xem con Bug này đi bao nhiêu bước rồi.

Dòng tiếp theo định nghĩa một **constructor**; đây là một phương thức đặc biệt để khởi tạo biến thực thể. Khi bạn tạo nên một Bug bằng cách kích hoạt new, Java sẽ kích hoạt constructor này.
Tham số cho constructor này là chiều dài cạnh.

Hành vi của Bug được điều khiển bởi phương thức act. Sau đây là phương thức act của BoxBug:

```
  public void act() {
    if (steps < sideLength && canMove()) {
      move();
      steps++;
    } else {
      turn();
      turn();
      steps = 0;
    }
  }
```

Nếu BoxBug có thể di chuyển, và chưa thực hiện đủ số bước đi theo yêu cầu, thì nó sẽ di chuyển và đồng thời tăng biến steps.
Nếu nó đụng phải tường hoặc đi hết một cạnh của hộp, thì con bọ sẽ quay 90 độ sang phải đồng thời chỉnh biến steps về 0.
Hãy chạy chương trình và xem nó làm gì. Bạn có thấy được con bọ có hành vi như dự kiến không?

10.1 Con mối

Tôi đã viết ra một lớp có tên Termite để mở rộng Bug và bổ sung khả năng tương tác với những bông hoa. Để chạy được lớp này, bạn hãy tải về những file sau rồi nhập chúng vào môi trường phát triển đang dùng:
http://thinkapjava.com/code/Termite.java
http://thinkapjava.com/code/Termite.gif
http://thinkapjava.com/code/TermiteRunner.java
http://thinkapjava.com/code/EternalFlower.java
Vì Termite mở rộng Bug, tất cả những phương thức của Bug đều hoạt động được với các Termite. Nhưng Termite có thêm những phương thức khác mà Bug không có.

```
/**
 * Trả lại true nếu con mối có mang bông hoa.
 */
  public boolean hasFlower();
/**
 * Trả lại true nếu con mối quay mặt về phía bông hoa.
 */
  public boolean seeFlower();
/**
```

```
 * Tạo nên bông hoa trừ khi con mối đã có sẵn một bông.
 */
  public void createFlower();
/**
 * Bỏ lại bông hoa tại vị trí con mối đang đứng.
 *
 * Lưu ý: trên mỗi ô chỉ có được một vật, bởi vậy hiệu ứng
 * của việc đánh rơi bông hoa sẽ được hoãn lại đến khi con mối di chuyển.
 */
  public void dropFlower();
/**
 * Ném bông hoa vào chỗ mà con mối đang hướng tới.
 */
  public void throwFlower();
/**
 * Nhặt bông hoa tại vị trí con mối hướng tới, nếu có,
 * và nếu con mối chưa mang theo hoa.
 */
  public void pickUpFlower();
```

Có những phương thức mà Bug cung cấp một lời định nghĩa này và Termite lại cung cấp cái khác. Trong trường hợp như vậy, phương thức Termite sẽ **ghi đè** lên phương thức Bug.
Chẳng hạn, Bug.canMove trả lại true nếu có một bông hoa ở vị trí kế tiếp, bởi vậy các có thể Bug có thể giẫm lên Flower. Còn Termite.canMove sẽ trả lại false nếu có bất kì đối tượng nào khác ở vị trí kế tiếp, nên biểu hiện của Termite sẽ khác đi.
Một ví dụ khác, các đối tượng con mối có một phiên bản turn trong đó nhận tham số là số nguyên chỉ độ góc. Sau cùng, đối tượng con mối córandomTurn, với tác dụng quay ngẫu nhiên qua trái hoặc phải với góc quay 45 độ.
Sau đây là mã lệnh từ file TermiteRunner.java:

```
public class TermiteRunner {
  public static void main(String[] args) {
    ActorWorld world = new ActorWorld();
    makeFlowers(world, 20);
    Termite alice = new Termite();
    world.add(alice);
    Termite bob = new Termite();
    bob.setColor(Color.blue);
    world.add(bob);
    world.show();
  }
```

```
  public static void makeFlowers(ActorWorld world, int n) {
    for (int i = 0; i<n; i++) {
      world.add(new EternalFlower());
    }
  }
}
```

Ở đây mọi thức có lẽ đều quen thuộc. TermiteRunner tạo nên một ActorWorld với
20 EternalFlowers và hai Termite.
Mỗi EternalFlower là một Flower ghi đè lên act sao cho các bông hoa không được tô thẫm đi.

```
public class EternalFlower extends Flower {
  public void act() {
  }
}
```

Nếu bạn chạy TermiteRunner.java, bạn sẽ thấy hai con mối di chuyển ngẫu nhiên quanh những bông hoa.
MyTermite.java giới thiệu những phương thức tương tác với các bông hoa. Sau đây là lời khai báo lớp này:

```
public class MyTermite extends Termite {
  public void act() {
    if (getGrid() == null)
      return;
    if (seeFlower()) {
      pickUpFlower();
    }
    if (hasFlower()) {
      dropFlower();
    }
    if (canMove()) {
      move();
    }
    randomTurn();
  }
}
```

MyTermite mở rộng Termite và ghi đè lên act. Nếu MyTermite thấy một bông hoa, nó sẽ nhặt lên. Nếu có bông hoa rồi, thì nó sẽ bỏ lại bông hoa này.

10.2 Con mối của Langton

Con kiến của Langton là một mô hình đơn giản về biểu hiện của kiến nhưng hiển thị những biểu hiện phức tạp đáng ngạc nhiên. Con kiến sống trong một lưới ô như GridWorld trong đó từng ô có màu trắng hoặc đen. Kiến di chuyển theo những quy tắc sau:

•Nếu con kiến đứng trên ô trắng; nó quay sang phải, tô màu ô thành đen, rồi tiến bước.
•Nếu con kiến đứng trên ô đen; nó quay sang trái, tô màu ô thành trắng, rồi tiến bước.

Vì những quy luật này rất đơn giản nên bạn sẽ trông đợi rằng con kiến này sẽ làm những điều đơn giản như chạy vòng quanh hoặc lặp lại một mẫu hình đơn giản. Song nếu kiến ta bắt đầu trên lưới ô toàn màu trắng thì nó sẽ đi hơn 10000 bước theo một dạng mẫu tưởng như ngẫu nhiên trước khi vào một vòng lặp gồm 104 bước.

Bạn có thể đọc thêm về con kiến Langton tại http://en.wikipedia.org/wiki/Langton_ant.
Thật không dễ lập nên con kiến Langton trong GridWorld vì ta không thể đặt màu của các ô. Song thay vào đó, ta có thể dùng những bông hoa để đánh dấu ô. Có điều là ta không thể có đồng thời cả kiến lẫn hoa trên cùng một ô, nên ta không hoàn toàn thực hiện đúng được những quy luật với con kiến.

Thay vào đó ta sẽ tạo nên một con mối có tên LangtonTermite, trong đó dùng seeFlower để kiểm tra xem ô trước mặt có bông hoa không, và nếu ô trước mặt có bông hoa, thì dùng pickUpFlower để hái nó, và throwFlower để đặt hoa xuống ô kế tiếp. Bạn có thể sẽ muốn đọc mã lệnh của những phương thức này để chắc rằng chúng làm gì.

10.3 Bài tập

Bài tập 1 Bây giờ bạn đã biết đủ kiến thức để làm bài tập trong cuốn Sách bài tập (Student Manual), Phần 2. Hãy làm những bài này, rồi xem tiếp những bài lý thú dưới đây.

Bài tập 2 Mục đích của bài tập này là khám phá biểu hiện của các con mối khi tương tác với những bông hoa. Hãy sửa chữa TermiteRunner.java để tạo nên những MyTermite thay vì các Termite. Sau đó chạy lại. MyTermite sẽ chạy vòng quanh một cách ngẫu nhiên, làm dịch chuyển những bông hoa. Tổng số bông hoa phải không đổi (kể cả những mông mà MyTermite đang giữ). Trong cuốn "Termites, Turtles and Traffic Jams", Mitchell Resnick đã mô tả một mô hình đơn giản cho biểu hiện của con mối:

- Nếu bạn thấy bông hoa, hãy nhặt nó lên. Trừ khi bạn đã có hoa rồi; trong trường hợp này thì vứt bỏ bông hoa hiện có.
- Tiến bước, nếu có thể.
- Quay sang trái hoặc phải một cách ngẫu nhiên.

Hãy sửa chữa MyTermite.java để thực hiện mô hình này. Theo bạn thì thay đổi trên sẽ có hiệu ứng gì đối với biểu hiện của các MyTermite?
Hãy thử chạy chương trình. Một lần nữa, tổng số bông hoa không đổi, nhưng dần dần hoa sẽ tụ lại thành một số ít các đống, nhiều khi chỉ là một đống.

Biểu hiện này là một **thuộc tính nổi**, mà bạn có thể tham khảo ở http://en.wikipedia.org/wiki/Emergence. Các con MyTermite tuân theo những quy tắc đơn giản chỉ bằng thông tin quy mô nhỏ, song kết quả sẽ là sự tổ chức có quy mô lớn.
Hãy thử nghiệm với những quy tắc khác nhau và xem chúng có tác động gì lên hệ thống. Những thay đổi nhỏ có thể gây nên kết quả không lường trước!

Bài tập 3

1. Sao chép lại file Termite.java rồi đặt tên thành LangtonTermite và sao chép TermiteRunner.java thành LangtonRunner.java. Hãy sửa chữa sao cho những định nghĩa lớp có tên trùng với tên file, và do đó LangtonRunner tạo nên một LangtonTermite.

2. Nếu bạn tạo một file tên là LangtonTermite.gif, GridWorld sẽ dùng nó để biểu diễn cho Termite của bạn. Bạn có thể tải về những ảnh côn trùng đẹp từ http://www.cksinfo.com/animals/insects/realisticdrawings/index.html. Để chuyển chúng về dạng GIF, bạn có thể dùng một ứng dụng như ImageMagick.
3. Sửa chữa act để thực hiện những quy tắc tương tự cho kiến Langton. Hãy thử những quy tắc khác nhau, và với cả hai góc quay 45 và 90 độ. Hãy tìm những quy tắc để chạy được nhiều ô nhất trước khi con mối bắt đầu chạy vòng.
4. Để cho mối có đủ chỗ chạy, bạn có thể nới rộng lưới ô hay chuyển sang dùng một UnboundedGrid.
5. Tạo nên nhiều con LangtonTermite rồi xem chúng tương tác như thế nào.

Chương 11: Tự tạo nên những đối tượng riêng

Trở về Mục lục cuốn sách

11.1 Lời định nghĩa lớp và các kiểu đối tượng

Trở về tận Mục 1.5 khi chúng ta định nghĩa lớp Hello, ta đồng thời tạo nên một kiểu đối tượng có tên Hello. Ta không tạo nên biến nào thuộc kiểu Hello này, và cũng không dùng new để tạo ra đối tượng Hello nào, song việc đó là hoàn toàn có thể!

Ví dụ đó chẳng có mấy tác dụng minh họa, bởi không lý gì để ta tạo ra một đối tượng Hello như vậy, và dù có tạo nên thì cũng chảng để làm gì. Trong chương này, ta sẽ xét đến những định nghĩa lớp để tạo nên các kiểu đối tượng *có ích*.

Sau đây là những ý tưởng quan trọng nhất trong chương:

•Việc định nghĩa một lớp mới đồng thời cũng tạo nên một đối tượng mới cùng tên.
•Lời định nghĩa lớp cũng giống như một bản mẫu cho các đối tượng: nó quy định những biến thực thể nào mà đối tượng đó chứa đựng, và những phương thức nào có thể hoạt động với chúng.
•Mỗi đối tượng thuộc về một kiểu đối tượng nào đó; như vậy, nó là một thực thể của một lớp nào đó.
•Khi bạn kích hoạt new để tạo nên một đối tượng, Java kích hoạt một phương thức đặc biệt có tên là **constructor** để khởi tạo các biến thực thể. Bạn cần cung cấp một hoặc nhiều constructor trong lời định nghĩa lớp.
•Các phương thức thao tác trên một kiểu được định nghĩa trong lời định nghĩa lớp cho kiểu đó.

Sau đây là một số vấn đề về lời định nghĩa lớp:

•Tên lớp (và do đó, tên của kiểu đối tượng) nên bắt đầu bàng một chữ in, để phân biệt chúng với các kiểu nguyên thủy và những tên biến.
•Bạn thường đặt một lời định nghĩa lớp vào trong mỗi file, và tên của file phải giống như tên của lớp, với phần mở rộng .java. Chẳng hạn, lớp Time được định nghĩa trong file có tên Time.java.
•Ở bất kì chương trình nào, luôn có một lớp được giao làm **lớp khởi động**. Lớp khởi động phải chứa một phương thức mang tên main, đó là nơi mà việc thực thi chương trình bắt đầu. Các lớp khác cũng *có thể* chứa phương thức cùng tên main, song phương thức đó sẽ không được thực thi từ đầu.

Khi đã nêu những vấn đề này rồi, ta hãy xét một ví dụ về lớp do người dùng định nghĩa, lớp Time.

11.2 Time

Một động lực chung cho việc tạo nên kiểu đối tượng, đó là để gói gọn những dữ liệu liên quan vào trong một đối tượng để ta có thể coi như một đơn vị duy nhất. Ta đã gặp hai kiểu như vậy, đó là Point và Rectangle.

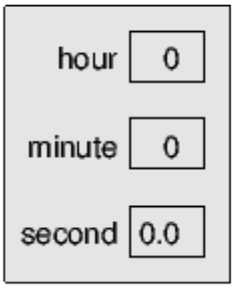
Một ví dụ khác, mà ta sẽ tự tay lập nên, là Time, để biểu diễn giờ đồng hồ. Dữ liệu được gói trong một đối tượng Time bao gồm có số giờ, số phút, và số giây. Bởi mỗi đối tượng Time đều chứa những dữ liệu này, nên ta cần biến thực thể để lưu giữ chúng.

Bước đầu tiên là xác định xem từng biến phải thuộc kiểu gì. Dường như rõ ràng là hour (giờ) và minute (phút) đều phải là những số nguyên. Để cho vấn đề được thú vị hơn, ta hãy đặt second (giây) là một double.

Các biến thực thể được định nghĩa ở đoạn đầu của lời khai báo lớp, bên ngoài bất kì lời khai báo phương thức nào khác, như sau:

```
class Time {
  int hour, minute;
  double second;
}
```

Đoạn mã này tự bản thân nó đã là lời khai báo lớp hợp lệ. Sơ đồ trạng thái cho một đối tượng Time sẽ trông như sau:



Sau khi khai báo các biến thực thể, bước tiếp theo là định nghĩa một constructor cho lớp mới này.

11.3 Constructor

Các constructor có nhiệmvụ khởi tạo các biến thực thể. Cú pháp của constructor cũng giống như của các phương thức khác, trừ ba điểm sau:

•Tên của constructor phải giống như tên lớp.
•Constructor không có kiểu trả về và cũng không có giá trị trả về.
•Từ khoá static được bỏ qua.

Sau đây là một ví dụ cho lớp Time:

```
public Time() {
  this.hour = 0;
  this.minute = 0;
  this.second = 0.0;
}
```

Ở chỗ mà bạn trông đợi một kiểu trả về, giữa public and Time, lại không có gì cả. Điều đó cho thấy cách mà chúng ta (và trình biên dịch nữa) có thể phân biệt được rằng đây là một constructor.

Constructor này không nhận tham số nào. Mỗi dòng của constructor khởi tạo một biến thực thể cho một giá trị mặc định (trong trường hợp này là nửa đêm). Cái tên this là một từ khóa đặc biệt để tham chiếu tới đối tượng mà ta đang tạo nên. Bạn có thể dùng this theo cách giống như dùng tên của bất kì đối tượng nào khác. Chẳng hạn, bạn có thể đọc và ghi các biến thực thể của this, và cũng truyền được this với vai trò tham số đến những phương thức khác.

Nhưng bạn không khai báo cái this này và cũng không thể gán giá trị cho nó. this được tạo bởi hệ thống; tất cả những gì bạn phải làm đó là khởi tạo các biến thực thể của nó.

Một lỗi thường gặp khi viết ra constructor là việc đưa câu lệnh return vào cuối. Hãy kiềm chế, tránh làm việc này.

11.4 Thêm các constructor

Constructor có thể được chồng chất [xem thêm phần "Quá tải"], cũng như các phương thức khác, theo nghĩa bạn có thể có nhiều constructor với các tham số khác nhau. Java biết rõ cần phải kích hoạt constructor nào bằng cách khớp những tham số của new với các tham số của constructor.
Việc có constructor không nhận đối số (như trên) là hoàn toàn bình thường, cũng như constructor nhận một danh sách tham số giống hệt với danh sách các biến thực thể. Chẳng hạn:

```
public Time(int hour, int minute, double second) {
  this.hour = hour;
  this.minute = minute;
  this.second = second;
}
```

Các tên và kiểu của những tham số cũng giống với tên và kiểu của các biến thực thể. Tất cả những gì mà constructor này làm chỉ là sao chép thông tin từ các tham số sang các biến thực thể.

Nếu xem tài liệu về Point và Rectangle, bạn sẽ thấy rằng cả hai lớp này đều có những constructor kiểu như trên. Việc chồng chất constructor cho phép linh hoạt tạo nên đối tượng trước rồi sau đó mới điền vào phần trống, hoặc để thu thập toàn bộ thông tin trước khi lập ra đối tượng.
Điều này nghe thì có vẻ không đáng quan tâm, song thực ra thì khác. Việc viết những constructor là quá trình máy móc, buồn tẻ. Một khi bạn đã viết được hai constructor rồi, bạn sẽ thấy rằng mình có thể viết chúng nhanh chóng chỉ qua việc trông vào danh sách các biến thực thể.

11.5 Tạo nên đối tượng mới

Mặc dù trông giống như phương thức, song constructor không bao giờ được kích hoạt trực tiếp. Thay vì vậy, khi bạn kích hoạt new, hệ thống sẽ huy động dung lượng bộ nhớ cho đối tượng mới và kích hoạt constructor này.
Chương trình sau giới thiệu hai cách làm để lập thành và khởi tạo các đối tượng Time:

```
class Time {
  int hour, minute;
  double second;
  public Time() {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
  }
  public Time(int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }
  public static void main(String[] args) {
```

```
    // một cách lập thành và khởi tạo đối tượng Time
    Time t1 = new Time();
    t1.hour = 11;
    t1.minute = 8;
    t1.second = 3.14159;
    System.out.println(t1);
    // một cách khác để thực hiện việc tương tự
    Time t2 = new Time(11, 8, 3.14159);
    System.out.println(t2);
  }
}
```

Trong main, lần đầu tiên kích hoạt new, ta không cấp cho đối số nào, bởi vậy Java kích hoạt constructor thứ nhất. Vài dòng phía dưới thực hiện gán giá trị cho các biến thực thể.
Lần thứ hai kích hoạt new, ta cấp các đối số khớp với các tham số của constructor thứ hai. Cách khởi tạo biến thực thể này gọn gàng hơn và hiệu quả hơn một chút, song cách làm này có thể khó đọc, bởi nó không rõ ràng là giá trị nào được gán cho biến thực thể nào.

11.6 In các đối tượng

Kết quả của chương trình nêu trên là:

```
Time@80cc7c0
Time@80cc807
```

Khi Java in giá trị của kiểu đối tượng do người dùng định nghĩa, nó sẽ in tên kiểu cùng một mã thập lục phân đặc biệt riêng của từng đối tượng. Mã này bản thân nó chẳng có ý nghĩa gì; thực tế nó khác nhau tuỳ máy tính và thậm chí tuỳ cả những lần chạy chương trình. Nhưng có thể nó giúp ích cho việc gỡ lỗi, trong trường hợp bạn muốn theo dõi từng đối tượng riêng rẽ.

Để in các đối tượng theo cách có ý nghĩa hơn đối với người dùng (chứ không phải đối với lập trình viên), bạn có thể viết một phương thức với tên gọi kiểu như printTime:

```
public static void printTime(Time t) {
  System.out.println(t.hour + ":" + t.minute + ":" + t.second);
}
```

Hãy so sánh phương thức này với phiên bản printTime ở Mục 3.10.
Kết quả của phương thức này, nếu ta truyền t1 hoặc t2 làm đối số, sẽ là 11:8:3.14159. Mặc dù ta có thể nhận ra đây là giờ đồng hồ, nhưng cách viết này không hề theo chuẩn quy định. Chẳng hạn, nếu số phút hoặc số giây nhỏ hơn 10, ta sẽ luôn dự kiến rằng có số 0 đi trước. Ngoài ra, có thể ta còn muốn bỏ phần thập phân của số giây đi. Nói cách khác, ta muốn kết quả kiểu như 11:08:03.
Trong đa số những ngôn ngữ lập trình, có nhiều cách đơn giản để điều khiển định dạng đầu ra cho kết quả số. Trong Java thì không có cách đơn giản nào.

Java có những công cụ mạnh dành cho việc in dữ liệu được định dạng như giờ đồng hồ và ngày tháng, đồng thời cũng có công cụ để diễn giải dữ liệu vào được định dạng. Song thật không may là những công

cụ như vậy không dễ sử dụng, nên tôi sẽ bỏ qua chúng trong khuôn khổ cuốn sách này. Nếu muốn, bạn có thể xem qua tài liệu của lớp Date trong gói java.util.

11.7 Các thao tác với đối tượng

Trong một vài mục tiếp theo, tôi sẽ giới thiệu ba dạng phương thức hoạt động trên các đối tượng:

hàm thuần tuý:
Nhận các đối tượng làm tham số nhưng không thay đổi chúng. Giá trị trả lại thuộc kiểu nguyên thuỷ hoặc một đối tượng mới tạo ra bên trong phương thức này.

phương thức sửa đổi:
Nhận đối số là các đối tượng rồi sửa đổi một vài, hoặc tất cả những đối tượng đó. Thường trả lại đối tượng rỗng (void).

phương thức điền:
Một trong các đối số là đối tượng "trống trơn" sẽ được phương thức điền thông tin vào. Về mặt kĩ thuật, đây cũng chính là một dang phương thức sửa đổi.

Với một phương thức cho trước ta thường có thể viết nó dưới dạng hàm thuần túy, phương thức sửa đổi hay phương thức điền. Tôi sẽ bàn thêm về ưu nhược điểm của từng hình thức một.

11.8 Các hàm thuần túy

Một phương thức được coi là hàm thuần túy nếu như kết quả chỉ phụ thuộc vào các đối số, và phương thức này không có hiệu ứng phụ như thay đổi một đối số hoặc in ra thông tin gì. kết quả duy nhất của việc kích hoạt một hàm thuần túy, đó là giá trị trả lại.

Một ví dụ là isAfter, để so sánh hai đối tượng Time rồi trả lại một boolean để chỉ định xem liệu toán hạng thứ nhất có xếp trước toán hạng thứ hai hay không:

```
public static boolean isAfter(Time time1, Time time2) {
  if (time1.hour > time2.hour)
    return true;
  if (time1.hour < time2.hour)
    return false;
  if (time1.minute > time2.minute)
    return true;
  if (time1.minute < time2.minute)
    return false;
  if (time1.second > time2.second)
    return true;
  return false;
}
```

Kết quả của phương thức này sẽ là gì nếu hai thời gian đã cho bằng nhau? Liệu đó có phải là kết quả phù hợp đối với phương thức này không? Nếu bạn viết tài liệu cho phương thức này, liệu bạn có đề cập rõ đến trường hợp đó không?

Ví dụ thứ hai là addTime, phương thức tính tổng hai thời gian. Chẳng hạn, nếu bây giờ là 9:14:30, và người làm bánh cần 3 giờ 35 phút, thì bạn có thể dùng addTime để hình dung ra khi nào bánh ra lò. Sau đây là bản sơ thảo của phương thức này; nó chưa thật đúng:

```
public static Time addTime(Time t1, Time t2) {
  Time sum = new Time();
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;
  return sum;
}
```

Mặc dù phương thức này trả lại một đối tượng Time, song nó không phải là constructor. Bạn cần xem lại và so sánh cú pháp của một phương thức dạng này với cú pháp của một constructor, vì chúng dễ gây nhầm lẫn.
Sau đây là một ví dụ về cách dùng phương thức. Nếu như currentTime chứa thời gian hiện tại và breadTime chứa thời gian cần để người thợ nướng bánh, thì bạn có thể dùng addTime để hình dung ra khi nào sẽ nướng xong bánh.

```
Time currentTime = new Time(9, 14, 30.0);
Time breadTime = new Time(3, 35, 0.0);
Time doneTime = addTime(currentTime, breadTime);
printTime(doneTime);
```

Kết quả của chương trình, 12:49:30.0, là đúng. Mặt khác, cũng có những trường hợp mà kết quả không đúng. Bạn có đoán được một trường hợp như vậy không?
Vấn đề là ở chỗ phương thức này không xử lý được tình huống khi số giây hoặc số phút cộng lại vượt quá 60. Trong trường hợp đó, ta phải "nhớ" số giây còn dư vào cột số phút, hoặc nhớ số phút dư vào cột giờ.

Sau đây là một dạng đúng của phương thức này.

```
public static Time addTime(Time t1, Time t2) {
  Time sum = new Time();
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;
  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
```

```
    }
    return sum;
  }
```

Mặc dù cách này đúng, song chương trình bắt đầu dài dòng. Sau này tôi sẽ gợi ý một giải pháp khác ngắn hơn nhiều.

Đoạn mã lệnh trên giới thiệu hai toán tử mà ta chưa từng gặp, += và -=. Những toán tử này cho ta viết ngắn gọn lệnh tăng hoặc giảm biến. Chúng cũng gần giống như ++ và --, chỉ khác ở chỗ (1) chúng làm việc được cả với double lẫn int, và (2) lượng tăng hoặc giảm không nhất thiết bằng 1. Câu lệnh sum.second -= 60.0; tương đương với sum.second = sum.second - 60;

11.9 Phương thức sửa đổi

Xét một ví dụ về phương thức sửa đổi, phương thức increment, nhằm tăng thêm một số giây cho trước vào một đối tượng Time. Một lần nữa, ta có bản nháp phương thức này như sau:

```
  public static void increment(Time time, double secs) {
    time.second += secs;
    if (time.second >= 60.0) {
      time.second -= 60.0;
      time.minute += 1;
    }
    if (time.minute >= 60) {
      time.minute -= 60;
      time.hour += 1;
    }
  }
```

Dòng đầu tiên thực hiện thao tác cơ bản; những dòng còn lại để xử lý các trường hợp ta đã xét.

Liệu phương thức này có đúng không? Điều gì sẽ xảy ra nếu đối số secs lớn hơn nhiều so với 60? Trong trường hợp như vậy, trừ đi 60 một lần là chưa đủ; ta phải tiếp tục trừ đến khi second nhỏ hơn 60. Ta có thể làm điều này bằng cách thay các lệnh if bằng các lệnh while:

```
  public static void increment(Time time, double secs) {
    time.second += secs;
    while (time.second >= 60.0) {
      time.second -= 60.0;
      time.minute += 1;
    }
    while (time.minute >= 60) {
      time.minute -= 60;
      time.hour += 1;
    }
  }
```

Giải pháp này đúng đắn, nhưng chưa hiệu quả lắm. Bạn có thể nghĩ ra lời giải nào không cần đến tính lặp hay không?

11.10 Các phương thức điền

Thay vì việc tạo nên đối tượng mới mỗi khi addTime được kích hoạt, ta có thể yêu cầu chương trình gọi hãy cung cấp một đối tượng nơi màaddTime lưu kết quả. Hãy so sánh đoạn mã sau với phiên bản trước:

```
public static void addTimeFill(Time t1, Time t2, Time sum) {
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;
  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
}
```

Kết quả được lưu trong sum, nên kiểu trả về là void.
Các phương thức sửa đổi và phương thức điền đều hiệu quả vì chúng không phải tạo nên đối tượng mới. Nhưng chúng lại gây khó khăn trong việc cô lập các phần khác nhau của chương trình; trong những dự án lớn chúng có thể gây nên lỗi rất khó tìm ra.

Các hàm thuần túy giúp ta quản lý tính chất phức tạp của những dự án lớn, phần là nhờ ngăn không cho những loại lỗi nhất định không thể xảy ra. Hơn nữa, hàm thuần túy còn thích hợp với những kiểu lập trình ghép và lồng. Và vì kết quả của hàm thuần túy chỉ phụ thuộc vào tham số, ta có thể tăng tốc cho nó bằng cách lưu giữ những giá trị đã tính toán từ trước.

Tôi gợi ý rằng bạn nên viết hàm thuần túy mỗi lúc thấy được, và chỉ dùng đến phương thức sửa đổi khi thấy rõ ưu điểm vượt trội.

11.11 Lập kế hoạch và phát triển tăng dần

Trong chương trình này tôi giới thiệu một quá trình phát triển chương trình với tên gọi **lập nguyên mẫu nhanh**[1]. Với từng phương thức, tôi viết một bản sơ thảo để thực hiện tính toán cơ bản, rồi kiểm tra nó với một vài trường hợp, sửa những lỗi bắt gặp được.
Cách tiếp cận này có thể hiệu quả, song nó có thể dẫn đến mã lệnh phức tạp một cách không cần thiết—vì nó xử lý quá nhiều trường hợp đặc biệt—và cũng kém tin cậy—vì thật khó tự thuyết phục rằng bạn đã tìm thấy *tất cả* những lỗi trong chương trình.

Một cách khác là xem xét kĩ hơn vấn đề nhằm tìm mấu chốt có thể giúp việc lập trình dễ dàng hơn.

Trong trường hợp này điểm mấu chốt bên trong là: Time thực ra là một số có ba chữ số trong hệ cơ số

60! Số giây, second, là hàng đơn vị, số phút, minute, là hàng 60, còn số giờ, hour, là hàng 3600.

Khi ta viết addTime và increment, thực chất là ta đang tính cộng ở hệ 60; đó là lý do tại sao ta phải "nhớ" từ hàng này sang hàng khác.
Một cách tiếp cận khác đối với tổng thể bài toán là chuyển Time thành double rồi lợi dụng khả năng tính toán của máy đối với double. Sau đây là một phương thức chuyển đổi Time thành double:

```
public static double convertToSeconds(Time t) {
  int minutes = t.hour * 60 + t.minute;
  double seconds = minutes * 60 + t.second;
  return seconds;
}
```

Bây giờ tất cả những gì ta cần là cách chuyển từ double sang đối tượng Time. Ta có thể viết một phương thức để thực hiện điều này, song có lẽ hợp lý hơn la viết một constructor thứ ba:

```
public Time(double secs) {
  this.hour =(int)(secs / 3600.0);
  secs -= this.hour * 3600.0;
  this.minute =(int)(secs / 60.0);
  secs -= this.minute * 60;
  this.second = secs;
}
```

Constructor này hơi khác những constructor khác; nó bao gồm những tính toán bên cạnh phép gán cho các biến thực thể.

Có thể bạn phải suy nghĩ để tự thuyết phục bản thân rằng kĩ thuật mà tôi dùng để chuyển từ hệ cơ số này sang cơ số kia là đúng. Nhưng một khi bạn đã bị thuyết phục rồi, ta có thể dùng những phương thức này để viết lại addTime:

```
public static Time addTime(Time t1, Time t2) {
  double seconds = convertToSeconds(t1) + convertToSeconds(t2);
  return new Time(seconds);
}
```

Mã lệnh trên ngắn hơn phiên bản gốc, và dễ thấy hơn hẳn rằng mã lệnh này đúng đắn (với giả thiết thường lệ rằng những phương thức nó kích hoạt cũng đều đúng). Việc viết lại increment theo cách tương tự được dành cho bạn như một bài tập.

11.12 Khái quát hóa

Trong chừng mực nào đó, việc chuyển đổi qua lại giữa các hệ cơ số 60 và 10 khó hơn việc xử lý thời gian đơn thuần. Việc chuyển hệ cơ số thì trừu tượng hơn, còn trực giác của ta xử lý thời gian tốt hơn.

Nhưng nếu ta có hiểu biết sâu để coi thời gian như các số trong hệ 60, và đầu tư công sức viết những phương thức chuyển đổi (convertToSeconds và constructor thứ ba), ta sẽ thu được một chương trình ngắn hơn, dễ đọc và gỡ lỗi, đồng thời đáng tin cậy hơn.
Việc bổ sung các đặc tính sau này cũng dễ dàng hơn. Hãy tưởng tượng ta cần trừ hai đối tượng Time để

tìm ra khoảng thời gian giữa chúng. Cách làmthực hiện tính trừ có nhớ. Nhưng dùng phương thức để chuyển đổi sẽ dễ hơn nhiều.
Điều trớ trêu là, đôi khi việc làm cho bài toán khó hơn (tổng quát hơn) lại khiến cho dễ dàng hơn (ít trường hợp đặc biệt, ít khả năng gây ra lỗi).

11.13 Thuật toán

Khi bạn viết một lời giải tổng quát cho một lớp các bài toán, thay vì tìm lời giải riêng cho một bài toán riêng lẻ, bạn đã viết một **thuật toán**. Thật không dễ định nghĩa thuật ngữ này, bởi vậy tôi sẽ cố gắng thử vài cái tiếp cận khác nhau.

Trước hết, hãy xét một số thứ không phải là thuật toán. Khi bạn học tính nhân giữa hai số, có lẽ bạn đã ghi nhớ bản cửu chương. Thật ra, bạn đã học thuộc lòng 100 lời giải cụ thể, bởi vậy kiến thức này thực sự không phải là thuật toán.

Nhưng nếu bạn "lười biếng," có thể bạn đã học hỏi được mấy mẹo vặt. Chẳng hạn, để tính tính của một số n với 9, bạn có thể viết $n–1$ là chữ số thứ nhất và $10–n$ là chữ số thứ hai. Mẹo này là lời giải tổng quát để nhân một số dưới mười bất kì với 9. Đó chính là thuật toán!

Tương tự, những kĩ thuật bạn học để cộng có nhớ, trừ có nhớ, và phép chia số lớn đều là những thuật toán. Một trong những đặc điểm của thuật toán là chúng không cần trí thông minh để thực hiện. Chúng chỉ là những cơ chế máy móc trong đó từng bước nối tiếp nhau theo một loạt những nguyên tắc đơn giản.

Theo ý kiến của tôi, thật đáng ngại khi thấy rằng chúng ta dành quá nhiều thời gian trên lớp để học cách thực hiện những thuật toán mà, nói thẳng ra là, không cần trí thông minh gì cả. Mặt khác, quá trình thiết kế những thuật toán lại thú vị, đầy thử thách trí tuệ, và là phần trung tâm của việc mà ta gọi là lập trình.

Có những việc mà con người làm theo lẽ tự nhiên, chẳng khó khăn hay phải suy nghĩ gì, lại là những thứ khó biểu diễn bằng thuật toán nhất. Việc hiểu ngôn ngữ là một ví dụ điển hình. Chúng ta ai cũng làm vậy, nhưng đến nay chưa ai giải thích được rằng ta làm vậy *bằng cách nào*, ít nhất là biểu diễn dưới dạng thuật toán.
Bạn sẽ sớm có cơ hội thiết kế những thuật toán đơn giản cho nhiều bài toán khác nhau.

11.14 Thuật ngữ

lớp:
Trước đây, tôi đã định nghĩa lớp là một tập hợp các phương thức có liên quan. Trong chương này ta còn được biết rằng lời định nghĩa lớp cũng đồng thời là một bản mẫu của một kiểu đối tượng mới.
thực thể:
Thành viên của một lớp. Mỗi đối tượng đều là thực thể của một lớp nào đó.
constructor:
Một phương thức đặc biệt để khởi tạo các biến thực thể của một đối tượng mới lập nên.
lớp khởi động:
Lớp có chứa phương thức main nơi bắt đầu việc thực thi chương trình.

hàm thuần túy:

Phương thức mà kết quả chỉ phụ thuộc vào các tham số của nó, và không gây hiệu ứng phụ nào ngoài việc trả lại một giá trị.

phương thức sửa đổi:

Phương thức làm thay đổi một hay nhiều đối tượng nhận làm tham số, và thường trả lại void.

phương thức điền:

Kiểu phương thức nhận tham số là một đối tượng "trống không" và điền vào những biến thực thể của nó thay vì việc phát sinh một giá trị trả lại.

thuật toán:

Một loạt những chỉ dẫn nhằm giải một lớp các bài toán theo một quá trình máy móc.

11.15 Bài tập

Bài tập 1 Trong trò chơi trên bàn có tên Scrabble$_2$, mỗi miếng vuông để xếp lên bàn sẽ chứa một chữ cái, để xêm nên các từ có nghĩa, và đồng thời có một điểm số; từ đó ta tính được điểm cho các từ khác nhau.

1. Hãy viết một định nghĩa lớp có tên Tile để biểu diễn các miếng vuông Scrabble. Các biến thực thể sẽ gồm một kí tự có tên letter và một số nguyên có tên value.
2. Hãy viết một constructor để nhận các tham số letter và value rồi khởi tạo các biến thực thể.
3. Viết một phương thức có tên printTile để nhận tham số là một đối tượng Tile rồi in ra các biến thực thể dưới định dạng mà người thường có thể đọc được.
4. Viết một phương thức có tên testTile để tạo nên một đối tượng Tile có chữ cái Z và giá trị 10, rồi dùng printTile để in ra trạng thái của đối tượng này.

   Mục đích của bài tập này là để luyện tập phần cơ chế tạo nên một lời định nghĩa lớp và mã lệnh để kiểm tra nó.

Bài tập 2 Hãy viết một định nghĩa lớp của Date, một kiểu đối tượng bao gồm ba số nguyên là year, month và day. Lớp này cần phải có hai constructor. Constructor thứ nhất không nhận tham số nào. Constructor thứ hai nhận các tham số mang tên year, month và day, rồi dùng chúng để khởi tạo các biến thực thể. Hãy viết một phương thức main để tạo nên một đối tượng Date mới có tên birthday. Đối tượng mới này để chứa ngày sinh nhật của bạn. Có thể dùng constructor nào cũng được.

Bài tập 3 Phân số là số có thể biểu điễn được dưới dạng tỉ số giữa hai số nguyên. Chẳng hạn, 2/3 là một phân số, và bạn cũng có thể coi 7 là một phân số với mẫu số ngầm định bằng 1. Ở bài tập này, bạn sẽ viết một lời định nghĩa lớp cho các phân số.

1. Lập một chương trình mới có tên Rational.java để định nghĩa một lớp tên là Rational. Một đối tượng Rational phải có hai biến thực thể số nguyên để lưu trữ tử số và mẫu số.
2. Viết một constructor không nhận tham số nào để đặt tử số bằng 0 và mẫu số bằng 1.
3. Viết một phương thức có tên printRational để nhận vào đối số là một đối tượng Rational rồi in nó ra theo định dạng hợp lý.
4. Viết một phương thức main để lập nên một đối tượng mới có kiểu là Rational, đặt các biến thực thể của nó bằng giá trị cụ thể, rồi in đối tượng này ra.
5. Đến đây, bạn đã có một chương trình tối thiểu có thể chạy thử được. Hãy chạy để kiểm tra nó, và gỡ lỗi, nếu cần.
6. Viết một constructor thứ hai cho lớp này có nhận vào hai đối số rồi sử dụng chúng để khởi tạo các biến thực thể.

7. Hãy viết một phương thức có tên negate để đảo dấu của phân số. Phương thức này phải là một phân thức sửa đổi, và vì vậy cần phải trả lại void. Hãy viết thêm dòng lệnh trong main để kiểm tra phương thức mới này.
8. Viết một phương thức có tên invert để nghịch đảo số bằng cách tráo đổi tử số và mẫu số. Hãy viết thêm dòng lệnh trong main để kiểm tra phương thức mới này.
9. Viết một phương thức có tên toDouble để chuyển đổi phân số thành một số double (số dấu phẩy động) rồi trả lại kết quả. Phương thức này là một hàm thuần tuý; nó không thay đổi đối tượng. Như thường lệ, hãy kiểm tra phương thức mới viết.
10. Viết một phương thức có tên reduce để rút gọn một phân số về dạng tối giản bằng cách tìm ước số chung lớn nhất của tử số và mẫu số rồi cùng chia cả tử lẫn mẫu cho ước chung này. Phương thức nêu trên phải là một hàm thuần tuý; nó không được phép thay đổi các biến thực thể của đối tượng mà nó được kích hoạt lên. Để tính ước số chung lớn nhất, hãy xem Bài tập 10 của Chương 8).
11. Viết một phương thức có tên add để nhận hai đối số là hai Rational rồi trả lại một đối tượng Rational mới. Đối tượng được trả lại phải chứa tổng của các đối số. có vài cách thực hiện phép cộng này. Bạn có thể dùng bất kì cách nào, nhưng hãy đảm bảo rằng kết quả của phép tính phải được rút gọn sao cho tử và mẫu không có ước số chung nào khác (ngoài 1).
    Mục đích của bài tập này là nhằm viết một lời định nghĩa hàm có chứa nhiều loại phương thức, bao gồm constructors, phương thức sửa đổi, và hàm thuần tuý.

1

Cái mà tôi gọi là "nguyên mẫu nhanh" (rapid prototyping) ở đây rất giống với cách phát triển dựa trên kiểm thử (test-driven development, TDD); sự khác biệt là ở chỗ TDD thường dựa trên kiểm thử tự động. Xem http://en.wikipedia.org/wiki/Test-driven_development.

2

Scrabble là một nhãn hiệu đã đăng kí ở Hoa Kì và Canada, thuộc về cty Hasbro Inc., và ở các nước còn lại trên thế giới, thì thuộc về J.W. Spear & Sons Limited ở Maidenhead, Berkshire, Anh Quốc, công ty nhánh của Mattel Inc.

Chương 12: Mảng

Trở về Mục lục cuốn sách

Mảng là một tập hợp các giá trị trong đó mỗi giá trị được xác định bởi một chỉ số. Bạn có thể lập nên các mảng int, mảng double, hay mảng chứa bất kì kiểu dữ liệu nào khác, nhưng các giá trị trong cùng một mảng phải có kiểu giống nhau.

Về mặt cú pháp, các kiểu mảng trông giống như các kiểu dữ liệu khác trong Java chỉ trừ đặc điểm: theo sau là []. Chẳng hạn, int[] là kiểu "mảng các số nguyên" còn double[] là kiểu "mảng các số phẩy động." Bạn có thể khai báo các biến với những kiểu như vậy theo cách thông thường:

```
int[] count;

double[] values;
```

Trước khi bạn khởi tạo các biến này, chúng được đặt về null. Để tự tay tạo các mảng, hãy dùng new.

```
count = new int[4];

values = new double[size];
```

Lệnh gán thứ nhất khiến cho count tham chiếu đến một mảng gồm 4 số nguyên; lệnh thứ hai tham chiếu khiến values tham chiếu đến một mảng các double. Số phần tử trong values phụ thuộc vào size. Bạn có thể dùng bất kì biểu thức nguyên nào để làm kích thước mảng.

Hình vẽ sau cho thấy cách biểu diễn mảng trong sơ đồ trạng thái:

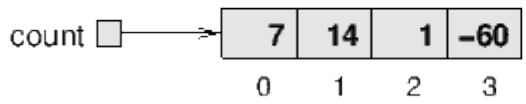
Các số lớn ghi bên trong các ô là những **phần tử** của mảng. Các con số nhỏ bên ngoài hộp là những chỉ số dùng để xác định từng ô. Khi bạn huy động một mảng các int, những phần tử của chúng đều được khởi tạo bằng không.

12.1 Truy cập các phần tử

Để lưu các giá trị trong mảng, hãy dùng toán tử [] operator. Chẳng hạn, count[0] tham chiếu đến phần tử "thứ không" của mảng, còn count[1] tham chiếu đến phần tử "thứ một". Bạn có thể dùng toán tử [] bất cứ đâu trong một biểu thức:

```
count[0] = 7;

count[1] = count[0] * 2;

count[2]++;

count[3] -= 60;
```

Tất cả đó đều là những phép gán hợp lệ. Sau đây là kết quả của đoạn mã trên:



Những phần tử của mảng được đánh số từ 0 tới 3, nghĩa là không có phần tử nào mang chỉ số 4. Điều này rất quen thuộc, bởi ta đã thấy điều tương tự trong chỉ số của String. Dù vậy, việc vượt quá phạm vi của mảng vẫn là kiểu lỗi thường gặp, bằng cách đó phát ra biệt lệArrayOutOfBoundsException.
Bạn có thể dùng bất kì biểu thức nào làm chỉ số cũng được, miễn là nó có kiểu int. Một trong những cách thông dụng nhất để đánh chỉ số của mảng là dùng biến vòng lặp. Chẳng hạn:

```
int i = 0;
while (i < 4) {
  System.out.println(count[i]); i++;
}
```

Đây là một vòng lặp while tiêu chuẩn để đếm từ 0 lên 4, và khi biến lặp i bằng 4, điều kiện lặp sẽ không thỏa mãn và vòng lặp kết thúc. Như vậy, phần thân vòng lặp chỉ được thực thi khi i là 0, 1, 2 và 3.
Mỗi lần qua vòng lặp ta dùng i làm chỉ số trong mảng, để in ra phần tử thứ i. Hình thức duyệt mảng này rất thông dụng.

12.2 Sao chép mảng

Khi bạn sao chép một biến mảng, hãy nhớ rằng bạn đang sao chép tham chiếu tới mảng. Ví dụ:

```
double[] a = new double [3];
double[] b = a;
```

Đoạn mã lệnh này tạo nên một mảng ba số double, rồi đặt hai biến khác nhau để tham chiếu tới nó. Trường hợp này cũng là một dạng trùng tên (aliasing).

Bất kì thay đổi nào đối với một trong hai mảng đều được phản ánh trên mảng còn lại. Thường thì đây không phải là điều bạn muốn; mà bạn muốn huy động một mảng mới rồi sao chép các phần tử từ mảng này sang mảng kia.

```
double[] b = new double [3];
int i = 0;
while (i < 4) {
  b[i] = a[i]; i++;
}
```

12.3 Mảng và đối tượng

Mảng giống với đối tượng ở nhiều điểm:

- Khi khai báo một biến mảng, bạn nhận được tham chiếu đến mảng.
- Bạn phải dùng new để tự tạo ra mảng.
- Khi truyền mảng làm đối số, bạn truyền một tham chiếu, nghĩa là phương thức được kích hoạt có thể thay đổi nội dung của mảng.

Một số đối tượng mà ta đã xét, như Rectangle, tương đồng với mảng ở chỗ chúng cũng là tập hợp các giá trị. Vậy nảy sinh câu hỏi, "Mảng bốn số nguyên thì khác một đối tượng Rectangle ở chỗ nào?" Nếu bạn quay về định nghĩa của "mảng" từ đầu chương, bạn sẽ thấy một khác biệt: các phần tử của mảng được xác định bằng chỉ số, còn các phần tử của đối tượng xác định bằng tên.

Một khác biệt nữa là các phần tử trong mảng phải có cùng kiểu. Còn đối tượng có thể chứa những biến thực thể khác kiểu nhau.

12.4 Vòng lặp for

Các vòng lặp mà ta đã dùng đều có một số điểm chung. Chúng đều bắt đầu bằng việc khởi tạo một biến; chúng đều có một phép kiểm tra, hay điều kiện, phụ thuộc vào biến đó; và bên trong vòng lặp thì chúng thực hiện tác động nhất định đến biến đó, như tăng giá trị.

Dạng vòng lặp này thông dụng đến nỗi còn một lệnh lặp khác, gọi là for, để diễn đạt một cách gọn gàng hơn. Cú pháp chung của nó như sau:

```
for (KHỞI TẠO; ĐIỀU KIỆN; GIA TĂNG) {
  PHẦN THÂN
}
```

Lệnh này tương đương với

```
KHỞI TẠO;
while (ĐIỀU KIỆN) {
  PHẦN THÂN
  GIA TĂNG
}
```

ngoại trừ nó gọn gàng hơn vì đã đặt tất cả những câu lệnh liên quan đến lặp vào một chỗ, và do đó dễ đọc hơn. Chẳng hạn:

```
for (int i = 0; i < 4; i++) {
  System.out.println(count[i]);
}
```

thì tương đương với

```
int i = 0;
while (i < 4) {
  System.out.println(count[i]);
  i++;
}
```

12.5 Chiều dài của mảng

Tất cả mảng đều có một biến thực thể tên là length. Chẳng cần nói thì bạn cũng biết, biến này chứa chiều dài của mảng (số phần tử). Nên lấy giá trị này làm giới hạn trên của vòng lặp thay vì một giá trị cố định. Làm như vậy, nếu như kích thước của mảng thay đổi thì bạn sẽ không phải dò lại cả chương trình

để thay đổi các vòng lặp; chương trình sẽ chạy được đúng với mọi kích cỡ mảng khác nhau.

```
for (int i = 0; i < a.length; i++) {
  b[i] = a[i];
}
```

Lần cuối cùng mà phần thân của vòng lặp được thực thi, i sẽ là a.length - 1, chỉ số của phần tử cuối. Khi i bằng với a.length, điều kiện sẽ không thỏa mãn và phần thần sẽ không được thực thi. Đây là điều tốt, vì sẽ có biệt lệ được phát ra. Đoạn mã này giả thiết rằng mảng b phải có bằng số phần tử, hoặc nhiều hơn so với a.

12.6 Số ngẫu nhiên

Đa số các chương trình máy tính đều làm cùng một công việc mỗi khi nó được thực thi; chương trình như vậy được gọi là có tính **tất định**. Thông thường, tất định là tính chất tốt, vì ta luôn trông đợi cùng một phép tính sẽ chỉ cho một kết quả. Song có những chương trình ứng dụng mà ta muốn kết quả phải không đoán trước được. Một ví dụ hiển nhiên là các trò chơi điện tử, song cũng có những ứng dụng khác nữa.

Để một chương trình thực sự **phi tất định** hóa ra lại không dễ chút nào, song ít nhất vẫn có những cách làm chương trình có vẻ như phi tất định. Một cách làm trong số đó là việc phát sinh những số ngẫu nhiên và dùng nó để quy định kết quả của chương trình. Java có một phương thức để phát sinh ra các số **giả ngẫu nhiên**, vốn không thực sự ngẫu nhiên, nhưng sẽ dùng được cho mục đích ta cần.

Hãy đọc tài liệu về phương thức random trong lớp Math. Giá trị trả lại là một doublenằm giữa 0.0 và 1.0. Chính xác là, nó lớn hơn hoặc bằng 0.0 và nhỏ hơn 1.0. Mỗi lần kích hoạt random bạn sẽ nhận được con số tiếp theo trong dãy số giả ngẫu nhiên. Để thấy được một mẫu của dãy ngẫu nhiên, hãy chạy vòng lặp sau:

```
for (int i = 0; i < 10; i++) {
  double x = Math.random();
  System.out.println(x);
}
```

Để phát sinh một số double giữa 0.0 và một giới hạn trên như high, bạn có thể nhân x với high.

12.7 Mảng các số ngẫu nhiên

Bằng cách nào để phát sinh một số nguyên ngẫu nhiên giữa low và high? Nếu phương thức randomInt bạn viết đã chính xác, thì mỗi giá trị trong khoảng từ low lên đến high-1 phải có cùng xác suất xuất hiện. Nếu bạn phát sinh một dãy số rất dài, thì mỗi giá trị phải xuất hiện ít nhất là có số lần xấp xỉ nhau.
Một cách kiểm tra phương thức vừa viết là phát inh rất nhiều số ngẫu nhiên, lưu trữ chúng vào một mảng, rồi đếm số lần từng giá trị xuất hiện.

Phương thức sau nhận một đối số duy nhất là kích thước của mảng. Phương thức có nhiệm vụ huy động một mảng số nguyên mới, điền vào những giá trị ngẫu nhiên, rồi trả lại tham chiếu đến mảng mới điền.

```
public static int[] randomArray(int n) {
  int[] a = new int[n];
```

```
    for (int i = 0; i<a.length; i++) {
      a[i] = randomInt(0, 100);
    }
    return a;
  }
```

Kiểu trả lại là int[], nghĩa là phương thức này trả lại một mảng các số nguyên. Để kiểm tra phương thức này, thật tiện nếu có một phương thức để in ra nội dung của mảng.

```
  public static void printArray(int[] a) {
    for (int i = 0; i<a.length; i++) {
      System.out.println(a[i]);
    }
  }
```

Đoạn mã sau đây phát sinh một mảng rồi in nó ra:

```
  int numValues = 8;
  int[] array = randomArray(numValues);
  printArray(array);
```

Trên máy tính của tôi, kết quả là

```
27
6
54
62
54
2
44
81
```

trông thật là ngẫu nhiên. Kết quả của bạn có thể sẽ khác đi.

Nếu đây là những điểm thi (và nếu vậy thì điểm thật tệ), giáo viên có thể biểu diễn kết quả trước lớp dưới dạng một **histogram**, vốn là một tập hợp những biến đếm để theo dõi số lần mỗi giá trị xuất hiện.

Với điểm thi, có thể ta dành ra 10 biến đếm để theo dõi bao nhiêu học sinh đạt điểm đầu 9 (90 – 99), bao nhiêu đạt điểm đầu 8, v.v. Một số mục tiếp theo sẽ dành cho việc phát triển mã lệnh tạo ra histogram.

12.8 Đếm

Một cách tiếp cận hay đến những bài toán như thế này là nghĩ về những phương thức đơn giản, dễ viết, rồi kết hợp chúng lại thành lời giải. Quá trình này được gọi là **phát triển từ dưới lên**.
Xem http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design.

Thật không dễ thấy điểm khởi đầu của quá trình, nhưng một cách hợp lý là tìm kiếm những bài toán nhỏ khớp với một dạng mẫu mà bạn đã biết trước.

Ở Mục 8.7 ta đã thấy một vòng lặp duyệt qua một chuỗi rồi đếm số lần xuất hiện một chữ cái cho trước. Bạn có thể coi chương trình này như một ví dụ về một mẫu có tên gọi "duyệt và đếm." Những yếu tố tạo nên dạng mẫu này là:

- Một tập hợp hoặc tập dữ liệu có thể duyệt được, như một mảng hoặc chuỗi.
- Một phép thử mà bạn có thể áp dụng cho từng phần tử trong tập đó.
- Một con trỏ để theo dõi xem có bao nhiêu phần tử đạt được phép thử này.

Trong trường hợp đang xét, tập hợp là một mảng các số nguyên. Phép thử là liệu rằng một điểm số cho trước có rơi vào một khoảng giá trị cho trước hay không.

Sau đây là một phương thức có tên inRange để đếm số phần tử trong mảng rơi vào một khoảng cho trước. Các tham số bao gồm mảng và hai số nguyên để quy định giới hạn dưới và trên của khoảng này.

```
public static int inRange(int[] a, int low, int high) {
  int count = 0;
  for (int i = 0; i < a.length; i++) {
    if (a[i] >= low && a[i] < high)
      count++;
  }
  return count;
}
```

Tôi đã không cụ thể hóa rằng liệu việc giá trị nào đó đúng bằng low hoặc high thì sẽ được coi là rơi vào khoảng hay không, nhưng từ mã lệnh bạn có thể thấy rằng low được coi là rơi vào trong còn high thì không. Điều này giúp ta tránh được việc đếm phần tử hai lần.
Bây giờ ta có thể đếm số điểm trong những khoảng cần quan tâm:

```
int[] scores = randomArray(30);
int a = inRange(scores, 90, 100);
int b = inRange(scores, 80, 90);
int c = inRange(scores, 70, 80);
int d = inRange(scores, 60, 70);
int f = inRange(scores, 0, 60);
```

12.9 Histogram

Mã lệnh này có sự lặp lại, nhưng cũng chấp nhận được khi có ít khoảng khác nhau. Nhưng thử tưởng tượng nếu ta muốn theo dõi số lần xuất hiện của từng điểm số, nghĩa là 100 giá trị có thể. Lúc đó liệu bạn còn muốn viết mã lệnh nữa không?

```
int count0 = inRange(scores, 0, 1);
int count1 = inRange(scores, 1, 2);
int count2 = inRange(scores, 2, 3);
```

```
...
int count3 = inRange(scores, 99, 100);
```

Tôi không nghĩ vậy. Điều mà ta thực sự mong muốn là cách để lưu trữ 100 số nguyên, tốt nhất là cách mà ta dùng được chỉ số để truy cập đến từng giá trị. Gợi ý: dùng mảng.

Dạng mẫu đếm cũng tương tự bất kể việc ta dùng một biến đếm hay một mảng các biến đếm. Trong trường hợp sau này, ta khởi tạo mảng bên ngoài vòng lặp. Sau đó, trong vòng lặp, ta kích hoạt inRange và lưu lại giá trị:

```
int[] counts = new int[100];
for (int i = 0; i < counts.length; i++) {
  counts[i] = inRange(scores, i, i+1);
}
```

Ở đây chỉ có một điều mẹo mực: chúng ta dùng biến lặp với hai tác dụng: làm chỉ số bên trong mảng, và làm tham số cho inRange.

12.10 Lời giải "một lượt"

Mã lệnh nói trên hoạt động được, song không hiệu quả như khả năng mà lẽ ra nó phải làm được. Mỗi lần đoạn chương trình kích hoạt inRange, nó duyệt toàn bộ mảng. Khi số các khoảng giá trị nhiều lên, sẽ có rất nhiều lần duyệt.

Sẽ tốt hơn nếu chỉ chạy một lượt qua mảng, và với mỗi giá trị, ta đi tính xem nó rơi vào khoảng nào. Tiếp theo ta có thể tăng biến đếm thích hợp. Ở ví dụ này, phép tính đó là nhỏ nhặt, bởi vì ta có thể dùng bản thân giá trị đó làm chỉ số cho mảng các biến đếm.

Sau đây là đoạn mã để duyệt một mảng các điểm số và phát sinh ra histogram.

```
int[] counts = new int[100];
for (int i = 0; i < scores.length; i++) {
  int index = scores[i];
  counts[index]++;
}
```

12.11 Thuật ngữ

mảng:
Một tập hợp các giá trị, trong đó những giá trị này phải cùng kiểu, và mỗi giá trị được xác định bằng một chỉ số.

phần tử:
Một trong số các giá trị thuộc mảng. Toán tử [] được dùng để lựa chọn phần tử.

chỉ số:
Một biến nguyên hoặc giá trị nguyên để chỉ định một phần tử của mảng.

tất định:
Một chương trình thực hiện đúng một công việc mỗi khi nó được kích hoạt.

giả ngẫu nhiên:
Một dãy con số trông có vẻ ngẫu nhiên, song thực ra là sản phẩm của những phép tính tất định.

histogram:

Một mảng các số nguyên trong đó từng số nguyên để đếm số các giá trị rơi vào một khoảng nhất định.

12.12 Bài tập

Bài tập 1 Hãy viết một phương thức có tên cloneArray để nhận vào tham số là một mảng các số nguyên, tạo ra một mảng mới cùng kích thước, sao chép các phần tử từ mảng đầu sang mảng mới tạo, rồi trả lại một tham chiếu đến mảng mới.

Bài tập 2 Viết một phương thức có tên randomDouble nhận vào hai số phẩy động, low và high, rồi trả lại một số phẩy động ngẫu nhiên, x, sao cho low ≤ x < high.

Bài tập 3 Viết một phương thức có tên randomInt nhận vào hai đối số, low và high, rồi trả lại một số nguyên ngẫu nhiên từ low đến high, nhưng không kể high.

Bài tập 4 Bao bọc mã lệnh trong Mục 12.10 vào một phương thức có tên makeHist để nhận một mảng các điểm số rồi trả lại một histogram các giá trị trong mảng.

Bài tập 5 Viết một phương thức có tên areFactors để nhận vào một số nguyên, n, và một mảng các số nguyên, và trả lại true nếu các số trong mảng đều là ước số của n (nghĩa là n chia hết cho tất cả những phần tử này). GỢI Ý: Xem bài tập 8.1.

Bài tập 6 Viết một phương thức nhận tham số gồm một mảng những số nguyên và một số nguyên tên là target, rồi trả lại chỉ số đầu tiên nơi mà target xuất hiện trong mảng, nếu có, hoặc -1 nếu không.

Bài tập 7 Có những lập trình viên phản đối quy tắc chung rằng các biến và phương thức phải được đặt tên có nghĩa. Thay vào đó, họ nghĩ rằng các biến và phương thức phải đặt tên là các loại hoa quả. Với từng phương thức sau đây, hãy viết một câu mô tả ý tưởng, nhiệm vụ của phương thức. Với mỗi biến, hãy xác định vai trò của nó.

```
public static int banana(int[] a) {
  int grape = 0;
  int i = 0;
  while (i < a.length) {
    grape = grape + a[i];
    i++;
  }
  return grape;
}
public static int apple(int[] a, int p) {
  int i = 0;
  int pear = 0;
  while (i < a.length) {
    if (a[i] == p)
      pear++;
    i++;
  }
  return pear;
```

```
}
public static int grapefruit(int[] a, int p) {
  for (int i = 0; i<a.length; i++) {
    if (a[i] == p)
      return i;
    }
  return -1;
}
```

Mục đích của bài tập này là thực hành đọc mã lệnh và nhận ra những dạng mẫu tính toán mà ta đã gặp.

Bài tập 8

1. Kết quả của chương trình sau là gì?
2. Hãy vẽ biểu đồ ngăn xếp để cho thấy trạng thái chương trình ngay trước khi mus trả về.
3. Diễn đạt bằng lời một cách ngắn gọn nhiệm vụ của mus.

```
public static int[] make(int n) {
  int[] a = new int[n];
  for (int i = 0; i < n; i++) {
    a[i] = i+1;
  }
  return a;
}
public static void dub(int[] jub) {
  for (int i = 0; i < jub.length; i++) {
    jub[i] *= 2;
  }
}
public static int mus(int[] zoo) {
  int fus = 0;
  for (int i = 0; i < zoo.length; i++) {
    fus = fus + zoo[i];
  }
  return fus;
}
public static void main(String[] args) {
  int[] bob = make(5);
  dub(bob);
  System.out.println(mus(bob));
}
```

Bài tập 9 Nhiều dạng mẫu để duyệt mảng mà ta đã gặp cũng có thể được viết theo cách đệ quy. Đó không phải là cách thường dùng, nhưng là một bài tập hữu ích.

1. Hãy viết một phương thức có tên maxInRange, nhận vào một mảng số nguyên mà một khoảng chỉ số (lowIndex và highIndex), rồi tìm giá trị lớn nhất trong mảng, nhưng chỉ xét những phần tử giữa lowIndex và highIndex, kể cả hai đầu này. Phương thức phải được viết theo cách đệ quy. Nếu chiều dài của khoảng bằng 1, nghĩa là nếu lowIndex == highIndex, thì ta biết ngay rằng phần tử duy nhất trong khoảng phải là giá trị lớn nhất. Do đó đây là trường hợp cơ sở. Nếu có nhiều phần tử trong khoảng, thì ta có thể chia mảng làm đôi, tìm cực đại trên mỗi phần, rồi sau đó lấy giá trị lớn hơn trong số hai cực đại tìm được.
2. Các phương thức như maxInRange có thể gây lúng túng khi dùng. Để tìm phần tử lớn nhất trong mảng, ta phải cung cấp một khoảng bao gồm toàn bộ mảng đó.

```
double max = maxInRange(array, 0, a.length-1);
```

Hãy viết một phương thức có tên max nhận tham số là một mảng rồi dùng maxInRange để tìm và trả lại giá trị lớn nhất. Các phương thức như max đôi khi còn được gọi là **phương thức gói bọc** vì chúng cung cấp một lớp khái niệm xung quanh một phương thức lủng củng và giúp nó dễ dùng. Phương thức mà thực sự thực hiện tính toán được gọi là **phương thức trợ giúp**.

3. Hãy viết một phiên bản find theo cách đệ quy và dùng đến dạng mẫu gói bọc-trợ giúp. find cần phải nhận một mảng các số nguyên và một số nguyên mục tiêu. Nó cần phải trả lại chỉ số của vị trí đầu tiên tại đó xuất hiện số nguyên mục tiêu, hoặc trả lại -1 nếu không xuất hiện.

Bài tập 10 Một cách không hiệu quả lắm để sắp xếp các phần tử trong mảng là tìm phần tử lớn nhất rồi đổi chỗ nó cho phần tử thứ nhất, sau đó tìm phần tử lớn thứ hai rồi đổi chỗ với phần tử thứ hai, và cứ như vậy. Cách này gọi là **sắp xếp chọn** (xem http://vi.wikipedia.org/wiki/Sắp_xếp_chọn).

1. Hãy viết một phương thức mang tên indexOfMaxInRange nhận vào một mảng số nguyên, tìm phần tử lớn nhất trong khoảng cho trước, rồi trả lại chỉ số của nó. Bạn có thể sửa lại phiên bản maxInRange hay bạn có thể viết từ đầu một phiên bản tương tác với máy.
2. Viết một phương thức có tên swapElement nhận một mảng số nguyên cùng hai chỉ số, rồi đổi chỗ hai phần tử tại các chỉ số đó.
3. Viết một phương thức có tên selectionSort nhận vào một mảng các số nguyên và trong đó dùngindexOfMaxInRange cùng swapElement để xếp mảng từ nhỏ đến lớn.

Bài tập 11 Viết một phương thức có tên letterHist nhận một chuỗi làm tham số rồi trả lại histogram của các chữ cái trong chuỗi. Phần tử thứ không của histogram cần phải chứa số chữ a trong chuỗi (cả chữ in và thường); phần tử thứ 25 cần phải chứa số chữ z. Lời giải của bạn chỉ được duyệt chuỗi này đúng một lần.

Bài tập 12 Một từ được gọi là “doubloon” nếu trong từ đó, mỗi chữ cái xuất hiện đúng hai lần. Chẳng hạn, các từ sau đây là doubloon mà tôi đã tìm thấy trong cuốn từ điển.

> Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

Hãy viết một phương thức có tên isDoubloon để trả lại true nếu từ đã cho là một doubloon và false nếu không phải.

Bài tập 13 Hai từ là từ đảo (anagram) nếu như chúng có chứa cùng những chữ cái (đồng thời cùng số

lượng từng chữ). Chẳng hạn, “stop” là từ đảo của “pots” còn “allen downey” là cụm từ đảo của “well annoyed.” Hãy viết một phương thức nhận vào hai String rồi trả lại true nếu như các String là từ đảo của nhau. Thêm phần thử thách: bạn chỉ được đọc các chữ cái của những String này đúng một lần.

Bài tập 14 Trong trò chơi Scrabble, mỗi người chơi có một tập hợp các miếng vuông với các chữ cái ghi trên đó, và mục tiêu của chò trơi là dùng những chữ cái đó ghép thành từ có nghĩa. Hệ thống tính điểm khá phức tạp, song thường thì các từ dài có giá trị cao hơn các từ ngắn. Giả dụ rằng bạn được cho trước các chữ cái dưới dạng một chuỗi, như "quijibo" và bạn nhận được một chuỗi khác để kiểm tra, như "jib". Hãy viết một phương thức có tên canSpell nhận vào hai chuỗi rồi trả lại true nếu tập hợp các miếng vuông xếp được thành từ có nghĩa. Bạn có thể có nhiều miếng ghi chữ giống nhau, nhưng chỉ được dùng mỗi miếng một lần. Thêm phần thử thách: bạn chỉ được đọc các chữ cái của những String này đúng một lần.

Bài tập 15 Thực ra trong Scrabble, còn những miếng vuông trắng có thể được dùng để biểu diễn chữ cái tùy ý. Hãy suy nghĩ một thuật toán cho canSpell xử lý được trường hợp chữ tùy ý như vậy. Đừng bận tâm đến những chi tiết thực hiện như bằng cách nào có thể biểu diễn những chữ tùy ý đó. Chỉ cần diễn đạt thuật toán bằng lời, bằng giả mã, hoặc bằng Java.

Chương 13: Mảng chứa các đối tượng

Trở về Mục lục cuốn sách

13.1 Con đường phía trước

Ở ba chương kế tiếp ta sẽ phát triển các chương trình chơi bài tây và với những cỗ bài. Trước khi đi vào chi tiết, sau đây là khái quát những bước đi:

1. Trong chương này ta sẽ định nghĩa một lớp Card rồi viết các phương thức để hoạt động với đối tượng Card và mảng chứa Card.
2. Trong Chương 14 ta sẽ tạo lập một lớp Deck rồi viết các phương thức hoạt động với các đối tượng Deck.
3. Trong Chương 15 tôi sẽ trình bày về lập trình hướng đối tượng (OOP) và ta sẽ chuyển đổi các lớp Card và Deck sang một phong cách giống như hướng đối tượng hơn.

Tôi nghĩ rằng tiến bước theo kiểu này khiến cho con đường đi dễ dàng hơn; song nhược điểm là ta sẽ thấy nhiều phiên bản của cùng đoạn mã lệnh, vì vậy có thể gây nhầm lẫn. Nếu được, bạn có thể tải về mã lệnh cho từng chương trong khi làm. Mã lệnh trong chương này ở đây: http://thinkapjava.com/code/Card1.java.

13.2 Các đối tượng Card

Nếu bạn chưa quen với bài tây, thì giờ là lúc thích hợp để kiểm một bộ, kẻo những gì trong chương này sẽ không có nhiều ý nghĩa. Hoặc bạn hãy đọc lấy http://en.wikipedia.org/wiki/Playing_card.

Có 52 lá bài trong một bộ, mỗi lá bài thuộc về một trong bốn chất và một trong 13 bậc. Các chất gồm Pích, Cơ, Rô, và Nhép (theo thứ tự giảm dần trong trò *bridge*). Các bậc gồm có A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, và K. Tùy theo trò chơi mà bạn quân A có thể cao hơn K hoặc thấp hơn 2.

Nếu bạn muốn định nghĩa một đối tượng mới để biểu diễn cho lá bài, rõ ràng các thuộc tính phải là: `rank` (bậc) và `suit` (chất). Còn việc chọn kiểu dữ liệu cho các thuộc tính lại không hiển nhiên. Một khả năng là dùng các chuỗi gồm những từ như `"Spade"` (Pích) cho chất và `"Queen"` cho bậc. Một vấn đề đặt ra với cách làm này là sẽ không dễ so sánh xem lá bài nào có bậc hoặc chất cao hơn.

Một cách khác là dùng số nguyên để **đánh số** cho các bậc và chất. Ở đây, "đánh số" không có nghĩa là ý mã hóa hoặc dịch thông điệp ra dạng mật mã như nhiều người thường nghĩ. Mà đối với nhà khoa học máy tính, "đánh số" nghĩa là "lập một phép ánh xạ từ con số đến dữ liệu cần biểu thị." Chẳng hạn:

Spades (Pích) ↦ 3
Hearts (Cơ) ↦ 2
Diamonds (Rô) ↦ 1
Clubs (Nhép) ↦ 0

Mã số này giúp so sánh các lá bài dễ hơn; vì chất cao hơn được ánh xạ đến số lớn hơn, và ta có thể so sánh chất bằng cách so các mã số của chúng. Ánh xạ đối với bậc thì khá dễ thấy; mỗi bậc số thì ánh xạ đến chính số nguyên tương ứng, còn với các bậc chữ:

J ↦ 11
Q ↦ 12
K ↦ 13

Ở đây tôi dùng kí hiệu toán học để biểu diễn ánh xạ là do ánh xạ không phải là một phần của chương trình. Đó là một phần của khâu thiết kế chương trình, nhưng không xuất hiện một cách cụ thể trên mã lệnh. Lời định nghĩa lớp cho kiểu `Card` sẽ như sau:

```
class Card {
  int suit, rank;

  public Card() {
    this.suit = 0;
    this.rank = 0;
  }

  public Card(int suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }

}
```

Như thường lệ, tôi cung cấp hai constructor: một cái nhận mỗi tham số ứng với từng biến thực thể; cái kia thì không nhận tham số nào.

Để tạo nên một đối tượng biểu diễn lá bài 3 Nhép, ta kích hoạt new:

```
    Card threeOfClubs = new Card(0, 3);
```

Đối số thứ nhất, 0 biểu thị chất Nhép.

13.3 Phương thức printCard

Khi bạn tạo nên một lớp mới, bước đầu tiên là khai báo các biến thực thể và viết các constructor. Bước thứ hai là viết những phương thức tiêu chuẩn mà từng đối tượng đều nên có, gồm một phương thức để in đối tượng ra, và một hoặc hai phương thức để so sánh các đối tượng. Ta hãy bắt đầu với printCard. Để in ra đối tượng Card theo cách mà mọi người dễ đọc, ta cần ánh xạ từ mã số đến các bậc và chất tương ứng. Một cách làm tự nhiên là dùng mảng chứa các chuỗi. Bạn có thể tạo một mảng các chuỗi theo cách giống như đã tạo ra mảng chứa những kiểu dữ liệu nguyên thuỷ:

```
    String[] suits = new String[4];
```

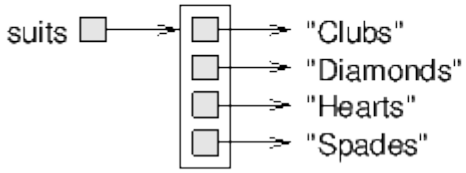
Sau đó ta có thể đặt giá trị của các phần tử trong mảng này.

```
    suits[0] = "Clubs";
    suits[1] = "Diamonds";
    suits[2] = "Hearts";
    suits[3] = "Spades";
```

Việc tạo ra một mảng và khởi tạo các phần tử trong nó là một thao tác thường gặp đến nỗi Java cung cấp luôn một cú pháp đặc biệt cho nó:

```
    String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
```

Câu lệnh này tương đương với các lệnh khai báo, huy động và gán. Sơ đồ trạng thái cho mảng này sẽ như sau:



Các phần tử của mảng này là những *tham chiếu* đến các chuỗi, thay vì là bản thân các chuỗi.
Bây giờ ta cần một mảng các chuỗi khác để giải mã các bậc của lá bài:

```
    String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6", "7", "8", "9",
"10", "Jack", "Queen", "King" };
```

Lý do có mặt "narf" là để đứng vào chỗ phần tử thứ không của mảng, vốn chẳng bao giờ được dùng đến (hay lẽ ra không có). Các bậc hợp lý chỉ có từ 1–13. Để tránh phần tử thường này, ta đã có thể bắt đầu từ 0, nhưng việc ánh xạ sẽ tự nhiên hơn nếu ta mã hóa 2 là 2, và 3 là 3, v.v.
Với các mảng này, ta có thể chọn được String thích hợp bằng cách dùng chỉ số là suit và rank. Trong phương thức printCard,

```
  public static void printCard(Card c) {
    String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
    String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6", "7", "8", "9",
"10", "Jack", "Queen", "King" };
    System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
  }
```

biểu thức suits[c.suit] có nghĩa là “dùng biến thực thể suit từ đối tượng c làm chỉ số trong mảng có tên suits, rồi chọn chuỗi thích hợp.” Kết quả của đoạn mã này

```
    Card card = new Card(1, 11);
    printCard(card);
```

là Jack of Diamonds.

13.4 Phương thức sameCard

Từ “same” (giống nhau, cùng) là một trong những hiện tượng ngôn ngữ trong tiếng Anh mà có vẻ ngoài quá rõ ràng, nhưng khi bạn suy nghĩ thì sẽ thấy còn có nhiều điều hơn bạn chờ đón ban đầu.

Chẳng hạn, nếu nói rằng “Chris và tôi có cùng (loại) xe,” thì tôi muốn nói rằng hai chiếc xe cùng nhãn hiệu, song là hai chiếc khác nhau. Còn nếu nói “Chris và tôi có cùng mẹ,” thì ý rằng mẹ cậu ta và mẹ tôi cùng là một người. Bởi vậy ý nghĩa của “cùng” thì lại khác nhau tùy theo ngữ cảnh.

Khi nói về các đối tượng, ta cũng gặp sự mập mờ tương tự. Chẳng hạn, nếu hai Card như nhau, thì liệu có nghĩa là chúng có cùng dữ liệu (bậc và chất), hay đó thực ra cùng là một đối tượng Card?
Để xem liệu có phải hai tham chiếu cùng chỉ đến một đối tượng hay không, ta dùng toán tử ==. Chẳng hạn:

```
    Card card1 = new Card(1, 11);
    Card card2 = card1;
```

```
if (card1 == card2) {
  System.out.println("card1 và card2 giống hệt nhau.");
}
```

Các tham chiếu đến cùng đối tượng thì **giống hệt** nhau. Còn các tham chiếu đến các đối tượng với dữ liệu như nhau thì sẽ **tương đồng** với nhau.

Để kiểm tra sự tương đồng, người ta thường viết một phương thức có tên gọi kiểu như sameCard.

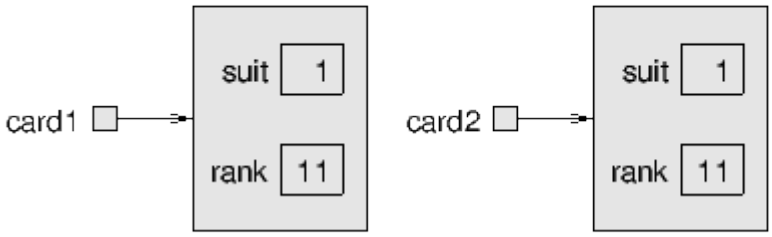
```
public static boolean sameCard(Card c1, Card c2) {
  return(c1.suit == c2.suit && c1.rank == c2.rank);
}
```

Sau đây là một ví dụ để tạo nên hai đối tượng có dữ liệu giống nhau, rồi dùng sameCard để kiểm tra xem liệu chúng có tương đồng không:

```
Card card1 = new Card(1, 11);
Card card2 = new Card(1, 11);
if (sameCard(card1, card2)) {
  System.out.println("card1 and card2 tuong dong nhau.");
}
```

Nếu các tham chiếu giống nhau thì chúng tương đồng. Song nếu chúng tương đồng thì chưa chắc chúng đã giống hệt nhau.

Ở đây, card1 và card2 tương đồng nhưng không giống hệt, cho nên sơ đồ trạng thái sẽ như sau:



Sơ đồ này sẽ trông thế nào nếu card1 và card2 giống hệt nhau?

Ở Mục 8.10, tôi đã nói rằng bạn không nên dùng toán tử == đối với String vì nó sẽ không hoạt động theo ý mình. Thay vì việc so sánh nội dung của các String (so sánh tương đồng), nó lại đi kiểm tra xem hai String này có phải cùng đối tượng (giống hệt) không.

13.5 Phương thức compareCard

Với những kiểu nguyên thủy, các toán tử điều kiện so sánh hai giá trị rồi quyết định xem cái nào lớn hay nhỏ hơn các kia. Những toán tử như vậy (< và > cùng những cái khác) không hoạt động được với kiểu đối tượng. Với các chuỗi, Java cung cấp một phương thức compareTo. Còn với Cards thì ta phải tự viết phương thức riêng, mà ta sẽ gọi là compareCard. Sau này, ta sẽ dùng phương thức này để sắp xếp một cỗ bài.

Có những tập hợp được xếp thứ tự hoàn toàn, theo nghĩa là bạn có thể so sánh hai phần tử bất kì trong đó để biết được phần tử nào lớn hơn. Lại có những tập hợp không sắp xếp được, theo nghĩa là chẳng có nghĩa lý gì để nói rằng phần tử này lớn hơn phần tử kia. Các số nguyên và số phẩy động là loại thứ tự

hoàn toàn. Còn các loại trái cây là không có thứ tự, vì vậy mà ta không thể so sánh táo với cam được. Trong Java, kiểu boolean là không thứ tự; ta không thể nói rằng true lớn hơn false.
Tập hợp các lá bài thì lại phần nào được xếp thứ tự, có nghĩa rằng đôi khi ta có thể so sánh lá bài và đôi khi không. Chẳng hạn, tôi biết rằng cây 3 Nhép thì cao hơn 2 Nhép và 3 Rô thì cao hơn 3 Nhép. Nhưng lá bài nào hơn, 3 Nhép hay 2 Rô? Một lá thì có bậc cao hơn, nhưng lá kia thì có chất cao hơn.

Để làm cho các lá bài so sánh được với nhau, ta phải quyết định xem thứ nào quan trọng hơn, bậc hay chất. Cách lựa chọn là tùy ý, nhưng khi bạn mua một cỗ bài mới, thì các quấn Nhép được xếp cạnh nhau, sau đó là các quân Rô, rồi cứ như vậy. Bởi thế ta hãy coi rằng chất thì quan trọng hơn.

Khi đã quyết định như vậy, ta có thể viết compareCard. Phương thức này nhận tham số là hai Card rồi trả lại 1 nếu lá bài thứ nhất hơn, -1 nếu lá bài thứ hai hơn, và 0 nếu chúng tương đồng.
Trước tiên, ta so sánh chất:

```
    if (c1.suit > c2.suit) return 1;
    if (c1.suit < c2.suit) return -1;
```

Nếu hai câu lệnh trên chẳng có câu lệnh nào đúng, thì các chất phải bằng nhau, và ta phải so sánh bậc:

```
    if (c1.rank > c2.rank) return 1;
    if (c1.rank < c2.rank) return -1;
```

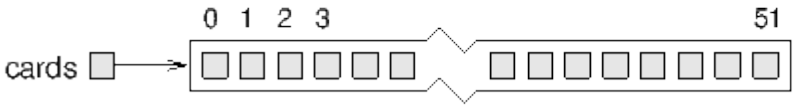
Nếu lại chẳng có câu nào đúng, thì hai bậc phải bằng nhau, và vì vậy ta phải trả lại 0.

13.6 Mảng các lá bài

Đến giờ ta đã thấy một vài ví dụ về phép hợp (khả năng kết hợp những đặc điểm của ngôn ngữ lập trình theo nhiều cách bố trí khác nhau). Một trong những ví dụ đầu tiên ta bắt gặp là việc dùng phép kích hoạt phương thức như là một phần của biểu thức. Một ví dụ khác là cấu trúc lồng ghép gồm các câu lệnh: bạn có thể đặt một lệnh if bên trong một vòng lặp while, hay bên trong một lệnh if khác, v.v.
Khi đã biết được dạng như vậy, và đã học được về mảng và đối tượng, thì có lẽ bạn chẳng ngạc nhiên khi được biết rằng ta có thể tạo nên mảng chứa những đối tượng. Và bạn có thể định nghĩa những đối tượng có biến thực thể là các mảng; bạn có thể lập nên những mảng chứa mảng khác; bạn có thể định nghĩa đối tượng chứa đối tượng khác, v.v. Trong hai chương tiếp theo, ta sẽ thấy những ví dụ về cách kết hợp như vậy trên cơ sở các đối tượng Card.
Ví dụ này tạo nên một mảng gồm 52 quân bài:

```
    Card[] cards = new Card[52];
```

Sau đây là sơ đồ trạng thái cho đối tượng này:



Mảng hiện tại có chứa các *tham chiếu* đến đối tượng; nó không chứa bản thân các đối tượng Card. Những phần tử này đều được khởi tạo vềnull. Bạn có thể truy cập từng phần tử trong mảng theo cách

thông thường:

```
if (cards[0] == null) {
  System.out.println("Chưa có quân bài nào!");
}
```

Nhưng nếu bạn cố thử truy cấp các biến thực thể của những Card chưa tồn tại, bạn sẽ nhận được biệt lệ NullPointerException.

```
cards[0].rank; // NullPointerException
```

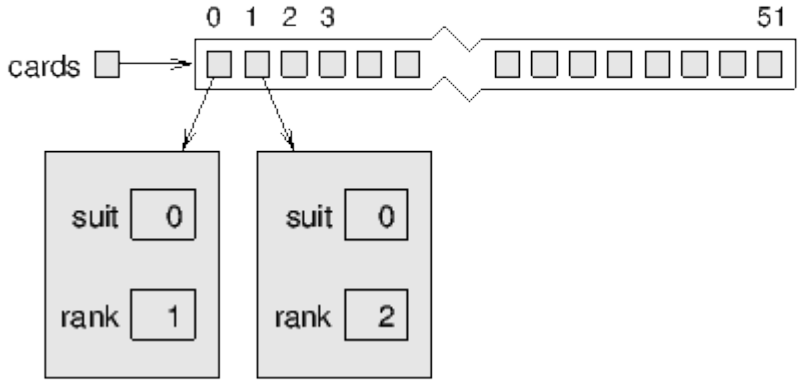
Nhưng đó lại là cú pháp đúng để truy cập rank (bậc) của lá bài "thứ không" trong cỗ. Đây là một ví dụ khác của phép hợp, bằng cách kết hợp cú pháp truy cập phần tử của mảng và truy cập một biến thực thể của đối tượng.

Cách dễ nhất để điền những đối tượng Card đầy vào cỗ bài là viết những vòng lặp for lồng ghép (nghĩa là vòng lặp này đặt trong vòng lặp khác):

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
  for (int rank = 1; rank <= 13; rank++) {
    cards[index] = new Card(suit, rank);
    index++;
  }
}
```

Vòng lặp ngoài cùng đếm các chất từ 0 tới 3. Với từng chất, vòng lặp trong đếm các bậc từ 1 đến 13. Vì vòng lặp ngoài chạy 4 lần, và vòng lặp trong chạy 13 lần, nên phần thân được thực hiện 52 lần.

Tôi đã dùng index để theo dõi lá bài tiếp theo sẽ cần phải đặt vào đâu trong cỗ bài. Sơ đồ trạng thái sau đây cho thấy cỗ bài như thế nào sau khi hai lá bài đầu tiên được huy động:



13.7 Phương thức printDeck

Khi làm việc với mảng, cách tiện lợi là có một phương thức để in ra nội dung. Ta đã vài lần thấy đượng dạng mẫu cho việc duyệt mảng, bởi vậy phương thức sau sẽ quen thuộc đối với bạn:

```
public static void printDeck(Card[] cards) {
  for (int i = 0; i < cards.length; i++) {
    printCard(cards[i]);
```

```
    }
  }
```

Vì cards có kiểu là Card[], nên một phần tử của cards thì có kiểu là Card. Bởi vậy cards[i] là một đối số hợp lệ cho printCard.

13.8 Tìm kiếm

Phương thức tiếp theo mà tôi sẽ viết là findCard, để tìm kiếm trong một mảng chứa Card, xem liệu rằng mảng này có chứa một lá bài cụ thể hay không. Phương thức này cho tôi một cơ hội biểu diễn hai thuật toán: **tìm kiếm tuyến tính** và **tìm kiếm phân đôi**.

Tìm kiếm tuyến tính thật dễ hiểu; ta duyệt cả cỗ bài rồi so sánh từng lá bài với lá mà ta đang tìm. Nếu thấy, ta sẽ trả về chỉ số tại đó lá bài xuất hiện. Nếu không có trong cỗ bài, ta trả về -1.

```
public static int findCard(Card[] cards, Card card) {
  for (int i = 0; i< cards.length; i++) {
    if (sameCard(cards[i], card)) {
      return i;
    }
  }
  return -1;
}
```

Các đối số của findCard là card và cards. Dường như thật kì quặc khi có một biến cùng tên với kiểu dữ liệu (biến card thuộc kiểu Card). Ta có thể nhận thấy sự khác biệt vì biến bắt đầu bằng chữ cái thường. Phương thức này trả lại ngay khi nó phát hiện ra lá bài cần tìm, ncos nghĩa là ta không cần phải duyệt cả cỗ bài nếu đã tìm được lá bài ta cần. Còn nếu ta đến điểm cuối vòng lặp, ta biết rằng lá bài đó không có trong cỗ.

Nếu các quân bài trong cỗ không được sắp xếp, thì chẳng có cách tìm kiếm nào nhanh hơn cách này. Ta phải nhìn từng lá bài một, bởi nếu không ta sẽ không chắc rằng quân bài mong muốn không ở đó.

Nhưng khi bạn tra từ trong một cuốn từ điển, bạn lại không tìm tuyến tính qua từng từ một, bởi lẽ các từ đều được xếp thứ tự rồi. Do vậy, có khả năng bạn sẽ dùng một thuật toán tương tự như tìm kiếm chia đôi:

1. Bắt đầu ở một chỗ giữa cuốn từ điển.
2. Chọn một từ trên trang đó rồi so sánh với từ cần tra.
3. Nếu bạn tìm thấy từ cần tra thì dừng lại.
4. Nếu từ cần tra xếp sau từ thấy được trên trang, thì hãy lật đến một chỗ nào đó phía sau của cuốn từ điển, rồi trở lại bước 2.
5. Nếu từ cần tra xếp trước từ thấy được trên trang, thì hãy lật đến một chỗ nào đó phía trước của cuốn từ điển, rồi trở lại bước 2.

   Nếu bạn đã tìm đến chỗ mà có hai từ liền kề nhau trong một trang, và từ cần tra lại nằm giữa hai từ đó, thì có thể kết luận rằng từ cần tra không có trong cuốn từ điển.

Quay trở lại với cỗ bài, nếu ta biết rằng các lá bài đã được xếp thứ tự, thì ta có thể viết một phiên bản khác findCard, nhưng chạy nhanh hơn. Cách tốt nhất để viết phương thức tìm kiếm chia đôi là dùng cách đệ quy, bởi việc chia đôi về bản chất là mang tính đệ quy.

Một mẹo là viết một phương thức có tên findBisect trong đó nhận vào tham số là hai chỉ số, low và high, quy định đoạn trong mảng cần được tìm kiếm (bao gồm cả low và high).

1. Để tìm kiếm trên mảng, hãy chọn một chỉ số giữa low và high (gọi nó là mid) rồi so sánh nó với lá bài cần tìm.
2. Nếu bạn đã tìm thấy nó thì dừng lại.
3. Nếu lá bài tại mid cao hơn lá bài cần tìm, thì tìm kiếm trong khoảng từ low đến mid-1.
4. Nếu lá bài tại mid thấp hơn lá bài cần tìm, thì tìm kiếm trong khoảng từ mid+1 đến high.

Các bước 3 và 4 trông giống những lời gọi đệ quy đến mức đáng ngờ. Sau đây là toàn bộ ý tưởng khi chuyển thành mã lệnh Java:

```
public static int findBisect(Card[] cards, Card card, int low, int high) {
  // CẦN LÀM: một trường hợp cơ sở
  int mid = (high + low) / 2;
  int comp = compareCard(cards[mid], card);
  if (comp == 0) {
    return mid;
  } else if (comp > 0) {
    return findBisect(cards, card, low, mid-1);
  } else {
    return findBisect(cards, card, mid+1, high);
  }
}
```

Mã lệnh này có chứa phần cốt lõi của phép tìm kiếm chia đôi, song vẫn thiếu một phần trong trọng, đó là lý do mà tôi đã ghi chú "CẦN LÀM". Như đã viết, phương thức này sẽ lặp đệ quy mãi mãi nếu như lá bài không có trong cỗ. Ta cần một trường hợp cơ bản để xử lý tình huống này.

Nếu high nhỏ hơn low, thì không có lá bài nào giữa chúng, bởi vậy ta sẽ kết luận rằng lá bài cần tìm không có trong cỗ. Nếu ta xử lý được trường hợp đó, thì phương thức sẽ hoạt động đúng:

```
public static int findBisect(Card[] cards, Card card, int low, int high) {
  System.out.println(low + ", " + high);
  if (high < low) return -1;
  int mid = (high + low) / 2;
  int comp = compareCard(cards[mid], card);
  if (comp == 0) {
    return mid;
  } else if (comp > 0) {
    return findBisect(cards, card, low, mid-1);
```

```
    } else {
      return findBisect(cards, card, mid+1, high);
    }
  }
```

Tôi đã bổ sung một lệnh in để có thể theo dõi được một loạt những lần kích hoạt đệ quy. Tôi đã thử đoạn mã sau:

```
    Card card1 = new Card(1, 11);
    System.out.println(findBisect(cards, card1, 0, 51));
```

và nhận được kết quả dưới đây:

```
0, 51
0, 24
13, 24
19, 24
22, 24
23
```

Sau đó tôi lập một lá bài không có trong cỗ (15 Rô), và thử cố tìm nó. Tôi đã nhận được kết quả:

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
-1
```

Những phép thử này không chứng minh được rằng chương trình đúng đắn. Thực tế là bao nhiêu kiểm thử cũng không thể chứng minh được tính đúng đắn nói trên. Song qua việc xem xét một vài trường hợp và kiểm tra mã lệnh, bạn có thể tự thuyết phục bản thân.

Số lần kích hoạt đệ quy thường từ 6 đến 7, vì vậy ta chỉ kích hoạt compareCard có 6 hoặc 7 lần thôi, so với tận 52 lần nếu tìm kiếm tuyến tính. Nói chung, phép chia đôi thì nhanh hơn nhiều so với tìm kiếm tuyến tính, và còn nhanh nữa với các mảng lớn.
Có hai lỗi thường gặp trong chương trình đệ quy, đó là quên đưa vào trường hợp cơ sở và viết lời gọi đệ quy song không bao giờ dẫn đến trường hợp cơ sở. Lỗi sai nào cũng dẫn đến đệ quy vô hạn, và biệt lệ StackOverflowException sẽ được phát ra. (Hãy hình dung một sơ đồ ngăn xếp cho một phương thức đệ quy không bao giờ kết thúc.)

13.9 Cỗ bài và cỗ bài con

Sau đây là nguyên mẫu (xem Mục 8.5) của findBisect:

```
  public static int findBisect(Card[] deck, Card card, int low, int high)
```

Ta có thể coi cards, low, và high chỉ là một thông số quy định một **cỗ bài con**. Cách suy nghĩ này rất

thông dụng, và đôi khi được gọi là**tham số trừu tượng**. Ở đây, "trừu tượng" có nghĩa là thứ mà đúng ra không có mặt trên mã lệnh chương trình, nhưng lại diễn tả tính năng của chương trình theo cấp độ ý tưởng cao hơn.
Chẳng hạn, khi bạn kích hoạt một phương thức rồi truyền vào một mảng cùng với các giới hạn low và high, không có gì ngăn cản ươợc phương thức đã kích hoạt khỏi truy cập phần của mảng nằm ngoài phạm vi giới hạn nói trên. Bởi vậy thật ra bạn không gửi một tập con của cỗ bài; bạn đang gửi toàn bộ cỗ bài. Nhưng miễn là bộ phận tiếp nhận (tức là phần nội dung phương thức) tuân theo luật chơi, thì ta có thể coi rằng đó chính là một cỗ bài con.
Hình thức suy nghĩ này, trong đó chương trình có hàm ý cao xa hơn là những câu mã lệnh, chính là một phần quan trọng trong tư duy nhà khoa học máy tính. Từ "trừu tượng" đã xuất hiện quá nhiều trong nhiều ngữ cảnh khác nhau và điều này khiến cho ý nghĩa của nó bị loãng đi. Mặc dù vậy, **trừu tượng** chính là một ý tưởng trọng tâm trong ngành khoa học máy tính (cũng như nhiều ngành khác).

Một định nghĩa khái quát hơn cho "trừu tượng" là "Quá trình mô hình hóa một hệ thống phức tạp bằng diễn giải được giản hóa, nhằm lược đi những chi tiết không liên quan đồng thời nắm bắt được những động thái mà ta cần quan tâm."

13.10 Thuật ngữ

mã hóa:
Việc biểu diễn một tập hợp các giá trị bằng một tập hợp các giá trị khác, bằng việc thiết lập một ánh xạ giữa chúng.
giống hệt:
Sự bằng nhau giữa các tham chiếu. Hai tham chiếu chỉ đến cùng một đối tượng trong bộ nhớ.
tương đồng:
Sự bằng nhau giữa các giá trị. Hai tham chiếu chỉ đến hai đối tượng chứa dữ liệu giống nhau.
tham số trừu tượng:
Một tập hợp gồm các tham số hoạt động cùng nhau như một tham số thống nhất.
trừu tượng:
> *Quá trình diễn giải một chương trình (hay thứ khác) ở một cấp độ cao hơn so với những gì được viết dưới dạng mã lệnh.*

13.11 Bài tập

Bài tập 1 Hãy gói bọc mã lệnh trong Mục 13.5 vào một phương thức. Sau đó chỉnh sửa nó để bậc của Át cao hơn K.
Bài tập 2 Hãy gói bọc mã lệnh thiết lập cỗ bài ở Mục 13.6 vào trong một phương thức có tên makeDeck không nhận tham số nào và trả lại một mảng đã điền đầy đủ những lá bài (Card).
Bài tập 3 Trong trò chơi Blackjack, mục tiêu là lấy được một nhóm cây bài có tổng điểm bằng 21. Điểm của nhóm bài bằng tổng các điểm trên những cây bài. Điểm cho những quân Át bằng 1, cho những quân bài mặt người bằng 10, và những quân khác thì điểm đúng bằng bậc. Chẳng hạn, nhóm ba quân bài (Ace, 10, Jack, 3) có tổng điểm là 1 + 10 + 10 + 3 = 24. Hãy viết một phương thức có tên handScore nhận vào đối óố là một mảng những lá bài rồi trả lại tổng điểm.
Bài tập 4 Trong trò chơi Poker, một "dây" (flush) là một nhóm lá bài có từ 5 lá trở lên cùng chất. Một nhóm bài có thể chứa bao nhiêu lá bài cũng được.

1. Hãy viết một phương thức có tên suitHist nhận tham số là một mảng gồm những Card rồi trả lại một histogram các chất trong nhóm. Lời giải của bạn chỉ được duyệt mảng đúng một lần.
2. Hãy viết một phương thức hasFlush nhận tham số là một mảng những Card rồi trả lại true nếu nhóm bài có chứa dây, và false nếu không.

   Bài tập 5 Làm việc với những cây bài sẽ hay hơn nếu bạn hiển thị được chúng lên màn hình. Nếu bạn chưa từng thử những ví dụ đồ họa ở Phụ lục A, bây giờ có thể sẽ là lúc thích hợp. Trước hết, hãy tải về http://thinkapjava.com/code/CardTable.java và http://thinkapjava.com/code/cardset.zip vào cùng một thư mục. Sau đó, giải nén cardset.zip, vốn có chứa một thư mục con cardset-oxymoron với tất cả hình của những quân bài. (Lưu ý rằng biếncardset trong CardTable.main chính là tên của thư mục này.) Chạy CardTable.java và bạn có thể thấy hình ảnh một cỗ bài trải ra trên bàn màu xanh. Bạn có thể dùng lớp này để khởi đầu lập nên những trò chơi bài riêng.

Chương 14: Đối tượng chứa các mảng

Trở về Mục lục cuốn sách

CẢNH BÁO: Trong chương này, ta tiến thêm một bước nữa về lập hướng đối tượng nhưng vẫn chưa hẳn đến được đó. Bởi vậy, nhiều ví dụ vẫn chưa đúng giọng Java, nghĩa là chưa phải mã lệnh Java chuẩn. Hình thức trung chuyển này (hi vọng rằng) sẽ giúp bạn học, nhưng thực tế tôi không viết mã lệnh như thế này.

Bạn có thể tải về mã lệnh cho chương này từ: http://thinkapjava.com/code/Card2.java.

14.1 Lớp Deck

Ở chương trước, ta đã làm việc với một mảng các đối tượng, nhưng cũng đề cập rằng hoàn toàn có thể có đối tượng có chứa biến thực thể là mảng. Trong chương này, ta tạo ra một đối tượng Deck có chứa một mảng những đối tượng Card.

Lời định nghĩa lớp sẽ trông như sau:

```
class Deck {
  Card[] cards;

  public Deck(int n) {
    this.cards = new Card[n];
  }
}
```

Ở đây, constructor khởi tạo biến thực thể là một mảng những lá bài, nhưng nó không tạo nên lá bài nào. Sau đây là sơ đồ trạng thái cho thấy Deck mà không có lá bài nào kèm theo:

Dưới đây là một constructor không có đối số để tạo nên một cỗ bài 52 lá rồi điền đầy những đối tượng Card vào nó:

```
  public Deck() {
    this.cards = new Card[52];
    int index = 0;
    for (int suit = 0; suit <= 3; suit++) {
      for (int rank = 1; rank <= 13; rank++) {
        cards[index] = new Card(suit, rank);
        index++;
      }
    }
  }
```

Phương thức này tương tự như makeDeck; ta chỉ việc thay đổi cú pháp để nó trở thành một constructor. Để kích hoạt nó, ta dùng new:

```
    Deck deck = new Deck();
```

Bây giờ việc đặt các phương thức thuộc về các đối tượng Deck vào trong lời định nghĩa lớp Deck là hợp lý. Khi xem xét những phương thức mà ta đã viết cho đến giờ, dễ thấy một ứng cử viên đó là printDeck (Mục 13.7). Sau đây là dáng vẻ của nó, được viết lại để hoạt động với Deck:

```
public static void printDeck(Deck deck) {
  for (int i = 0; i < deck.cards.length; i++) {
    Card.printCard(deck.cards[i]);
  }
}
```

Một sự thay đổi là kiểu của tham số, từ Card[] sang Deck.
Thay đổi thứ hai là ta không còn dùng được deck.length để lấy chiều dài của mảng, bởi giờ đây deck đã là một đối tượng Deck, chứ không phải một mảng. Nó chứa một mảng, nhưng nó không phải là mảng. Bởi v ậy ta phải viết deck.cards.length để kết xuất được mảng từ đối tượng Deck rồi lấy chiều dài của mảng này.
Với lý do tương tự, ta phải dùng deck.cards[i] để truy cập một phần tử của mảng, thay vì chỉ viết deck[i].
Sự thay đổi cuối cùng là việc kích hoạt printCard phải nói rõ rằng printCard được định nghĩa trong lớp Card.

14.2 Tráo bài

Trong phần lớn các trò chơi bài tây, bạn cần phải tráo cỗ bài; nghĩa là xếp bài theo một trật tự ngẫu nhiên. Ở Mục 12.6 ta đã thấy cách phát sinh số ngẫu nhiên, song thật không dễ thấy cách áp dụng để tráo cỗ bài.

Một khả năng là mô phỏng cách con người tráo bài, thường là chia cỗ bài làm đôi rồi chọn bài đan xen từ từng phần. Bởi người thường không thể tráo chính xác theo cách này được, nên sau chừng 7 lần lặp lại thao tác thì cỗ bài dường như đã hoàn toàn ngẫu nhiên. Song một chương trình máy tính thì lại có đặc điểm luôn trộn bài thật hoàn hảo nên kết quả sẽ không thật ngẫu nhiên. Thực tế là, sau 8 lần trộn, máy sẽ làm cho cỗ bài về nguyên trạng. Bạn có thể xem thông tin thêm ở http://en.wikipedia.org/wiki/Faro_shuffle.
Một thuật toán trộn bài hợp lý hơn là trong mỗi lần duyệt chỉ lật một lá bài, và mỗi lần lặp thì chọn lấy hai lá bài rồi đổi chỗ chúng.

Sau đây là phác thảo cách hoạt động của thuật toán này. Để phác họa chương trình, tôi kết hợp câu lệnh Java với ngôn ngữ nói, mà đôi khi được gọi là **giả mã**:

```
    for (int i = 0; i < deck.cards.length; i++) {
      // chọn một số nằm giữa 1 và deck.cards.length-1
      // đổi chỗ lá bài thứ i và lá bài ngẫu nhiên được chọn
    }
```

Điều hay ở giả mã là nó thường làm rõ những phương thức nào mà bạn sắp cần có. Trong trường hợp này, ta cần một thứ như randomInt, để chọn một số nguyên ngẫu nhiên giữa low và high, và swapCards để nhận vào hai chỉ số rồi đổi chỗ hai lá bài ở vị trí các chỉ số đó.

Quá trình này—viết giả mã trước rồi mới viết phương thức thực hiện sau—được gọi là **phát triển từ trên xuống** (xem http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design).

14.3 Sắp xếp

Bây giờ khi đã làm cỗ bài lẫn lung tung lên, ta cần một cách khiến nó trở lại trật tự. Có một thuật toán sắp xếp giống với thuật toán trộn đến không ngờ. Nó được gọi là **sắp xếp chọn** bởi nó hoạt động dựa trên việc duyệt mảng lặp đi lặp lại và mỗi lần duyệt thì chọn lấy lá bài thấp nhất còn lại.

Trong lần lặp thứ nhất, ta tìm lấy lá bài thấp nhất rồi đổi chỗ cho lá bài ở vị trí thứ 0. Trong lần lặp thứ i, ta tìm lấy lá bài thấp nhất bên phải vị trí i rồi đổi chỗ nó cho lá bài thứ i.
Sau đây là giả mã cho cách sắp xếp chọn:

```
    for (int i = 0; i < deck.cards.length; i++) {
      // tìm lấy lá bài thấp nhất tại vị trí i, hoặc bên phải chỗ đó
      // đổi chỗ lá bài thứ i với lá bài thấp nhất tìm được }
```

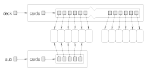
Một lần nữa, giả mã giúp cho việc thiết kế các **phương thức trợ giúp**. Trong trường hợp này, ta có thể dùng lại swapCards, bởi vậy ta chỉ cần có một phương thức mới, có tên indexLowestCard, để nhận một mảng những lá bài và một chỉ số nơi cần bắt đầu tìm kiếm.

14.4 Cỗ bài con

Vậy ta nên biểu diễn một phần bài hay một dạng tập con của cỗ bài đủ như thế nào? Một khả năng là tạo nên một lớp mới có tên Hand, và nó có thể mở rộng Deck. Một khả năng khác, như tôi trình bày ở đây, là biểu diễn một phần bài bằng đối tượng Deck nhưng có ít hơn 52 lá bài.
Ta có thể muốn một phương thức, subdeck, để nhận một cỗ bài và một khoảng chỉ số, rồi trả lại một cỗ bài mới chứa tập con những lá bài đã chỉ định:

```
public static Deck subdeck(Deck deck, int low, int high) {
  Deck sub = new Deck(high-low+1);
  for (int i = 0; i<sub.cards.length; i++) {
    sub.cards[i] = deck.cards[low+i];
  }
  return sub;
}
```

Chiều dài của cỗ bài con là high-low+1 bởi cả lá bài thấp (low) lẫn cao (high) đều được tính vào. Cách tính có thể gây nhầm lẫn, và dẫn đến lỗi "lệch một". Cách tránh lỗi này tốt nhất thường là vẽ hình minh họa.
Vì ta đã cung cấp new vào một đối số, nên contructor được kích hoạt sẽ là cái đầu tiên, vốn chỉ huy động mảng mà không huy động bất kì lá bài nào. Bên trong vòng lặp for, cỗ bài con được điền đầy những bản sao tham chiếu từ cỗ bài lớn.
Dưới đây là sơ đồ trạng thái của cỗ bài con được tạo nên bằng những tham số low=3 và high=7. Kết quả là một phần bài gồm 5 lá được chung với cỗ bài ban đầu; nghĩa là theo cách đặt bí danh (alias).



Cách đặt bí danh này thường không phải ý tưởng hay, bởi những thay đổi trong một cỗ bài con lại được thể hiện ở những cỗ khác; đây không phải là động thái mà bạn trông đợi ở những bộ bài thật. Song nếu các lá bài là đối tượng không thay đổi được, thì việc đặt bí danh ít nguy hiểm hơn. Trong trường hợp này, có lẽ chẳng có lý do nào để thay đổi bậc hay chất của lá bài. Thay vì vậy, ta có thể mỗi lúc lại tạo ra một lá bài rồi coi nó như một đối tượng không thay đổi được. Bởi vậy, đối với Card, việc đặt bí danh là lựa chọn hợp lý.

14.5 Tráo bài và chia bài

Ở Mục 14.2, tôi đã viết giả mã cho thuật toán trộn bài. Coi như ta đã có một phương thức shuffleDeck nhận tham số là một cỗ bài rồi trộn nó lên, ta có thể sử dụng nó để chia thành nhiều phần bài:

```
Deck deck = new Deck();

shuffleDeck(deck);

Deck hand1 = subdeck(deck, 0, 4);

Deck hand2 = subdeck(deck, 5, 9);

Deck pack = subdeck(deck, 10, 51);
```

Mã lệnh này đặt 5 lá bài đầu tiên vào phần thứ nhất, 5 lá bài kế tiếp vào phần thứ hai, và phần còn lại giữ ở cỗ.

Khi bạn hình dung cách chia bài, bạn có nghĩ rằng ta nên chia vòng tròn như đánh bài thật không? Tôi cũng nghĩ về điều này, song nhận thấy rằng với chương trình máy tính thì như vậy không cần thiết. Quy tắc chia vòng tròn chỉ để giảm thiểu khả năng tráo bài chưa kĩ và để cho người chia khó chơi ăn gian hơn. Hai điều này không thành vấn đề đối với máy tính.

Ví dụ này là lời nhắc hở hữu ích về một trong những nguy hiểm của phép ẩn dụ trong kĩ thuật: đôi khi ta quy định những hạn chế không cần thiết lên máy tính, hoặc trông chờ những tính năng không có sẵn, bởi ta đã không suy nghĩ khi mở rộng một hình ảnh ẩn dụ vượt quá giới hạn của nó rồi.

14.6 Sắp xếp trộn

Ở Mục 14.3, ta đã thấy một thuật toán sắp xếp đơn giản nhưng hóa ra không hiệu quả. Để sắp xếp n phần tử, nó phải duyệt mảng n lần, và mỗi lần duyệt tốn một khoảng thời gian tỉ lệ với n. Do đó, thời gian tổng cộng sẽ tỉ lệ với n_2.

Trong mục này tôi phác thảo một thuật toán hiệu quả hơn, có tên gọi **sắp xếp trộn**. Để sắp xếp n phần tử, phương pháp này tốn một thời gian tỉ lệ với $n \log n$. Như vậy có vẻ chưa ấn tượng lắm, song khi n lớn lên, hiệu số giữa n_2 và $n \log n$ có thể sẽ rất lớn. Hãy thử vài giá trị của n xem sao.

Ý tưởng cơ bản phía sau phép sắp xếp trộn là thế này: Nếu bạn có 2 phần bài, từng phần đã được sắp xếp rồi, thì rất dễ (và nhanh chóng) để trộn ghép chúng thành một cỗ bài duy nhất được sắp xếp đúng. Hãy thử làm điều này với một cỗ bài xem sao:

1. Hình thành hai phần bài với mỗi phần khoảng 10 lá rồi sắp xếp chúng để cho khi đặt ngửa mặt lên thì các lá bài thấp ở trên. Đặt hai phần bài này trước mặt bạn.
2. So sánh hai lá bài ở trên cùng của mỗi phần rồi chọn lá bài thấp hơn. Lật úp nó rồi đưa nó vào phần bài riêng được sắp xếp.
3. Lặp lại bước Hai đến khi một phần bài đã hết. Sau đó lấy những lá bài còn lại rồi thêm vào phần bài được sắp xếp.

Kết quả ta sẽ được một phần bài chung được xếp đúng. Sau đây là cách làm bằng giả mã:

```
  public static Deck merge(Deck d1, Deck d2) {
    // tạo nên một cỗ bài đủ lớn chứa hết các quân bài
    Deck result = new Deck(d1.cards.length + d2.cards.length);
    // dùng chỉ số i để theo dõi vị trí hiện tại trên
    // phần bài thứ nhất, và chỉ số j cho phần bài thứ hai
    int i = 0;
    int j = 0;
    // chỉ số k duyệt theo phần bài được xếp đúng
    for (int k = 0; k < result.cards.length; k++) {
      // nếu d1 rỗng thì d2 thắng; nếu d2 rỗng, d1 thắng;
      // nếu không, đi so sánh hai lá bài
      // thêm lá bài thắng vào phần bài được xếp đúng
    }
    return result;
}
```

Cách hay nhất để kiểm tra merge là lập nên và trộn một cỗ bài, dùng phương thức subdeck để hình thành nên hai phần bài (nhỏ), rồi dùng thủ tục sort từ chương trước để sắp xếp hai nửa. Sau đó bạn có thể truyền hai nửa này vào merge để xem phương thức mới này có hoạt động không.
Nếu bạn có thể làm cho mọi thứ như trên hoạt động được, hãy thử một phiên bản mergeSort đơn giản:

```
  public static Deck mergeSort(Deck deck) {
    // tìm điểm giữa của cỗ bài
    // chia cỗ bài thành hai phần nhỏ
    // xếp phần nhỏ bằng sortDeck
    // trộn hai nửa rồi trả lại kết quả
  }
```

Sau đó, nếu như bạn làm cho phương thức hoạt động được, sẽ có cái hay! Điều kì diệu về sắp xếp trộn là nó có tính đệ quy. Ở lúc mà bạn bắt đầu sắp xếp các phần bài, tại sao phải kích hoạt phương thức sort cũ và chậm chạp? Sao không kích hoạt phương thức mergeSort mới toanh, đang được viết? Đây không chỉ là một ý tưởng tốt, mà *cần thiết* phải đạt được ưu thế về hiệu năng của chương trình mà tôi đã hứa. Nhưng để chương trình hoạt động, bạn cần phải có một trường hợp cơ sở. Nếu không, đệ quy sẽ mãi mãi. Một trường hợp cơ sở đơn giản là một phần bài với 0 hoặc 1 lá bài. Nếu như mergesort nhận được một phần bài nhỏ như vậy, thì nó có thể trả lại kết quả y nguyên, bởi phần bài nhỏ này đã được sắp xếp.
Phiên bản đệ quy của mergesort có thể sẽ trông như sau:

```
  public static Deck mergeSort(Deck deck) {
    // nếu cỗ bài chỉ gồm 0 hoặc 1 lá bài, thì trả lại nó
    // tìm ra điểm giữa của cỗ bài
    // chia cỗ bài thành hai phần bài
```

```
    // sắp xếp các phần bài này bằng mergesort
    // trộn ghép lại hai nửa rồi trả lại kết quả
  }
```

Như thường lệ, có hai cách nghĩ về chương trình đệ quy: Bạn có thể nghĩ qua toàn bộ luồng thực thi, hay bạn có thể dựa vào "niềm tin" (xem Mục 6.9). Tôi đã xây dựng ví dụ này để khuyến khích bạn tư duy theo niềm tin như vậy.

Khi dử dụng sortDeck để sắp xếp các phần bài, bạn không thấy bị thôi thúc phải theo luồng thực thi, phải không? Bạn chỉ việc giả sử rằng nó hoạt động được vì bạn đã gỡ lỗi cho nó rồi. Ồ, tất cả những điều mà bạn đã làm cho mergeSort trở nên đệ quy là thay thế một thuật toán sắp xếp này với thuật toán khác. Không có lý do gì để đọc chương trình khác đi cả.
Thực ra, bạn phải nghĩ một chút mới lập được trường hợp cơ sở đúng đắn và đảm bảo rằng cuối cùng bạn sẽ đạt đến trường hợp cơ sở này. Song ngoài điều đó ra, việc viết nên phiên bản đệ quy hẳn sẽ không còn vấn đề nữa. Chúc bạn may mắn!

14.7 Biến lớp

Đến giờ ta đã thấy những biến địa phương, vốn được khai báo bên trong phương thức, và biến thực thể, vốn được khai báo ở lời định nghĩa lớp, thường đi trước các định nghĩa phương thức.

Các biến địa phương được tạo nên khi một phương thức được kích hoạt và phá hủy khi phương thức kết thúc. Các biến thực thể được tạo nên khi bạn tạo nên một đối tượng; các biến này bị phá hủy khi đối tượng bị thu hồi rác bộ nhớ.

Bây giờ là lúc biết về **biến lớp**. Giống như biến thực thể, biến lớp được định nghĩa trong một lời định nghĩa lớp trước những định nghĩa phương thức, song chúng được nhận diện bằng từ khóa static. Chúng được tạo nên khi chương trình khởi đầu và còn tồn tại đến tận khi chương trình kết thúc.
Bạn có thể tham chiếu tới biên thực thể từ bất kì đâu bên trong lời khai báo lớp. Những biến lớp thường được dùng để lưu các giá trị hằng số cần thiết ở nhiều chỗ.

Lấy ví dụ, sau đây là một phiên bản của Card trong đó suits và ranks là những biến lớp:

```
class Card {
  int suit, rank;
  static String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
  static String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6", "7", "8", "9",
"10", "Jack", "Queen", "King" };

  public static void printCard(Card c) {
    System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
  }
}
```

Bên trong printCard ta có thể tham chiếu tới suits và ranks như thể chúng là các biến địa phương.

14.8 Thuật ngữ

giả mã:
Một cách thiết kế chương trình bằng cách viết các bản nháp kết hợp ngôn ngữ nói và Java.

phương thức trợ giúp:
Thường là một phương thức nhỏ chẳng làm nhiều điều có ích, song phương thức này trợ giúp một phương thức khác có ích hơn.

biến lớp:
Một biến được khai báo trong lớp theo dạng static; luôn chỉ có một bản sao của biến này tồn tại.

14.9 Bài tập

Bài tập 1 Mục đích của bài tập này là thi hành các thuật toán tráo bài và sắp xếp trong chương.

1. Hãy tải về mã lệnh cho chương này từ http://thinkapjava.com/code/Card2.java rồi nhập nó vào môi trường phát triển đang dùng. Tôi đã cung cấp phần khung của những phương thức mà bạn sẽ viết, do vậy mà chương trình sẽ được biên dịch thông suốt. Nhưng khi chạy, nó sẽ in các dòng thông báo cho biết rằng những phương thức rỗng chưa làm việc được. Khi bạn hoàn thành hết các phương thức này, những lời thông báo đó sẽ biến mất.
2. Nếu đã làm Bài tập 3 trong Chương 12, bạn đã viết randomInt. Nếu không, bây giờ hãy viết phương thức này rồi thêm vào mã lệnh để kiểm tra.
3. Hãy viết một phương thức có thên swapCards để nhận vào một cỗ bài (mảng chứa những quân bài) cùng hai chỉ số, rồi đổi chỗ hai lá bài ở những vị trí đó. GỢI Ý: phương thức này phải đổi chỗ tham chiếu, chứ không phải nội dung của các đối tượng. Việc này nhanh hơn; đồng thời cũng xử lý đúng trường hợp các lá bài cùng tham chiếu đến một đối tượng (tức "đặt bí danh").
4. Viết một phương thức có tên shuffleDeck để dùng thuật toán trong Mục 14.2. Có thể bạn sẽ muốn dùng phương thức randomInt từ Bài tập 3 Chương 12.
5. Viết một phương thức có tên indexLowestCard để dùng phương thức compareCard nhằm tìm kiếm lá bài thấp nhất trong một khoảng cho trước của cỗ bài (từ lowIndex đến highIndex, kể cả hai đầu).
6. Viết một phương thức có tên sortDeck để sắp xếp cỗ bài từ thấp lên cao.
7. Dùng giả mã trong Mục 14.6, hãy viết phương thức có tên merge. Đảm bảo rằng bạn kiểm tra nó trước khi dùng nó làm một phần trong mergeSort.
8. Viết một dạng đơn giản của mergeSort, dạng chia cỗ bài làm đôi, rồi dùng sortDeck để sắp xếp hai nửa, và dùng merge để tạo nên cỗ bài mới, sắp xếp đúng.
9. Viết một dạng đệ quy hoàn chỉnh của mergeSort. Nhớ rằng sortDeck là một phương thức sửa đổi còn mergeSort là một hàm; như vậy chúng được kích hoạt theo cách khác nhau:

```
sortDeck(deck); // sửa đổi cỗ bài sẵn có

deck = mergeSort(deck); // thay thế cỗ bài cũ bằng cỗ mới
```

Chương 15: Lập trình hướng đối tượng

Trở về Mục lục cuốn sách

15.1 Các ngôn ngữ và phong cách lập trình

Có nhiều ngôn ngữ lập trình khác nhau, và không kém mấy về số lượng là các phong cách lập trình (còn gọi là mẫu hình). Những chương trình mà ta đã viết đến giờ đều thuộc phong cách **thủ tục**, bởi chú ý được dồn vào việc quy định các thủ tục tính toán.

Đa số các chương trình Java đều **hướng đối tượng**, có nghĩa là tập trung về những đối tượng và tương tác giữa chúng. Sau đây là một số đặc tính của lập trình hướng đối tượng:

•Đối tượng thường biểu diễn cho những thực thể ngoài đời. Trong chương trước, việc tạo nên lớp Deck là một bước hướng tới lập trình hướng đối tượng.
•Đa số các phương thức là phương thức đối tượng (như những phương thức mà ta kích hoạt lên các Strings) thay vì các phương thức lớp (như các phương thức Math). Những phương thức mà đến giờ ta đã viết vẫn là phương thức lớp. Ở chương này ta sẽ viết một số phương thức đối tượng.
•Các đối tượng cô lập khỏi nhau bằng cách hạn chế những cách thức tương tác giữa chúng, đặc biệt bằng cách ngăn không cho chúng truy cập các biến thực thể mà không kích hoạt các phương thức.
•Các lớp được tổ chức trong cây gia đình, ở đó những lớp mới thì mở rộng từ lớp cũ, qua việc bổ sung những phương thức mới và thay thế phương thức sẵn có.
Ở chương này tôi chuyển chương trình Card ở chương trước, từ phong cách thủ tục sang hướng đối tượng. Bạn có thể tải về mã lệnh từ chương này tại http://thinkapjava.com/code/Card3.java.

15.2 Các phương thức đối tượng và phương thức lớp

Có hai kiểu phương thức trong Java, gọi là **phương thức lớp** và **phương thức đối tượng**. Phương thức lớp dễ nhận thấy bởi có từ khóastatic ngay trên dòng đầu. Còn bất kì phương thức nào *không* có từ khóa static này thì đều là phương thức đối tượng.
Dù chưa viết được phương thức đối tượng nào, song ta đã kích hoạt một số phương thức như vậy. Mỗi khi bạn kích hoạt một phương thức "lên" một đối tượng, thì đó chính là phương thức đối tượng. Chẳng hạn, charAt và những phương thức khác mà ta kích hoạt lên những đối tượng String đều là các phương thức đối tượng.
Bất cứ thứ gì viết được là phương thức lớp cũng có thể được viết thành phương thức đối tượng, và ngược lại. Song đôi khi, sẽ có một cách viết tự nhiên hơn cách kia.

Chẳng hạn, sau đây là printCard viết dưới dạng phương thức lớp:

```
public static void printCard(Card c) {
  System.out.println(ranks[c.rank] + " of " + suits[c.suit]);
}
```

Còn sau đây, nó được viết lại thành phương thức đối tượng:

```
public void print() {
  System.out.println(ranks[rank] + " of " + suits[suit]);
}
```

Sau đây là những thay đổi:

1.Tôi đã xóa từ static.
2.Tôi thay đổi tên của phương thức để giống với giọng đọc Java hơn.
3.Tôi bỏ tham số đi.
4.Bên trong một phương thức đối tượng, bạn có thể tham chiếu đến các biến thực thể như thể chúng là những biến địa phương, vì vậy tôi đổi c.rank thành rank, và tương tự với suit.
Đây là cách kích hoạt phương thức mới này:

```
    Card card = new Card(1, 1);
    card.print();
```

Khi bạn kích hoạt một phương thức lên một đối tượng thì đối tượng đó trở thành **đối tượng hiện hành**, còn được gọi là this. Bên trong print, từ khóa this tham chiếu đến lá bài mà phương thức được kích hoạt lên đó.

15.3 Phương thức toString

Từng kiểu đối tượng đều có một phương thức mang tên toString để trả lại một chuỗi biểu diễn cho đối tượng đó. Khi bạn in ra đối tượng bằng lệnh print hoặc println, Java sẽ kích hoạt phương thức toString của đối tượng này.
Phiên bản mặc định của toString trả lại một chuỗi có chứa kiểu của đối tượng và một số nhận diện duy nhất (xem Mục 11.6). Khi bạn định nghĩa một kiểu đối tượng mới, bạn có thể **sửa đè** lên hành vi mặc định này bằng cách cung cấp một phương thức mới chứa hành vi mà bạn muốn.
Chẳng hạn, sau đây là một phương thức toString đối với Card:

```
  public String toString() {
    return ranks[rank] + " of " + suits[suit];
  }
```

Kiểu trả lại là String, theo lẽ tự nhiên; và phương thức này không nhận tham số nào. Bạn có thể kích hoạt toString theo lối thông thường:

```
    Card card = new Card(1, 1);
    String s = card.toString();
```

hoặc bạn cũng có thể kích hoạt gián tiếp nó thông qua println:

```
    System.out.println(card);
```

15.4 Phương thức equals

Ở Mục 13.4 ta đã nói về hai hình thức cân bằng: sự giống hệ, nghĩa là hai biến cùng tham chiếu tới một đối tượng, và sự tương đương, tức là hai biến có cùng giá trị.

Toán tử == kiểm tra sự giống hệt, nhưng không có toán tử nào để kiểm tra sự tương đồng, bởi "tương đồng" thế nào thì còn phụ thuộc vào kiểu của đối tượng nữa. Thay vì vậy, các đối tượng lại cung cấp một phương thức có tên equals để định nghĩa sự tương đồng này.
Các lớp trong Java cung cấp những phương thức equals để làm điều đúng đắn. Nhưng với những kiểu do người dùng định nghĩa thì cách ứng xử mặc định của phương thức này cũng chẳng khác gì sự giống hệt; đây không phải là điều bạn mong muốn.
Đối với Card ta đã có một phương thức để kiểm tra sự tương đồng:

```
public static boolean sameCard(Card c1, Card c2) {
  return (c1.suit == c2.suit && c1.rank == c2.rank);
}
```

Bởi vậy tất cả những điều ta cần làm là viết lại nó dưới dạng một phương thức cho đối tượng:

```
public boolean equals(Card c2) {
  return (suit == c2.suit && rank == c2.rank);
}
```

Một lần nữa, tôi đã bỏ đi từ khóa static cũng như thông số đầu, c1. Sau đây là cách kích hoạt phương thức mới này:

```
Card card = new Card(1, 1);
Card card2 = new Card(1, 1);
System.out.println(card.equals(card2));
```

Bên trong equals, card là đối tượng hiện hành còn card2 là tham số, c2. Đối với những phương thức hoạt động trên hai đối tượng có cùng kiểu, đôi khi tôi dùng hẳn từ khóa this đồng thời gọi tham số kia là that:

```
public boolean equals(Card that) {
  return (this.suit == that.suit && this.rank == that.rank);
}
```

Tôi nghĩ rằng theo cách này, mã lệnh sẽ dễ đọc hơn.

15.5 Những điều kì quặc và lỗi sai

Nếu bạn có các phương thức đối tượng và lớp đối tượng ở bên trong cùng một lớp, thì thật dễ nhầm lẫn. Một cách thông thường để tổ chức lời định nghĩa lớp là đặt tất cả những constructor ở đầu, theo sau là tất cả những phương thức đối tượng và tiếp theo là phương thức lớp.

Bạn có thể có một phương thức đối tượng trùng tên với phương thức lớp, miễn là chúng không có cùng số lượng cũng như kiểu các tham số. Giống các hình thức quá tải (overloading) khác, Java quyết định xem cần kích hoạt dạng nào bằng cách nhìn vào những tham số mà bạn cung cấp.

Bây giờ khi đã biết ý nghĩa của từ khóa static, có lẽ bạn đã hình dung ra được rằng main là một phương thức lớp, nghĩa là không có một "đối tượng hiện thời" nơi nó được kích hoạt. Vì không có đối tượng hiện thời trong một phương thức lớp, nên sẽ có lỗi khi dùng từ khóa this. Nếu bạn thử thì sẽ nhận được một thông báo lỗi kiểu như "Undefined variable: this."
Đồng thời, bạn cũng không thể tham chiếu đến những biến thực thể mà không dùng kí pháp dấu chấm lẫn cung cấp một tên đối tượng. Nếu thử làm, bạn sẽ nhận một thông báo lỗi như "non-static variable... cannot be referenced from a static context." Nói "non-static variable" nghĩa là biến thực thể ("instance variable.")

15.6 Thừa kế

Đặc điểm ngôn ngữ thường gắn với lập trình hướng đối tượng nhất là tính **thừa kế**. Thừa kế là khả năng định nghĩa được một lớp mới là phiên bản sửa đổi từ một lớp sẵn có. Mở rộng hình ảnh ví von

này, lớp sẵn có đôi khi còn được gọi là lớp **cha mẹ** và lớp mới được gọi là lớp **con**.

Ưu điểm cơ bản của đặc điểm này là bạn có thể bổ sung được những phương thức và biến thực thể mà không cần sửa đổi lớp cha mẹ. Điều này đặc biệt hữu ích đối với các lớp Java, vì bạn có muốn cũng chẳng thể sửa đổi được chúng.

Nếu bạn đã làm các bài tập GridWorld rồi (ở các Chương 5 và 10), bạn sẽ thấy một số ví dụ về thừa kế:

```
public class BoxBug extends Bug {
  private int steps;
  private int sideLength;
  public BoxBug(int length) {
    steps = 0;
    sideLength = length;
  }
}
```

BoxBug extends Bug nghĩa là BoxBug là một loại Bug mới đựa kế thừa những phương thức và biến thực thể của Bug. Ngoài ra:

•Lớp con có thể có thêm các biến thực thể khác. Trong ví dụ này, các BoxBug có steps và sideLength.

•Lớp con có thể có thêm các phương thức khác. Trong ví dụ này, các BoxBug có thêm một constructor nhận vào tham số nguyên.

•Lớp con có thể **ghi đè** lên một phương thức thừa hưởng từ lớp cha mẹ. Trong ví dụ này, lớp con cung cấp phương thức act (không chỉ ra ở đây), để ghi đè lên phương thức act của lớp cha mẹ.

Nếu bạn đã làm các bài tập về đồ họa ở Phụ lục A, bạn còn thấy một ví dụ nữa:

```
public class MyCanvas extends Canvas {
  public void paint(Graphics g) {
    g.fillOval(100, 100, 200, 200);
  }
}
```

MyCanvas là một kiểu mới của Canvas, chẳng có thêm phương thức hay biến thực thể nào, song nó ghi đè lên paint.

Nếu bạn chưa từng làm bài nào trong số đó thì giờ đã là lúc rồi!

15.7 Cấu trúc thừa kế lớp

Trong Java, tất cả mọi lớp đều mở rộng từ một lớp nào đó khác. Lớp cơ bản nhất được gọi là Object. Nó không chứa biến thực thể nào, nhưng có cung cấp các phương thức equals và toString, cùng những phương thức khác.

Nhiều lớp mở rộng Object, gồm cả hầu hết những lớp ta đã viết và nhiều lớp Java khác, như java.awt.Rectangle. Bất kì lớp nào không nói rõ tên lớp cha mẹ ra, thì đều mặc định là thừa hưởng từ Object.

Tuy vậy, một số chuỗi thừa kế thì dài hơn. Chẳng hạn, javax.swing.JFrame mở rộng java.awt.Frame, đến lượt nó lại mở rộng Window, đến lượt nó mở rộng Container, đến lượt nó mở rộng Component,

đến lượt nó mở rộng Object. Bất kể chuỗi này có dài như thế nào thì Object vẫn là "tổ tiên" chung của tất cả các lớp.
"Cây gia đình" của các lớp được gọi là thừa kế lớp. Object thường xuất hiện ở trên cùng, và tất cả những lớp "con" thì được xếp dưới. Chẳng hạn, nếu bạn nhìn vào tài lệu của JFrame, bạn sẽ thấy rằng phần của sự thừa kế cho ra JFrame.

15.8 Thiết kế hướng đối tượng

Thừa kế là một đặc điểm quan trọng. Có những chương trình sẽ trở nên rất phức tạp nếu không diễn đạt được một cách gọn gàng, đơn giản bằng đặc điểm nói trên. Hơn nữa, thừa kế có thể giúp tận dụng lại mã lệnh, vì bạn có thể chỉnh lại theo ý thích cách ứng xử của những lớp sẵn có mà không cần sửa đổi chúng.

Mặt khác, thừa kế có thể làm cho chương trình rất khó đọc. Khi bạn thấy một lời kích hoạt phương thức, thật khó để hình dung ra phương thức nào được kích hoạt.

Ngoài ra, nhiều thứ có thể thực hiện bằng cách thừa kế cũng có thể làm được hoặc thậm chí tốt hơn mà không dùng cách này. Cách làm thay thế thường gặp là **tổng hợp**, trong đó các đối tượng được kết hợp từ những đối tượng có sẵn, qua đó bổ sung thêm tính năng mà không cần thừa kế.

Việc thiết kế nên những đối tượng và mối liên hệ giữa chúng là chủ đề nghiên cứu của **thiết kế hướng đối tượng**, một lĩnh vực nằm ngoài phạm vi cuốn sách này. Song nếu bạn quan tâm, tôi sẽ gợi ý bạn đọc quyển *Head First Design Patterns*, của nhà xuất bản O'Reilly Media.

15.9 Thuật ngữ

phương thức đối tượng:

Một phương thức được kích hoạt lên một đối tượng, đồng thời hoạt động trên đối tượng đó. Các phương thức đối tượng thì không có chứa từ khóa static.

phương thức lớp:

Một phương thức có từ khóa static. Phương thức lớp không được kích hoạt trên đối tượng và chúng không có đối tượng hiện hành.

đối tượng hiện hành:

Đối tượng mà trên đó một phương thức đối tượng được kích hoạt. Bên trong phương thức, đối tượng hiện hành được tham chiếu đến bằng this.

ngầm:

Thứ được lướt qua không nói đến, hay được ngụ ý. Bên trong một phương thức đối tượng, bạn có thể tham chiếu đến những biến thực thể một cách ngầm (nghĩa là không nhắc đến tên đối tượng).

tường minh:

Thứ được ghi rõ ra. Bên trong một phương thức lớp, tất cả những tham chiếu đến biến thực thể phải được viết tường minh.

15.10 Bài tập

Bài tập 1 Tải về các file http://thinkapjava.com/code/CardSoln2.java và http://thinkapjava.com/code/CardSoln3.java. File CardSoln2.java chứa lời giải những bài tập của chương trước. Nó chỉ dùng các phương thức lớp (trừ các constructor). CardSoln3.java cũng chứa chương trình này, nhưng đa số các phương thức đều là

phương thức đối tượng. Tôi vẫn để nguyênmerge mà không thay đổi vì tôi nghĩ nó là phương thức lớp thì sẽ dễ đọc hơn. Hãy chuyển merge thành một phương thức đối tượng, và chuyển mergeSort một cách tương ứng. Bạn thích phiên bản merge nào hơn?

Bài tập 2 Hãy chuyển phương thức lớp sau đây thành phương thức đối tượng.

```
public static double abs(Complex c) {
  return Math.sqrt(c.real * c.real + c.imag * c.imag);
}
```

Bài tập 3 Hãy chuyển phương thức lớp sau đây thành phương thức đối lớp.

```
public boolean equals(Complex b) {
  return(real == b.real && imag == b.imag);
}
```

Bài tập 4 Bài tập này là sự tiếp nối theo Bài tập 3 của Chương 11. Mục đích là nhằm thực hành cú pháp của những phương thức đối tượng và làm quen với những thông báo lỗi có liên quan.

1. Hãy chuyển các phương thức trong lớp Rational từ phương thức lớp sang phương thức đối tượng, đồng thời thực hiện những chuyển đổi cần thiết trong main.
2. Cố ý mắc một số lỗi. Thử kích hoạt các phương thức lớp như thể chúng là phương thức đối tượng, và ngược lại. Hãy thử tìm hiểu xem thế nào là hợp lệ và thế nào không, và hiểu thông báo lỗi bạn nhận được khi mọi việc rối lên.
3. Hãy nghĩ về ưu và nhược điểm của các phương thức lớp và phương thức đối tượng? Cách nào (thường) viết gọn hơn? Cách nào diễn đạt tính toán một cách tự nhiên hơn (hoặc xét công bằng, những kiểu phép tính nào có thể được diễn đạt một cách tự nhiên nhất theo mỗi phong cách)?

Bài tập 5 Mục đích của bài tập này là viết một chương trình để phát sinh ra những phần bài poker ngẫu nhiên rồi phân loại chúng, để ta ước tính được xác suất của các dạng phần bài khác nhau. Nếu bạn không chơi poker, bạn có thể đọc về nó ở đây http://en.wikipedia.org/wiki/List_of_poker_hands.

1. Bắt đầu bằng http://thinkapjava.com/code/CardSoln3.java rồi đảm bảo chắc rằng bạn biên dịch và chạy được chương trình.
2. Hãy viết lời định nghĩa cho một lớp có tên PokerHand (phần bài), mở rộng từ Deck.
3. Viết một phương thức trong Deck có tên deal để tạo ra một PokerHand, chuyển các lá bài từ cỗ bài vào phần bài, rồi trả lại phần bài này.
4. Trong main, hãy dùng shuffle và deal để phát sinh và in ra bốn PokerHand, mỗi phần bài gồm 5 lá. Bạn có thu được kết quả tốt không?
5. Viết một phương thức PokerHand có tên hasFlush để trả lại một giá trị boolean để chỉ định xem liệu phần bài này có một flush (5 lá đồng chất) hay không.
6. Viết một phương thức có tên hasThreeKind để chỉ định xem liệu phần bài có bộ ba hay không.
7. Viết một vòng lặp để phát sinh ra vài nghìn phần bài rồi kiểm tra xem chúng có chứa 5 lá đồng chất, hay bộ ba không. Ước tính xác suất để nhận được một trong hai dạng phần bài kể trên. Hãy so sánh kết quả thu được với các xác suất ở http://en.wikipedia.org/wiki/List_of_poker_hands.
8. Viết các phương thức để kiểm tra cho những dạng phần bài khác. Có dạng dễ, có dạng khó. Đôi khi bạn sẽ thấy cần viết một vài phương thức trợ giúp phục vụ cho nhiều phép kiểm tra khác nhau.
9. Có những trò chơi poker mà người chơi lấy 7 lá bài, rồi chọn ra 5 lá bài đẹp nhất. Hãy sửa lại chương trình của bạn để phát sinh ra các phần bài 7 lá rồi tính lại những xác suất nêu trên.

Chương 16: GridWorld, phần 3

Trở về Mục lục cuốn sách

Nếu bạn chưa làm bài tập trong các Chương 5 và 10, bạn hãy nên làm đi trước khi đọc chương này. Xin được nhắc lại, bạn có thể tìm tài liệu về các lớp GridWorld ở http://www.greenteapress.com/thinkapjava/javadoc/gridworld/.
Phần 3 của cuốn Hướng dẫn sinh viên về GridWorld trình bày những lớp cấu thành GridWorld và các mối tương tác giữa chúng. Đây là một ví dụ về thiết kế hương đối tượng và làm một cơ hội để ta bàn luận những vấn đề thiết kế hướng đối tượng.

Nhưng trước khi bạn đọc cuốn Hướng dẫn sinh viên, sau đây có thêm một số điều mà bạn cần biết.

16.1 ArrayList

GridWorld sử dụng java.util.ArrayList, một đối tượng gần giống với mảng. Đó là một **tập hợp**, tức là đối tượng để chứa những đối tượng khác. Java cung cấp những tập hợp khác với nhiều tính năng khác nhau, nhưng để dùng GridWorld ta chỉ cần đến các ArrayList.
Để thấy một ví dụ, hãy tải về http://thinkapjava.com/code/BlueBug.java và http://thinkapjava.com/code/BlueBugRunner.java. BlueBug là một con bọ di chuyển ngẫu nhiên và đi tìm các tảng đá. Nếu nó thấy một tảng đá, con bọ sẽ làm tảng đá hóa màu xanh.
Sau đây là cách hoạt động của BlueBug. Khi act được kích hoạt, BlueBug lấy vị trí của nó cùng một tham chiếu đến lưới:

```
    Location loc = getLocation();

    Grid<Actor> grid = getGrid();
```

Kiểu dữ liệu nằm trong cặp ngoặc góc (<>) là một **tham số kiểu** để quy định nội dung của grid. Nói cách khác, grid không chỉ là một Grid, mà nó là Grid có chứa những Actor.
Bước tiếp theo là thu lấy những vị trí lân cận với chỗ hiện tại. Grid cung cấp một phương thức chỉ để làm việc này:

```
    ArrayList<Actor> neighbors = grid.getNeighbors(loc);
```

Kết quả trả lại từ getNeighbors là một ArrayList gồm các Actor. Phương thức size trả lại chiều dài của ArrayList, và get thì chọn lấy một phần tử. Bởi vậy ta có thể in ra những vị trí lân cận như sau.

```
    for (int i = 0; i < neighbors.size(); i++) {

      Actor actor = neighbors.get(i);

      System.out.println(actor);

    }
```

Việc duyệt một ArrayList là thao tác thông dụng đến nỗi có một cú pháp đặc biệt dành cho nó: **vòng lặp for-each**. Bởi vậy ta có thể viết:

```
    for (Actor actor : neighbors) {

      System.out.println(actor);

    }
```

Ta biết rằng các lân cận đều là những Actor, song lại không biết kiểu của chúng là gì: từng lân cận có thể là Bug, Rock, v.v. Để tìm tảng đá (Rock), ta sử dụng toán tử instanceof, vốn để kiểm tra xem liệu một đối tượng có là thực thể của một lớp hay không.

```
    for (Actor actor : neighbors) {
      if (actor instanceof Rock) {
        actor.setColor(Color.blue);
      }
    }
```

Để làm cho toàn bộ hoạt động được, ta cần phải nhập những lớp cần dùng đến:

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Bug;
import info.gridworld.actor.Rock;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
import java.awt.Color;
import java.util.ArrayList;
```

16.2 Giao diện

GridWorld cũng sử dụng các **giao diện** của Java, bởi vậy tôi muốn giải thích ý nghĩa của chúng. "Giao diện" có nhiều nghĩa trong những ngữ cảnh khác nhau, nhưng trong Java, thuật ngữ này dùng để chỉ một đặc điểm cụ thể của ngôn ngữ: giao diện là một lời định nghĩa lớp mà trong đó các phương thức không có phần thân.

Trong lời định nghĩa lớp thông thường, mỗi phương thức có một nguyên mẫu và một phần thân (xem Mục 8.5). Nguyễn mẫu còn được gọi là**phần quy định** bởi nó quy định tên, các thông số, và kiểu trả lại của phương thức đó. Phần thân được gọi là **phần thực hiện** bởi nó thực hiện phần quy định trên. Trong một giao diện Java, các phương thức không có phần thân, bởi vậy giao diện chỉ quy định các phương thức mà không thực hiện chúng.

Chẳng hạn, java.awt.Shape là một giao diện với các nguyên mẫu cho contains, intersects, cùng một số phương thức khác. java.awt.Rectangle cung cấp phần thực hiện của những phương thức này, bởi vậy ta nói rằng "Rectangle thực hiện Shape." Thực ra, dòng đầu tiên của lời định nghĩa lớp Rectangle là:

```
public class Rectangle extends Rectangle2D implements Shape, Serializable
```

Rectangle thừa kế các phương thức từ Rectangle2D và cung cấp phần thực hiện cho các phương thức trong Shape và Serializable.

Trong GridWorld, lớp Location thực hiện giao diện java.lang.Comparable bằng cách cung cấp compareTo, vốn tương tự với compareCards ở Mục 13.5. GridWorld cũng định nghĩa một giao diện mới, Grid, để quy định các phương thức mà một Grid cần phải cung cấp. Đồng thời, GridWorld cũng bao gồm hai phần thực hiện, BoundedGrid và UnboundedGrid.

Quyển hướng dẫn có dùng chữ viết tắt **API**, mà chữ đầy đủ là "application programming interface" (giao diện lập trình ứng dụng). API là một tập hợp các phương thwusc dành sẵn cho bạn, người lập

trình ứng dụng, để sử dụng. Hãy
xemhttp://en.wikipedia.org/wiki/Application_programming_interface.

16.3 public và private

Hãy nhớ lại từ Chương 1, tôi đã nói rằng tôi sẽ giải thích tại sao phương thức main lại có từ khóa public chứ? Rốt cuộc, đã đến lúc cần giải thích rồi.
public nghĩa là phương thức được xét có thể được kích hoạt từ những phương thức khác. Lựa chọn còn lại là private, có nghĩa là phương thức đang xét chỉ có thể kích hoạt được trong lớp mà nó được định nghĩa.
Các biến thực thể cũng có thể là public hoặc private, với kết quả tương tự: một biến thực thể private chỉ có thể truy cập được từ bên trong lớp mà nó được định nghĩa.
Lý do cơ bản cho việc đặt những phương thức và biến thực thể dưới dạng private là nhằm hạn chế sự tương tác giữa các lớp để có thể giữ mức độ phức tạp ở mức chấp nhận được.

Chẳng hạn, lớp Location giữ các biến thực thể dưới dạng private. Nó có các phương thức truy cập getRow là getCol, nhưng lại không cung cấp phương thức nào để sửa đổi các biến thực thể. Hệ quả là, các đối tượng Location đều không thể biến đổi, theo nghĩa rằng chúng đều có thể được chia sẻ mà ta không lo chúng bộc lộ động thái không mong đợi do xuất hiện bí danh (alias).
Việc đặt các phương thức dưới dạng private giúp ta giữ cho API được đơn giản. Các lớp thường kèm theo những phương thức trợ giúp vốn được dùng để thực hiện các phương thức khác, song nếu để cho những phương thức này tham gia vào trong API public có thể sẽ không cần thiết và dễ gây lỗi.

Các phương thức và biến thực thể private là đặc điểm ngôn ngữ giúp cho lập trình viên đảm bảo được sự **bao bọc dữ liệu**, theo nghĩa là các đối tượng thuộc lớp này thì được cô lập khỏi những lớp khác.

16.4 Trò chơi Life

Nhà toán học John Conway đã phát minh ra "Trò chơi Life," mà ông gọi là một "trò chơi không người" vì chẳng cần có người chơi để lựa chọn chiến thuật hay ra quyết định. Sau khi thiết lập điều kiện ban đầu, bạn chỉ việc xem trò chơi tự nó phát triển. Nhưng điều này hóa ra còn hay hơn so với thoạt nghe; bạn có thể đọc thêm ở http://en.wikipedia.org/wiki/Conways_Game_of_Life.
Mục đích của bài tập này là thực hiện trò chơi Life trong GridWorld. "Bàn cờ" chính là là lưới ô, và những "quân cờ" chính là đối tượng Rock (viên đá).

Trò chơi được tiến hành theo từng lượt, hay từng **bước thời gian**. Ở lúc bắt đầu một bước thời gian, từng viên đá có trạng thái "sống" hoặc "chết". Trên màn hình, màu sắc của viên đá này thể hiện trạng thái của nó. Trạng thái của từng viên đá lại phụ thuộc vào trạng thái của những viên **lân cận** với nó. Mỗi viên đá có 8 viên lân cận, trừ những viên nằm dọc theo cạnh của lưới ô. Sau đây là luật chơi:

- Nếu một viên đá chết có đúng 3 viên lân cận, thì nó sẽ sống lại! Nếu không, thì nó vẫn chết.
- Nếu một viên đá sống có 2 hoặc 3 viên lân cận, thì nó vẫn sống. Còn không thì nó chết đi.

Từ quy tắc này sẽ có một vài hệ quả: Nếu tất cả viên đá đều chết rồi, thì chẳng có viên nào sống lại. Nếu lúc đầu bạn có mỗi một viên đá sống, thì nó sẽ chết đi. Nhưng nếu có 4 viên cạnh nhau xếp thành hình vuông thì chúng sẽ giữ cho nhau còn sống, bởi vậy đây là một cấu trúc bền vững.

Đa số các cấu hình đơn giản lúc đầu sẽ nhanh chóng chế đi, hoặc đạt đến một cấu hình ổn định. Song

cũng có một ít điều kiện ban đầu cho thấy độ phức tạp đáng kể. Một trong những điều kiện đầu như vậy là r-pentomino: bắt đầu chỉ với 5 viên đá, cấu hình này chạy suốt 1103 bước thời gian rồi kết thúc ở một cấu hình bền vững với 116 viên đá sống (xem http://www.conwaylife.com/wiki/R-pentomino).
Các mục tiếp sau đây là những gợi ý để thực hiện trò chơi Life trong GridWorld. Bạn có thể tải về lời giải của tôi
tại http://thinkapjava.com/code/LifeRunner.java và http://thinkapjava.com/code/LifeRock.java.

16.5 LifeRunner

Hãy sao lại một bản file BugRunner.java rồi đặt tên thành LifeRunner.java, sau đó bổ sung những phương thức với nguyên mẫu như sau:

```
/**
 * Lập nên một lưới ô cho trò chơi Life, cùng cấu hình r-pentomino. */
public static void makeLifeWorld(int rows, int cols)

/**
 * Xếp các viên đá LifeRock lên lưới ô. */
public static void makeRocks(ActorWorld world)
```

Phương thức makeLifeWorld cần phải tạo nên một Grid chứa các Actor cùng một ActorWorld, sau đó kích hoạt makeRocks, đến lượt phương thức này sẽ phải đặt một LifeRock vào mỗi ô trong Grid.

16.6 LifeRock

Hãy sao lại một bản của file BoxBug.java rồi đặt tên là LifeRock.java. Lớp LifeRock phải mở rộng từ Rock. Hãy bổ sung thêm một phương thức act chẳng để làm gì cả. Bây giờ mã lệnh phải chạy thông được và bạn sẽ thấy một Grid chứa đầy Rock.
Để theo dõi trạng thái của những viên đá này, bạn có thể bổ sung một biến thực thể mới, hoặc bạn có thể dùng màu sắc (Color) của Rock để biểu thị trạng thái. Bằng cách nào đi nữa, hay viết những phương thức có các nguyên mẫu sau đây:

```
/**
 * Trả về true nếu viên đá còn sống. */
public boolean isAlive()

/**
 * Làm cho viên đá sống lại. */
public void setAlive()

/**
 * Làm viên đá chết đi. */
public void setDead()
```

Hãy viết một constructor để kích hoạt setDead rồi khẳng định chắc rằng tất cả viên đá đều chết.

16.7 Cập nhật đồng thời

Trong trò chơi Life, tất cả viên đá đều được cập nhật một cách đồng thời; nghĩa là từng viên đá đều

kiểm tra trạng thái của các viên lân cận trước khi những viên lân cận này thay đổi trạng thái. Nếu không, động thái của hệ thống sẽ còn phụ thuộc vào thứ tự của phép cập nhật nữa.

Để thực hiện cập nhật đồng thời, tôi gợi ý rằng bạn nên viết một phương thức act gồm có hai khâu. Ở khâu thứ nhất, tất cả viên đá phải đếm số lân cận với nó rồi ghi lại kết quả. Và ở khâu thứ hai, tất cả viên đá cập nhật trạng thái của chúng.
Phương thức act tôi đã viết trông như sau:

```
/**
 * Kiểm tra xem ta đang ở khâu nào và gọi phương thức tương ứng.
 * Chuyển đến khâu tiếp theo. */
public void act() {
  if (phase == 1) {
    numNeighbors = countLiveNeighbors();
    phase = 2;
  } else {
    updateStatus();
    phase = 1;
  }
}
```

phase và numNeighbors là các biến thực thể. Và sau đây là những nguyên mẫu
cho countLiveNeighbors và updateStatus:

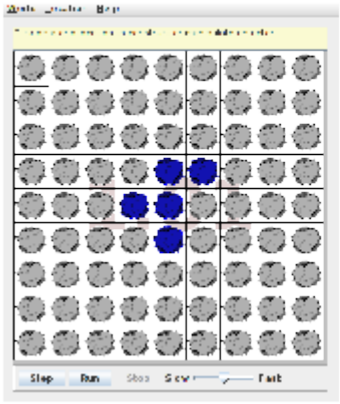
```
/**
 * Đếm số viên đá lân cận còn sống. */
public int countLiveNeighbors()

/**
 * Cập nhật trạng thái viên đá (sống hoặc chết) dựa trên
 * số các viên lân cận với nó. */
public void updateStatus()
```

Hãy bắt đầu với một phiên bản đơn giản của updateStatus chỉ để chuyển viên đá sống thành chết và ngược lại. Bây giờ hãy chạy chương trình rồi khẳng định rằng viên đá đã đổi màu. Cứ hai bước trong World (môi trường) thì tương ứng với một bước thời gian trong trò chơi Life.
Bây giờ hãy điền nội dung vào phần thân của các phương
thức countLiveNeighbors và updateStatus theo luật chơi và xem hệ thống có động thái giống như ta dự liệu hay không.

16.8 Điều kiện đầu

Để thay đổi điều kiện đầu, bạn có thể dùng các menu bật của GridWorld để đặt trạng thái của các viên đá bằng cách kích hoạt setAlive. Hoặc bạn cũng có thể viết các phương thức để tự động hóa quy trình. Trong LifeRunner, hãy bổ sung một phương thức có tên makeRow để tạo nên cấu hình ban đầu
với n viên đá sống liền nhau cùng một hàng ở giữa lưới ô. Điều gì sẽ xảy ra với các giá trị khác nhau

của n?
Hãy bổ sung một phương thức có tên makePentomino để tạo nên một r-pentomino ở chính giữa lưới ô.
Cấu hình ban đầu phải có dạng như sau:



Nếu bạn chạy cấu hình này trong nhiều bước, nó sẽ lan tỏa đến cuối lưới ô. Bốn đường biên của lưới ô đã làm thay đổi động thái của hệ thống. Để thấy được sự phát triển toàn vẹn của r-pentomino, thì lưới ô phải đủ lớn. Bạn có thể phải thử nghiệm để tìm ra kích cỡ lưới ô thích hợp; và tùy thuộc vào tốc độ máy tính đang dùng, công việc này có thể mất thời gian.

Trang web về trò chơi có mô tả những điều kiện đầu khác mà cho ta kết quả thú vị (http://www.conwaylife.com/). Hãy chọn lấy điều kiện đầu mà bạn ưa thích rồi thực hiện nó.
Còn có những biến thể của trò chơi Life với luật chơi khác nhau. Hãy thử chơi một biến thể trong số đó để xem có gì hay.

16.9 Bài tập

Bài tập 1 Khởi đầu bằng một bản sao của BlueBug.java, bạn hãy viết một lời định nghĩa lớp cho một kiểu đối tượng Bug mới có khả năng tìm kiếm và ăn những đóa hoa. Ở đây, "ăn" bông hoa có thể được thực hiện bằng kích hoạt removeSelfFromGrid lên nó.
Bài tập 2 Bây giờ bạn biết hết những gì cần biết để đọc Phần 3 của Cuốn hướng dẫn sinh viên về GridWorld rồi làm các bài tập.
Bài tập 3 Nếu bạn đã thiết lập trò chơi Life, bạn đã hoàn toàn sẵn sàng làm Phần 4 của cuốn Hướng dẫn sinh viên về GridWorld. Hãy đọc nó rồi làm các bài tập.
Chúc mừng, bạn đã học xong!

Phụ lục A: Đồ họa

Trở về Mục lục cuốn sách

A.1 Đồ họa Java 2 chiều

Phụ lục này đưa ra các ví dụ và bài tập minh họa cho tính năng đồ họa trong Java. Có một số cách tạo nên đồ họa trong Java; cách đơn giản nhất là dùng java.awt.Graphics. Sau đây là một ví dụ hoàn chỉnh:

```
import java.awt.Canvas;
import java.awt.Graphics;
import javax.swing.JFrame;

public class MyCanvas extends Canvas {

  public static void main(String[] args) {
    // tạo một khung (frame)
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // thêm một nền vẽ (canvas)
    Canvas canvas = new MyCanvas();
    canvas.setSize(400, 400);
    frame.getContentPane().add(canvas);
    // hiển thị khung
    frame.pack();
    frame.setVisible(true);
  }

  public void paint(Graphics g) {
    // vẽ hình tròn
    g.fillOval(100, 100, 200, 200);
  }
}
```

Bạn có thể tải đoạn mã lệnh này về từ http://thinkapjava.com/code/MyCanvas.java.

Những dòng lệnh đầu có nhiệm vụ nhập các lớp mà ta cần từ java.awt và javax.swing.

MyCanvas mở rộng Canvas, nghĩa là một đối tượng MyCanvas là một kiểu Canvas mà cung cấp các phương thức để vẽ những đối tượng đồ họa.

Trong main, ta đã

1.Tạo nên một JFrame, vốn là một cửa sổ có thể chứa nền vẽ (canvas), nút bấm (buttons), trình đơn (menu), cùng các thành phần cửa sổ khác;

2.Tạo nên MyCanvas, ấn định bề rộng và chiều cao của nó, rồi thêm nó lên khung, sau đó

3.Hiển thị khung này lên màn hình.
paint là một phương thức đặc biêt được kích hoạt khi MyCanvas cần được vẽ. Nếu bạn chạy mã lệnh này, bạn sẽ thấy một hình tròn đen trên nền xám.

A.2 Các phương thức Graphics

Để vẽ lên nền Canvas, bạn kích hoạt các phương thức thuộc đối tượng Graphics. Ví dụ trước đây sử dụng fillOval. Các phương thức khác gồm có drawLine, drawRect v.v. Bạn có thể đọc tài liệu của những phương thức này ở http://download.oracle.com/javase/6/docs/api/java/awt/Graphics.html.
Sau đây là nguyên mẫu của fillOval:

```
  public void fillOval(int x, int y, int width, int height)
```

Các tham số quy định một **hình bao**, vốn là hình chữ nhật bao lấy hình trái xoan được vẽ (xem phía dưới). Bản thân hình bao thì không được vẽ lên.

x và y quy định vị trí góc trái bên trên của hình bao trong **hệ tọa độ** đồ họa.

A.3 Hệ tọa độ

Có thể bạn đã quen thuộc với tọa độ Đề-các trong không gian hai chiều; trong đó mỗi vị trí được xác định bằng một tọa độ x (khoảng cách dọc trục x) và một tọa độ y. Theo quy ước, các tọa độ Đề-các tăng dần qua bên phải và lên phía trên, như ở hình vẽ sau.

Theo quy ước, hệ thống đồ họa máy tính sử dụng một hệ tọa độ trong đó gốc ở góc trái trên, và hướng dương của trục y chỉ *xuống*. Java tuân theo quy ước này.
Các tọa độ đều được đo bằng điểm ảnh; mỗi điểm ảnh tương ứng với một chấm trên màn hình. Một màn hình thường có bề rộng khoảng 1000 điểm ảnh. Các tọa độ đều luôn là số nguyên. Nếu muốn dùng một giá trị phẩy động để làm tọa độ, bạn phải làm tròn giá trị này (xem Mục 3.2).

A.4 Màu sắc

Để chọn màu của một hình, bạn hãy kích hoạt setColor lên đối tượng đồ họa:

```
    g.setColor(Color.red);
```

setColor thay đổi màu hiện hành; mọi thứ được vẽ đều bằng màu hiện hành.
Color.red là một giá trị cho bởi lớp Color; để dùng màu này bạn phải nhập java.awt.Color. Các màu khác gồm có:

```
black     blue    cyan   darkGray   gray   lightGray

magenta   orange  pink   red        white  yellow
```

Bạn có thể tạo nên những màu khác bằng cách chỉ định các thành phần đỏ, lục, lam (RGB).
Xemhttp://download.oracle.com/javase/6/docs/api/java/awt/Color.html.
Bạn có thể điều khiển màu nền của Canvas bằng cách kích hoạt Canvas.setBackground.

A.5 Chuột Mickey

Giả dụ ta muốn vẽ một chú chuột Mickey. Ta có thể dùng hình oval như vừa vẽ làm khuôn mặt, sau đó

thêm vào đôi tai. Để làm cho mã lệnh dễ đọc hơn, hãy dùng Rectangle (hình chữ nhật) để biểu diễn các hình bao.

Sau đây là một phương thức nhận vào một Rectangle rồi kích hoạt fillOval.

```
public void boxOval(Graphics g, Rectangle bb) {
  g.fillOval(bb.x, bb.y, bb.width, bb.height);
}
```

Và sau đây là một phương thức để vẽ Mickey:

```
public void mickey(Graphics g, Rectangle bb) {
  boxOval(g, bb);
  int dx = bb.width/2;
  int dy = bb.height/2;
  Rectangle half = new Rectangle(bb.x, bb.y, dx, dy);
  half.translate(-dx/2, -dy/2);
  boxOval(g, half);
  half.translate(dx*2, 0);
  boxOval(g, half);
}
```

Dòng thứ nhất vẽ khuôn mặt. Ba dòng tiếp theo tạo nên một hình chữ nhật nhỏ hơn làm đôi tai. Ta dịch chuyển hình chữ nhật này lên trên và bên trái để tạo thành tai thứ nhất, sau đó dịch sang phải làm tai thứ hai.

Kết quả trông sẽ như sau:

Bạn có thể tải mã lệnh về từ http://thinkapjava.com/code/Mickey.java.

A.6 Thuật ngữ

tọa độ:

Một biến hay giá trị quy định vị trí trong một cửa sổ đồ họa hai chiều.

điểm ảnh:

Đơn vị đo tọa độ.

hình bao:

Một cách thông thường quy định tọa độ của một vùng chữ nhật.

A.7 Bài tập

Bài tập 1 Vẽ lá cờ Nhật Bản, một hình tròn đỏ trên nền trắng có bề rộng hơn so với chiều cao.

Bài tập 2 Sửa lại Mickey.java để vẽ những đôi tai trên cả đôi tai, rồi tai mới trên tai này, và cứ như vậy đến khi tai nhỏ nhất có bề rộng chỉ 3 điểm ảnh. Kết quả dường như giống Hươu Mickey:

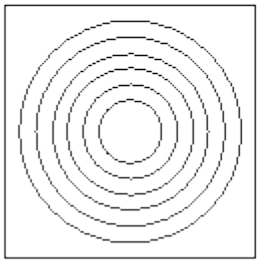
Gợi ý: bạn chỉ được bổ sung hay sửa đổi một vài dòng lệnh.

Bạn có thể tải về một lời giải từ http://thinkapjava.com/code/MickeySoln.java.

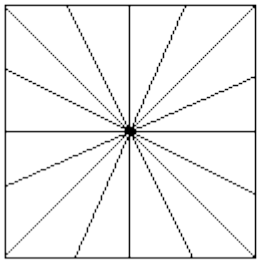
Bài tập 3

1. Tải về http://thinkapjava.com/code/Moire.java rồi nhập nó vào môi trường phát triển hiện hành.
2. Đọc phương thức paint và phác thảo công dụng mà bạn phán đoán. Bây giờ chạy phương thức này. Bạn có thấy kết quả như dự đoán không? Một lời giải thích cho điều này có thể xem ở http://en.wikipedia.org/wiki/Moire_pattern.
3. Sửa lại chương trình để khoảng cách giữa các đường tròn rộng ra hoặc hẹp lại. Xem có gì trong hình ảnh.
4. Sửa lại chương trình để các đường tròn đồng tâm được vẽ từ tâm màn hình, như ở hình dưới, bên trái. Khoảng cách giữa các đường tròn cần phải đủ nhỏ để thấy được sự giao hòa Moiré .

concentric circles



radial Moire pattern



5. Hãy viết một phương thức có tên radial để vẽ một loạt các đường thẳng đồng quy như ở hình (phải), nhưng phải đủ sát nhau để tạo nên một dạng mẫu Moiré.
6. Gần như mọi loại dạng mẫu đồ họa cũng có thể tạo nên dạng mẫu giao hòa kiểu Moiré. Hãy nghịch chơi và quan sát sản phẩm bạn tạo nên.

Phụ lục B: Đầu vào và đầu ra trong Java

Trở về Mục lục cuốn sách

B.1 Đối tượng System

Lớp System cung cấp các phương thức và đối tượng thu nhận đầu vào từ bàn phím, in dòng chữ lên màn hình, và thực hiện vào ra (input/output, I/O) đối với file. System.out là đối tượng để hiển thị lên màn hình. Khi bạn kích hoạt print và println, bạn đã kích hoạt chúng từ System.out.

Thậm chí bạn có thể dùng chính System.out để in ra System.out:

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

Kết quả là:

```
java.io.PrintStream@80cc0e5
```

Khi Java in ra một đối tượng, nó in ra kiểu của đối tượng này (PrintStream) cùng với gói mà kiểu đó được định nghĩa (java.io), và một số nhận diện duy nhất cho đối tượng này. Trên máy tính tôi dùng, số nhận diện nói trên là 80cc0e5, nhưng vẫn với mã lệnh này mà bạn chạy thì có thể sẽ nhận được kết quả khác.

Cũng có một đối tượng có tên System.in cho phép ta nhận đầu vào từ bàn phím. Tuy vậy không may là đối tượng trên không giúp cho việc lấy dữ liệu bàn phím dễ dàng cho lắm.

B.2 Đầu vào từ bàn phím

Trước hết, bạn phải dùng System.in để tạo nên một InputStreamReader mới.

```
    InputStreamReader in = new InputStreamReader(System.in);
```

Sau đó bạn dùng in để tạo nên một BufferedReader mới:

```
    BufferedReader keyboard = new BufferedReader(in);
```

Sau cùng, bạn có thể kích hoạt readLine lên keyboard, để lấy kết quả đầu vào từ bàn phím rồi chuyển nó thành một String.

```
    String s = keyboard.readLine();

    System.out.println(s);
```

Chỉ có một vấn đề. Có thể xuất hiện trục trặc khi bạn kích hoạt readLine, và chúng có thể phát biệt lệ IOException. Một phương thức phát ra biệt lệ phải bao gồm biệt lệ này trong phần nguyên mẫu của phương thức đó, như sau:

```
  public static void main(String[] args) throws IOException {

    // phần thân của main

  }
```

B.3 Đầu vào từ file

Sau đây là một chương trình đọc vào các dòng trong một file rồi in những dòng đó ra:

```
import java.io.*;

public class Words {

  public static void main(String[] args) throws FileNotFoundException, IOException

{
```

```
    processFile("words.txt");
  }

  public static void processFile(String filename) throws FileNotFoundException,
IOException {
    FileReader fileReader = new FileReader(filename);
    BufferedReader in = new BufferedReader(fileReader);
    while (true) {
      String s = in.readLine();
      if (s == null)
        break;
      System.out.println(s);
    }
  }
}
```

Dòng đầu tiên làm nhiệm vụ nhập java.io, gói chương trình có chứa FileReader, BufferedReader, và phần còn lại trong thư mục thừa kế lớp để thực hiện những công việc giản đơn thông thường. Dấu * có nghĩa là nó sẽ nhập vào toàn bộ các lớp trong gói chương trình này.
Sau đây cũng là chương trình đó được viết lại bằng ngôn ngữ Python:

```
for word in open('words.txt'):
    print word
```

Tôi không đùa. Từng đó đã đủ một chương trình, với tính năng tương tự.

B.4 Bắt biệt lệ

Ở ví dụ trước, processFile có thể phát những biệt lệ FileNotFoundException và IOException. Và vì main gọi đến processFile, nó phải khai báo cùng những biệt lệ đó. Trong một chương trình lớn hơn, main có thể khai báo từng biệt lệ có mặt.
Một cách làm khác là **bắt** biệt lệ này bằng câu lệnh try. Sau đây là một ví dụ:

```
  public static void main(String[] args) {
    try {
      processFile("words.txt");
    }
    catch (Exception ex) {
      System.out.println("Cách này không có tác dụng. Sau đây là lý do:");
      ex.printStackTrace();
    }
  }
```

Cấu trúc này tương tự như một lệnh if. Nếu “nhánh” thứ nhất chạy mà không gây ra một biệt lệ nào, thì

nhánh thứ hai sẽ được bỏ qua.
Nếu như nhánh thứ nhất gây nên một biệt lệ, thì luồng thực thi sẽ nhảy đến nhánh thứ hai, vốn để xử lý điều kiện biệt lệ (bằng cách nói "sai rồi" theo một cách lịch thiệp). Trong trường hợp này nó in ra một thông báo lỗi cùng với kết quả lần dấu ngăn xếp.

Bạn có thể tải đoạn mã lệnh này về từ http://thinkapjava.com/code/Words.java và danh sách từ vựng ở http://thinkapjava.com/code/words.txt. Hãy đảm bảo chắc rằng hai file này đặt trong cùng thư mục. (Nếu bạn dùng môi trường phát triển tích hợp như NetBeans hoặc Eclipse, hãy đảm bảo rằng file words.txt nằm trong thư mục dự án hiện thời.)
Bây giờ hãy đi làm các Bài tập 9, 10, 11 của Chương 10 – Chuỗi kí tự.

Phụ lục C: Phát triển chương trình

Trở về Mục lục cuốn sách

C.1 Các chiến lược

Trong cuốn sách tôi đã trình bày những chiến lược khác nhau để phát triển chương trình, bởi vậy giờ đây tôi muốn tập hợp chúng lại. Nền tảng của tất cả chiến lược này cùng là **phát triển tăng dần**, vốn như sau:

1.Lấy điểm khởi đầu là chương trình chạy được, chỉ thực hiện một động tác dễ thấy, chẳng hạn in dữ liệu nào đó.
2.Mỗi lúc chỉ bổ sung thêm một ít dòng lệnh, và cứ thay đổi một lần lại phải kiểm tra chương trình.
3.Lặp lại công đoạn đến khi chương trình thực hiện được việc dự kiến.
Sau mỗi thay đổi, chương trình phải cho kết quả nhìn thấy được để kiểm tra đoạn mã lệnh mới bổ sung. Cách tiếp cận lập trình như thế này có thể tiết kiệm cho ta nhiều thời gian.

Bởi mỗi lúc bạn chỉ thêm có một vài dòng lệnh, nên rất dễ tìm ra các lỗi cú pháp. Và vì mỗi phiên bản chương trình lại tạo nên kết quả nhìn thấy được, nên bạn liên tục kiểm tra mô hình nhận thức của mình về cách chương trình hoạt động. Nếu mô hình nhận thức bị sai, bạn sẽ đối mặt với mâu thuẫn (và có cơ hội sửa chữa mô hình đó) trước khi viết ra nhiều dòng lệnh sai.

Thử thách của việc phát triển tăng dần là không dễ hình dung ra con đường dẫn từ khởi điểm đến chỗ chương trình hoàn thiện và đúng đắn. Để giúp cho điều này, có một vài chiến lược ta có thể chọn lựa:

Đóng gói và khái quát hoá:

> *Nếu bạn chưa biết cách chia bài toán thành các phương thức, thì hãy viết mã lệnh trong main, rồi tìm những bó lệnh rõ rệt để gói chúng vào một phương thức, rồi khái quát hoá một cách phù hợp.*

Lập nguyên mẫu nhanh:
Nếu bạn đã biết cần viết phương thức gì, song chưa biết cách viết nó thế nào, thì hãy bắt tay viết một bản nháp để xử lý trường hợp đơn giản nhất, sau đó thử nó với những trường hợp khác, vừa viết vừa mở rộng và sửa lỗi.
Hướng từ dưới lên:
Bắt đầu bằng việc viết những phương thức đơn giản, rồi ghép chúng lại thành lời giải.
Hướng từ trên xuống:
Dùng giả mã để thiết kế cấu trúc của bài toán rồi nhận diện những phương thức mà bạn cần. Sau đó viết các phương thức rồi thay thế giả mã với mã thật.
Trong quá trình phát triển, bạn có thể cần phải dựng "dàn giáo". Chẳng hạn, mỗi lớp cần phải có một phương thức toString để cho phép bạn in ra trạng thái của một đối tượng dưới dạng người đọc được. Phương thức này rất có ích cho việc gỡ lỗi, song thường không thuộc về chương trình hoàn thiện.

C.2 Các hình thức thất bại

Nếu bạn đang phải mất quá nhiều thời gian vào việc gỡ lỗi, có thể là do bạn đang dùng một chiến lược phát triển không hiệu quả. Sau đây là những hình thức thất bại mà tôi thường gặp nhất (và cũng đôi khi

mắc phải):

Phát triển không tăng dần:
Nếu như bạn đang viết nhiều dòng lệnh cùng lúc mà không biên dịch và chạy thử, thì bạn đang tự chuốc lấy phiền phức. Có lần tôi hỏi một sinh viên xem bài tập làm đến đâu rồi, cậu ta trả lời, "Tuyệt! Em đã viết hết chương trình rồi. Giờ chỉ việc gỡ lỗi thôi."
Bám chặt lấy mã lệnh sai:
Nếu bạn viết ra nhiều dòng lệnh một lúc mà không biên dịch và kiểm tra chương trình, thì có thể bạn còn không gỡ lỗi được nữa. Đôi khi chiến thuật duy nhất là (than ôi!) xoá đi mã lệnh sai rồi làm lại từ đầu (bằng chiến thuật tăng dần). Nhưng người mới lập trình thường có tình cảm gắn bó với mã lệnh họ viết ra, dù cho mã lệnh này không hoạt động được. Cách duy nhất để thoát khỏi cái bẫy này là phải tàn nhẫn.
Lập trình bước ngẫu nhiên:
Đôi khi tôi hướng dẫn sinh viên mà dường như họ lập trình kiểu ngẫu nhiên. Họ sửa một chỗ trong chương trình, chạy, nhận lấy thông báo lỗi, rồi lại sửa, lại chạy, v.v. Vấn đề là không hề có mối liên hệ rõ ràng nào giữa kết quả chương trình và chỗ sửa đổi đó. Nếu bạn nhận được thông báo lỗi, thì hãy dành thời gian để đọc nó. Tổng quát hơn, hãy dành thời gian suy nghĩ.
Phó mặc cho trình biên dịch:
Các thông báo lỗi đều có ích, song không phải lúc nào chúng cũng đúng. Chẳng hạn, nếu thông báo ghi là, "Semi-colon expected on line 13," (thiếu một dấu chấm phẩy ở dòng 13), thì điều này nghĩa là có lỗi cú pháp ở gần dòng 13. Song cách giải quyết không phải lúc nào cũng là điền dấu chấm phẩy vào dòng 13. Đừng phó mặc chương trình của mình cho trình biên dịch.
Chương tiếp theo sẽ trình bày thêm các gợi ý về cách gỡ lỗi hiệu quả.

Phụ lục D: Gỡ lỗi

Trở về Mục lục cuốn sách

Chiến thuật gỡ lỗi hay nhất còn tuỳ thuộc vào loại lỗi bạn mắc phải:

- Lỗi cú pháp tạo ra bởi trình biên dịch, nhằm chỉ định có trục trặc trong cú pháp của chương trình. Chẳng hạn: bỏ mất dấu chấm phẩy ở cuối câu lệnh.
- Các biệt lệ được tạo ra nếu có điều gì trục trặc khi chương trình đang chạy. Chẳng hạn: một vòng đệ quy vô hạn cuối cùng sẽ gây nên biệt lệ StackOverflowException.
- Lỗi logic khiến cho chương trình thực hiện việc làm sai. Chẳng hạn, một biểu thức có thể không được tính toán đúng theo trình tự mà bạn định liệu, cho ra kết quả không lường trước.

Các mục tiếp sau đây được xếp theo kiểu lỗi; có những kĩ thuật xử lý dùng được cho nhiều loại lỗi khác nhau.

D.1 Lỗi cú pháp

Hình thức gỡ lỗi hay nhất là ở đó bạn không phải làm gì, bởi ngay từ đầu bạn đã tránh mắc phải lỗi. Trong mục trước, tôi đã đề xuất những chiến lược phát triển để giảm thiểu lỗi và tạo điều kiện phát hiện sớm lỗi khi mắc phải chúng. Điểm mấu chốt là lấy một chương trình chạy được làm điểm khởi đầu, và mỗi lúc chỉ thêm rất ít mã lệnh. Khi có lỗi, bạn sẽ biết rõ là lỗi này nằm ở đâu.

Dù vậy, bạn có thể sẽ rơi vào một trong những tình huống sau. Với mỗi tình huống, tôi lại có đề xuất cách xử lý thích hợp.

TRÌNH BIÊN DỊCH BÀY RA LA LIỆT NHỮNG THÔNG BÁO LỖI.

Nếu trình biên dịch có báo đến 100 lỗi đi nữa, thì điều này cũng không có nghĩa là chương trình bạn có 100 lỗi. Mỗi khi trình biên dịch gặp một lỗi, nó thường đi chệch ra khỏi luồng thực thi một quãng. Nó sẽ cố gắng hồi phục và tiếp tục theo luồng thực thi sau lỗi đầu tiên, nhưng đôi khi nó thông báo những lỗi đáng ngờ.

Chỉ có thông báo lỗi đầu tiên mới đáng tin cậy. Tôi gợi ý rằng bạn sửa từng lỗi một, rồi biên dịch lại chương trình. Bạn có thể thấy một dấu chấm phẩy có thể "sửa được" 100 lỗi.

TÔI ĐANG GẶP MỘT LỖI BIÊN DỊCH THẬT KÌ QUẶC VÀ NÓ CHẲNG BIẾN ĐI.

Trước hết, hãy đọc kĩ thông báo lỗi này. Thông báo được viết bằng ngôn ngữ chuyên dụng và rất ngắn ngủi, song thường ẩn chứa một thông tin cốt lõi.

Nếu không có gì khác, thông báo sẽ cho bạn biết trục trặc xảy ra ở đâu trong chương trình. Thực ra, nó cho bạn biết trình biên dịch đang ở đâu khi phát hiện thấy lỗi, chứ không nhất thiết là nơi có lỗi. Hãy dùng thông tin thu nhận từ trình biên dịch như một chỉ dẫn, song nếu bạn không thấy lỗi theo hướng chỉ dẫn đó thì hãy mở rộng việc tìm kiếm ra.

Nói chung lỗi sẽ nằm ở trước vị trí thông báo lỗi, song cũng có những trường hợp mà lỗi nằm ở nơi khác

hẳn. Chẳng hạn, nếu bạn nhận được thông báo lỗi ở một lời kích hoạt phương thức, thì có khi lỗi thực sự lại nằm ở lời định nghĩa phương thức.

Nếu bạn chưa nhanh chóng tìm ra được lỗi, thì hãy lấy hơi thật sâu rồi nhìn rộng ra cả chương trình. Hãy đảm bảo chắc rằng chương trình được viết thụt đầu dòng đúng chuẩn; điều này giúp ta phát hiện lỗi cú pháp dễ dàng hơn.

Bây giờ, hãy tìm kiếm những lỗi dễ mắc phải:

1. Kiểm tra tất cả những cặp ngoặc tròn và ngoặc nhọn phải cân xứng và được lồng ghép đúng thứ tự. Tất cả lời định nghĩa phương thức phải được lồng trong một lời định nghĩa lớp. Tất cả các câu lệnh của chương trình phải đặt trong định nghĩa phương thức.
2. Hãy nhớ rằng viết chữ in thì khác với chữ thường.
3. Kiểm tra dấu chấm phẩy ở cuối câu lệnh (và không có dấu chấm phẩy theo sau ngoặc nhọn).
4. Hãy đảm bảo chắc rằng mỗi chuỗi kí tự trong mã lệnh phải có đôi dấu nháy kép. Đảm bảo chắc rằng bạn dùng nháy kép cho chuỗi và nháy đơn cho kí tự.
5. Với từng câu lệnh gán, hãy đảm bảo rằng kiểu dữ liệu ở bên vế trái cũng giống như kiểu vế phải. Hãy đảm bảo rằng biểu thức bên vế trái là một tên biến hoặc đối tượng nào khác mà bạn có thể gán giá trị vào cho nó (như một phần tử của mảng).
6. Với từng lần kích hoạt phương thức, hãy đảm bảo rằng các đối số được cung cấp đã xếp đúng vị trí, và có đúng kiểu, và đối tượng mà bạn đang kích hoạt phương thức lên cũng có đúng kiểu.
7. Nếu bạn đang kích hoạt một phương thức trả giá trị, hãy đảm bảo chắc rằng bạn thao tác với giá trị trả về này. Nếu bạn kích hoạt một phương thức rỗng, hãy đảm bảo chắc rằng mình *không* thử làm gì với kết quả.
8. Nếu bạn đang kích hoạt một phương thức đối tượng, hãy chắc rằng bạn đang kích hoạt nó với một đối tượng đúng kiểu. Nếu bạn đang kích hoạt một phương thức lớp từ bên ngoài phương thức mà nó được định nghĩa, hãy đảm bảo chắc rằng bạn đã chỉ định tên lớp này.
9. Bên trong một phương thức đối tượng, bạn có thể tham chiếu tới các biến thực thể mà không quy định đối tượng nào. Nếu bạn thử làm điều này trong một phương thức lớp, bạn sẽ nhận được thông báo kiểu như, “Tham chiếu tĩnh tới biến không tĩnh.”
   Nếu không có giải pháp nào kể trên phát huy tác dụng, hãy xem mục kế tiếp…

TÔI KHÔNG THỂ BIÊN DỊCH ĐƯỢC CHƯƠNG TRÌNH DÙ ĐÃ CỐ GẮNG MỌI CÁCH.

Nếu như trình biên dịch nói rằng có lỗi mà bạn không nhìn thấy, thì có khả năng là do bạn và trình biên dịch không cùng nhìn vào đoạn mã lệnh. Hãy kiểm tra môi trường phát triển đang dùng để đảm bảo chắc rằng chương trình bạn đang soạn thảo chính là chương trình đang được biên dịch. Nếu bạn còn chưa chắc chắn, hãy thử cố tình đưa vào một lỗi cú pháp ngay ở đầu chương trình. Bây giờ hãy biên dịch lại. Nếu trình biên dịch vẫn không tìm thấy lỗi mới đó, thì có lẽ bạn đã thiết lập môi trường tích hợp sai quy cách.

Nếu bạn đã kiểm tra mã lệnh một lượt rồi, và chắc chắn là trình biên dịch đang làm việc với đúng mã lệnh mình soạn thảo, thì đã đến lúc dùng phương pháp “tuyệt vọng”: **gỡ lỗi bằng cách chia đôi**.

- Hãy tạo một bản sao của file hiện hành. Nếu bạn đang soạn file Bob.java, hãy lập bản sao có

tên Bob.java.old.

- Xóa bớt một nửa mã lệnh từ file Bob.java. Thử biên dịch lại.
- Nếu giờ đây chương trình biên dịch được thì bạn biết rằng lỗi nằm ở nửa kia. Hãy phục hồi lại nửa vừa xóa rồi lặp lại cách thử.
- Nếu chương trình vẫn không biên dịch nổi, thì lỗi sai phải nằm ở nửa còn lại này. Hãy xóa đi một nửa số mã lệnh rồi lặp lại cách thử.
- Một khi bạn đã tìm thấy và sửa được lỗi, thì hãy dần dần phục hồi lại phần mã lệnh đã xóa, từng ít từng ít một.

  Quá trình này thật lộm cộm, nhưng thật ra là nhanh hơn so với bạn nghĩ; và cách này cũng rất đáng tin cậy.

TÔI ĐÃ LÀM THEO CHỈ DẪN CỦA TRÌNH BIÊN DỊCH MÀ VẪN CHƯA CÓ TÁC DỤNG.

Một số thông báo của trình biên dịch lại có đoạn lời khuyên nhủ, chẳng hạn "class Golfer must be declared abstract. It does not define int compareTo(java.lang.Object) from interface java.lang.Comparable." (lớp Golfer phải được khai báo là trừu tượng. Nó không định nghĩa int compareTo(java.lang.Object) từ trong interface java.lang.Comparable.) Nghe có vẻ như trình biên dịch đang bảo bạn khai báo Golfer là lớp trừu tượng, và nếu bạn đọc cách này thì có lẽ chẳng hiểu đó là gì hoặc cách làm thế nào.

Thật may là, trình biên dịch đã sai. Trong trường hợp này, giải pháp là đảm bảo rằng Golfer có một phương thức mang tên compareTo để nhận tham số là một Object.

Đừng để trình biên dịch dắt mũi bạn. Các thông báo lỗi cho bạn chứng cớ là đã có trục trặc, nhưng cách khắc phục mà nó chúng đưa ra đều không đáng tin cậy.

D.2 Lỗi thực thi

CHƯƠNG TRÌNH TÔI VIẾT BỊ TREO.

Nếu một chương trình dừng lại và hình như không làm gì, ta nói rằng nó đã bị **treo**. Thường thì điều này nghĩa là nó mắc phải một vòng lặp vô hạn hoặc đệ quy vô hạn.

- Nếu có một vòng lặp cụ thể mà bạn nghi ngờ có vấn đề, hãy thêm một lệnh `print` ngay trước vòng lặp, để in ra "tien vao vong lap" và một lệnh khác ngay sau vòng lặp, in ra "thoat khoi vong lap". Chạy chương trình. Nếu bạn thấy được thông điệp thứ nhất mà không thấy cái thứ hai thì đã có một vòng lặp vô hạn. Xem tiếp mục "Vòng lặp vô hạn" dưới đây.
- Ở hầu hết trường hợp, đệ quy vô hạn sẽ làm cho chương trình chạy một lúc và sau đó phát ra biệt lệ StackOverflowException. Nếu điều này xảy ra, hãy xem tiếp mục "Đệ quy vô hạn" sau đây. Nếu bạn không gặp biệt lệ StackOverflowException này nhưng nghi ngờ rằng có vấn đề xảy ra với một phương thức hoặc hàm đệ quy, bạn vẫn có thể sử dụng các kĩ thuật trong mục "Đệ quy vô hạn".
- Nếu cách này cũng không có tác dụng thì có thể là bạn chưa hiểu luồng thực hiện của chương trình. Hãy đọc tiếp mục "Luồng thực thi" bên dưới.

VÒNG LẶP VÔ HẠN

Nếu bạn nghĩ rằng bạn có một vòng lặp vô hạn và cho rằng mình đã biết được vòng lặp nào gây ra vấn đề, thì hãy thêm một lệnh `print` tại điểm cuối vòng lặp và in ra giá trị các biến trong điều kiện cùng với giá trị của điều kiện.

Chẳng hạn:

```
    while (x > 0 && y < 0) {
      // thao tác gì đó với x
      // thao tác gì đó với y
      System.out.println("x: " + x);
      System.out.println("y: " + y);
      System.out.println("điều kiện: " + (x > 0 && y < 0));
    }
```

Bây giờ khi chạy chương trình, bạn sẽ thấy ba dòng kết quả với mỗi lần chạy qua vòng lặp. Lần cuối cùng chạy qua vòng lặp điều kiện sẽ phải là `false`. Nếu vòng lặp tiếp tục chạy, bạn sẽ nhìn được các giá trị của `x` và `y`, và có thể hình dung được tại sao chúng không được cập nhật đúng.

ĐỆ QUY VÔ HẠN

Trong nhiều trường hợp, một vòng lặp đệ quy sẽ khiến chương trình phát biệt lệ StackOverflowException. Nhưng nếu chương trình chậm chạp có thể nó sẽ tốn nhiều thời gian để bị đầy ngăn xếp.

Nếu bạn nghi ngờ rằng một hàm hoặc phương thức nào đó gây ra đệ quy vô hạn, hãy bắt đầu kiểm tra để chắc rằng có một trường hợp cơ sở. Nói cách khác, cần phải có điều kiện nào đó để khiến cho hàm hoặc phương thức trả về mà không gọi đệ quy nữa. Nếu không, bạn cần phải nghĩ lại thuật toán và tìm ra một trường hợp cơ sở.

Nếu có một trường hợp cơ sở nhưng chương trình dường như không đạt đến đó, thì hãy thêm câu lệnh `print` vào điểm đầu của hàm hoặc phương thức để in ra các tham biến. Bây giờ khi chạy chương trình, bạn sẽ thấy một ít dòng kết quả mỗi lần hàm hoặc phương thức được gọi đến, và sẽ thấy giá trị các tham số. Nếu tham số không thay đổi với xu hướng về trường hợp cơ sở, bạn sẽ thấy được tại sao.

LUỒNG THỰC THI

Nếu bạn không chắc chắn về luồng thực hiện trong chương trình, hãy thêm các câu lệnh `print` vào điểm đầu của mỗi hàm với thông báo kiểu như “bắt đầu phương thức `foo`”, trong đó `foo` là tên phương thức.

Bây giờ khi chạy chương trình, nó sẽ in ra một dấu vết của mỗi phương thức khi được kích hoạt đến.

Bạn cũng có thể in ra những đối số mà từng phương thức nhận được. Khi chạy chương trình, hãy kiểm tra xem các giá trị này hợp lý không, và kiểm tra một trong những lỗi thường mắc phải nhất—cung cấp các đối số sai thứ tự.

KHI CHẠY CHƯƠNG TRÌNH TÔI NHẬN ĐƯỢC MỘT BIỆT LỆ.

Khi có biệt lệ xảy ra, Java sẽ in một thông báo trong đó có tên của biệt lệ, dòng lệnh có vấn đề, và một lần dấu vết trên ngăn xếp (stack trace). Bản thân cái lần vết này chứa thông tin về phương thức đang được chạy, và phương thức kích hoạt nó, rồi phương thức kích hoạt phương thức đó, và cứ như vậy.

Bước đầu tiên là kiểm tra vị trí trong chương trình nơi mà lỗi xuất hiện, đồng thời thử hình dung điều gì đã xảy ra.

NullPointerException:

Bạn cố gắng truy cập một biến thực thể hoặc kích hoạt một phương thức trên đối tượng mà bản thân nó đang là null. Bạn cần phải hình dung ra biến nào là null rồi hình dung xem bằng cách nào dẫn đến hiện tượng đó. Hãy nhớ rằng khi khai báo một biến với một kiểu đối tượng, thì ban đầu nó vẫn là null đến tận khi bạn gán giá trị cho. Chẳng hạn, đoạn mã sau gây ra biệt lệ NullPointerException:

```
Point blank;

System.out.println(blank.x);
```

ArrayIndexOutOfBoundsException:

Chỉ số mà bạn đang dùng để truy cập một mảng đã lớn hơn array.length-1. Nếu bạn tìm được vị trí lỗi, hãy thêm vào câu lệnh `print` *vào ngay trước lỗi này để hiển thị giá trị của chỉ số cùng với chiều dài của mảng. Liệu mảng này có kích thước đúng chưa? Chỉ số có đúng không? Bây giờ tìm ngược lại dọc chương trình và xem mảng này cùng với chỉ số đó bắt nguồn từ đâu. Hãy tìm lệnh gán gần nhất và xem nó có thực hiện đúng không. Nếu không có cái nào là tham số, thì hãy đến chỗ phương thức được kích hoạt và xem các giá trị này đến từ đâu.*

StackOverFlowException:

Xem "Đệ quy vô hạn."

FileNotFoundException:

Điều này nghĩa là Java không tìm thấy file cần thiết. Nếu bạn đang dùng một môi trường phát triển dựa trên các dự án, như Eclipse, thì có khả năng là bạn sẽ phải nhập file đó vào trong dự án đang mở. Còn không thì hãy đảm bảo chắc rằng file đó tồn tại và đường dẫn đến nó được ghi đúng. Vấn đề này tuỳ thuộc vào hệ thống file trên máy tính của bạn, bởi vậy có thể khó dò tìm.

ArithmeticException:

Biệt lệ phát ra khi có trục trặc với phép toán số học, thường là phép chia cho số không.

```
TÔI ĐÃ THÊM VÀO QUÁ NHIỀU LỆNH PRINT ĐẾN NỖI BÂY GIỜ NGẬP TRÀN KẾT QUẢ ĐẦU RA.
```

Một trong những vấn đề khi dùng lệnh `print` để gỡ lỗi là việc bạn có thể bị chìm trong kết quả ra. Có hai cách tiếp tục: đơn giản hóa đầu ra hoặc đơn giản hóa chương trình.

Để giản hóa kết quả đầu ra, bạn cần xóa bỏ hoặc đưa vào chú thích những dòng lệnh `print` vốn không có tác dụng, hoặc kết hợp chúng lại, hoặc sửa định dạng đầu ra để dễ hiểu hơn.
Để giản hóa chương trình, có vài cách làm được. Trước hết, hãy giảm quy mô của bài toán xuống. Chẳng hạn, nếu bạn cần tìm kiếm trong mảng, hãy làm với một mảng *nhỏ*. Nếu chương trình nhận đầu vào từ phía người dùng, hãy cho những dữ liệu vào đơn giản mà gây ra lỗi.
Đồng thời hãy dọn dẹp chương trình. Hãy bỏ những đoạn mã chết và tổ chức lại chương trình để nó càng dễ đọc càng tốt. Chẳng hạn, nếu bạn nghi rằng vấn đề nằm ở một đoạn nằm sâu trong chương trình, hãy thử viết lại nó với cấu trúc đơn giản hơn. Nếu bạn nghi ngờ rằng có một phương thức lớn, hãy thử chẻ nhỏ thành những phương thức nhỏ và kiểm tra lần lượt.

Thông thường quá trình tìm ra trường hợp thử đơn giản nhất sẽ dẫn bạn đến điểm gây lỗi. Chẳng hạn, nếu bạn thấy chương trình chạy được trong trường hợp số phần tử trong mảng là chẵn nhưng không được khi số phần tử là lẻ, thì điều đó sẽ là dấu vết cho thấy điều gì đang diễn ra.

Việc tổ chức lại một đoạn mã có thể giúp bạn phát hiện những lỗi nhỏ. Nếu bạn thực hiện sửa đổi mà

nghĩ rằng nó không ảnh hưởng gì đến chương trình, và lúc có ảnh hưởng thì đó sẽ là bài học cho bạn.

D.3 Lỗi logic

CHƯƠNG TRÌNH TÔI VIẾT RA KHÔNG HOẠT ĐỘNG ĐÚNG.

Lỗi logic rất khó tìm, vì trình biên dịch và hệ thống lúc thực thi không cung cấp thông tin gì về sự trục trặc. Chỉ có bạn mới biết rằng chương trinh cần phải thực hiện điều gì.

Bước đầu tiên là tạo lập một kết nối giữa nội dung chương trình và biểu hiện mà bạn quan sát được. Bạn cần giả thiết về điều thật sự mà chương trình đang thực hiện. Bạn cần tự hỏi mình những điều sau:

- Có điều gì mà chương trình cần phải làm nhưng dường như nó không làm hay không? Hãy tìm ra đoạn mã lệnh thực hiện tính năng đó và chắc rằng nó được thực thi khi bạn nghĩ rằng lẽ ra nó phải chạy.
- Có điều gì đang diễn ra mà lẽ ra không nên có nó? Hãy tìm đoạn mã trong chương trình mà thực hiện tính năng đó rồi xem liệu nó có được thực khi trong khi đáng lẽ thì không.
- Có đoạn mã nào tạo ra một hiệu ứng mà không như bạn mong đợi không? Hãy chắc rằng bạn hiểu được đoạn mã nghi vấn, đặc biệt khi nó liên quan đến việc kích hoạt phương thức Java. Hãy đọc tài liệu về những phương thức, rồi thử bằng những trường hợp kiểm tra đơn giản. Có khi chúng lại không làm việc mà bạn nghĩ rằng chúng sẽ làm.

Để lập trình, bạn phải có một mô hình tưởng tượng về cách thức hoạt động của chương trình. Nếu bạn viết một chương trình mà không thực hiện đúng việc bạn mong đợi, thì thường là vấn đề không nằm ở chương trình; nó nằm ở mô hình tưởng tượng của bạn.

Cách tốt nhất để sửa mô hình tưởng tượng cho đúng là chia chương trình thành những bộ phận (thường là các lớp và phương thức) rồi kiểm tra chạy thử từng bộ phận một cách độc lập. Một khi bạn thấy sự khác biệt giữa mô hình và thực tế, bạn sẽ có thể giải quyết vấn đề.

Sau đây là một số lỗi logic thông thường cần phải kiểm tra:

- Luôn nhớ rằng phép chia nguyên làm tròn xuống. Nếu bạn muốn cả phần thập phân, hãy dùng số double.
- Số phẩy động chỉ là gần đúng, nên bạn đừng lệ thuộc vào độ chính xác tuyệt đối.
- Nói chung, hãy dùng số nguyên cho những thứ đếm được và dùng số phẩy động cho thứ đo được.
- Nếu bạn dùng toán tử gán (=) thay vì toán tử bằng (==) trong điều kiện của một lệnh if, while, hoặc for, bạn có thể sẽ nhận một biểu thức về mặt cú pháp thì đúng nhưng về ngữ nghĩa thì sai.
- Khi bạn áp dụng toán tử bằng (==) với đối tượng, nó sẽ kiểm tra identity. Nếu bạn có ý muốn kiểm tra độ tương đương, hãy dùng phương thức equals.
- Đối với các kiểu dữ liệu do người dùng định nghĩa, equals sẽ kiểm tra identity. Nếu bạn muốn một kí hiệu khác cho tương đồng, bạn phải ghi đè lên nó.
- Kế thừa có thể dẫn đến những lỗi logic rất chi li, bởi bạn có thể chạy mã lệnh được kế thừa mà không nhận ra nó. Hãy xem mục “Luồng thực thi” ở trên.

TÔI CÓ MỘT BIỂU THỨC LỚN VÀ GAI GÓC MÀ CHẲNG HOẠT ĐỘNG THEO SỰ MONG ĐỢI.

Việc viết những biểu thức phức tạp cũng tốt miễn là chúng dễ đọc, nhưng chúng có thể làm việc gỡ lỗi gặp khó khăn. Thông thường nên chẻ nhỏ một biểu thức thành một loạt các lệnh gán cho những biến tạm thời.

Chẳng hạn:

```
    rect.setLocation(rect.getLocation().translate( -rect.getWidth(),
-rect.getHeight()));
```

Có thể được viết lại thành

```
    int dx = -rect.getWidth();
    int dy = -rect.getHeight();
    Point location = rect.getLocation();
    Point newLocation = location.translate(dx, dy);
    rect.setLocation(newLocation);
```

Dạng mã lệnh chi tiết thì dễ đọc hơn vì tên biến cho ta bản thân đã giúp giải thích rõ thêm, và cũng dễ gỡ lỗi hơn vì bạn có thể kiểm tra kiểu của những biến trung gian cùng việc hiển thị giá trị của chúng.

Một vấn đề khác có thể xảy ra với những biểu thức lớn là thứ tự thực hiện phép tính có thể không như bạn mong muốn. Chẳng hạn, để lượng giá biểu thức x / 2π , có thể bạn đã viết:

```
    double y = x / 2 * Math.PI;
```

Điều này không đúng vì các phép nhân và chia có cùng thứ tự ưu tiên và được lượng giá từ trái sang phải. Vì vậy biểu thức này sẽ tính *x*π / 2.
Một cách hay để gỡ lỗi biểu thức là thêm vào những cặp ngoặc đơn để giúp cho thứ tự lượng giá được rõ ràng:

```
    double y = x / (2 * Math.PI);
```

Phiên bản này thì đúng đắn, và dễ đọc hơn đối với người không ghi nhớ thứ tự thực hiện phép toán.

PHƯƠNG THỨC TÔI ĐÃ VIẾT KHÔNG TRẢ LẠI GIÁ TRỊ NHƯ DỰ KIẾN.

Nếu bạn viết một câu lệnh return (trả về) với một biểu thức phức tạp, thì bạn đã không có cơ hội in ra giá trị này trước khi trả nó về. Một lần nữa, bạn có thể dùng biến tạm. Chẳng hạn, thay vì

```
  public Rectangle intersection(Rectangle a, Rectangle b) {
    return new Rectangle( Math.min(a.x, b.x), Math.min(a.y, b.y),
Math.max(a.x+a.width, b.x+b.width)-Math.min(a.x, b.x) Math.max(a.y+a.height,
b.y+b.height)-Math.min(a.y, b.y) );
  }
```

bạn đã có thể viết

```
  public Rectangle intersection(Rectangle a, Rectangle b) {
    int x1 = Math.min(a.x, b.x);
    int y2 = Math.min(a.y, b.y);
    int x2 = Math.max(a.x+a.width, b.x+b.width);
    int y2 = Math.max(a.y+a.height, b.y+b.height);
    Rectangle rect = new Rectangle(x1, y1, x2-x1, y2-y1);
```

```
    return rect;
  }
```

Giờ thì bạn đã có cơ hội hiển thị bất kì biến trung gian nào trước khi trả về. Và bằng cách dùng lại x1 cùng y1, bạn cũng làm mã lệnh gọn hơn

CÂU LỆNH PRINT MÀ TÔI VIẾT CHẲNG LÀM ĐƯỢC GÌ CẢ

Nếu bạn dùng phương thức println, kết quả đầu ra sẽ hiện lên ngay; nhưng nếu bạn dùng print (ít nhất là có những môi trường phát triển như vậy), kết quả sẽ được lưu lại mà không hiện lên cho đến tận khi có dấu xuống dòng tiếp theo. Nếu chương trình kết thúc mà không in ra một dòng mới thì có thể bạn chẳng còn nhìn thấy được kết quả lưu lại nữa.
Nếu bạn nghi ngờ là đã có điều này xảy ra, hãy chuyển một số hoặc tất cả các lệnh print trong chương trình thành println.

THẬT SỰ TÔI RẤT, RẤT VƯỚNG MẮC VÀ CẦN ĐƯỢC GIÚP ĐỠ.

Trước hết, hãy thử rời khỏi máy tính trong vài phút. Máy tính phát ra sóng từ gây ảnh hưởng đến não, với các triệu chứng sau:

- Cáu giận.
- Tin tưởng vào lực siêu nhiên (“máy tính này ghét tôi”) và những ảo tưởng (“chương trình chỉ chạy khi tôi đội ngược mũ”).
- Lập trình bước ngẫu nhiên (nỗ lực lập trình bằng cách viết tất cả các trường hợp chương trình có thể có và chọn ra một phiên bản hoạt động đúng).

Nếu bạn tự thấy mình mắc phải một trong số các triệu chứng trên, hãy đứng dậy và đi dạo. Khi đã tĩnh tâm hẳn, hãy nghĩ lại chương trình. Nó đang làm điều gì? Đâu là các nguyên nhân gây ra biểu hiện đó? Lần cuối cùng chương trình cọn chạy được là lúc nào, và sau đó bạn thực hiện những điều gì?

Đôi khi phát hiện lỗi chỉ là vấn đề thời gian. Tôi thường tìm thấy lỗi trong lúc rời xa khỏi máy tính và để trí óc khuây khỏa. Một số nơi tốt nhất để thoát khỏi máy gồm có trên tàu, khi đi tắm, và trước khi đi ngủ.

KHÔNG, TÔI THẬT SỰ MUỐN GIÚP ĐỠ.

Điều đó xảy ra. Ngay cả những lập trình viên giỏi nhất đôi lúc cũng bị bí. Đôi khi bạn làm một chương trình lâu quá đến nỗi không thể phát hiện ra lỗi. Tìm một người có góc nhìn khác chính là điều cần thiết.

Trước khi yêu cầu giúp đỡ, bạn hãy chuẩn bị kĩ. Chương trình phải càng đơn giản càng tốt, và hãy phân tích trên dữ liệu đầu vào nhỏ nhất có thể gây lỗi. Bạn cần có các lệnh `print` ở những vị trí thích hợp (và kết quả đầu ra phải dễ hiểu). Bạn cần hiểu rõ vấn đề để có thể diễn đạt nó một cách ngắn gọn.
Khi đưa người đến giúp, hãy chắc chắn rằng bạn cung cấp đủ thông tin mà họ cần:

- Nếu có thông báo lỗi, thông báo đó là gì và nó chỉ định phần nào trong chương trình?
- Việc cuối cùng mà bạn thao tác trước khi lỗi này xảy ra là gì? Những dòng lệnh nào bạn vừa mới viết gần đây nhất, hay trường hợp chạy thử gần đây nhất mới bị thất bại là gì?
- Bạn đã thử những biện pháp gì rồi, và thu hoạch được gì?

Đến khi bạn giải thích được khúc mắc cho người ta, có thể bạn sẽ thấy kết quả. Hiện tượng này thường

gặp đến nỗi người ta gợi ý một kĩ thuật gỡ lỗi có tên "vịt cao su." Sau đây là cách hoạt động:

1. Mua một con vịt cao su chuẩn.
2. Khi bạn thực sự đã vướng mắc trong lập trình, hãy đặt con vịt cao su trước mặt rồi nói, "Vịt ơi, tao đang vướng mắc đây. Hoàn cảnh là như thế này..."
3. Trình bày vấn đề cho con vịt.
4. Tìm thấy hướng giải quyết.
5. Cám ơn con vịt cao su.

Tôi không hề nói đùa. Hãy xem http://en.wikipedia.org/wiki/Rubber_duck_debugging.

TÔI ĐÃ TÌM THẤY LỖI RỒI!

Khi bạn tìm thấy lỗi, thông thường cách sửa sẽ là hiển nhiên. Nhưng không phải luôn luôn như vậy. Đôi khi cái mà có vẻ như lỗi lại là một dấu hiệu cho thấy bạn chưa hiểu chương trình viết ra, hoặc là có một lỗi trong thuật toán bạn dùng. Với những trường hợp này, bạn có thể sẽ phải nghĩ lại thuật toán, hay chỉnh lại mô hình nhận thức của mình. Hãy dành thời gian rời xa máy tính, để suy nghĩ, tự tính tay các phép thử, hoặc vẽ sơ đồ biểu diễn bài toán.

Sau khi sửa xong lỗi, bạn đừng chuyển sang lỗi mới. Hãy nghĩ một lát xem vừa rồi là loại lỗi gì, tại sao bạn mắc phải lỗi này, làm thế nào mà lỗi đã lộ diện, và đáng ra bạn có thể tìm lỗi này nhanh hơn bằng cách nào. Lần sau khi bạn thấy điều tương tự, có thể bạn sẽ chóng phát hiện ra lỗi hơn.