Nhập môn lập trình Java

Roy Miller, Tác giả, The Other Road, LLC

Tóm tắt: Ngôn ngữ Java, và nền tảng Java luôn phát triển là một cuộc cách mạng trong lập trình. Mục tiêu của hướng dẫn này là giới thiệu cho bạn cú pháp của Java mà bạn hầu như chắc chắn sẽ gặp trên con đường nghề nghiệp và cho bạn thấy những thành tố đặc thù (idioms) của nó giúp bạn tránh khỏi những rắc rối. Theo bước Roy Miller, chuyên gia Java khi ông hướng dẫn bạn những điểm cốt yếu của lập trình Java, bao gồm mẫu hình hướng đối tượng (OPP) và cách thức áp dụng nó vào lập trình Java; cú pháp của ngôn ngữ Java và cách sử dụng; tạo ra đối tượng và thêm các hành vi, làm việc với các sưu tập (collections), xử lý lỗi; các mẹo để viết mã lệnh tốt hơn.

Về tài liệu này

Tài liệu đề cập đến những gì?

Hướng dẫn này giới thiệu cho bạn phương pháp lập trình hướng đối tượng (OOP) bằng ngôn ngữ Java. Nền tảng Java là một chủ đề rộng lớn, bởi vậy chúng ta không thể đề cập đến tất cả trong tài liệu này, nhưng chúng ta sẽ trình bày đủ để bạn có thể khởi đầu. Tài liệu tiếp theo sẽ cung cấp thêm các thông tin và hướng dẫn bạn trong nỗ lực lập trình bằng Java.

Ngôn ngữ Java tất nhiên có cả ưu và nhược điểm, nhưng ảnh hưởng của nó đến ngành công nghiệp phát triển phần mềm là không thể chối cãi. Mặt tốt là ngôn ngữ Java gây ít khó khăn hơn so với C++. Java giảm bớt một số nhiệm vụ lập trình phiền phức, như quản lý tường minh bộ nhớ, và cho phép người lập trình tập trung vào logic nghiệp vụ. Mặt kém của nó, theo những người thuần túy chủ nghĩa hướng đối tượng, là Java có quá nhiều tàn dư phi đối tượng để được coi là một công cụ tốt. Tuy nhiên, bất kể vị trí của bạn như thế nào thì biết cách sử dụng ngôn ngữ này như một công cụ khi nó là công cụ thích hợp nhất phục vụ cho công việc của bạn vẫn là một lựa chọn nghề nghiệp sáng suốt.

Tôi nên đọc tài liệu này không chứ?

Nội dung của tài liệu hướng dẫn này nhằm đến những người mới bắt đầu lập trình Java, những người có thể chưa quen với các khái niệm lập trình hướng đối tượng hoặc đặc biệt là chưa quen với nền tảng Java. Nó coi như người đọc đã có kiến thức chung để tải về và cài đặt phần mềm, kiến thức chung về lập trình và cấu trúc dữ liệu (như mảng), nhưng không yêu cầu nhiều hơn việc làm quen sơ qua với lập trình hướng đối tượng.

Tài liệu này sẽ hướng dẫn bạn từ việc thiết lập nền Java trên máy của bạn, cài đặt và làm việc với Eclipse, một môi trường phát triển tích hợp (IDE) miễn phí để viết mã Java. Từ điểm này trở đi, bạn sẽ học những điều căn bản của lập trình Java, bao gồm mẫu hình lập trình hướng đối tượng và cách áp dụng nó vào lập trình Java; cú pháp và cách dùng ngôn ngữ Java; tạo ra các đối tượng và thêm các hành vi, làm việc với các sưu tập, xử lý lỗi; các mẹo để viết mã lệnh tốt hơn. Sau khi học hết tài liệu này, bạn sẽ trở thành một lập trình viên Java – một lập trình viên Java khởi đầu, nhưng dù sao vẫn là một lập trình viên Java.

Các yêu cầu về phần mềm

Để chạy các ví dụ hoặc các đoạn mã mẫu trong hướng dẫn này, bạn sẽ cần có Java 2 Platform, Standard Edition (J2SE), phiên bản 1.4.2 hoặc mới hơn, và máy của bạn phải cài IDE Eclipse. Đừng lo nếu bạn còn chưa cài đặt những gói này – chúng tôi sẽ chỉ cho bạn cách làm trong phần khởi động. Tất cả ví dụ mã lệnh trong hướng dẫn này được kiểm thử với J2SE 1.4.2 chạy trên nền Windows XP. Tuy nhiên, một điều tuyệt vời của nền tảng Eclipse là nó chạy trên hầu hết các hệ điều hành mà có khả năng bạn đang dùng, bao gồm Windows 98/ME/2000/XP, Linux, Solaris, AIX, HP-UX, và thậm chí cả Mac OS X.

Khởi động

Các chỉ dẫn cài đặt

Trong mấy phần tiếp theo, tôi sẽ hướng dẫn bạn qua từng bước tải về và cài đặt Java 2 Platform Standard Edition (J2SE), phiên bản 1.4.2 và IDE Eclipse. Nền Java giúp bạn dịch và chạy chương trình Java. Còn IDE Eclipse mang lại cho bạn công cụ hùng mạnh và thân thiện với người sử dụng để viết mã lệnh bằng ngôn ngữ Java. Nếu bạn đã cài đặt Java SDK và Eclipse, hãy vui lòng chuyển ngay sang phần Tìm hiểu nhanh về Eclipse hoặc mục tiếp theo, Các khái niệm lập trình hướng đối tượng, nếu bạn cảm thấy thuận tiện.

Cài đặt Java SDK

Mục đích ban đầu của ngôn ngữ Java là cho phép các lập trình viên viết một chương trình để chạy trên bất cứ nền tảng nào, một ý tưởng gói gọn trong cụm từ "Viết một lần, chạy bất cứ đâu" (WORA). Trong thực tế, điều này hoàn toàn không đơn giản, nhưng nó đang trở nên dễ dàng hơn. Nhiều thành phần khác nhau trong công nghệ Java hỗ trợ cho nỗ lực này. Java có 3 ấn bản, ấn bản chuẩn (Standard), ấn bản doanh nghiệp (Enterprise), và ấn bản di động (Mobile), hai ấn bản sau tương ứng dành cho việc phát triển ứng dụng doanh nghiệp và thiết bị cầm tay. Chúng ta sẽ làm việc với J2SE, bao gồm tất cả các thư viện lõi của Java. Tất cả những gì bạn cần làm là tải về và cài đặt.

Để tải về bộ phát triển phần mềm J2SE (J2SE SDK), làm theo các bước sau:

1. Mở trình duyệt và đi đến trang chủ Công nghệ Java. Tại giữa đầu trang bạn sẽ thấy nhiều đường kết nối đến các vùng chủ đề công nghệ Java khác nhau. Chọn **J2SE (Core/Desktop)**.

2. Trong danh sách các bản phát hành J2SE hiện tại, chọn **J2SE 1.4.2**.

3. Tại cột dẫn hướng bên trái của trang kết quả, nhấp chuột vào **Downloads**.

4. Có một vài liên kết tải về trên trang này. Tìm và nhấp chọn liên kết **Download J2SE SDK**.

5. Chấp nhận các điều kiện về giấy phép sử dụng và nhấp chọn **Continue**.

6. Bạn sẽ thấy một danh sách các gói tải về theo từng nền hệ điều hành. Chọn gói tải về thích hợp với bất cứ nền hệ điều hành nào mà bạn đang dùng.

7. Ghi lưu tệp vào ổ cứng của bạn.

8. Khi tải về xong, chạy chương trình cài đặt để cài đặt SDK trên ổ cứng của bạn, nên chọn thư mục có tên thích hợp ngay trong thư mục gốc của ổ cứng.

Tuyệt! Bây giờ bạn đã có môi trường Java trên máy mình. Bước tiếp theo là cài đặt môi trường phát triển tích hợp (IDE).

Cài đặt Eclipse

Môi trường phát triển tích hợp (IDE) che giấu đi nhiều chi tiết công nghệ trần tục trong khi làm việc với ngôn ngữ Java, do đó bạn có thể tập trung vào viết và chạy mã lệnh. Bộ JDK mà bạn vừa cài đặt bao gồm vài công cụ dòng lệnh cho phép bạn biên dịch và chạy các chương trình Java mà không cần có IDE, nhưng sử dụng những công cụ này nhanh chóng trở nên rất vất vả chỉ trừ những chương trình đơn giản nhất. Sử dụng một IDE che giấu đi nhiều chi tiết sẽ mang lại cho bạn những công cụ mạnh để giúp bạn lập trình nhanh hơn và tốt hơn, và đơn giản nó là một cách lập trình rất dễ chịu.

Không còn cần thiết phải trả tiền để mua một IDE tuyệt hảo. IDE Eclipse là một dự án nguồn mở và nó là của bạn, tải về miễn phí. Eclipse lưu trữ và theo dõi mã lệnh Java của bạn trong những tệp dữ liệu dễ đọc nằm trong hệ thống tệp của bạn. (Bạn cũng có thể dùng Eclipse để làm việc với mã lệnh trong kho CVS). Tin tốt lành là Eclipse để bạn làm việc với tệp nếu bạn muốn, nhưng nó ẩn giấu đi chi tiết về tệp nếu bạn chỉ muốn làm việc với các cấu trúc Java khác như các lớp chẳng hạn (ta sẽ thảo luận chi tiết sau).

Việc tải về và cài đặt Eclipse rất đơn giản. Hãy làm theo những bước sau:

1. Mở trình duyệt và đi đến trang web của Eclipse.

2. Nhấn chọn đường liên kết **Downloads** ở bên trái của trang.

3. Nhấn chọn đường liên kết **Main Eclipse Download Site** để vào trang tải về của dự án Eclipse.

4. Bạn sẽ thấy một danh sách các kiểu xây dựng (build types) và tên. Chọn mục **3.0**.

5. Ở giữa trang, bạn sẽ thấy một danh sách các SDK Eclipse tùy theo nền hệ điều hành; chọn cái thích hợp với hệ thống của bạn.

6. Ghi lưu tệp vào ổ cứng.

7. Khi tải về xong, chạy trình cài đặt và cài đặt Eclipse vào ổ cứng của bạn, nên chọn thư mục có tên thích hợp ngay trong thư mục gốc của ổ cứng.

Tất cả những việc còn lại bây giờ là thiết đặt IDE.

Thiết đặt Eclipse

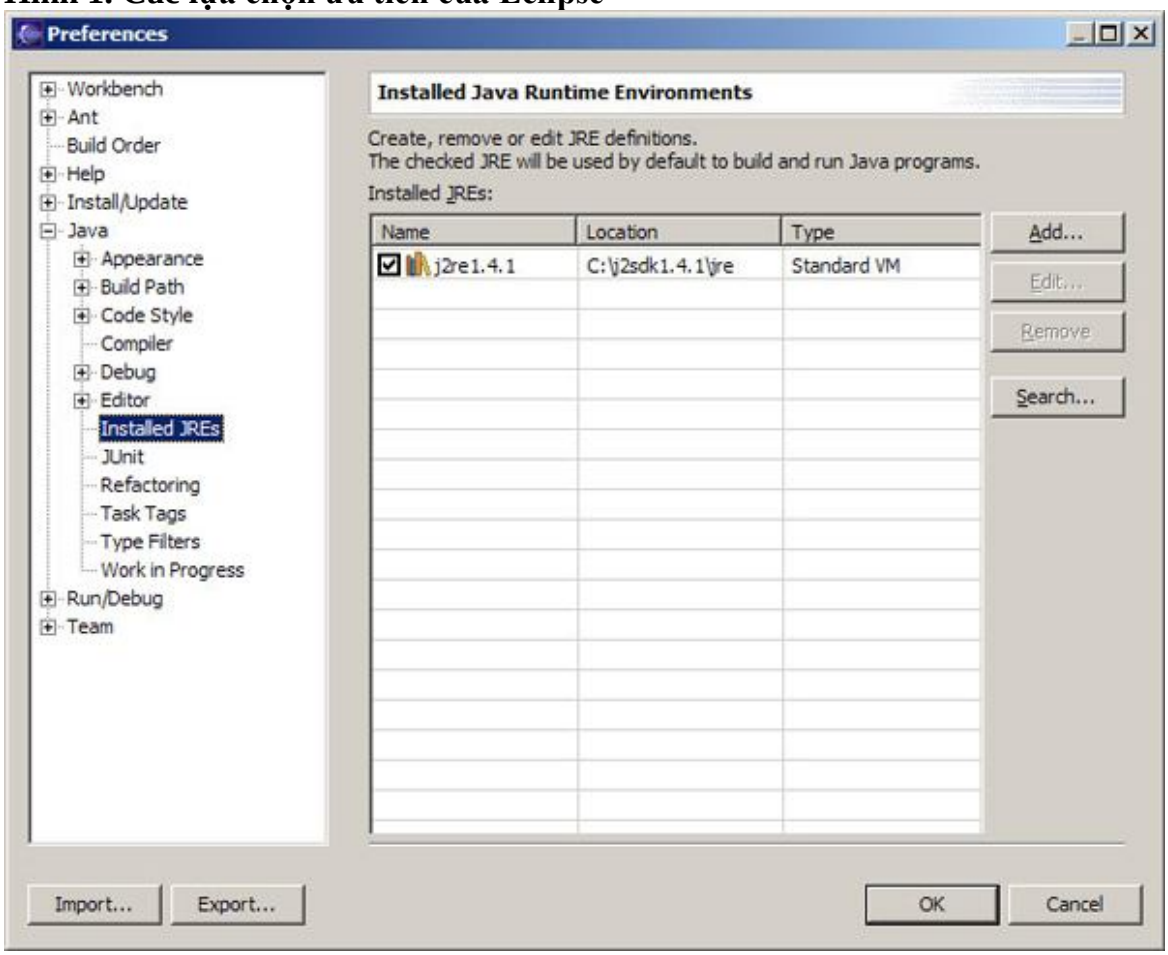
Để dùng Eclipse viết mã Java, bạn phải cho Eclipse biết vị trí Java ở đâu trên máy của bạn. Hãy làm theo các bước sau:

1. Khởi chạy Eclipse bằng cách nhấn đúp chuột vào tệp eclipse.exe, hoặc tệp chạy thi hành tương đương trên hệ điều hành của bạn.

2. Khi màn hình Welcome xuất hiện, nhấn đường liên kết **Go To The Workbench**. Thao tác này sẽ đưa bạn đến bối cảnh tài nguyên (sẽ đề cập chi tiết sau).

3. Nhấn chọn **Window>Preferences>Installed JREs,** thao tác này cho phép bạn chỉ rõ vị trí nơi môi trường Java đã được cài đặt vào máy bạn (xem hình 1).

Hình 1. Các lựa chọn ưu tiên của Eclipse



4. Nhiều khả năng là Eclipse sẽ tìm thấy JRE đã cài đặt sẵn, nhưng bạn nên chỉ rõ ràng đến JRE mà bạn đã cài trong mục Cài đặt Java SDK. Bạn có thể làm điều đó trong hộp thoại Preferences (các lựa chọn ưu tiên). Nếu Eclipse liệt kê một JRE đang có sẵn, nhấn chuột chọn nó và nhấn **Edit**, trái lại, nhấn **Add.**

5. Chỉ rõ đường dẫn đến thư mục JRE của JDK mà bạn đã cài trong mục Cài đặt Java SDK.

6. Nhấn **OK**.

Bây giờ thì Eclipse đã được thiết đặt để biên dịch và chạy mã lệnh Java. Trong phần tiếp theo chúng ta sẽ đi một vòng xem qua môi trường Eclipse để bạn làm quen với công cụ này.

Một vòng xem qua Eclipse

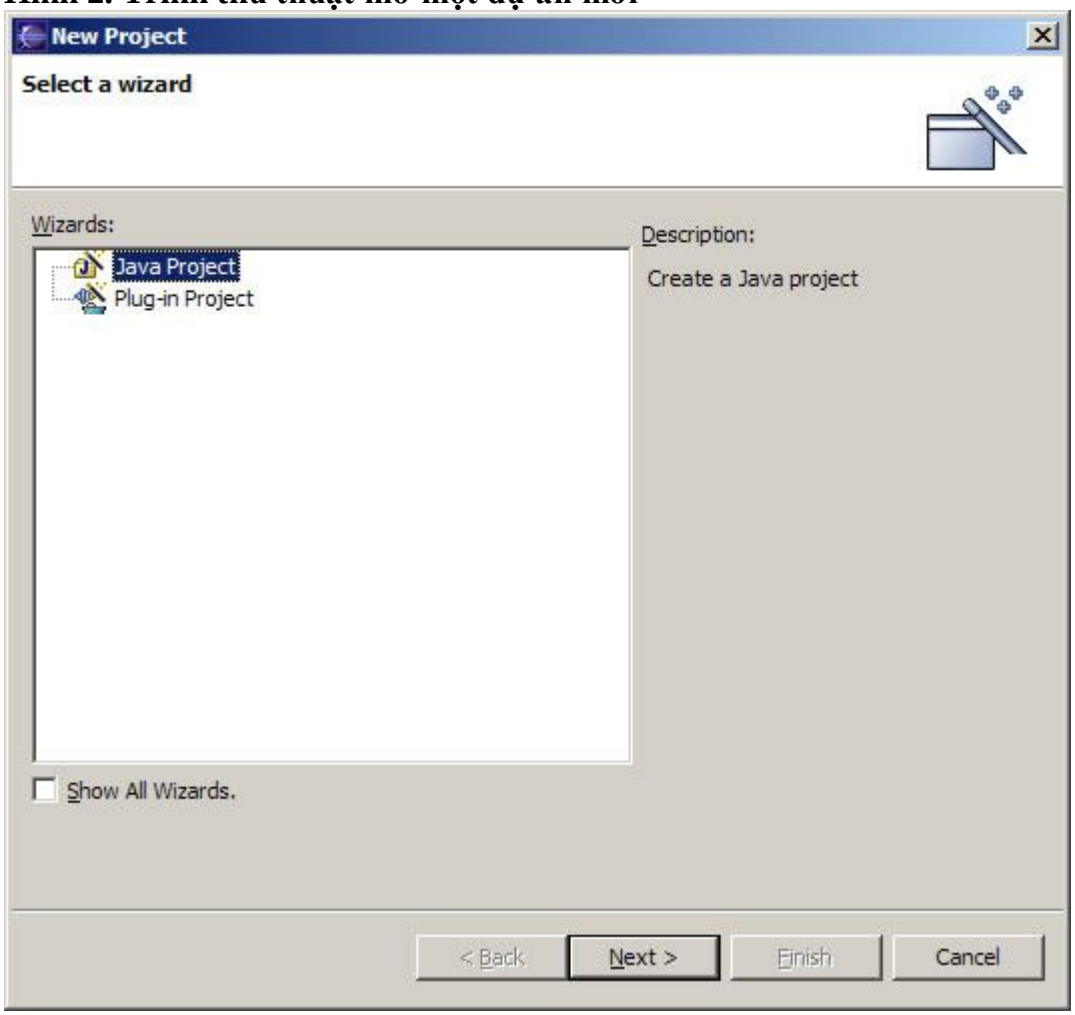
Làm việc với Eclipse là một chủ đề lớn và hầu như vượt ra ngoài phạm vi của tài liệu hướng dẫn này. Xem phần Các tài nguyên để có thêm thông tin về Eclipse. Ở đây, chúng ta sẽ chỉ trình bày vừa đủ để bạn làm quen với cách làm việc của Eclipse và sử dụng nó để lập trình Java như thế nào.

Giả sử bạn vẫn có Eclipse đang chạy, bạn ngừng xem phần phối cảnh Tài nguyên. Eclipse cung cấp một tập *các phối cảnh* trên mã lệnh bạn viết. Phối cảnh Tài nguyên (Resource) sẽ hiển thị một khung nhìn hệ thống tệp tin của bạn trong vùng làm việc (workspace) Eclipse mà bạn đang sử dụng. Một *vùng làm việc* giữ tất cả các tệp liên quan đến việc phát triển trong Eclipse. Bây giờ thì chưa có gì trong vùng làm việc của bạn để bạn phải thực sự quan tâm.

Nhìn chung, Eclipse có các phối cảnh, chứa các khung nhìn. Trong phối cảnh Tài nguyên, bạn sẽ thấy có khung nhìn Navigator, khung nhìn Outline, … Bạn có thể kéo thả tất cả những khung nhìn này vào vị trí bất kỳ bạn muốn. Nó là môi trường tùy biến hầu như vô hạn. Thế nhưng bây giờ, những xếp đặt mặc định cũng đủ tốt. Nhưng những gì chúng ta thấy chưa cho phép ta làm những gì ta muốn. Bước đầu tiên để viết mã lệnh Java trong Eclipse là tạo một dự án Java. Đây không phải là một cấu trúc ngôn ngữ Java; nó đơn giản là một cấu trúc Eclipse giúp bạn tổ chức mã lệnh Java của mình. Làm theo các bước sau để tạo dự án Java:

1. Nhấn chuột vào **File>New>Project** để hiển thị cửa sổ thủ thuật tạo dự án mới (New Project wizard - xem hình 2). Đây thực sự là một trình thủ thuật của thủ thuật, cho phép bạn lựa chọn sử dụng thủ thuật nào bạn muốn (thủ thuật New Project, thủ thuật New File, vân vân).

Hình 2. Trình thủ thuật mở một dự án mới



2. Hãy đảm bảo là bạn chọn thủ thuật Java Project và nhấn **Next**.

3. Nhập bất kỳ tên dự án nào bạn muốn (tên "Intro" có thể là một gợi ý hay), giữ nguyên các lựa chọn mặc định và nhấn **Finish**.

4. Ở bước này, Eclipse sẽ hỏi bạn liệu có nên chuyển sang phối cảnh Java không. Nhấn chọn **No**.

Bạn vừa tạo một dự án Java có tên là Intro, bạn sẽ phải nhìn thấy nó trong khung nhìn Navigator ở góc trên bên trái của màn hình. Chúng ta sẽ không chuyển sang phối cảnh Java sau khi tạo dự án vì có một phối cảnh tốt hơn để dùng cho những mục đích hiện tại của chúng ta. Nhấn nút Open Perspective trên phiếu nằm ở góc trên bên phải của cửa sổ, sau đó chọn phối cảnh Java Browsing. Phối cảnh này sẽ hiển thị cho bạn những gì bạn cần thấy để dễ dàng viết một chương trình Java. Khi viết mã lệnh Java, ta sẽ duyệt qua một vài đặc tính Eclipse nữa để bạn có thể học cách viết, sửa đổi và quản lý mã lệnh của mình. Tuy nhiên, trước khi thực hiện

điều này, chúng ta phải trình bày một vài khái niệm cơ bản về lập trình hướng đối tượng, ta sẽ thực hiện điều này ở phần tiếp theo. Ngay bây giờ, chúng ta sẽ kết thúc phần này bằng cách xem một số tài liệu Java trực tuyến.

API Java trực tuyến

Giao diện lập trình ứng dụng (API) Java rất nhiều, bởi vậy học cách tìm kiếm như thế nào là điều quan trọng. Nền tảng java đủ lớn để cung cấp cho bạn hầu như bất cứ công cụ nào mà bạn cần khi lập trình. Học cách khai thác các khả năng cũng cần nhiều nỗ lực như khi học các cơ chế của ngôn ngữ.

Nếu bạn vào trang tài liệu Java của Sun (xem phần Các tài nguyên để tìm các đường liên kết), bạn sẽ thấy một đường liên kết tới các tài liệu API cho từng phiên bản SDK. Đi theo liên kết dành cho phiên bản 1.4.2 để xem tài liệu đó như thế nào.

Bạn sẽ thấy có 3 khung trên trình duyệt:

- Danh sách các gói có sẵn ở khung trên cùng bên trái
- Danh sách tất cả các lớp ở khung dưới bên trái
- Chi tiết của cái mà bạn đã chọn ở phía bên phải

Tất cả các lớp trong SDK đều có ở đây. Chọn lớp HashMap. Ở bên phải bạn sẽ thấy mô tả chi tiết của lớp này. Phía trên đỉnh là tên và gói chứa lớp này, hệ phân bậc các lớp, các giao diện mà lớp thực hiện (phần này nằm ngoài phạm vi của tài liệu này), và mọi lớp con trực tiếp mà lớp này hiện có. Tiếp sau, bạn sẽ thấy mô tả chi tiết về lớp. Đôi khi trong mô tả còn có cả ví dụ cách dùng, các liên kết có liên quan, gợi ý về kiểu cách, … Sau phần mô tả, bạn sẽ thấy danh sách các hàm tạo (constructors), tiếp đó là danh sách tất cả các phương thức của lớp, tiếp nữa là toàn bộ các phương thức thừa kế, sau đó là mô tả chi tiết của tất cả các phương thức. Thông tin rất đầy đủ và có một bảng chỉ mục chi tiết ở trên và dưới khung bên tay phải.

Nhiều thuật ngữ trong đoạn trên đây (như gói – package) là mới với bạn vào lúc này. Đừng lo lắng. Chúng ta sẽ trình bày chúng một cách chi tiết. Bây giờ, điều quan trọng là bạn đã biết rằng tài liệu về ngôn ngữ Java luôn sẵn sàng trực tuyến cho bạn dùng.

Các khái niệm OOP

Một đối tượng là gì?

Java được biết đến như một ngôn ngữ *hướng đối tượng* (OO - object-oriented), bạn có thể sử dụng ngôn ngữ này để lập trình hướng đối tượng. Điều này rất khác so với lập trình thủ tục, và có thể hơi lạ lùng đối với hầu hết các lập trình viên không hướng đối tượng. Bước đầu tiên bạn phải hiểu đối tượng là gì, vì đó là khái niệm cơ sở của OOP.

Một *đối tượng* là một bó mã lệnh tự thân trọn vẹn (self-contained), tự hiểu chính mình và có thể nói cho các đối tượng khác về chính mình nếu chúng đưa ra các yêu cầu mà nó hiểu được. Một đối tượng có các *thành phần dữ liệu* (các biến) và *các phương thức*, chính là những yêu cầu mà nó biết cách trả lời (dù chúng không được diễn đạt bằng lời như các câu hỏi). Tập các phương thức mà một đối tượng biết cách trả lời được gọi là *giao diện* của đối tượng. Một vài phương thức là mở công cộng, nghĩa là các đối tượng khác có thể gọi đến chúng. Tập các phương thức này được gọi là *giao diện công cộng* của đối tượng.

Khi một đối tượng gọi phương thức của một đối tượng khác, thì được gọi là gửi một thông điệp (*sending a message* hoặc *message send*). Cụm từ này là thuật ngữ của OO nhưng hầu hết trong giới Java mọi người hay nói, "gọi phương thức này" hơn là "gửi thông điệp này". Trong phần tiếp theo, chúng ta sẽ xem xét một ví dụ minh họa khái niệm giúp bạn hiểu vấn đề này rõ ràng hơn.

Ví dụ minh họa khái niệm đối tượng

Giả sử chúng ta có đối tượng Person. Mỗi Person có tên, tuổi, chủng tộc và giới tính. Mỗi Person cũng biết nói và biết đi. Một Person có thể hỏi tuổi của một Person khác, hoặc yêu cầu một Person khác bắt đầu đi (hay dừng). Diễn đạt theo thuật ngữ lập trình, bạn có thể tạo một đối tượng Person và khai báo một số biến (như tên và tuổi). Nếu bạn tạo một đối tượng Person thứ hai, đối tượng này có thể hỏi tuổi của đối tượng thứ nhất hoặc yêu cầu đối tượng thứ nhất bắt đầu đi. Nó có thể thực hiện những điều ấy bằng cách gọi đến các phương thức của đối tượng Person đầu tiên. Khi chúng ta bắt đầu viết mã lệnh bằng ngôn ngữ Java, bạn sẽ hiểu ngôn ngữ này triển khai thực hiện khái niệm đối tượng ra sao.

Nói chung, khái niệm đối tượng là như nhau trong ngôn ngữ Java và các ngôn ngữ hướng đối tượng khác, mặc dù việc triển khai thực hiện là khác nhau giữa các ngôn ngữ. Các khái niệm là phổ quát. Vì sự thật này, lập trình viên hướng đối tượng, bất chấp họ lập trình bằng ngôn ngữ nào, có xu hướng phát biểu khác so với những lập trình viên thủ tục. Các lập trình viên thủ tục thường nói về các hàm và các mô đun. Lập trình viên hướng đối tượng lại nói về các đối tượng và họ

thường nói về các đối tượng này bằng cách sử dụng các đại từ nhân xưng. Chẳng hề bất thường khi bạn nghe một lập trình viên hướng đối tượng nói với đồng nghiệp, "đối tượng Supervisor nói với đối tượng Employee, 'cho tôi ID của cậu,'" vì anh ta cần những thứ này để gán nhiệm vụ cho Employee.

Lập trình viên hướng thủ tục có thể nghĩ cách nói chuyện này thật lạ lùng, nhưng nó lại hoàn toàn bình thường đối với lập trình viên hướng đối tượng. Trong thế giới lập trình của họ, mọi thứ đều là đối tượng (cũng có một vài ngoại lệ đáng chú ý trong ngôn ngữ Java) và các chương trình là chỉ là sự tương tác (hay nói chuyện) giữa các đối tượng với nhau.

Các nguyên tắc hướng đối tượng cơ bản

Khái niệm đối tượng là trọng yếu đối với lập trình hướng đối tượng, và dĩ nhiên, ý tưởng các đối tượng giao tiếp với nhau bằng các thông điệp cũng vậy. Nhưng có 3 nguyên tắc cơ bản mà bạn cần hiểu.

Bạn có thể nhớ 3 nguyên tắc hướng đối tượng cơ bản bằng cụm viết tắt PIE:

- Đa hình (**P**olymorphism)
- Thừa kế (**I**nheritance)
- Bao gói (**E**ncapsulation)

Đó là những từ trừu tượng nhưng những khái niệm này thực sự không quá khó hiểu. Trong các phần tiếp theo, chúng ta sẽ bàn về từng khái niệm này ở mức độ chi tiết hơn, theo thứ tự ngược lại.

Bao gói

Hãy nhớ rằng, một đối tượng là tự thân trọn vẹn, chứa đựng các thành phần dữ liệu và hành động mà nó có thể thực hiện trên các thành phần dữ liệu ấy. Đây là việc triển khai thực hiện nguyên lý gọi là *ẩn giấu thông tin*. Ý tưởng của nó là một đối tượng tự nó hiểu mình. Nếu một đối tượng khác muốn điều gì từ đối tượng này

thì nó phải hỏi. Theo thuật ngữ lập trình hướng đối tượng, phải gửi một thông điệp đến một đối tượng khác để hỏi về tuổi. Theo thuật ngữ Java, phải gọi một phương thức của đối tượng khác để nó trả lại kết quả là tuổi.

Sự bao gói đảm bảo rằng mỗi đối tượng là khác nhau và chương trình là một cuộc chuyện trò giữa các đối tượng. Ngôn ngữ Java cho phép các lập trình viên vi phạm nguyên lý này nhưng hầu như luôn là một ý tưởng tồi nếu làm như thế.

Thừa kế

Khi bạn được sinh ra, nói về khía cạnh sinh học, bạn là tổ hợp DNA của cha mẹ mình. Bạn không hoàn toàn giống ai trong số họ, mà bạn giống cả hai người. OO cũng có nguyên tắc tương tự đối với các đối tượng. Quay lại với đối tượng Person. Ta nhớ lại rằng mỗi người có một chủng tộc. Không phải tất cả các Person đều cùng chủng tộc, nhưng dù sao thì họ cũng có điểm tương tự như nhau chứ? Chắc chắn vậy! Họ chẳng phải ngựa, tinh tinh hay cá voi mà là người. Mọi con người đều có những điểm chung nhất định và điều này giúp phân biệt con người với các loài động vật khác. Nhưng giữa mọi người cũng có khác biệt với nhau. Một đứa trẻ có giống hệt một người trưởng thành không? Không. Đi lại và nói là khác nhau rồi. Nhưng một đứa trẻ thì vẫn chắc chắn là một con người.

Theo ngôn ngữ hướng đối tượng, Person và Baby là các *lớp* sự vật hiện tượng thuộc cùng một *hệ thống phân bậc*, và Baby *thừa kế* các đặc tính và hành vi từ *lớp cha của nó*. Chúng ta có thể nói rằng một Baby cụ thể là một kiểu Person hay Baby thừa kế từ Person. Nhưng không có chiều ngược lại – một Person không nhất thiết phải là một Baby. Mỗi đối tượng Baby là một *cá thể* của lớp Baby và khi chúng ta tạo một đối tượng Baby, chúng ta *cá thể hóa* lớp này. Hãy coi lớp như là khuôn mẫu chung cho các cá thể của lớp đó. Nói chung, đối tượng có thể làm những gì tùy thuộc vào kiểu của đối tượng đó là gì – hoặc nói theo cách khác, đối tượng đó là cá thể của lớp nào. Cả Baby và Adult đều thuộc kiểu Person, nhưng một đối tượng (Adult) có thể có một việc làm (job) còn đối tượng kia (Baby) thì không.

Theo thuật ngữ Java, Person là một *lớp bậc trên* (superclass) của Baby và Adult, và các lớp này là *lớp con* của Person. Một khái niệm có liên quan khác là ý tưởng về *trừu tượng hóa*. Person có mức trừu tượng hóa cao hơn Baby hay Adult. Cả hai đều là kiểu Person nhưng có những khác biệt nho nhỏ. Tất cả các đối tượng Person đều có những điểm chung (như tên và tuổi). Bạn có thể cá thể hóa một Person? Thực sự là không! Bạn hoặc có một Baby hoặc có một Adult. Trong Java, Person

được gọi là *lớp trừu tượng*. Bạn không thể trực tiếp có một cá thể của lớp Person. Bạn sẽ có Baby hoặc Adult, cả hai đều là kiểu Person, nhưng là Person đã được thực tế hóa. Các lớp trừu tượng nằm ngoài phạm vi của tài liệu này, chúng tôi sẽ không nói thêm về chúng nữa.

Bây giờ, ta hãy suy nghĩ xem với một Baby, "nói" (speak) có nghĩa là gì. Chúng ta sẽ xét đến các hệ quả trong phần thảo luận tiếp theo.

Đa hình

Baby có "nói" như Adult không? Dĩ nhiên là không rồi. Một Baby có thể ê a, nhưng không nhất thiết nói ra những lời hiểu được như Adult. Do đó, nếu tôi cá thể hóa một đối tượng Baby (hay là "cá thể hóa một Baby" cũng có cùng ý nghhĩa – từ "đối tượng" được ngầm hiểu) và cho nó nói, thì nó chỉ có nghĩa là những tiếng ê a. Ta hy vọng rằng Adult "nói" thì mạch lạc hơn.

Trong hệ thống phân bậc con người, chúng ta có Person nằm ở đỉnh với Baby và Adult nằm phía dưới nó, là các lớp con. Tất cả mọi người đều có thể nói, Baby và Adult cũng vậy, nhưng sẽ nói khác nhau. Baby chỉ ê a và phát những âm thanh đơn giản. Adult nói thành lời. Đó chính là *sự đa hình*: các đối tượng làm việc theo cách riêng của chúng.

Ngôn ngữ Java là (và không là) OO ở chỗ nào?

Như chúng ta sẽ thấy, ngôn ngữ Java cho phép bạn tạo các đối tượng hạng nhất (first-class), nhưng không phải bất cứ cái gì trong ngôn ngữ này đều là đối tượng. Một số ngôn ngữ OO như Smalltalk lại hoàn toàn khác. Smalltalk hoàn toàn là OO, có nghĩa là mọi thứ trong ngôn ngữ này đều là đối tượng. Java là ngôn ngữ lai tạp giữa đối tượng và phi đối tượng. Nó cho phép một đối tượng biết rõ các đối tượng khác, nếu với tư cách là một lập trình viên bạn cho phép điều đó xảy ra. Điều này vi phạm nguyên lý bao gói.

Tuy nhiên, ngôn ngữ Java cũng cung cấp cho tất cả các lập trình viên OO những công cụ cần thiết để tuân theo mọi quy tắc OO và viết mã lệnh OO rất chuẩn.

Nhưng làm được như vậy cần phải tự có kỷ luật. Ngôn ngữ không ép bạn làm việc đúng đắn được.

Trong khi những người thuần túy chủ nghĩa hướng đối tượng tranh luận xem liệu Java là hướng đối tượng hay không, thực sự đây không phải là một lý lẽ mang lại ích lợi. Nền tảng Java sẽ giữ vững vị trí của nó. Hãy học cách lập trình hướng đối tượng tốt nhất có thể với mã lệnh Java và cứ để những lý lẽ thuần túy chủ nghĩa cho những người khác. Ngôn ngữ Java giúp bạn viết chương trình rõ ràng, khá ngắn gọn, dễ bảo trì, điều này là khá đủ trong cuốn sách của tôi đối với hầu hết các tình huống nghề nghiệp.

Ngôn ngữ Java bên dưới cái vỏ ngoài

Nền tảng Java hoạt động như thế nào

Khi bạn viết mã lệnh bằng ngôn ngữ Java, giống như nhiều ngôn ngữ khác, bạn viết *mã nguồn*, sau đó bạn biên dịch nó; trình biên dịch kiểm tra mã lệnh của bạn và đối chiếu với các quy tắc cú pháp của ngôn ngữ. Nhưng nền tảng Java bổ sung thêm một bước khác nữa ngoài các bước trên. Khi bạn biên dịch mã Java, bạn sẽ nhận được kết quả là *mã byte* (bytecodes). Sau đó, máy ảo Java (JVM) sẽ thông dịch các mã byte này lúc chạy thi hành– đó là khi bạn yêu cầu Java chạy chương trình.

Theo thuật ngữ hệ thống tệp, khi bạn viết mã, bạn sinh ra một tệp .java. Khi bạn biên dịch tệp này, trình biên dịch của Java sinh ra một tệp .class, chứa các mã byte. JVM đọc và thông dịch tệp .class này lúc chạy thi hành và nó hoạt động như thế nào là tùy thuộc vào nền hệ thống mà bạn đang chạy. Để chạy trên các nền hệ thống khác nhau, bạn phải biên dịch mã nguồn của mình đối với các thư viện dành riêng cho nền hệ thống đó. Bạn có thể hình dung, lời hứa hẹn "viết một lần, chạy mọi nơi" sẽ trở thành "viết một lần, kiểm thử mọi nơi". Đó là có những sự khác biệt mong manh (hay không mong manh cho lắm) giữa các nền hệ thống, có thể khiến cho mã lệnh của bạn hành xử khác nhau trên những nền tảng khác nhau.

Thu dọn rác

Khi bạn tạo các đối tượng Java, JRE sẽ tự động cấp phát không gian bộ nhớ cho các đối tượng này từ *heap*, đây là vùng bộ nhớ lớn có sẵn để cấp trong máy tính của bạn. Quá trình chạy thi hành sẽ theo vết của những đối tượng này giùm bạn. Khi chương trình của bạn không sử dụng các đối tượng đó nữa thì JRE sẽ vứt bỏ chúng. Bạn không phải để tâm đến chúng nữa.

Nếu bạn đã từng viết bất cứ phần mềm nào bằng ngôn ngữ C++, cũng là một ngôn ngữ hướng đối tượng (người ta cho rằng thế), với tư cách là lập trình viên, bạn phải cấp phát và lấy lại bộ nhớ dành cho đối tượng mình tạo ra một cách tường minh bằng cách sử dụng các hàm malloc() và free(). Điều đó đối với các lập trình viên thật là phiền hà. Nó cũng nguy hiểm nữa, vì nó mở đường cho việc thất thoát bộ nhớ len lỏi vào trong chương trình của bạn. Thất thoát bộ nhớ gây ra việc chương trình của bạn ngốn bộ nhớ với tốc độ phát hoảng, điều này gây sức ép lên bộ vi xử lý của máy tính đang chạy chương trình. Nền tảng Java giúp bạn loại bỏ nỗi lo về tất cả những vấn đề đó vì nó có *thành phần thu dọn rác*.

Bộ thu dọn rác của Java là một tiến trình nền phía sau để loại các đối tượng không còn được dùng tới nữa, chứ không buộc bạn phải tường minh làm điều đó. Máy tính rất thích hợp trong việc lưu giữ vết của hàng ngàn thứ và cấp phát tài nguyên.

Nền tảng Java giúp cho phép máy tính của bạn thực hiện điều đó. Nó duy trì số đếm các tham chiếu đang dùng đến mọi đối tượng trong bộ nhớ. Khi con số này chạm mức 0, bộ thu dọn rác sẽ lấy lại vùng bộ nhớ mà đối tượng ấy đã sử dụng. Bạn có thể trực tiếp gọi bộ thu dọn rác, nhưng tôi không bao giờ phải làm điều đó. Nó thường tự xử lý và tất nhiên là cũng sẽ tự xử lý trong mọi mã ví dụ trong tài liệu này.

IDE so với các công cụ dòng lệnh

Như chúng ta đã lưu ý trước đây, nền tảng Java đi kèm với các công cụ dòng lệnh cho phép bạn biên dịch (javac) và chạy (java) các chương trình Java. Vậy tại sao ta lại sử dụng một IDE như Eclipse? Đơn giản chỉ vì việc sử dụng các công cụ dòng lệnh có thể rất phiền phức, bất kỳ chương trình có độ phức tạp như thế nào. Các công cụ dòng lệnh có sẵn nếu bạn cần đến chúng, nhưng sử dụng một IDE thường là lựa chọn khôn ngoan hơn.

Lý do chính của khẳng định này là IDE quản lý tệp và đường dẫn giúp bạn, và có các trình hướng dẫn tương tác để hỗ trợ bạn khi bạn muốn thay đổi môi trường chạy thi hành của mình. Khi tôi muốn biên dịch một chương trình Java bằng công dụ dòng lệnh javac, tôi phải lo việc thiết đặt biến môi trường CLASSPATH từ lúc đầu để JRE có thể biết nơi đặt các lớp của tôi, hoặc tôi phải thiết đặt giá trị cho biến này lúc biên dịch. Trong một IDE như Eclipse, tất cả những gì tôi phải làm là cho Eclipse biết tìm JRE ở đâu. Nếu mã lệnh của tôi dùng các lớp không do tôi viết ra, tất cả những gì tôi phải làm là cho Eclipse biết những thư viện mà dự án của tôi tham chiếu đến là gì và tìm chúng ở đâu. Điều này đơn giản hơn nhiều so với việc dùng dòng lệnh để gõ những câu lệnh dài đến phát khiếp để chỉ rõ đường dẫn đến lớp.

Nếu bạn muốn hay cần dùng các công cụ dòng lệnh, bạn có thể tìm thấy cách sử dụng chúng ở trang Web về công nghệ Java của Sun (xem Các tài nguyên).

Lập trình hướng đối tượng với công nghệ Java

Giới thiệu

Công nghệ Java bao trùm nhiều thứ, nhưng bản thân ngôn ngữ Java lại không lớn lắm. Tuy nhiên, trình bày nó bằng ngôn ngữ thông thường, lại không phải là nhiệm vụ đơn giản. Phần này sẽ không bàn kỹ về ngôn ngữ này. Thay vào đó, sẽ nêu những gì bạn cần biết để khởi đầu và những gì bạn hầu như chắc chắn sẽ gặp với tư cách là lập trình viên mới vào nghề. Các tài liệu hướng dẫn khác (xem Các tài nguyên để nhận được những gợi ý về một số tài liệu này) sẽ trình bày các khía cạnh khác nhau của ngôn ngữ này, các thư viện hữu ích bổ trợ do Sun cung cấp, các tài nguyên khác và thậm chí cả các IDE.

Chúng tôi sẽ trình bày đầy đủ ở đây cả lời dẫn giải và các ví dụ mã lệnh để bạn có thể bắt đầu viết các chương trình Java và học cách lập trình hướng đối tượng đúng đắn trong môi trường Java. Từ đó, vấn đề chỉ còn là việc thực hành và học tập.

Hầu hết các tài liệu hướng dẫn nhập môn đọc lên giống như những cuốn sách tham khảo đặc tả ngôn ngữ. Đầu tiên bạn thấy tất cả các quy tắc cú pháp, sau đó bạn xem các ví dụ áp dụng, tiếp đó là nói về những chủ đề nâng cao hơn, như các đối tượng chẳng hạn. Ở đây chúng tôi sẽ không đi theo con đường đó. Đó là vì nguyên nhân chính dẫn đến các mã lệnh hướng đối tượng tồi tệ viết bằng ngôn ngữ Java là những lập trình viên mới vào nghề không đặt mình vào môi trường hướng đối tượng ngay từ đầu. Các đối tượng có khuynh hướng bị đối xử như một chủ đề phụ thêm (add-on) hay lệ thuộc. Thay vào đó, chúng tôi sẽ đan xen việc học cú pháp Java thông qua quá trình học Java hướng đối tượng. Bằng cách này, bạn sẽ có được một bức tranh mạch lạc về việc sử dụng ngôn ngữ ra sao trong bối cảnh hướng đối tượng.

Cấu trúc của một đối tượng Java

Hãy nhớ rằng, đối tượng là một thứ *được bao gói kín*, tự biết mọi điều về mình và có thể làm một số việc khi được yêu cầu thích hợp. Mọi ngôn ngữ đều có quy tắc định nghĩa một đối tượng như thế nào. Trong ngôn ngữ Java, đối tượng nói chung nom giống như liệt kê dưới đây, mặc dù chúng có thể thiếu hụt một số thành phần:

package *packageName*;

```
import  packageNameToImport;


 accessSpecifier class  ClassName {
            accessSpecifier
                dataType
                variableName [=  initialValue ];
            ...

            accessSpecifier ClassName(  arguments ) {
                        constructor statement(s)
            }

            accessSpecifier
                returnValueDataType
                methodName (  arguments ) {
                        statement(s)
            }
}
```

Ở đây có một số khái niệm mới mà chúng ta sẽ thảo luận trong vài phần tiếp sau.

Các gói

Khai báo gói phải xuất hiện đầu tiên khi bạn định nghĩa một lớp:

package *packageName*;

Mọi đối tượng Java đều nằm trong một package. Nếu bạn không nói rõ ràng nó thuộc gói nào, Java sẽ đặt nó vào trong *gói mặc định*. Một package chỉ đơn giản là một tập các đối tượng, tất cả (thường là thế) liên quan với nhau theo một cách nào đó. Các package quy chiếu theo đường dẫn đến tệp tin trong hệ thống tệp của bạn. Tên của các gói dùng ký pháp dấu chấm (.) để dịch đường dẫn tệp tin này thành một thứ mà nền tảng Java hiểu được. Mỗi mẩu trong tên package gọi là một *nút* (node).

Ví dụ, trong gói có tên là java.util.ArrayList, java là một nút, util là một nút và ArrayList là một nút. Nút cuối cùng trỏ đến tệp ArrayList.java.

Các câu lệnh nhập khẩu

Tiếp theo là các câu lệnh *nhập khẩu*, khi bạn định nghĩa một lớp:

import *packageNameToImport*;

...

S

Khi đối tượng của bạn sử dụng các đối tượng trong các gói khác, trình biên dịch của Java cần biết tìm chúng ở đâu. Một lệnh *nhập khẩu* (import) sẽ cho trình biên dịch biết nơi tìm những lớp bạn cần dùng. Ví dụ, nếu bạn muốn dùng lớp ArrayList từ gói java.util, bạn cần nhập khẩu theo cách sau:

```
import java.util.ArrayList;
```

Mỗi lệnh import kết thúc bằng một dấu chấm phẩy (;), giống như hầu hết các câu lệnh trong ngôn ngữ Java. Bạn có thể viết bao nhiêu câu lệnh nhập khẩu cũng được khi bạn cần cho Java biết tìm tất cả các lớp mà bạn dùng ở đâu. Ví dụ, nếu tôi muốn dùng lớp ArrayList từ gói java.util, lớp BigInteger từ gói java.math, tôi sẽ nhập khẩu chúng như sau:

```
import java.util.ArrayList;

import java.math.BigInteger;
```

Nếu bạn nhập khẩu nhiều hơn một lớp từ cùng một gói, bạn có thể dùng cách viết tắt để cho biết bạn muốn nạp tất cả các lớp trong gói này. Ví dụ, nếu tôi muốn dùng cả ArrayList và HashMap, cả hai đều từ gói java.util, tôi sẽ nhập khẩu chúng như sau:

```
import java.util.*;
```

Bạn muốn nhập khẩu gói nào thì phải có lệnh nhập khẩu riêng cho gói đó.

Khai báo một lớp

Tiếp theo là *khai báo lớp*, khi bạn định nghĩa một lớp:

```
        accessSpecifier class  ClassName {
    accessSpecifier
        dataType
        variableName [=  initialValue ];
    ...

    accessSpecifier ClassName(  arguments ) {
            constructor statement(s)
    }

    accessSpecifier
        returnValueDataType
        methodName (  arguments ) {
            statement(s)
    }
}
```

Bạn định nghĩa một đối tượng Java như một lớp. Hãy nghĩ rằng lớp là khuôn mẫu của đối tượng, một thứ gì đó giống như máy cắt bánh quy vậy. Lớp định nghĩa kiểu của đối tượng bạn có thể tạo ra từ lớp. Bạn có thể dập khuôn ra bao nhiêu đối tượng thuộc cùng kiểu đó như bạn muốn. Khi bạn làm thế, bạn đã tạo ra một cá thể của lớp – hoặc nói theo cách khác là bạn đã cụ thể hóa một đối tượng. (Chú ý: từ đối tượng được dùng hoán đổi lẫn lộn để chỉ một lớp lẫn một cá thể của lớp.)

Định tố truy cập (access specifier) của một lớp có thể có nhiều giá trị, nhưng hầu hết đều là public (công cộng), và đó cũng là tất cả những gì chúng ta sẽ nói tới trong tài liệu hướng dẫn này. Bạn có thể đặt một cái tên bất kỳ nào bạn thích cho một lớp, nhưng tên của lớp theo quy ước bắt đầu bằng bằng một chữ cái viết hoa, và mỗi từ tiếp theo trong tên cũng bắt đầu bằng một chữ cái viết hoa.

Lớp có hai kiểu thành phần: *các biến* (hay thành phần dữ liệu) và *các phương thức*. Tất cả các thành phần của lớp đều được định nghĩa trong *thân lớp*, nằm giữa *cặp ngoặc nhọn* của lớp.

Các biến

Giá trị của các biến trong một lớp là cái để phân biệt từng cá thể của lớp, đó là lý do vì sao chúng thường được gọi là các *biến cá thể*. Một biến có một định tố truy cập chỉ rõ những đối tượng nào được phép *truy cập nó, một kiểu dữ liệu, một tên và (tùy chọn) một giá trị khởi tạo*. Đây là danh sách các định tố truy cập và ý nghĩa của chúng:

- public (công cộng): Bất kỳ đối tượng nào trong bất kỳ gói nào đều có thể thấy biến này.
- protected (có bảo vệ): Bất kỳ một cá thể nào của lớp, lớp con trong cùng một gói và bất kỳ lớp nào không phải là lớp con nhưng nằm trong cùng một gói có thể thấy biến này. Lớp con trong các gói khác không thể thấy nó.
- private (riêng tư): Không một đối tượng nào ngoài cá thể cụ thể của lớp có thể thấy được biến, thậm chí cả lớp con.
- Không có định tố, (hoặc *package protected* (có bảo vệ theo gói)): Chỉ có các lớp trong cùng một gói với lớp chứa biến là có thể thấy biến mà thôi.

Nếu bạn cố truy cập một biến không thể truy cập được, trình biên dịch sẽ thông báo biến đó là không nhìn thấy đối với bạn. Bạn sẽ dùng định tố truy cập gì trong những hoàn cảnh nào là nhờ vào óc suy xét, và chúng ta sẽ quay trở lại vấn đề này sau.

Các phương thức

Các phương thức của một lớp định nghĩa lớp có thể làm những gì. Có hai loại phương thức trong ngôn ngữ Java:

- Hàm tạo
- Các phương thức khác

Cả hai đều có định tố truy cập (để chỉ ra những đối tượng nào có thể sử dụng chúng) và phần thân (giữa cặp ngoặc nhọn), có chứa một hay nhiều câu lệnh. Ngoài điều này ra, khuôn dạng và chức năng của chúng rất khác nhau. Chúng ta sẽ đề cập đến từng loại phương thức này ở hai phần tiếp sau.

Hàm tạo

Các hàm tạo cho phép bạn chỉ rõ cách cá thể hóa một lớp. Bạn khai báo một hàm tạo như sau:

accessSpecifier ClassName(*arguments*) {

constructor statement(s)

}

Bạn nhận được sẵn một *hàm tạo mặc định* (không có tham số truyền vào) cho mọi lớp mà bạn tạo ra mà không phải làm gì. Thậm chí bạn không phải định nghĩa nó. Các hàm tạo trông khác với các phương thức khác ở chỗ chúng không có kiểu dữ liệu của giá trị trả về. Đó là vì kiểu dữ liệu giá trị trả lại của nó chính là lớp đó. Bạn viết mã lệnh gọi một hàm tạo như sau:

ClassName variableHoldingAnInstanceOfClassName = new ClassName(*arguments*);

Khi bạn gọi một hàm tạo, bạn dùng từ khóa new. Các hàm tạo có thể nhận tham số truyền vào hoặc không (hàm tạo mặc định không có tham số vào). Nghiêm túc mà nói, các hàm tạo không phải là phương thức hay thành viên của lớp. Chúng là một sinh thể đặc biệt trong ngôn ngữ Java. Tuy vậy, trong thực tế, vẻ bề ngoài và hoạt động của chúng nhiều lúc cũng giống các phương thức khác, và nhiều người gộp cả hai vào với nhau. Hãy ghi nhớ rằng chúng là đặc biệt.

Các phương thức không là hàm tạo

Các phương thức không là hàm tạo trong ngôn ngữ Java là thứ mà bạn thường sử dụng nhất. Bạn khai báo chúng như sau:

accessSpecifier

returnValueDataType

methodName (*arguments*) {

statement(s)

}

Tất cả các phương thức đều có kiểu trả về, nhưng không phải mọi phương thức đều trả lại thứ gì đó. Nếu phương thức không trả lại gì, bạn dùng từ khóa void để chỉ kiểu trả về. Bạn có thể đặt bất cứ tên gì mà bạn thích cho phương thức miễn là cái tên đó hợp lệ (không được khởi đầu bằng dấu chấm, ví dụ thế), nhưng theo quy ước thì tên phương thức là:

- Là xâu ký tự các chữ cái
- Bắt đầu bằng một ký tự viết thường
- Bắt đầu các từ tiếp theo bằng ký tự viết hoa.

Bạn *gọi* một phương thức như sau:

```
returnType variableForReturnValue =

    instanceOfSomeClass.methodName(parameter1, parameter2, ...);
```

Như vậy, bạn đang gọi phương thức methodName() của đối tượng instanceOfSomeClass, và truyền cho phương thức này một vài đối số. Sự khác biệt giữa tham số và đối số là không nhiều, nhưng chúng khác nhau. Phương thức nhận các tham số. Khi bạn truyền giá trị cụ thể vào phương thức lúc bạn gọi thì những giá trị này được gọi là đối số của lời gọi.

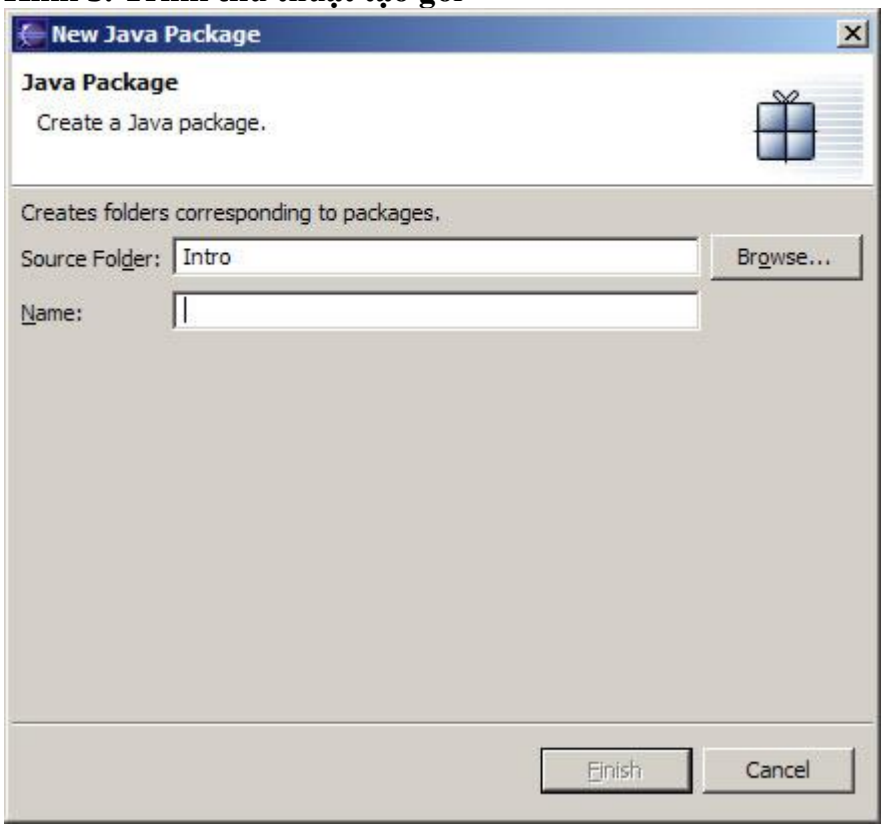
Đối tượng Java đầu tiên của bạn

Tạo một gói

Nếu bạn còn chưa ở đó, hãy di chuyển đến phối cảnh Duyệt Java (Java Browsing) trong Eclipse. Chúng ta sẽ thiết đặt để tạo lớp Java đầu tiên của bạn. Bước thứ nhất là tạo một nơi cho lớp này tồn tại.

Thay vì dùng gói mặc định, tốt hơn là ta hãy tạo một gói riêng cho dự án Intro của mình. Nhấn chuột chọn **File>New>Package**. Thao tác này sẽ mở Trình hướng dẫn tương tác Package (xem hình 3)

Hình 3. Trình thủ thuật tạo gói



Gõ nhập **intro.core** là tên đặt cho gói và nhấn **Finish**. Bạn có thể thấy gói sau đây trong khung nhìn Packages trong vùng làm việc:

intro.core

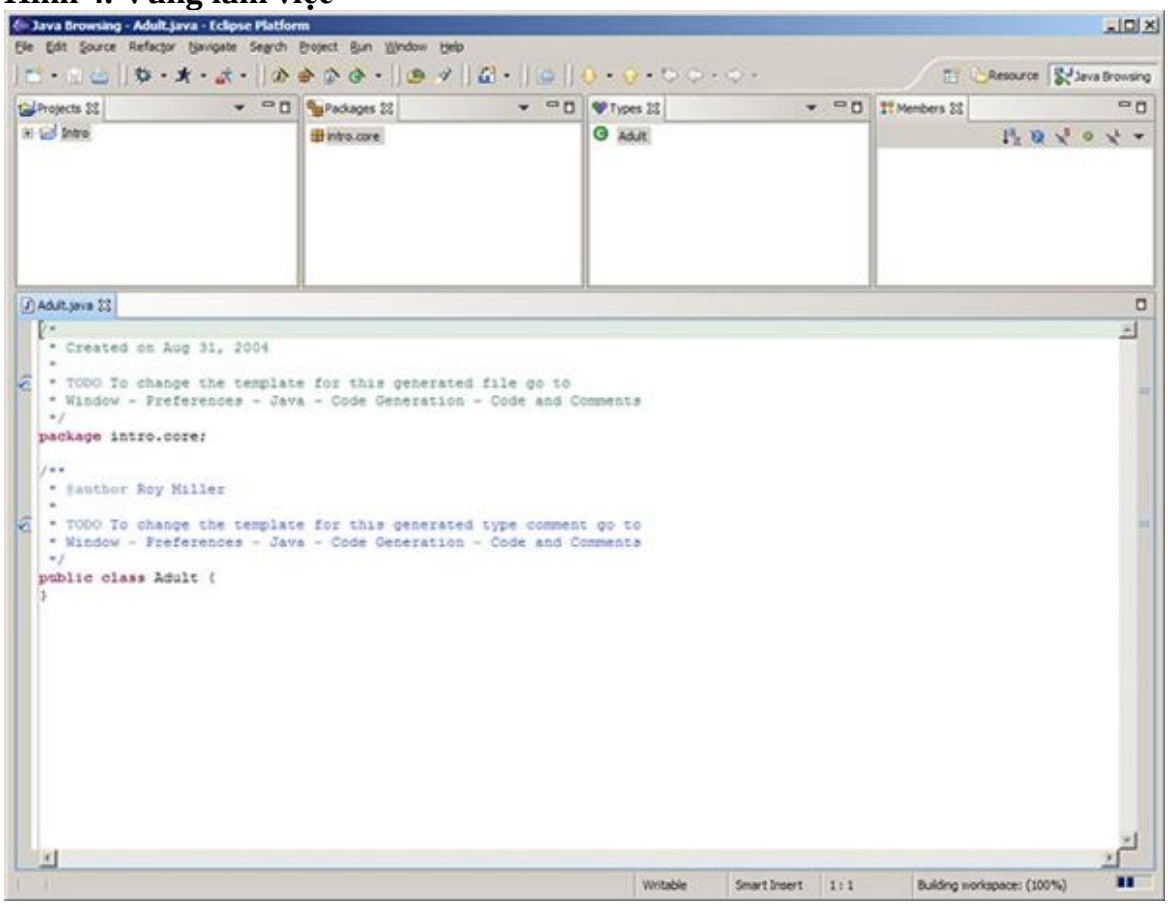
Lưu ý rằng biểu tượng ở bên trái gói trông mờ mờ -- nhìn giống như một dạng mờ xám đi của biểu tượng gói. Đó là quy ước chung của giao diện người dùng trong Eclipse đối với những hạng mục trống rỗng. Gói của bạn chưa có bất cứ lớp Java nào bên trong nên biểu tượng của nó bị mờ đi.

Khai báo một lớp

Bạn có thể tạo một lớp Java trong Eclipse bằng cách chọn **File>New,**, nhưng chúng ta sẽ dùng công cụ thay thế. Nhìn lên phía trên khung nhìn Packages để thấy thanh công cụ tạo các dự án, gói và lớp. Nhấn chuột chọn công cụ New Java Class (chữ "C" màu xanh lục) để mở trình hướng dẫn tương tác New Java Class. Đặt tên lớp là **Adult** và chấp nhận các mặc định bằng cách nhấp **Finish.** Bạn sẽ thấy có một vài thay đổi:

- Lớp Adult xuất hiện trên khung nhìn Classes, phía bên phải của khung nhìn Packages (xem hình 4).

Hình 4. Vùng làm việc



- biểu tượng gói intro.core không còn bị mờ nữa.
- Trình soạn thảo của Adult.java hiển thị bên dưới.

Tại thời điểm này, lớp trông như sau:

```
package intro.core;


public class Adult {

}
```

Eclipse sinh ra một cái vỏ hay một khung mẫu cho lớp của bạn, và bao gồm luôn câu lệnh package nằm ở ngay đầu. Thân của lớp hiện giờ là rỗng. Đơn giản là chúng ta phải thêm da thịt vào. Bạn có thể cấu hình khung mẫu cho các lớp mới, phương thức mới, vân vân trong trình hướng dẫn tương tác Preferences mà bạn đã từng dùng trước đây (**Window>Preferences**). Bạn có thể cấu hình khuôn mẫu cho mã lệnh trong đường dẫn đến các lựa chọn ưu tiên **Java>Code Style>Code Templates**. Thực tế, để đơn giản mọi thứ trong việc hiển thị mã lệnh, từ đây trở đi tôi đã loại bỏ tất cả các chú thích khỏi các khuôn mẫu này – nghĩa là bất kỳ dòng nào có dấu // ở trước hoặc bất kỳ dòng nào nằm trong cặp /* … */ hoặc cặp /** … **/. Từ bây giờ trở đi, bạn sẽ không nhìn thấy bất kỳ dòng chú thích nào trong mã lệnh nữa, trừ khi chúng ta đặc biệt thảo luận về cách dùng chúng ở phần sau.

Tuy nhiên, trước khi chúng ta tiếp tục, ta hãy minh họa cách thức mà IDE Eclipse làm cho công việc của bạn dễ dàng hơn. Trong trình soạn thảo, hãy sửa đổi từ **class** thành **clas** và chờ vài giây. Bạn nhận thấy là Eclipse sẽ gạch chân chữ này bằng một đường lượn sóng màu đỏ. Nếu bạn di chuột qua một mục được gạch chân, Eclipse sẽ bật một cửa sổ thông tin để báo cho bạn biết bạn đang mắc lỗi cú pháp. Eclipse giúp bạn bằng cách liên tục biên dịch mã lệnh của bạn và kín đáo cảnh báo bạn nếu có vấn đề xảy ra. Nếu bạn dùng công cụ dòng lệnh javac, bạn sẽ phải biên dịch mã lệnh và chờ xem lỗi. Điều đó có thể làm chậm việc phát triển chương trình của bạn. Eclipse loại bỏ rắc rối đó.

Các chú thích

Giống như hầu hết các ngôn ngữ khác, ngôn ngữ Java hỗ trợ các chú thích, mà đơn giản là các câu lệnh mà trình biên dịch sẽ bỏ qua khi nó kiểm tra cú pháp xem có đúng không. Java có nhiều kiểu chú thích:

// *Chú thích một dòng. Trình biên dịch bỏ qua đoạn văn bản đi sau cặp dấu gạch xiên.*

/* *Chú thích nhiều dòng. Trình biên dịch bỏ qua đoạn văn bản nằm giữa hai dấu sao.* */

/** *Chú thích javadoc. Trình biên dịch bỏ qua đoạn văn bản nằm giữa hai dấu sao,*

*và công cụ javadoc sẽ sử dụng chúng. */*

Kiểu cuối cùng là đáng chú ý nhất. Nói rất ngắn gọn, công cụ javadoc đi kèm với bản phân phối SDK Java mà bạn đã cài đặt có thể giúp bạn sinh ra tài liệu HTML cho mã lệnh của bạn. Bạn có thể sinh ra tài liệu cho các lớp riêng của mình và chúng sẽ trông tương tự như những gì bạn thấy trong tài liệu API Java mà bạn đọc trong The Java API online. Một khi bạn đã chú thích hợp lý cho mã lệnh của mình, bạn có thể chạy lệnh javadoc từ dòng lệnh. Bạn có thể tìm thấy chỉ dẫn thực hiện việc này, và tất cả thông tin đã có sẵn về javadoc trong trang web về công nghệ Java (xem Các tài nguyên).

Các từ dành riêng

Còn một mục nữa cần đề cập trước khi chúng ta bắt đầu viết mã lệnh để bộ biên dịch kiểm tra. Java có một số từ mà bạn không được dùng để đặt tên cho các biến. Sau đây là danh sách các từ này:

abstract	boolean	break	byte
case	catch	char	class
const	continue	char	class
default	do	double	else
extends	false	final	finally
float	for	goto	if

implements	import	int	instanceof
interface	long	int	native
new	null	package	private
protected	public	package	private
static	strictfp	super	switch
synchronized	short	super	this
throw	throws	true	try
transient	return	void	volatile
while	assert	true	false
null			

Danh sách này không dài, nhưng Eclipse sẽ tô đậm các từ này khi bạn gõ, bởi vậy bạn không phải ghi nhớ chúng. Tất cả, trừ ba từ cuối, trong danh sách là *từ khóa* Java. Ba từ này là các *từ dành riêng*. Sự khác biệt không quan trọng đối với những mục tiêu của chúng ta; bạn không thể sử dụng cả hai loại.

Giờ ta hãy xem một số mã lệnh thực sự.

Thêm các biến

Như tôi trước đây đã nói, Adult biết tên, tuổi, chủng tộc và giới tính của nó. Chúng ta có thể bổ sung thêm các mẩu dữ liệu ấy cho lớp Adult của chúng ta bằng cách khai báo chúng dưới dạng các biến. Sau đó thì mọi cá thể của lớp Adult sẽ có các biến này. Mỗi Adult sẽ có giá trị khác nhau cho các biến này. Đó là lý do tại sao các biến của mỗi đối tượng lại thường được gọi là *biến cá thể* - chúng là cái để phân biệt mỗi cá thể của lớp. Ta hãy thêm biến, dùng định tố truy nhập protected cho mỗi biến:

```
package intro.core;



public class Adult {

        protected int age;

        protected String name;

        protected String race;

        protected String gender;

}
```

Bây giờ thì mọi cá thể của Adult đều sẽ chứa những mẩu dữ liệu này. Lưu ý ở đây mỗi dòng mã lệnh kết thúc bằng dấu chấm phẩy (;). Ngôn ngữ Java yêu cầu điều đó. Cũng lưu ý thêm là mỗi biến có một kiểu dữ liệu. Ta có một biến kiểu số nguyên và ba biến kiểu xâu. Kiểu dữ liệu của các biến có hai dạng:

- Các kiểu dữ liệu nguyên thủy
- Các đối tượng (hoặc do người dùng định nghĩa hoặc là có sẵn trong ngôn ngữ Java), còn gọi là *biến tham chiếu*.

Kiểu dữ liệu nguyên thủy

Có chín kiểu dữ liệu nguyên thủy như bạn thường thấy:

Type	**Size**	**Default value**	**Example**
boolean	N/A	false	true
byte	8 bits	0	2
char	16 bits	'u/0000'	'a'
short	16 bits	0	12
int	32 bits	0	123
long	64 bits	0	9999999
float	32 bits with a decimal point	0.0	123.45
double	64 bits with a decimal point	0.0	999999999.99999999

Chúng ta dùng kiểu int cho dữ liệu age vì chúng ta không cần giá trị thập phân và một số nguyên là đủ lớn để mô tả tuổi thực của con người. Ta dùng kiểu String cho ba biến khác vì chúng không phải là số. String là một lớp trong gói java.lang, bạn có thể truy nhập vào mã lệnh của lớp này một cách tự động bất cứ lúc nào bạn muốn (chúng ta sẽ đề cập về nó trong phần Strings). Bạn cũng có thể khai báo các biến là kiểu do người dùng định nghĩa, ví dụ như Adult.

Chúng ta định nghĩa mỗi biến trên một dòng riêng biệt, nhưng thực tế không cần phải làm như vậy. Khi bạn có hai hoặc nhiều hơn các biến có cùng kiểu, bạn có thể

định nghĩa chúng trên cùng một dòng, chỉ cần ngăn cách chúng bằng dấu phẩy, như thế này:

accessSpecifier dataType variableName1,

variableName2,

variableName3,...

Nếu chúng ta muốn khởi tạo các biến này khi khai báo chúng, ta chỉ cần thêm giá trị khởi tạo sau tên từng biến:

accessSpecifier dataType variableName1 = initialValue,

variableName2 = initialValue, ...

Bây giờ lớp của chúng ta đã tự hiểu chính nó, chúng ta có thể minh chứng điều này như sau đây.

Phương thức main()

Có một phương thức đặc biệt mà bạn có thể đưa vào trong bất kỳ lớp nào để JRE có thể thi hành mã lệnh. Tên phương thức này là main(). Mỗi lớp chỉ có một phương thức main(). Dĩ nhiên, không phải mọi lớp đều có phương thức này nhưng vì Adult là lớp duy nhất mà hiện thời ta đang có, chúng ta sẽ bổ sung phương thức main() vào để có thể tạo một cá thể Adult và kiểm tra các biến cá thể của nó:

```
package intro.core;


public class Adult {
	protected int age;
	protected String name;
	protected String race;
	protected String gender;

	public static void main(String[] args) {
		Adult myAdult = new Adult();


		System.out.println("Name: " + myAdult.name);
		System.out.println("Age: " + myAdult.age);
		System.out.println("Race: " + myAdult.race);
		System.out.println("Gender: " + myAdult.gender);
	}
}
```

Trong thân của phương thức main(), chúng ta tạo một cá thể Adult, sau đó in ra giá trị của các biến cá thể. Hãy nhìn vào dòng đầu tiên. Đây là chỗ mà những người thuần túy chủ nghĩa hướng đối tượng khó chịu với ngôn ngữ Java. Họ cho rằng new nên là một phương thức của Adult và bạn phải gọi nó theo kiểu: Adult.new(). Dĩ nhiên tôi hiểu quan điểm của họ, nhưng ngôn ngữ Java không làm theo cách ấy, và đó là lý do để những người thuần túy chủ nghĩa có thể kêu ca

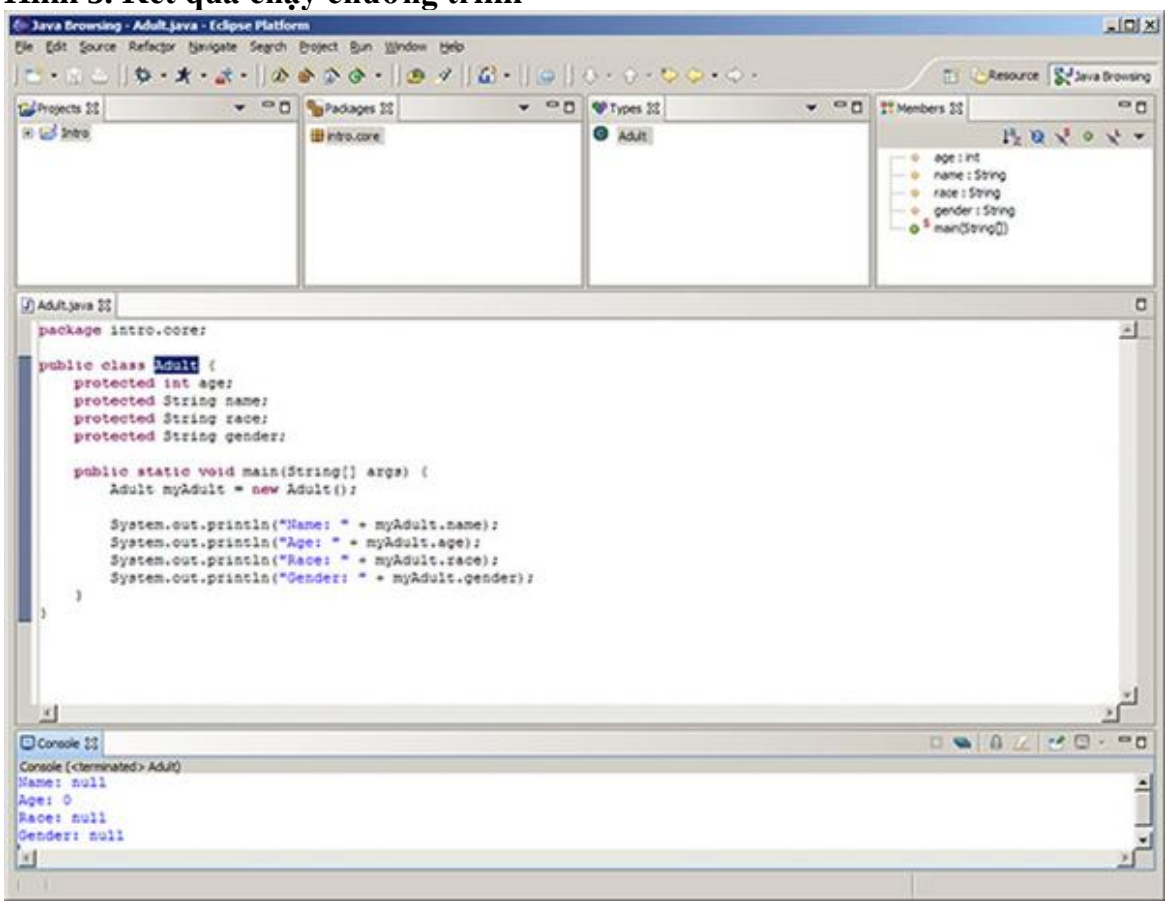
hoàn toàn đúng rằng Java không thuần là hướng đối tượng. Hãy xem lại dòng đầu tiên một lần nữa. Nhớ rằng tất cả lớp Java đều có hàm tạo mặc định, là thứ mà ta đang dùng ở đây.

Sau khi chúng ta tạo một cá thể Adult, chúng ta lưu nó trong một *biến cục bộ* gọi là myAdult. Sau đó ta in ra các giá trị của các biến cá thể của nó. Trong hầu hết các ngôn ngữ, bạn có thể in mọi thứ ra màn hình. Ngôn ngữ Java cũng không ngoại lệ. Cách bạn làm việc này trong mã lệnh Java là gọi phương thức println() của luồng out của đối tượng System. Đừng lo lắng về chuyện bạn phải hiểu tất cả chi tiết của quá trình ấy vào lúc này. Chỉ biết rằng chúng ta đang sử dụng một lời gọi phương thức hữu ích để in ra một thứ gì đó. Với mỗi lời gọi, ta chuyển giao một xâu trực kiện (string literal) và nối thêm giá trị của biến cá thể của myAdult. Chúng ta sẽ quay lại xem chi tiết phương thức này về sau.

Thi hành mã lệnh trong Eclipse

Để thi hành mã lệnh này, bạn phải thực hiện một ít thao tác trong Eclipse. Nhấn chọn lớp Adult trong khung nhìn Types và nhấn chọn biểu tượng "người đang chạy" trên thanh công cụ. Bạn sẽ thấy hộp thoại Run, nơi cho phép bạn tạo ra cấu hình khởi chạy cho chương trình của mình. Chọn kiểu cấu hình muốn tạo ra là **Java Application** rồi chọn **New.** Eclipse sẽ chỉ rõ tên mặc định đặt cho cấu hình này là "Adult", và thế là ổn. Nhấn **Run** để thấy kết quả. Eclipse sẽ hiển thị khung nhìn Console bên dưới bộ soạn thảo mã lệnh giống như ở hình 5.

Hình 5. Kết quả chạy chương trình



Lưu ý rằng các biến đều mang giá trị mặc định của chúng. Theo mặc định, bất kỳ biến cá thể nào dù là do người dùng định nghĩa hay có sẵn trong ngôn ngữ đều mang giá trị null. Khởi tạo tường minh các biến, đặc biệt là các đối tượng luôn luôn là một ý tưởng hay, nhờ đó bạn có thể chắc chắn về giá trị của các biến mà chúng có bên trong. Ta hãy quay lại và khởi tạo cho các biến những giá trị như sau:

Variable	**Value**
name	"Bob"
age	25

race	"inuit"
gender	"male"

Chạy lại mã lệnh bằng cách nhấn chọn một lần nữa vào biểu tượng người đang chạy. Bạn sẽ thấy giá trị mới hiện lên trên console.

Bây giờ thì hãy làm sao để Adult của chúng ta có khả năng nói cho những đối tượng khác biết về dữ liệu của nó.

Thêm hành vi

Các phương thức truy cập

Xem xét bên trong đối tượng Adult của chúng ta chỉ bằng cách tham chiếu trực tiếp đến các biến thì cũng thuận tiện nhưng thường thì đó không phải là một ý tưởng hay khi một đối tượng lại moi móc vào bên trong một đối tượng khác theo cách đó. Điều đó vi phạm nguyên lý bao gói mà chúng ta đã nói trước đó, và nó cũng cho phép một đối tượng chọc ngoáy vào trạng thái nội tại của đối tượng khác. Lựa chọn khôn ngoan hơn là cho một đối tượng có khả năng nói cho đối tượng khác biết giá trị các biến cá thể của nó khi được hỏi tới. Bạn dùng các *phương thức truy cập* để làm điều này.

Các phương thức truy cập là các phương thức giống như những phương thức khác nhưng chúng thường tuân thủ theo một quy ước đặt tên riêng. Để cung cấp giá trị một biến cá thể cho đối tượng khác, hãy tạo ra một phương thức có tên là getVariableName(). Tương tự như thế, để cho phép các đối tượng khác thiết đặt các biến cá thể của đối tượng của bạn, hãy tạo ra phương thức setVariableName().

Trong cộng đồng Java, các phương thức truy cập này thường được gọi là các *getter* và các *setter* vì tên chúng bắt đầu bằng get và set. Chúng là những phương thức đơn giản nhất mà bạn từng thấy, vì thế chúng là những ví dụ tốt để minh họa cho những khái niệm phương thức đơn giản. Bạn nên biết rằng phương thức truy cập là thuật ngữ khái quát để chỉ những phương thức nhận thông tin về một đối tượng nào đó. Không phải tất cả các phương thức truy cập đều tuân theo quy tắc đặt tên dành cho getter và setter như chúng ta sẽ thấy sau này.

Đây là một số đặc tính chung của các getter và setter:

- Định tố truy cập của các getter và setter điển hình là public.
- Các getter điển hình là không nhận tham số nào.
- Các setter điển hình là nhận chỉ một tham số, đó là giá trị mới cho biến cá thể mà chúng thiết đặt.
- Kiểu trả về của getter điển hình là cùng kiểu với biến cá thể mà nó báo lại giá trị.
- Kiểu trả lại của setter điển hình là void, nghĩa là chúng không trả lại gì hết (chúng chỉ đặt giá trị cho biến cá thể).

Khai báo các phương thức truy cập

Chúng ta có thể thêm các phương thức truy cập cho biến cá thể age của Adult như sau:

```
public int getAge() {
        return age;
}
public void setAge(int anAge) {
        age = anAge;
}
```

Phương thức getAge() trả lại giá trị của biến age bằng cách dùng từ khóa return. Các phương thức không trả lại giá trị gì thì ngầm hiểu có câu lệnh return void; ở cuối cùng. Trong phương thức lấy giá trị này, chúng ta tham chiếu đến biến age bằng cách dùng tên của biến.

Chúng ta cũng có thể viết return this.age;;. Biến this quy về đối tượng hiện tại. Nó được ngầm hiểu khi bạn tham chiếu trực tiếp đến một biến cá thể. Một số lập trình viên hướng đối tượng Smalltalk thích dùng this bất cứ khi nào họ nói đến một biến cá thể, cũng giống như họ luôn dùng từ khóa self khi viết mã lệnh bằng Smalltalk. Bản thân tôi cũng thích như thế nhưng Java không yêu cầu như vậy và làm thế sẽ chỉ thêm chật chỗ trên màn hình, vì vậy các ví dụ trong tài liệu này sẽ không dùng this trừ trường hợp mã lệnh sẽ không tường minh nếu thiếu nó.

Gọi các phương thức

Giờ ta đã có các phương thức truy cập, chúng ta sẽ thay việc truy cập trực tiếp đến biến age trong phương thức main() bằng lời gọi phương thức. Bây giờ main() sẽ như sau:

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

        Adult myAdult = new Adult();

        System.out.println("Name: " + myAdult.name);

        System.out.println("Age: " + myAdult.getAge());

        System.out.println("Race: " + myAdult.race);

        System.out.println("Gender: " + myAdult.gender);

}
```

Nếu bạn chạy lại mã lệnh, sẽ cho ra kết quả như cũ. Lưu ý rằng gọi phương thức của một đối tượng rất dễ dàng. Hãy sử dụng khuôn dạng sau:

instanceName.methodName()

Nếu phương thức này không cần tham số (ví dụ như getter), bạn vẫn phải viết cặp ngoặc đơn sau tên phương thức khi gọi. Nếu phương thức cần tham số (như setter), thì đặt chúng trong cặp ngoặc đơn, phân cách bởi dấu phẩy nếu có hơn một tham số.

Một lưu ý nữa về setter trước khi ta chuyển sang chủ đề khác: nó nhận một tham số kiểu int có tên là anAge. Sau đó nó gán giá trị của tham số này cho biến cá thể age. Chúng ta có thể đặt cho tham số này cái tên bất kỳ mà ta muốn. Tên không quan trọng nhưng khi bạn tham chiếu đến nó trong phương thức thì phải gọi chính xác cái tên mình đã đặt.

Trước khi chuyển sang phần khác, hãy thử dùng qua setter. Thêm dòng sau vào main() ngay sau khi chúng ta khởi tạo một đối tượng Adult:

```
myAdult.setAge(35);
```

Bây giờ thì chạy lại mã lệnh. Kết quả cho thấy tuổi là 35. Những gì diễn ra phía sau khung cảnh này là:

- Chúng tra truyền một *giá trị* số nguyên cho phương thức thông qua tham số.
- JRE cấp bộ nhớ cho tham số này và đặt tên cho nó là anAge.

Các phương thức không là phương thức truy cập

Các phương thức truy cập thật hữu ích, nhưng ta muốn các đối tượng Adult của ta có thể làm điều gì đó hơn là chỉ chia sẻ dữ liệu của chúng, bởi vậy chúng ta cần bổ sung thêm các phương thức khác. Chúng ta muốn Adult có thể nói, vậy hãy bắt đầu từ đây. Phương thức speak() sẽ như sau:

```
public String speak() {
        return "hello";
}
```

Bây giờ thì cú pháp này đã quen thuộc với bạn. Phương thức này trả lại một xâu trực kiện. Hãy dùng phương thức này và làm sáng sủa phương thức main(). Thay đổi lời gọi đầu tiên đến println() thành:

```
System.out.println(myAdult.speak());
```

Chạy lại mã lệnh bạn sẽ nhìn thấy dòng chữ hello trên màn hình.

Các xâu ký tự

Cho đến nay chúng ta đã sử dụng một vài biến kiểu String (xâu ký tự) nhưng chúng ta vẫn chưa thảo luận về chúng. Xử lý xâu trong C tốn rất nhiều công sức vì chúng là các mảng các ký tự 8 bít kết thúc bằng null mà bạn phải tự thao tác. Trong ngôn ngữ Java, xâu là đối tượng thuộc hạng nhất, có kiểu String, kèm theo đó là các phương thức cho phép bạn thao tác với nó. Mã lệnh Java gần nhất gần với ngôn ngữ C về vấn đề xâu là kiểu dữ liệu nguyên thủy char, chứa chỉ một ký tự Unicode đơn lẻ, ví dụ như 'a'.

Chúng ta đã biết cách khởi tạo một đối tượng String và thiết đặt giá trị cho nó, nhưng có nhiều cách khác để thực hiện việc này. Sau đây là vài cách để khởi tạo một cá thể String có giá trị là "hello":

```
String greeting = "hello";

String greeting = new String("hello");
```

Vì xâu trong ngôn ngữ Java là đối tượng hạng nhất, bạn có thể dùng new để khởi tạo một đối tượng thuộc kiểu xâu. Đặt một biến kiểu String cho ta kết quả tương tự, vì Java tạo ra một đối tượng String để chứa xâu trực kiện, sau đó gán đối tượng này cho biến cá thể.

Chúng ta có thể làm nhiều thứ với các String và lớp này có rất nhiều phương thức hữu ích. Thậm chí không cần dùng một phương thức, ta đã có thể làm vài việc lý thú với các xâu bằng cách nối một cặp xâu, nghĩa là kết hợp chúng xâu này tiếp sau xâu kia:

```
System.out.println("Name: " + myAdult.getName());
```

Thay vì dùng dấu +, chúng ta có thể gọi phương thức concat() của đối tượng String để nối nó với một đối tượng String khác:

```
System.out.println("Name: ".concat(myAdult.getName()));
```

Mã lệnh này trông hơi lạ, ta hãy duyệt qua một chút, từ trái sang phải:

- System là đối tượng có sẵn cho phép bạn tương tác với nhiều thứ trong môi trường hệ thống (bao gồm cả một vài khả năng của chính nền tảng Java)
- out là *biến lớp* của System, nghĩa là nó có thể truy cập được mà không cần phải có một cá thể của System. Nó đại diện cho màn hình.
- println() là phương thức của out nhận tham số kiểu String, in nó ra màn hình, và tiếp ngay sau là một ký tự xuống dòng để bắt đầu một dòng mới.
- "Name: " là một xâu trực kiện. Nền tảng Java coi xâu trực kiện này là một cá thể String, bởi vậy chúng ta có thể gọi phương thức trực tiếp trên nó.
- concat() là một phương thức của cá thể String, nhận tham số kiểu String và nối nó với chính cá thể String mà bạn đã gọi phương thức của nó.
- myAdult là một cá thể Adult của ta.
- getName() là phương thức truy cập biến cá thể name.

Như thế, JRE sẽ lấy tên của Adult, gọi concat(), và thêm “Bob” vào sau “Name: ".

Trong Eclipse, bạn có thể thấy các phương thức có sẵn trên bất kỳ đối tượng nào bằng cách đặt con chạy dấu chèn vào sau dấu chấm sau tên biến chứa cá thể, sau đó nhấn Ctrl-thanh phím khoảng trống. Thao tác này làm hiện ra ở bên trái dấu chấm một danh sách các phương thức của đối tượng. Bạn có thể cuộn lên cuộn xuống danh sách này bằng các phím mũi tên trên bàn phím, điểm sáng một phương thức bạn muốn rồi sau đó nhấn phím Enter để chọn nó. Ví dụ, để xem tất

cả các phương thức có sẵn của đối tượng String, đặt con chạy dấu chèn vào sau dấu chấm sau chữ “Name: ” rồi nhấn Ctrl-thanh phím khoảng trống.

Sử dụng xâu

Bây giờ ta hãy dùng phép nối xâu trong lớp Adult. Đến lúc này, ta có một biến cá thể là name. Một ý tưởng hay là có một biến firstname và một biến lastname, sau đó nối chúng vào với nhau khi ai đó hỏi tên của Adult. Không vấn đề gì! Hãy thêm phương thức sau đây:

```
public String getName() {

        return firstname + " " + lastname;

}
```

Eclipse sẽ hiển thị những đường lượn sóng màu đỏ tại phương thức này vì các biến cá thể ấy còn chưa có, như vậy có nghĩa là mã lệnh sẽ không biên dịch được. Bây giờ ta thay biến cá thể name bằng 2 biến sau đây (với giá trị mặc định làm rõ ý nghĩa hơn):

```
protected String firstname = "firstname";

protected String lastname = "lastname";
```

Tiếp đó, thay đổi lời gọi println() đầu tiên như sau:

```
System.out.println("Name: " + myAdult.getName());
```

Bây giờ ta có một getter có trang trí hơn để lấy ra các biến tên. Nó sẽ nối các xâu tốt đẹp để tạo ra một cái tên đầy đủ cho Adult. Chúng ta có thể viết phương thức getName() như sau:

```
public String getName() {
        return firstname.concat(" ").concat(lastname);
}
```

Mã lệnh này cũng làm cùng công việc đó, nhưng nó minh họa việc sử dụng tường minh một phương thức của String, nó cũng minh họa việc móc xích các lời gọi phương thức. Khi ta gọi concat() của firstname với một xâu trực kiện (dấu cách), nó sẽ trả lại một đối tượng String mới là ghép của hai xâu. Ngay sau đó ta lại tiếp tục gọi concat() của xâu ghép này để nối tên (firstname) và một dấu cách với lastname. Kết quả ta có tên đầy đủ theo khuôn dạng.

Các toán tử số học và toán tử gán

Adult của chúng ta có thể nói, nhưng không thể di chuyển. Hãy thêm một vài hành vi để nó có thể đi lại.

Đầu tiên, hãy thêm một biến cá thể để lưu lại số bước chân mà mỗi đối tượng Adult sẽ bước:

```
public int progress = 0;
```

Bây giờ bổ sung thêm phương thức có tên là walk():

```
public String walk(int steps) {
```

```
        progress = progress + steps;

        return "Just took " + steps + " steps";

}
```

Phương thức của chúng ta nhận một tham số là số nguyên để chỉ số bước cần bước, cập nhật progress để ghi nhận số bước chân, sau đó báo kết quả. Cũng là sáng suốt nếu ta bổ sung thêm một getter cho progress nhưng không thêm setter. Tại sao? Vì cho phép đối tượng khác bốc thẳng ta tiến lên phía trước một số bước chân nào đó thì chẳng khôn ngoan chút nào. Nếu đối tượng khác muốn đề nghị ta bước đi, nó có thể gọi walk(). Đó là theo tiếng nói của óc suy xét và đây là một ví dụ khá tầm thường. Ở các dự án thực sự, các quyết định thiết kế loại này luôn phải được đưa ra và thường không thể quyết định sớm từ trước, bất kể các đại cao thủ (gurus) thiết kế hướng đối tượng nói gì.

Trong phương thức của chúng ta, chúng ta đã cập nhật progress bằng cách thêm steps vào. Chúng ta lại lưu kết quả trong progress. Ta đã dùng *toán tử gán* cơ bản nhất, =, để lưu kết quả. Ta đã dùng toán tử số học + để cộng hai số hạng. Có một cách khác để đạt được cùng mục đích. Mã lệnh sẽ làm cùng một việc ấy:

```
public String walk(int steps) {

        progress += steps;

        return "Just took " + steps + " steps";

}
```

Sử dụng toán tử gán += sẽ ít dài dòng hơn cách đầu tiên ta dùng. Nó tránh cho ta khỏi phải tham chiếu đến biến progress hai lần. Nhưng nó thực hiện cùng một việc: nó cộng steps vào progress và lưu kết quả trong progress.

Bảng dưới đây là danh sách và mô tả ngắn gọn của các toán tử số học và toán tử gán thường gặp nhất trong Java (lưu ý rằng một số toán tử số học là nhị nguyên), có hai toán hạng, và một số khác là đơn nguyên, có một toán hạng).

Toán tử	Cách dùng	Mô tả
+	a + b	Cộng a và b
+	+a	Nâng a lên thành kiểu int nếu a là kiểu byte, short, hoặc char
-	a - b	Lấy a trừ đi b
-	-a	Âm a
*	a * b	Nhân a với b
/	a / b	Chia a cho b
%	a % b	Trả lại phần dư của phép chia a cho b (nói cách khác, đây là toán tử modulus)
++	a++	Tăng a thêm 1 đơn vị; tính giá trị của a trước khi tăng
++	++a	Tăng a thêm 1 đơn vị; tính giá trị của a sau khi tăng
--	a--	Giảm a đi 1 đơn vị; tính giá trị của a trước khi tăng
--	--a	Giảm a đi 1 đơn vị; tính giá trị của a sau khi tăng
+=	a += b	Giống như a = a + b

-=	a -= b	Giống như a = a - b
*=	a *= b	Giống như a = a * b
%=	a %= b	Giống như a = a % b

Chúng ta cũng đã biết một số thứ khác được gọi là *toán tử* trong ngôn ngữ Java. Ví dụ, dấu chấm . phân định tên của các gói và lời gọi phương thức; cặp ngoặc đơn (*params*) để phân định danh sách các tham số phân cách bằng dấu phẩy của một phương thức; và new, khi theo sau nó là tên của hàm tạo, để khởi tạo một cá thể. Chúng ta sẽ xem xét thêm một số cái trong phần tiếp theo.

Thực thi điều kiện

Giới thiệu về thực thi điều kiện

Mã lệnh đơn giản chạy từ câu lệnh đầu tiên đến câu lệnh cuối cùng mà không đổi hướng thực sự không thể làm được gì nhiều. Để trở nên hữu ích, chương trình của bạn cần đưa ra các quyết định, và có thể hành động khác đi trong những tình huống khác nhau. Giống như bất cứ một ngôn ngữ hữu dụng nào, Java cung cấp cho bạn những công cụ để làm điều đó, dưới dạng các câu lệnh và toán tử. Phần này sẽ đề cập đến các yếu tố sẵn có ấy khi bạn viết mã lệnh Java.

Các toán tử quan hệ và toán tử điều kiện

Java cung cấp cho bạn một vài toán tử và câu lệnh điều khiển luồng để cho phép bạn ra quyết định trong mã lệnh của mình. Rất thông thường, một quyết định lựa chọn trong mã lệnh bắt đầu bằng một biểu thức logic (là biểu thức được đánh giá bằng hai giá trị đúng hoặc sai – true/false). Những biểu thức đó sử dụng các *toán tử quan hệ*, chúng so sánh một toán hạng hoặc một biểu thức với một với một toán hạng hay biểu thức khác, và *các toán tử điều kiện* nữa. Đây là danh sách:

Toán tử	**Cách sử dụng**	**Trả về true nếu...**
>	a > b	a lớn hơn b
>=	a >= b	a lớn hơn hay bằng b
<	a < b	a nhỏ hơn b
<=	a <= b	a nhỏ hơn hay bằng b
==	a == b	a bằng b

! =	a != b	a không bằng b
&&	a && b	a và b cả hai đều true, tính b có điều kiện (nếu a là true thì không cần tính b nữa)
||	a || b	a hoặc b là true, tính b có điều kiện (nếu a là true thì không cần tính b nữa)
!	! a	a là false
&	a & b	a và b cả hai là true, luôn luôn tính b
|	a | b	a hoặc b là true, luôn luôn tính b
^	a ^ b	a và b là khác nhau (true nếu a là true và b là false, hoặc ngược lại, nhưng không được đồng thời là true hoặc false)

Thực thi điều kiện với lệnh if

Giờ ta phải dùng những toán tử đó. Hãy thêm một vài biểu thức logic đơn giản vào phương thức walk() của chúng ta:

```
public String walk(int steps) {

        if (steps > 100)
```

```
                return "I can't walk that far at once";


        progress = progress + steps;

        return "Just took " + steps + " steps.";

}
```

Bây giờ logic hành động trong phương thức này sẽ kiểm tra xem số lượng steps lớn như thế nào. Nếu quá lớn, phương thức sẽ ngay lập tức trả về và thông báo rằng thế là quá xa. Mỗi phương thức có thể trả về chỉ một lần. Nhưng không phải có hai giá trị trả về ở đây sao? Đúng, nhưng chỉ một lần trả về được thực thi. Điều kiện if của Java có khuôn dạng như sau:

```
if (  boolean expression ) {

        statements to execute if true...

} [else {

        statements to execute if false...

}]
```

Cặp ngoặc nhọn không bắt buộc nếu chỉ có một lệnh đơn sau từ khóa if hay sau từ khóa else, đó là lý do vì sao mã lệnh của chúng ta không dùng đến cặp ngoặc này. Bạn không buộc phải có mệnh đề else, và trong mã lệnh của ta cũng không có. Chúng ta có thể đặt các mã lệnh còn lại của phương thức trong mệnh đề else nhưng hiệu lực cũng sẽ tương đương và những thứ thêm vào một cách không cần thiết như thế được gọi là *gia vị cú pháp vô ích*, nó làm giảm tính dễ đọc của mã lệnh.

Phạm vi của biến

Tất cả các biến trong ứng dụng Java đều có một *phạm vi* (scope), hay là các đặc trưng xác định nơi bạn có thể truy cập biến chỉ bằng tên của nó. Nếu biến nằm *trong vùng phạm vi*, bạn có thể tương tác với nó bằng tên. Nếu biến nằm *ngoài vùng phạm vi* thì điều này là không thể.

Có nhiều mức phạm vi trong ngôn ngữ Java, được xác định bởi vị trí khai báo của biến ở đâu (**Lưu ý**: không phải tất cả đều là chính thức, theo như tôi biết, nhưng chúng thường là những cái tên mà người lập trình vẫn dùng).

```
public class SomeClass {


        member variable scope


      public void someMethod(  parameters ) {

            method parameter scope


            local variable declaration(s)

            local scope


            someStatementWithACodeBlock {

                  block scope

            }

      }

}
```

Phạm vi của một biến trải rộng cho đến cuối đoạn (hoặc cuối khối) mã lệnh mà nó được khai báo trong đó. Ví dụ, trong phương thức walk(), chúng ta tham chiếu đến tham số steps bằng cái tên đơn giản của nó, vì nó nằm trong phạm vi. Ở ngoài phương thức này, khi nói đến steps thì trình biên dịch sẽ báo lỗi. Mã lệnh cũng có thể gọi đến các biến đã được khai báo trong phạm vi rộng hơn mã đó. Ví dụ, chúng ta được phép tham chiếu đến biến cá thể progress bên trong phương thức walk().

Các dạng khác của lệnh if

Chúng ta có thể tạo ra một câu lệnh kiểm tra điều kiện đẹp hơn bằng cách viết lệnh if dưới dạng khác:

if (*boolean expression*) {

    statements to execute if true...

} else if (*boolean expression*) {

    statements to execute if false...

} else if (*boolean expression*) {

    statements to execute if false...

} else {

    default statements to execute...

}

Phương thức của chúng ta có thể giống như sau:

if (steps > 100)

```
        return "I can't walk that far at once";
else if (steps > 50)
        return "That's almost too far";
else {
        progress = progress + steps;
        return "Just took " + steps + " steps.";
}
```

Có một dạng viết tắt của lệnh if trông hơi xấu, nhưng cùng đạt được mục đích, mặc dù dạng viết tắt này không cho phép có nhiều câu lệnh cả trong phần if lẫn trong phần else. Dó là dùng toán tử tam nguyên ?:. (Toán tử tam nguyên là toán tử có ba toán hạng). Chúng ta có thể viết lại câu lệnh if đơn giản theo cách sau:

```
return (steps > 100) ? "I can't walk that far at once" : "Just took " + steps + " steps.";
```

Tuy nhiên, câu lệnh này không đạt được mục đích vì khi steps nhỏ hơn 100, chúng ta muốn trả về một thông điệp và đồng thời muốn cập nhật biến progress. Bởi vậy trong trường hợp này, sử dụng toán tử tắt ?: không phải là một lựa chọn vì chúng ta không thể thực thi nhiều câu lệnh với dạng viết tắt này.

Lệnh switch

Lệnh if là chỉ là một trong số các câu lệnh cho phép bạn kiểm tra điều kiện trong mã lệnh. Một câu lệnh khác bạn rất có thể đã gặp là lệnh switch. Nó đánh giá một biểu thức số nguyên, sau đó thực thi một hay nhiều câu lệnh dựa trên giá trị của biểu thức này. Cú pháp của lệnh như sau:

```
switch ( integer expression ) {
        case 1:
                statement(s)
                [break;]
        case 2:
                statement(s)
                [break;]
        case n:
                statement(s)
                [break;]
        [default:
                statement(s)
                break;]
}
```

JRE đánh giá *biểu thức số nguyên*, chọn ra trường hợp áp dụng, sau đó thực thi câu lệnh của trường hợp này. Câu lệnh cuối cùng của mỗi trường hợp ngoại trừ trường hợp cuối là break;. Nó là "lối thoát ra" của câu lệnh switch, và điều khiển sẽ tiếp tục xử lý dòng tiếp theo trong mã lệnh sau câu lệnh switch. Về mặt kỹ thuật, không cần có lệnh break;. Câu lệnh break cuối cùng lại càng đặc biệt không cần thiết vì dù gì thì điều khiển cũng tự thoát khỏi câu lệnh. Nhưng cách làm tốt là cứ thêm chúng vào. Nếu bạn không thêm lệnh break; ở cuối mỗi trường hợp, việc thực thi chương trình sẽ rơi vào trường hợp kế tiếp và tiếp tục chạy, cho đến khi gặp một câu lệnh break; hoặc đến hết câu lệnh switch. Trường hợp default sẽ được thực hiện nếu giá trị số nguyên không kích hoạt bất cứ trường hợp nào khác. Nó là không bắt buộc.

Về bản chất, câu lệnh switch thực sự là câu lệnh if-else if với điều kiện của lệnh if là số nguyên; nếu điều kiện của bạn dựa trên một giá trị số nguyên đơn lẻ, bạn có thể dùng hoặc là lệnh switch hoặc là lệnh if-else. Vậy chúng ta có thể viết lại điều kiện if của chúng ta trong phương thức walk dưới dạng câu lệnh switch được không? Câu trả lời là không vì chúng ta đã kiểm tra một biểu thức logic (steps > 100). Câu lệnh switch không cho phép kiểm tra như thế.

Ví dụ về câu lệnh switch

Đây là một ví dụ tầm thường về việc sử dụng câu lệnh switch (nó là một ví dụ khá cổ điển):

```
int month = 3;

switch (month) {

        case 1: System.out.println("January"); break;

        case 2: System.out.println("February"); break;

        case 3: System.out.println("March"); break;

        case 4: System.out.println("April"); break;

        case 5: System.out.println("May"); break;

        case 6: System.out.println("June"); break;

        case 7: System.out.println("July"); break;

        case 8: System.out.println("August"); break;

        case 9: System.out.println("September"); break;

        case 10: System.out.println("October"); break;

        case 11: System.out.println("November"); break;
```

```
        case 12: System.out.println("December"); break;

        default:  System.out.println("That's not a valid month number."); break;
}
```

month là một biến nguyên biểu thị số hiệu của tháng. Vì là số nguyên nên câu lệnh switch ở đây là hợp lệ. Với mỗi trường hợp hợp lệ, chúng ta in ra tên tháng, sau đó thoát khỏi câu lệnh. Trường hợp mặc định quản lý các số nằm ngoài phạm vi hợp lệ của các tháng.

Sau rốt, đây là một ví dụ về việc dùng nhiều trường hợp thông nhau có thể là một mẹo nhỏ thú vị:

```
int month = 3;

switch (month) {

  case 2:

  case 3:

  case 9: System.out.println(

    "My family has someone with a birthday in this month."); break;

  case 1:

  case 4:

  case 5:

  case 6:

  case 7:

  case 8:

  case 10:
```

```
  case 11:

  case 12: System.out.println("Nobody in my family has a birthday in this
month."); break;

            default: System.out.println("That's not a valid month number."); break;

}
```

Ở đây ta thấy trường hợp 2, 3 và 9 được xử lý giống nhau; các trường hợp còn lại có một kiểu xử lý khác. Lưu ý rằng các trường hợp này không phải xếp liền theo thứ tự và có nhiều trường hợp thông nhau là những gì chúng ta cần trong tình huống này.

Chạy lại nhiều lần

Ở ngay đầu tài liệu này, chúng ta đã làm mọi thứ vô điều kiện, tạm thời cũng ổn, nhưng nó có những hạn chế. Tương tự thế, thỉnh thoảng chúng ta muốn mã lệnh thi hành đi thi hành lại cùng một việc cho đến khi công việc hoàn tất. Ví dụ, giả sử ta muốn đối tượng Adult của chúng ta nói hơn một lần câu "hello". Điều này khá dễ thực hiện trong mã lệnh Java (mặc dù không dễ thực hiện trong các ngôn ngữ kịch bản lệnh như Groovy chẳng hạn). Java cung cấp cho bạn các cách sau để lặp đi lặp lại mã lệnh, hoặc thực hiện mã lệnh hơn một lần:

- câu lệnh for
- câu lệnh do
- câu lệnh while

Chúng thường được gọi chung là *các vòng lặp* (ví dụ, vòng lặp for), vì chúng lặp đi lặp lại cả khối mã lệnh cho đến khi bạn ra lệnh cho chúng dừng lại. Trong các phần tiếp sau, chúng ta sẽ nói ngắn gọn về từng lệnh một và dùng chúng trong phương thức speak() để trò chuyện chút ít.

Vòng lặp for

Cấu trúc lặp cơ bản nhất trong ngôn ngữ Java là câu lệnh for, câu lệnh này cho phép bạn lặp từng bước trên một phạm vi giá trị cho phép xác định số lần thực thi vòng lặp. Cú pháp thường sử dụng nhất của vòng lặp for như sau:

for (*initialization; termination; increment*) {

    statement(s)

}

Biểu thức khởi tạo (initialization) xác định vòng lặp bắt đầu ở đâu. *Biểu thức dừng* (termination) xác định vòng lặp kết thúc ở đâu. *Biểu thức tăng* (increment) xác định biến khởi tạo sẽ tăng bao nhiêu mỗi lần đi qua vòng lặp. Mỗi lần lặp, vòng lặp sẽ thực hiện các câu lệnh trong khối lệnh, là tập hợp các câu lệnh nằm giữa dấu ngoặc nhọn (hãy nhớ rằng bất kỳ khối nào trong mã lệnh Java cũng được đặt giữa hai dấu ngoặc nhọn, chứ không phải chỉ mã lệnh của vòng lặp for).

Cú pháp và khả năng của vòng lặp for có khác trong Java phiên bản 5.0, do vậy hãy đọc bài viết của John Zukowski về các đặc tính mới, thú vị trong ấn bản mới ra gần đây của ngôn ngữ này. (xem Các tài nguyên).

Sử dụng vòng lặp for

Ta hãy biến đổi phương thức speak() sao cho nó nói từ “hello” ba lần, dùng vòng lặp for. Khi làm việc này, chúng ta sẽ tìm hiểu về một lớp có sẵn của Java, làm việc lắp ghép các xâu một cách ngon lành:

public String speak() {

```
        StringBuffer speech = new StringBuffer();
        for (int i = 0; i < 3; i++) {
                speech.append("hello");
        }
        return speech.toString();
}
```

Lớp StringBuffer trong gói java.lang cho phép bạn thao tác các xâu dễ dàng và rất tuyệt vời trong việc nối các xâu lại với nhau (giống như ta móc chúng lại với nhau). Đơn giản là chúng ta khởi tạo một xâu, sau đó gọi phương thức append() mỗi lần muốn thêm một điều gì đó vào lời nói của từng Adult. Vòng lặp for là nơi mọi việc thực sự diễn ra. Trong cặp ngoặc đơn của vòng lặp, chúng ta đã khai báo một biến nguyên i để làm số đếm cho vòng lặp (các ký tự *i, j và k* thường được dùng làm biến đếm vòng lặp, nhưng bạn có thể đặt bất kỳ tên nào mà bạn muốn cho biến này). Biểu thức tiếp theo cho biết chúng ta sẽ tiếp tục lặp cho đến khi biến này đạt đến một giá trị nhỏ hơn ba. Biểu thức sau cho biết chúng ta sẽ tăng biến đếm lên một sau mỗi lần lặp (hãy nhớ toán tử ++). Mỗi lần đi qua vòng lặp, chúng ta sẽ gọi phương thức append() của speech và dán một từ "hello" khác vào cuối.

Bây giờ, thay phương thức speak() cũ bằng phương thức speak() mới, loại bỏ mọi lệnh println trong main() và thêm một lệnh để gọi phương thức speak() của Adult. Khi bạn thực hiện, lớp có thể giống như sau:

```
package intro.core;

public class Adult {
        protected int age = 25;
```

```
protected String firstname = "firstname";
protected String lastname = "lastname";
protected String race = "inuit";
protected String gender = "male";
protected int progress = 0;

public static void main(String[] args) {
        Adult myAdult = new Adult();
        System.out.println(myAdult.speak());
}
public int getAge() {
        return age;
}
public void setAge(int anAge) {
        age = anAge;
}
public String getName() {
        return firstname.concat(" ").concat(lastname);
}
public String speak() {
        StringBuffer speech = new StringBuffer();
        for (int i = 0; i < 3; i++) {
```

```
                speech.append("hello");
            }
            return speech.toString();
        }
        public String walk(int steps) {
            if (steps > 100)
                return "I can't walk that far at once";
            else if (steps > 50)
                return "That's almost too far";
            else {
                progress = progress + steps;
                return "Just took " + steps + " steps.";
            }
        }
}
```

Khi bạn chạy đoạn mã lệnh này, bạn sẽ nhận được kết quả là dòng hellohellohello trên màn hình. Nhưng dùng vòng lặp for chỉ là một cách để làm việc này. Java còn cho bạn hai cấu trúc thay thế khác mà bạn sẽ thấy tiếp theo đây.

Vòng lặp while

Đầu tiên hãy thử vòng lặp while. Phiên bản sau đây của phương thức speak() cho ra cùng một kết quả như cách làm mà bạn thấy ở phần trên:

```
public String speak() {
        StringBuffer speech = new StringBuffer();
        int i = 0;
        while (i < 3) {
                speech.append("hello");
                i++;
        }
        return speech.toString();
}
```

Cú pháp cơ bản của vòng lặp while như sau:

```
while ( boolean expression ) {
        statement(s)
}
```

Vòng lặp while thực hiện mã lệnh trong khối cho đến khi biểu thức của nó trả lại giá trị là false. Vậy bạn điều khiển vòng lặp như thế nào? Bạn phải đảm bảo là biểu thức sẽ trở thành false tại thời điểm nào đó, nếu không, bạn sẽ có một vòng lặp vô hạn. Trong trường hợp của chúng ta, ta khai báo một biến địa phương là *i* ở bên ngoài vòng lặp, khởi tạo cho nó giá trị là 0, sau đó kiểm tra giá trị của nó trong

biểu thức lặp. Mỗi lần đi qua vòng lặp, chúng ta lại tăng *i* lên. Khi nó không còn nhỏ hơn 3 nữa, vòng lặp sẽ kết thúc và chúng ta sẽ trả lại xâu ký tự lưu trong bộ đệm.

Ở đây ta thấy vòng for thuận tiện hơn. Khi dùng vòng for, chúng ta khai báo và khởi tạo biến điều khiển, kiểm tra giá trị của nó và tăng giá trị của nó chỉ bằng một dòng mã lệnh. Dùng vòng while đòi hỏi nhiều việc hơn. Nếu chúng ta quên tăng biến đếm, chúng ta sẽ có một vòng lặp vô hạn. Nếu ta không khởi tạo biến đếm, trình biên dịch sẽ nhắc nhở. Nhưng vòng while lại rất tiện lợi nếu bạn phải kiểm tra một biểu thức logic phức tạp (đóng hộp tất cả vào vòng lặp for thú vị ấy sẽ làm cho nó rất khó đọc).

Bây giờ ta đã thấy vòng lặp for và while, nhưng phần tiếp theo sẽ minh họa cho ta thấy còn một cách thứ ba nữa.

Vòng lặp do

Đoạn mã lệnh sau đây sẽ thực thi chính xác những điều mà ta thấy ở hai vòng lặp trên:

```
public String speak() {
        StringBuffer speech = new StringBuffer();
        int i = 0;
        do {
                speech.append("hello");
                i++;
        } while (i < 3) ;
        return speech.toString();
}
```

Cú pháp cơ bản của vòng lặp do như sau:

```
do {
        statement(s)
} while ( boolean expression ) ;
```

Vòng lặp do gần như giống hệt vòng lặp while, ngoại trừ việc nó kiểm tra biểu thức logic *sau* khi thực thi khối lệnh lặp. Với vòng lặp while, chuyện gì sẽ xảy ra nếu biểu thức cho giá trị false ngay lần đầu tiên kiểm tra? Vòng lặp sẽ không thực hiện dù chỉ một lần. Còn với vòng lặp do, bạn sẽ được đảm bảo là vòng lặp sẽ thực hiện ít nhất 1 lần. Sự khác biệt này có thể có ích vào nhiều lúc.

Trước khi tạm biệt các vòng lặp, hãy xem lại hai câu lệnh rẽ nhánh. Chúng ta đã thấy lệnh break khi ta nói đến câu lệnh switch. Nó cũng có hiệu quả tương tự trong vòng lặp: nó dừng vòng lặp. Mặt khác, lệnh continue giúp dùng ngay lần lặp hiện tại và chuyển tới lần lặp tiếp theo. Đây là một ví dụ thường thấy:

```
for (int i = 0; i < 3; i++) {
        if (i < 2) {
                System.out.println("Haven't hit 2 yet...");
                continue;
        }
```

```
        if (i == 2) {
                System.out.println("Hit 2...");
                break;
        }
}
```

Nếu bạn đưa đoạn mã này vào trong phương thức main() và chạy, bạn sẽ nhận được kết quả như sau:

```
Haven't hit 2 yet...
Haven't hit 2 yet...
Hit 2...
```

Hai lần đầu đi qua vòng lặp, i đều nhỏ hơn 2, do vậy chúng ta in ra dòng chữ “Haven't hit 2 yet...”, sau đó thực hiện lệnh continue, lệnh này chuyển luôn đến lần lặp tiếp theo. Khi i bằng 2, khối mã lệnh trong phần lệnh if đầu tiên không được thi hành. Chúng ta nhảy đến lệnh if thứ hai, in ra dòng chữ "Hit 2...", sau đó lệnh break thoát khỏi vòng lặp.

Trong phần tiếp theo, chúng ta sẽ tăng thêm sự phong phú của các hành vi có thể bổ sung thêm thông qua việc trình bày về việc xử lý *các sưu tập*.

Các sưu tập

Giới thiệu về các sưu tập

Hầu hết các ứng dụng phần mềm của thế giới thực đều có liên quan đến các sưu tập sự vật nào đó (các tệp, các biến, các dòng của tệp, …). Thông thường, các chương trình hướng đối tượng đều có liên quan đến sưu tập các đối tượng. Ngôn ngữ Java có một Khung công tác các sưu tập (Collections Framework) khá tinh vi cho phép bạn tạo và quản lý các sưu tập đối tượng thuộc các kiểu khác nhau. Bản thân khung công tác này đã có thể đủ để viết riêng nguyên cả một cuốn sách hướng dẫn, do đó chúng tôi sẽ không bàn tất cả trong tài liệu này. Thay vào đó, chúng tôi sẽ đề cập đến sưu tập thường dùng nhất và một vài kỹ thuật sử dụng nó. Những kỹ thuật đó áp dụng cho hầu hết các sưu tập có trong ngôn ngữ Java.

Mảng

Hầu hết các ngôn ngữ lập trình đều có khái niệm mảng để chứa một sưu tập các sự vật và Java cũng không ngoại lệ. Mảng thực chất là một sưu tập các phần tử có cùng kiểu.

Có hai cách để khai báo một mảng:

- Tạo một mảng có kích thước cố định và kích thước này không bao giờ thay đổi.

- Tạo một mảng với một tập các giá trị ban đầu. Kích thước của tập giá trị này sẽ quyết định kích cỡ của mảng – nó sẽ vừa đủ lớn để chứa toàn bộ các giá trị đó. Sau đó thì kích cỡ này sẽ cố định mãi.

Nói chung, bạn khai báo một mảng như sau:

new *elementType* [*arraySize*]

Để tạo một mảng số nguyên gồm có 5 phần tử, bạn phải thực hiện theo một trong hai cách sau:

```
int[] integers = new int[5];
```

```
int[] integers = new int[] { 1, 2, 3, 4, 5 };
```

Câu lệnh đầu tiên tạo một mảng rỗng gồm có 5 phần tử. Câu lệnh thứ hai là cách tắt để khởi tạo một mảng. Câu lệnh này cho phép bạn xác định một danh sách các giá trị khởi tạo, phân tách nhau bằng dấu phẩy (,), nằm trong cặp ngoặc nhọn. Chú ý là chúng ta không khai báo kích cỡ trong cặp ngoặc vuông – số các mục trong khối khởi tạo quyết định kích cỡ của mảng là 5 phần tử. Cách làm này dễ hơn là tạo một mảng rồi sau đó viết mã lệnh cho một vòng lặp để đặt các giá trị vào, giống như sau:

```
int[] integers = new int[5];

for (int i = 1; i <= integers.length; i++) {

        integers[i] = i;

        System.out.print(integers[i] + " ");

}
```

Đoạn mã lệnh này cũng khai báo một mảng số nguyên có 5 phần tử. Nếu ta thử xếp nhiều hơn 5 phần tử vào mảng này, ta sẽ gặp ngay vấn đề khi chạy đoạn mã lệnh này. Để nạp mảng, chúng ta phải lặp đi qua các số nguyên từ 1 cho đến số bằng chiều dài của mảng, chiều dài của mảng ta có thể biết được nhờ truy cập phương thức length() của đối tượng mảng. Mỗi lần lặp qua mảng, chúng ta đặt một số nguyên vào mảng. Khi gặp số 5 thì dừng lại.

Khi mảng đã nạp xong, chúng ta có thể truy nhập vào các phần tử trong mảng nhờ vòng lặp tương tự:

```
for (int i = 0; i < integers.length; i++) {

        System.out.print(integers[i] + " ");

}
```

Bạn hãy coi mảng như một dãy các thùng. Mỗi phần tử trong mảng nằm trong một thùng, mỗi thùng được gán một chỉ số khi bạn tạo mảng. Bạn truy nhập vào các phần tử nằm trong thùng cụ thể nào đó bằng cách viết:

arrayName [*elementIndex*]

Chỉ số của mảng bắt đầu từ 0, có nghĩa là phần tử đầu tiên ở vị trí số 0. Điều đó làm cho vòng lặp thêm ý nghĩa. Chúng ta bắt đầu vòng lặp bằng số 0 vì mảng được đánh chỉ số bắt đầu từ 0 và chúng ta lặp qua từng phần tử trong mảng, in ra giá trị của từng chỉ số phần tử.

Sưu tập là gì?

Mảng cũng tốt, nhưng làm việc với chúng cũng có đôi chút bất tiện. Nạp giá trị cho mảng cũng mất công, và một khi khai báo mảng, bạn chỉ có thể nạp vào mảng những phần tử đúng kiểu đã khai báo và với số lượng phần tử đúng bằng số lượng mà mảng có thể chứa. Mảng chắc chắn là không có vẻ hướng đối tượng lắm. Thực tế, lý do chính để Java có mảng là vì nó được giữ lại để dùng như di sản từ những ngày tiền lập trình hướng đối tượng. Mảng có trong mọi phần mềm, bởi vậy không có mảng sẽ khiến cho ngôn ngữ khó mà tồn tại trong thế giới thực, đặc biệt khi bạn phải tương tác với các hệ thống khác có dùng mảng. Nhưng Java cung cấp cho bạn nhiều công cụ để quản lý sưu tập hơn. Những công cụ này thực sự rất hướng đối tượng.

Khái niệm sưu tập không khó để có thể hiểu được. Khi bạn cần một số lượng cố định các phần tử có cùng kiểu, bạn có thể dùng mảng. Khi bạn cần các phần tử có kiểu khác nhau hoặc số lượng các phần tử có thể thay đổi linh hoạt, bạn dùng sưu tập của Java.

Danh sách mảng

Trong tài liệu này, chúng tôi sẽ đề cập đến chỉ một dạng của sưu tập, đó là ArrayList. Trong lúc trình bày, bạn sẽ biết được một lý do khác khiến cho nhiều người thuần túy chủ nghĩa hướng đối tượng công kích ngôn ngữ Java.

Để dùng ArrayList, bạn phải thêm một lệnh quan trọng vào lớp của mình:

```
import java.util.ArrayList;
```

Bạn khai báo một ArrayList rỗng như sau:

```
ArrayList referenceVariableName = new ArrayList();
```

Bổ sung và loại bỏ các phần tử trong danh sách khá dễ dàng. Có nhiều phương thức để làm điều ấy, nhưng có hai phương thức thường dùng nhất như sau:

```
someArrayList.add(someObject);

Object removedObject = someArrayList.remove(someObject);
```

Đóng hộp và mở hộp các kiểu nguyên thủy.

Các sưu tập Java chứa các đối tượng, chứ không phải là các kiểu nguyên thủy. Mảng có thể chứa cả hai, nhưng lại không hướng đối tượng như ta muốn. Nếu bạn muốn lưu trữ bất cứ kiểu gì là kiểu con của Object vào một danh sách, bạn đơn giản chỉ cần gọi một trong số nhiều phương thức của ArrayList để làm việc này. Cách đơn giản nhất là:

```
referenceVariableName.add(someObject);
```

Câu lệnh này thêm một đối tượng vào cuối danh sách. Cho đến đây mọi việc đều ổn. Nhưng liệu điều gì sẽ xảy ra khi bạn muốn thêm một kiểu nguyên thủy vào danh sách? Bạn không thể làm việc này trực tiếp. Thay vào đó, bạn phải *bọc* kiểu nguyên thủy thành đối tượng. Mỗi kiểu nguyên thủy có một lớp bao bọc tương ứng:

- Boolean dành cho các boolean
- Byte dành cho các byte
- Character dành cho các char
- Integer dành cho các int
- Short dành cho các short
- Long dành cho các long
- Float dành cho các float
- Double dành cho các double

Ví dụ, để đưa kiểu nguyên thủy int vào một ArrayList, chúng ta sẽ phải viết mã lệnh như sau:

```
Integer boxedInt = new Integer(1);

someArrayList.add(boxedInt);
```

Bao bọc kiểu nguyên thủy trong một cá thể của lớp bao bọc (wrapper) cũng được gọi là *thao tác đóng hộp* (boxing) kiểu nguyên thủy. Để nhận lại kiểu nguyên thủy ban đầu, ta phải *mở hộp* (unboxing) nó. Có nhiều phương thức hữu dụng trong các lớp bao bọc, nhưng sử dụng chúng khá phiền toái đối với hầu hết các lập trình viên vì nó đòi hỏi nhiều thao tác phụ thêm để sử dụng kiểu nguyên thủy với các sưu

tập. Java 5.0 đã giảm bớt những vất vả ấy bằng cách hỗ trợ các thao tác *đóng hộp/mở hộp tự động*.

Sử dụng các sưu tập

Trong đời thực hầu hết người trưởng thành đều mang theo tiền. Giả sử các Adult đều có ví để đựng tiền của mình. Với hướng dẫn này, chúng ta sẽ giả sử rằng:

- Chỉ các tờ giấy bạc là biểu hiện của tiền tệ
- Mệnh giá của tờ giấy bạc (như một số nguyên) đồng nhất với tờ giấy bạc đó.
- Tất cả tiền trong ví đều là đô la Mỹ.
- Mỗi đối tượng Adult khởi đầu cuộc đời được lập trình của nó không có đồng tiền nào

Bạn nhớ mảng các số nguyên chứ? Thay vào đó ta hãy tạo một ArrayList. Nhập khẩu gói ArrayList, sau đó thêm một ArrayList vào lớp Adult ở cuối danh sách các biến cá thể khác:

```
protected ArrayList wallet = new ArrayList();
```

Chúng ta tạo một ArrayList và khởi tạo nó là danh sách rỗng vì đối tượng Adult phải kiếm từng đồng đô la. Chúng ta cũng có thể bổ sung thêm vài phương thức truy cập wallet nữa:

```
public ArrayList getWallet() {
        return wallet;
}
```

```
public void setWallet(ArrayList aWallet) {

        wallet = aWallet;

}
```

Cung cấp những phương thức truy cập nào là tùy theo óc suy xét, nhưng trong trường hợp này ta đi đến những phương thức truy cập thông thường. Chẳng có lý do gì mà chúng ta không thể gọi setWallet() giống như gọi resetWallet(), hay thậm chí là goBankrupt() vì chúng ta đang thiết đặt lại nó thành ArrayList rỗng. Liệu một đối tượng khác có thể thiết đặt lại wallet của chúng ta với một giá trị mới không? Một lần nữa ta lại phải viện đến óc xét đoán. Đó là những gì mà thiết kế hướng đối tượng tính đến (OOD)!

Bây giờ chúng ta sẽ thiết đặt mọi thứ để bổ sung một vài phương thức cho phép ta tương tác với wallet:

```
public void addMoney(int bill) {

        Integer boxedBill = new Integer(bill);

        wallet.add(boxedBill);

}

public void spendMoney(int bill) {

        Integer boxedBill = new Integer(bill);

        boolean haveThatBill = wallet.contains(boxedBill);



        if(haveThatBill) {

                wallet.remove(boxedBill);

        } else {
```

```
            System.out.println("I don't have that bill.");
        }
}
```

Chúng ta sẽ nghiên cứu chi tiết hơn trong mấy phần tiếp theo đây.

Tương tác với sưu tập

Phương thức addMoney() cho phép chúng ta đưa thêm một tờ giấy bạc vào ví. Ta hãy nhớ lại rằng tờ giấy bạc của chúng ta ở đây chỉ đơn giản là những số nguyên. Để thêm chúng vào sưu tập, ta phải bao bọc một số kiểu int thành đối tượng Integer.

Phương thức spendMoney() lại nhảy vũ điệu đóng hộp để kiểm tra tờ giấy bạc có trong wallet không bằng cách gọi contains(). Nếu ta có tờ giấy bạc đó, ta gọi remove() để lấy nó đi. Nếu ta không thực hiện thì ta cũng nói như vậy.

Hãy dùng các phương thức này trong main(). Thay thế nội dung hiện tại trong main() bằng nội dung sau:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();


        myAdult.addMoney(5);
        myAdult.addMoney(1);
        myAdult.addMoney(10);
```

```
        StringBuffer bills = new StringBuffer();

        Iterator iterator = myAdult.getWallet().iterator();

        while (iterator.hasNext()) {

                Integer boxedInteger = (Integer) iterator.next();

                bills.append(boxedInteger);

        }

        System.out.println(bills.toString());

}
```

Cho đến thời điểm này ta thấy phương thức main() tổng hợp rất nhiều thứ. Đầu tiên, chúng ta gọi phương thức addMoney() vài lần để nhét tiền vào trong wallet. Sau đó ta lặp đi qua nội dung của wallet để in ra những gì có trong đó. Chúng ta dùng vòng lặp while để làm điều này, nhưng ta còn phải làm thêm một số việc nữa. Đó là:

- Lấy một Iterator cho danh sách, nó sẽ giúp chúng ta truy nhập từng phần tử trong danh sách.
- Gọi hasNext() của Iterator với vai trò biểu thức logic để chạy vòng lặp xem liệu ta có còn phần tử nào cần xử lý nữa không
- Gọi next() của Iterator để lấy phần tử tiếp theo mỗi lần đi qua vòng lặp
- *Ép kiểu* đối tượng trả về thành *kiểu mà ta biết* trong danh sách (trong trường hợp này là Integer)

Đó là cách diễn đạt chuẩn dành cho vòng lặp qua một sưu tập trong ngôn ngữ Java. Một cách làm khác là ta có thể gọi phương thức toArray() của danh sách và nhận lại một mảng, sau đó ta có thể lặp qua mảng này, sử dụng vòng for như ta đã làm với vòng while. Cách làm hướng đối tượng hơn là khai thác sức mạnh của khung công tác sưu tập của Java.

Khái niệm mới duy nhất ở đây là ý tưởng *ép kiểu* (casting). Đó là gì? Như ta đã biết, đối tượng trong ngôn ngữ Java có kiểu, hay là lớp. Nếu bạn nhìn vào chữ ký của phương thức next(), bạn sẽ thấy nó trả lại một Object, chứ không phải là một lớp con cụ thể của Object. Tất cả các đối tượng trong thế giới lập trình Java đều là lớp con của Object, nhưng Java cần biết kiểu chính xác của đối tượng để bạn có thể gọi các phương thức tương ứng với kiểu mà bạn muốn có. Nếu bạn không ép kiểu, bạn sẽ bị giới hạn chỉ được dùng các phương thức có sẵn dành cho Object, thực sự chỉ gồm một danh sách ngắn mà thôi. Trong ví dụ cụ thể này, chúng ta không cần gọi bất kỳ phương thức nào của Integer mà không có trong danh sách, nhưng nếu ta cần gọi thì ta phải ép kiểu trước đã.

Nâng cấp đối tượng của bạn

Giới thiệu về việc nâng cấp đối tượng của bạn

Bây giờ thì Adult của ta đã khá hữu ích, nhưng chưa thực sự hữu ích như nó cần phải có. Trong phần này, chúng ta sẽ nâng cấp đối tượng khiến nó dễ sử dụng hơn và cũng hữu ích hơn. Công việc bao gồm:

- Tạo ra vài hàm tạo hữu ích.
- *Nạp chồng* một vài phương thức để tạo ra một giao diện công cộng thuận tiện hơn
- Thêm mã lệnh để hỗ trợ so sánh các Adult s
- Thêm mã lệnh để dễ dàng gỡ lỗi khi sử dụng Adult s

Đồng thời, chúng ta sẽ tìm hiểu về các kỹ thuật *tái cấu trúc mã* và xem xem có thể xử lý những sai sót mà ta gặp trong khi chạy mã lệnh như thế nào.

Xây dựng các hàm tạo

Trước đây chúng ta đã đề cập đến các hàm tạo. Bạn có thể nhớ rằng tất cả các đối tượng trong mã lệnh Java đều có sẵn một hàm tạo không tham số mặc định. Bạn không phải định nghĩa nó, và bạn sẽ không thấy nó xuất hiện trong mã lệnh của mình. Thực tế, chúng ta đã dùng lợi thế ấy trong lớp Adult. Bạn không thấy có sự xuất hiện của một hàm tạo trong lớp này.

Tuy nhiên, trong thực tiễn sáng suốt hơn là nên định nghĩa hàm tạo của riêng bạn. Khi bạn làm vậy, bạn có thể hòan toàn yên tâm là ai đó khi khảo sát lớp của bạn sẽ biết cách xây dựng nó theo cách mà bạn muốn. Bởi vậy hãy định nghĩa một hàm tạo không có tham số riêng của mình. Ta nhắc lại cấu trúc cơ bản của một hàm tạo:

accessSpecifier ClassName(*arguments*) {

constructor statement(s)

}

Định nghĩa một hàm tạo không tham số cho Adult thật là đơn giản:

```
public Adult {

}
```

Chúng ta đã làm xong. Hàm tạo không có tham số của chúng ta chẳng làm gì cả, thực thế, ngoại trừ việc tạo ra một Adult. Bây giờ khi ta gọi new để sinh một Adult, chúng ta sẽ dùng hàm tạo không tham số của chúng ta để thay thế cho hàm tạo mặc định. Nhưng điều gì sẽ xảy ra nếu ta muốn hàm tạo do ta xây dựng thực hiện một số việc? Trong trường hợp của Adult, sẽ tiện lợi hơn nhiều nếu có thể chuyển thêm tên và họ dưới dạng String, và yêu cầu hàm tạo thiết đặt các biến cá thể với các giá trị khởi tạo đó. Điều này cũng được làm đơn giản như thế này:

```
public Adult(String aFirstname, String aLastname) {

        firstname = aFirstname;

        lastname = aLastname;

}
```

Hàm tạo này nhận hai tham số và sẽ gán chúng cho các biến cá thể. Hiện tại chúng ta có hai hàm tạo. Chúng ta thực sự không cần hàm tạo đầu tiên nữa, nhưng không hề gì nếu giữ nó lại. Nó mang lại cho người dùng lớp này một tùy chọn. Họ có thể tạo một đối tượng Adult với tên mặc định hoặc tạo một đối tượng Adult có tên xác định mà họ đưa vào.

Những gì ta vừa làm, thậm chí là bạn có lẽ còn chưa biết, được gọi là *nạp chồng* (overload) phương phức. Chúng ta sẽ thảo luận về khái niệm này chi tiết hơn trong phần tiếp theo.

Nạp chồng phương thức

Khi bạn tạo hai phương thức có cùng tên, nhưng số lượng tham số khác nhau (hoặc kiểu của tham số khác nhau), bạn đã *nạp chồng* phương thức đó. Đây là một mặt mạnh của đối tượng. Môi trường chạy thi hành của Java sẽ quyết định phiên bản nào của phương thức được gọi, dựa trên những thứ mà bạn truyền vào. Trong trường hợp các hàm tạo của chúng ta, nếu bạn không truyền vào bất cứ tham số nào thì JRE sẽ dùng hàm tạo không tham số. Nếu ta truyền vào hai đối tượng kiểu String thì môi trường chạy thi hành sẽ dùng phiên bản nhận hai tham số String. Nếu ta truyền vào các tham số có kiểu khác (hoặc là chỉ một String) thì môi trường chạy thi hành sẽ nhắc rằng không có hàm tạo nào nhận những tham số kiểu đó.

Bạn có thể nạp chồng bất cứ phương thức nào chứ không phải chỉ hàm tạo, việc này khiến cho việc tạo các giao diện thuận tiện cho người dùng sử dụng lớp của bạn trở nên dễ dàng. Hãy thử bằng cách bổ sung thêm phiên bản khác của phương thức addMoney() của bạn. Lúc này, phương thức đó nhận một tham số kiểu int. Tốt thôi, nhưng điều gì sẽ xảy ra nếu chúng ta muốn nạp thêm 100$ vào quỹ của Adult? Chúng ta phải gọi đi gọi lại phương thức này để thêm vào một loạt tờ giấy bạc có tổng giá trị là 100$. Thật là vô cùng bất tiện. Sẽ hay hơn nhiều nếu có thể truyền vào một mảng các phần tử int biểu thị cho một tập nhiều tờ giấy bạc. Ta hãy nạp chồng phương thức này để nhận tham số là một mảng. Đây là phương thức mà ta có:

```
public void addMoney(int bill) {

        Integer boxedBill = new Integer(bill);

        wallet.add(boxedBill);

}
```

Còn đây là phiên bản nạp chồng:

```
public void addMoney(int[] bills) {

        for (int i = 0; i < bills.length; i++) {
```

```
            int bill = bills[i];
            Integer boxedBill = new Integer(bill);
            wallet.add(boxedBill);
        }
}
```

Phương thức này trông rất giống với một phương thức addMoney() khác của chúng ta, nhưng nó nhận tham số là một mảng. Ta thử dùng phương thức này bằng cách biến đổi phương thức main() của Adult giống như sau:

```
public static void main(String[] args) {
        Adult myAdult = new Adult();

        myAdult.addMoney(new int[] { 1, 5, 10 });
        System.out.println(myAdult);
}
```

Khi chạy mã lệnh này, ta có thể thấy Adult có một wallet bên trong có 16$. Đây là giao diện tốt hơn nhiều. Nhưng chưa xong. Hãy nhớ rằng chúng là lập trình viên chuyên nghiệp, và ta muốn giữ cho mã lệnh của mình được sáng sủa. Bạn có thấy sự trùng lặp mã lệnh nào trong hai phương thức của chúng ta chưa? Hai dòng trong phiên bản đầu tiên xuất hiện nguyên văn trong phiên bản thứ hai. Nếu ta muốn thay đổi những gì ta làm khi thêm tiền vào, ta phải biến đổi mã lệnh ở hai nơi, đó là ý tưởng kém. Nếu ta bổ sung phiên bản khác của phương thức này để nó nhận tham số là ArrayList thay vì nhận một mảng, chúng ta phải biến đổi mã lệnh ở ba nơi. Điều đó nhanh chóng trở nên không thể chấp nhận nổi. Thay vào đó, chúng ta có thể *tái cấu trúc* (refactor) mã lệnh để loại bỏ sự trùng lặp. Ở phần tiếp

theo, chúng ta sẽ thực hiện một *thao tác tái cấu trúc* gọi là trích xuất phương thức (Extract Method) để hoàn thành việc này.

Tái cấu trúc khi nâng cao

Tái cấu trúc (refactoring) là quá trình thay đổi cấu trúc của mã lệnh hiện có mà không làm biến đổi chức năng của nó. Ứng dụng của bạn phải sản sinh cùng một kết quả đầu ra như cũ sau quá trình tái cấu trúc, nhưng mã lệnh của bạn sẽ trở nên rõ ràng hơn, sáng sủa hơn, và ít trùng lặp. Thường thuận lợi hơn để làm tái cấu trúc mã lệnh trước khi thêm một đặc tính (để bổ sung vào dễ hơn hoặc làm rõ hơn cần bổ sung thêm vào đâu), và sau khi thêm một đặc tính (để làm sạch sẽ những gì đã làm khi bổ sung vào). Trong trường hợp này, chúng ta đã thêm vào một phương thức mới và ta thấy một số mã lệnh trùng lặp. Chính là lúc để tái cấu trúc!

Đầu tiên, chúng ta cần tạo ra một phương thức nắm giữ hai dòng mã lệnh trùng lặp. Chúng ta gọi phương thức đó là addToWallet():

```
protected void addToWallet(int bill) {

        Integer boxedBill = new Integer(bill);

        wallet.add(boxedBill);

}
```

Chúng ta đặt chế độ truy nhập cho phương thức này là protected vì nó thực sự là phương thức phụ trợ nội tại của riêng chúng ta, chứ không phải là một phần của giao diện công cộng của lớp do chúng ta xây dựng. Bây giờ hãy thay các dòng mã lệnh trong phương thức bằng lời gọi đến phương thức mới:

```
public void addMoney(int bill) {

        addToWallet(bill);
```

```
}
```

Đây là phiên bản được nạp chồng:

```
public void addMoney(int[] bills) {
        for (int i = 0; i < bills.length; i++) {
                int bill = bills[i];
                addToWallet(bill);
        }
}
```

Nếu bạn chạy lại mã lệnh, bạn sẽ thấy cùng một kết quả. Kiểu tái cấu trúc này nên trở thành một thói quen, và Eclipse sẽ khiến nó trở nên dễ dàng hơn đối với bạn bằng cách đưa vào thêm nhiều công cụ tái cấu trúc tự động. Việc đi sâu tìm hiểu chi tiết về chúng nằm ngoài phạm vi của tài liệu hướng dẫn này, nhưng bạn có thể thử nghiệm chúng. Nếu chúng ta chọn hai dòng mã lệnh trùng lặp trong phiên bản đầu của addMoney(), chúng ta có thể nhấn chuột phải vào mã lệnh đã chọn và chọn **Refactor>Extract Method.** Eclipse sẽ từng bước dẫn dắt chúng ta qua quá trình tái cấu trúc. Đây là một trong những đặc tính mạnh nhất của IDE này.

Các thành phần của lớp

Các biến và phương thức mà chúng ta có trong Adult là các biến cá thể và phương thức cá thể. Mỗi đối tượng sẽ có các biến và phương thức cá thể như thế.

Bản thân các lớp cũng có các biến và phương thức. Chúng được gọi chung là *các thành phần của lớp,* và bạn khai báo chúng bằng từ khóa static. Sự khác nhau giữa các thành phần của lớp và các biến cá thể là:

- Tất cả các cá thể của một lớp sẽ chia sẻ chung một bản sao đơn lẻ của biến lớp (class variable).
- Bạn có thể gọi các phương thức lớp (class method) ngay trên bản thân lớp đó mà không cần có một cá thể của lớp.
- Các phương thức của cá thể có thể truy cập các biến lớp, nhưng các phương thức lớp không thể truy cập vào biến cá thể
- Các phương thức lớp chỉ có thể truy cập biến lớp.

Khi nào thì việc thêm các biến lớp hay phương thức lớp trở nên có ý nghĩa? Quy tắc xuyên suốt là hiếm khi làm điều đó, để bạn không lạm dụng chúng. Một số cách dùng thông thường là:

- Để khai báo các hằng số mà bất cứ cá thể nào của lớp cũng có thể sử dụng được
- Để theo vết "bộ đếm" các cá thể của lớp.
- Trên một lớp với các phương thức tiện ích mà không bao giờ cần đến một cá thể, vẫn giúp ích được (như là phương thức Collections.sort())

Các biến lớp

Để tạo một biến lớp, ta dùng từ khóa static khi khai báo:

accessSpecifier static *variableName*

[= *initialValue*];

JRE tạo một bản sao của các biến cá thể của lớp cho mọi cá thể của lớp đó. JRE chỉ sinh duy nhất một bản sao cho mỗi biến lớp, không phụ thuộc vào số lượng cá thể, khi lần đầu tiên nó gặp lời gọi lớp trong chương trình. Tất cả các cá thể sẽ chia sẻ chung (và có thể sửa đổi) bản sao riêng lẻ này. Điều này làm cho các biến lớp

trở thành một lựa chọn tốt để chứa *các hằng số* mà tất cả các cá thể đều có thể sử dụng.

Ví dụ, chúng ta đang dùng các số nguyên để mô tả "tờ giấy bạc" trong wallet của Adult. Điều đó hoàn toàn chấp nhận được, nhưng sẽ thật tuyệt nếu ta đặt tên cho các giá trị nguyên này để chúng ta có thể dễ dàng hiểu con số đó biểu thị cho cái gì khi ta đọc mã lệnh. Hãy khai báo một vài hằng số để làm điều này, ở chính ngay nơi ta khai báo các biến cá thể trong lớp của mình:

```
protected static final int ONE_DOLLAR_BILL = 1;

protected static final int FIVE_DOLLAR_BILL = 5;

protected static final int TEN_DOLLAR_BILL = 10;

protected static final int TWENTY_DOLLAR_BILL = 20;

protected static final int FIFTY_DOLLAR_BILL = 50;

protected static final int ONE_HUNDRED_DOLLAR_BILL = 100;
```

Theo quy ước, các hằng số của lớp đều được viết bằng chữ in hoa, các từ phân tách nhau bằng dấu gạch dưới. Ta dùng từ khóa static để khai báo chúng như là các biến lớp, và ta thêm từ khóa final vào để đảm bảo là không một cá thể nào có thể thay đổi chúng được (nghĩa là biến chúng trở thành hằng số). Bây giờ ta có thể biến đổi main() để thêm một ít tiền cho Adult của ta, sử dụng các hằng được đặt tên mới:

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

        Adult myAdult = new Adult();

        myAdult.addMoney(new int[] { Adult.ONE_DOLLAR_BILL,
Adult.FIVE_DOLLAR_BILL });

        System.out.println(myAdult);

}
```

Đọc đoạn mã này sẽ giúp làm sáng tỏ những gì ta bổ sung vào wallet wallet.

Các phương thức lớp

Như ta đã biết, ta gọi một phương thức cá thể như sau:

```
variableWithInstance.methodName();
```

Chúng ta đã gọi phương thức trên một biến có tên, biến đó chứa một cá thể của lớp. Khi bạn gọi một phương thức lớp, bạn sẽ gọi như sau:

```
ClassName.methodName();
```

Chúng ta không cần đến một cá thể để gọi phương thức này. Chúng ta đã gọi nó thông qua chính bản thân lớp. Phương thức main() mà ta đang dùng chính là một phương thức lớp. Hãy nhìn chữ ký của nó. Chú ý rằng nó được khai báo với từ khóa public static. Chúng ta đã biết định tố truy nhập này từ trước đây. Còn từ khóa static chỉ ra rằng đây là một phương thức lớp, đây chính là lý do mà các phương thức kiểu này đôi khi được gọi là các*phương thức static*. Chúng ta không cần có một cá thể của Adult để gọi phương thức main().

Chúng ta có thể xây dựng các phương thức lớp cho Adult nếu ta muốn, mặc dù thực sự không có lý do để làm điều đó trong trường hợp này. Tuy nhiên, để minh họa cách làm, ta sẽ bổ sung thêm một phương thức lớp tầm thường:

```
public static void doSomething() {

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

}

Thêm dấu chú thích vào các dòng lệnh hiện có của main() để loại bỏ chúng và bổ sung thêm dòng sau:

```
Adult.doSomething();

Adult myAdult = new Adult();

myAdult.doSomething();
```

Khi bạn chạy mã lệnh này, bạn sẽ thấy thông điệp tương ứng trên màn hình hai lần. Lời gọi thứ nhất gọi doSomething() theo cách điển hình khi gọi một phương thức lớp. Bạn cũng có thể gọi chúng thông qua một cá thể của lớp, như ở dòng thứ ba của mã lệnh. Nhưng đó không phải là cách hay. Eclipse sẽ cảnh báo cho bạn biết bằng cách dùng dòng gạch chân dạng sóng màu vàng và đề nghị bạn nên truy cập phương thức này theo "cách tĩnh" (static way), nghĩa là trên lớp chứ không phải là trên cá thể.

So sánh các đối tượng với toán tử ==

Có hai cách để so sánh các đối tượng trong ngôn ngữ Java:

- Toán tử ==
- Toán tử equals()

Cách đầu tiên, và là cách cơ bản nhất, so sánh các đối tượng theo tiêu chí *ngang bằng đối tượng* (object equality). Nói cách khác, câu lệnh:

```
a == b
```

sẽ trả lại giá trị true nếu và chỉ nếu a và b trỏ tới chính xác cùng một cá thể của một lớp (tức là cùng một đối tượng). Các kiểu nguyên thủy là ngoại lệ riêng. Khi ta so sánh hai kiểu nguyên thủy bằng toán tử ==, môi trường chạy thi hành của Java sẽ so sánh các giá trị của chúng (hãy nhớ rằng dù gì thì chúng cũng không phải là đối tượng thực sự). Hãy thử ví dụ này trong main() và xem kết quả trên màn hình.

```
int int1 = 1;

int int2 = 1;

Integer integer1 = new Integer(1);

Integer integer2 = new Integer(1);

Adult adult1 = new Adult();

Adult adult2 = new Adult();



System.out.println(int1 == int2);

System.out.println(integer1 == integer2);

integer2 = integer1;

System.out.println(integer1 == integer2);

System.out.println(adult1 == adult2);
```

Phép so sánh đầu tiên trả lại giá trị true, vì ta đang so sánh hai kiểu nguyên thủy có cùng giá trị. Phép so sánh thứ hai trả lại giá trị false, vì hai biến không tham chiếu đến cùng một đối tượng cá thể. Phép so sánh thứ ba trả lại giá trị true, vì bây giờ hai biến trỏ đến cùng một cá thể. Hãy thử với lớp của chúng ta, ta cũng nhận được giá trị false vì adult1 và adult2 không chỉ đến cùng một cá thể.

So sánh các đối tượng bằng equals()

Bạn gọi phương thức equals() trên một đối tượng như sau:

a.equals(b);

Phương thức equals() là một phương thức của lớp Object, vốn là lớp cha của mọi lớp trong ngôn ngữ Java. Điều đó có nghĩa là bất cứ lớp nào bạn xây dựng nên cũng sẽ thừa kế hành vi cơ sở equals() từ lớp Object. Hành vi cơ sở này không khác so với toán tử ==. Nói cách khác, mặc định là hai câu lệnh này cùng sử dụng toán tử == và trả lại giá trị false:

a == b;

a.equals(b);

Hãy nhìn lại phương thức spendMoney() của lớp Adult. Chuyện gì xảy ra đằng sau khi ta gọi phương thức contains()của đối tượng wallet của ta? Ngôn ngữ Java sử dụng toán tử == để so sánh các đối tượng trong danh sách với một đối tượng mà ta yêu cầu. Nếu Java thấy khớp, phương thức sẽ trả lại giá trị true; các trường hợp khác trả lại giá trị false. Bởi vì ta đang so sánh các kiểu nguyên thủy, Java có thể thấy sự trùng khớp dựa theo giá trị của các số nguyên (hãy nhớ rằng toán tử == so sánh các kiểu nguyên thủy dựa trên giá trị của chúng).

Thật tuyệt với đối với các kiểu nguyên thủy, nhưng liệu sẽ thế nào nếu ta so sánh nội dung của các đối tượng? Toán tử == không thể làm việc này. Để so sánh nội dung của các đối tượng, chúng ta phải đè chồng phương thức equals() của lớp mà a là cá thể của lớp đó. Điều đó có nghĩa là bạn tạo ra một phương thức có cùng chữ ký chính xác như chữ ký của phương thức của một trong các lớp bậc trên (superclasses), nhưng bạn sẽ triển khai thực hiện phương thức này khác với phương thức của lớp bậc trên. Nếu làm như vậy, bạn có thể so sánh nội dung của hai đối tượng để xem liệu chúng có giống nhau không chứ không phải là chỉ kiểm tra xem liệu hai biến đó có trỏ tới cùng một cá thể không.

Hãy thử ví dụ này trong main(), và xem kết quả trên màn hình:

```
Adult adult1 = new Adult();

Adult adult2 = new Adult();


System.out.println(adult1 == adult2);

System.out.println(adult1.equals(adult2));


Integer integer1 = new Integer(1);

Integer integer2 = new Integer(1);


System.out.println(integer1 == integer2);

System.out.println(integer1.equals(integer2));
```

Phép so sánh đầu tiên trả lại giá trị false vì adult1 và adult2 trỏ đến các cá thể khác nhau của lớp Adult. Phép so sánh thứ hai cũng trả lại giá trị false vì triển khai mặc định của equals() đơn giản là so sánh hai biến để xem liệu chúng có trỏ tới cùng một cá thể không. Nhưng hành vi mặc định này của equals() thường không phải là cái ta mong muốn. Chúng ta muốn so sánh nội dung của hai Adult để xem liệu chúng có giống nhau không. Ta có thể đè chồng phương thức equals() để làm điều này. Như bạn thấy kết quả của hai phép so sánh cuối cùng trong ví dụ trên, lớp Integer đè chồng lên phương thức này sao cho toán tử ==trả lại giá trị false, nhưng equals() lại so sánh các giá trị int đã bao bọc để xem có bằng nhau không. Chúng ta sẽ làm tương tự với Adult trong phần tiếp theo.

Đè chồng phương thức equals()

Để đè chồng phương thức equals() nhằm so sánh các đối tượng thì thực tế chúng ta phải đè chồng hai phương thức:

```
public boolean equals(Object other) {
        if (this == other)
                return true;

        if ( !(other instanceof Adult) )
                return false;

        Adult otherAdult = (Adult)other;
        if (this.getAge() == otherAdult.getAge() &&
                this.getName().equals(otherAdult.getName()) &&
                this.getRace().equals(otherAdult.getRace()) &&
                this.getGender().equals(otherAdult.getGender()) &&
                this.getProgress() == otherAdult.getProgress() &&
                this.getWallet().equals(otherAdult.getWallet()))
                return true;
        else
                return false;
}

public int hashCode() {
```

```
        return firstname.hashCode() + lastname.hashCode();
}
```

Chúng ta đè chồng phương thức equals() theo cách sau, là cách diễn đạt tiêu biểu của Java:

- Nếu đối tượng được so sánh chính là đối tượng so sánh thì hai đối tượng này là rõ ràng là bằng nhau, bởi vậy ta trả lại giá trị true
- Chúng ta kiểm tra để chắc chắn rằng đối tượng mà chúng ta sẽ đem so sánh là một cá thể của lớp Adult (nếu không thì hai đối tượng này không thể như nhau được)
- Chúng ta ép kiểu đối tượng được gửi đến thành một Adult để có thể gọi các phương thức phù hợp của nó
- Chúng ta so sánh các mảnh của hai Adult, chúng sẽ phải giống nhau nếu hai đối tượng là "bằng nhau" (dù theo bất cứ định nghĩa nào về phép bằng mà chúng ta sử dụng)
- Nếu bất cứ mảnh nào không bằng nhau thì chúng ta sẽ trả về giá trị là false; ngược lại trả về giá trị true

Lưu ý rằng chúng ta có thể so sánh age bằng toán tử == vì age là giá trị nguyên thủy. Chúng ta dùng phương thức equals() để so sánh các String, vì lớp đó đè chồng phương thức equals() để so sánh nội dung của các String (nếu ta dùng toán tử ==, chúng ta sẽ luôn nhận được kết quả trả về là false, vì hai String không bao giờ cùng là một đối tượng). Chúng ta làm tương tự với ArrayList, vì nó đè chồng phương thức equals() để kiểm tra xem hai danh sách có cùng các phần tử theo cùng thứ tự hay không, như thế là đủ cho ví dụ đơn giản của chúng ta.

Bất cứ khi nào bạn đè chồng phương thức equals(), bạn cũng nên viết đè chồng cả phương thức hashCode() nữa. Lý do vì sao lại như thế nằm ngoài phạm vi của tài liệu hướng dẫn này, nhưng hiện giờ, chỉ cần biết rằng ngôn ngữ Java dùng các giá trị được trả về từ phương thức hashCode() này để đặt các cá thể của lớp của bạn vào các sưu tập, các sưu tập này lại dùng thuật toán băm để sắp đặt các đối tượng (như HashMap). Quy tắc nghiêm ngặt và nhanh chóng để quyết định hashCode() phải trả lại giá trị gì (ngoài việc nó phải trả lại một số nguyên) là nó phải trả về:

- Cùng giá trị giống nhau cho cùng một đối tượng vào mọi thời điểm.
- Các giá trị bằng nhau đối với các đối tượng bằng nhau.

Thông thường, việc trả về giá trị mã băm của một vài hoặc toàn bộ các biến cá thể của một đối tượng là một cách thích hợp để tính toán ra mã băm. Một lựa chọn khác là chuyển đổi các biến thành String, nối chúng lại và sau đó trả về mã băm của String kết quả. Một lựa chọn khác nữa là để một hoặc một vài biến kiểu số với một hằng số nào đó để làm cho kết quả trở nên có tính duy nhất hơn nữa, nhưng việc này thường là quá mất công.

Đè chồng phương thức toString()

Lớp Object có một phương thức toString(), mọi lớp sau này do bạn tạo ra sẽ thừa kế nó. Nó trả về một biểu diễn dạng String của đối tượng của bạn và rất hữu dụng cho việc gỡ lỗi. Để xem phiên bản triển khai thực hiện mặc định của phương thức toString() làm gì thì ta hãy thử ví dụ sau trong main():

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

        Adult myAdult = new Adult();



        myAdult.addMoney(1);

        myAdult.addmoney(5);



        System.out.println(myAdult);

}
```

Kết quả ta nhận được trên màn hình sẽ như sau:

intro.core.Adult@b108475c

Phương thức println() gọi phương thức toString() của đối tượng đã truyền đến nó. Vì chúng ta còn chưa đè chồng phương thức toString() nên chúng ta sẽ nhận được kết quả đầu ra mặc định, đó là ID của đối tượng. Tất cả đối tượng đều có ID nhưng chúng không cho bạn biết gì nhiều về đối tượng. Sẽ tốt hơn khi bạn đè chồng lên phương thức toString() để đưa ra cho chúng ta một bức tranh được định dạng đẹp đẽ của các nội dung của đối tượng Adult():

```
public String toString() {

        StringBuffer buffer = new StringBuffer();



        buffer.append("And Adult with: " + "\n");

        buffer.append("Age: " + age + "\n");

        buffer.append("Name: " + getName() + "\n");

        buffer.append("Race: " + getRace() + "\n");

        buffer.append("Gender: " + getGender() + "\n");

        buffer.append("Progress: " + getProgress() + "\n");

        buffer.append("Wallet: " + getWallet());



        return buffer.toString();

}
```

Chúng ta tạo ra một StringBuffer để xây dựng một biểu diễn dạng String của đối tượng của chúng ta, sau đó trả về String này. Khi bạn chạy lại thì màn hình sẽ cho ta kết quả xuất ra đẹp đẽ như sau:

An Adult with:

Age: 25

Name: firstname lastname

Race: inuit

Gender: male

Progress: 0

Wallet: [1, 5]

Thế này rõ ràng là thuận tiện và có ích hơn là một ID đối tượng khó hiểu.

Các lỗi

Thật tuyệt nếu như mã lệnh của chúng ta không bao giờ có bất kỳ sai sót nào nhưng điều này là phi thực tế. Đôi khi mọi thứ không xuôi chèo mát mái như ta muốn, và có những khi vấn đề xảy ra còn tệ hơn là việc sản sinh ra những kết quả không mong muốn. Khi điều đó xảy ra, JRE sẽ *đưa ra một lỗi ngoại lệ* (throws an exception). Ngôn ngữ này có bao gồm những câu lệnh đặc biệt cho phép bạn bắt lỗi và quản lý lỗi một cách thích hợp. Sau đây là khuôn dạng chung của những câu lệnh này:

```
try {
        statement(s)
```

} catch (*exceptionType*

name) {

statement(s)

} finally {

statement(s)

}

Lệnh try bao bọc đoạn mã lệnh có thể gây ra lỗi. Nếu có lỗi, việc thi hành sẽ lập tức nhảy tới khối catch, cũng gọi là trình xử lý lỗi. Khi đã qua khối try và khối catch, việc thi hành sẽ tiếp tục đến khối finally, bất chấp việc liệu có lỗi xảy ra hay không. Khi bạn bắt được lỗi, bạn có thể thử phục hồi lại sau lỗi hoặc bạn có thể thoát ra khỏi chương trình (hay phương thức) một cách nhẹ nhàng.

Xử lý lỗi

Thử ví dụ sau trong main():

```
public static void main(String[] args) {
        Adult myAdult = new Adult();


        myAdult.addMoney(1);
        String wontWork = (String) myAdult.getWallet().get(0);
}
```

Khi chúng ta chạy mã lệnh này, chúng ta sẽ nhận được báo lỗi. Màn hình sẽ hiển thị như sau:

```
java.lang.ClassCastException
        at intro.core.Adult.main(Adult.java:19)
Exception in thread "main"
```

Lưu vết của ngăn xếp sẽ báo cho biết kiểu của lỗi và số hiệu của dòng xuất hiện lỗi. Hãy nhớ rằng chúng ta phải ép kiểu (cast) khi gỡ bỏ một Object khỏi sưu tập. Chúng ta có sưu tập các đối tượng Integer nhưng chúng ta đã thử lấy đối tượng thứ nhất bằng lệnh get(0) (trong đó 0 là chỉ số của phần tử đầu tiên trong danh sách vì danh sách bắt đầu từ 0, cũng như mảng) và ép kiểu nó thành String. Môi trường chạy thi hành của Java sẽ kêu ca vì lỗi này. Lúc đó thì chương trình sẽ ngừng. Hãy làm sao để nó chấm dứt nhẹ nhàng hơn bằng cách xử lý lỗi này:

```
try {
        String wontWork = (String) myAdult.getWallet().get(0);
} catch (ClassCastException e) {
        System.out.println("You can't cast that way.");
}
```

Tại đây chúng ta bắt lỗi và in ra một thông báo lịch sự. Một cách khác là ta có thể không làm gì trong khối catch, in ra thông báo lịch sự trong khối finally, nhưng điều đó không cần thiết. Trong một vài trường hợp, đối tượng lỗi (thường có tên khởi đầu bằng e hoặc ex, nhưng không nhất thiết phải thế) có thể cung cấp cho bạn nhiều thông tin hơn về lỗi, chúng có thể giúp bạn nắm được thông tin tốt hơn hoặc sửa chữa lỗi một cách dễ dàng.

Hệ phân cấp lỗi

Ngôn ngữ Java tích hợp chặt chẽ trọn vẹn một hệ phân cấp lỗi, điều đó có nghĩa là có rất nhiều kiểu lỗi. Ở mức cao nhất, một số lỗi được kiểm tra nhờ *trình biên dịch*, và một số lỗi khác, được gọi là RuntimeException, thì trình biên dịch không kiểm tra được. Quy tắc của Java là bạn phải bắt lỗi hoặc xác định rõ lỗi của mình. Nếu một phương thức có thể đưa ra một lỗi không phải RuntimeException, phương thức đó hoặc là phải xử lý lỗi, hoặc là phải chỉ rõ rằng phương thức gọi nó phải làm việc này. Bạn làm việc này với biểu thức throws trong chữ ký của phương thức. Ví dụ:

```
protected void someMethod() throws IOException
```

Trong mã lệnh của bạn, nếu bạn gọi một phương thức mà phương thức này chỉ rõ rằng nó đưa ra một hoặc các kiểu lỗi, bạn phải xử lý nó bằng một cách nào đó, hoặc bổ sung thêm một mệnh đề throws vào chữ ký của phương thức của bạn để chuyển tiếp đến ngăn xếp các lời gọi phương thức đã được gọi trong mã lệnh của bạn. Trong trường hợp xảy ra sự kiện lỗi, môi trường chạy thi hành của ngôn ngữ Java sẽ tìm trình xử lý lỗi ở đâu đó, tới tận *ngăn xếp* nếu không có trình xử lý nào ở nơi mà lỗi phát sinh ra. Nếu không tìm thấy trình xử lý lỗi cho đến khi truy đến đỉnh ngăn xếp thì môi trường chạy thi hành của Java sẽ lập tức dừng chương trình lại.

Một tin tốt lành là hầu hết các IDE (Eclipse hiển nhiên nằm trong số này) sẽ thông báo cho bạn nếu mã lệnh của bạn cần phải bẫy lỗi có thể được đưa ra bởi phương thức mà bạn gọi. Sau đó bạn có thể quyết định sẽ làm gì với nó.

Còn nhiều điều để nói về xử lý lỗi, dĩ nhiên là thế, nhưng lại quá nhiều để trình bày trong tài liệu này. Hy vọng những gì chúng ta đã bàn đến ở đây sẽ giúp bạn hiểu cái gì đang đợi bạn.

Các ứng dụng Java

Ứng dụng là gì?

Chúng ta đã thấy một ứng dụng rồi, dù là ứng dụng rất đơn giản. Lớp Adult có một phương thức main() ngay từ khi mới xuất hiện. Phương thức này cần thiết vì bạn cần một phương thức như vậy để Java thực thi mã lệnh của bạn. Thông thường, các đối tượng lĩnh vực ứng dụng của bạn sẽ không có phương thức main(). Ứng dụng Java điển hình thường bao gồm::

- Chỉ một lớp có phương thức main() để khởi động mọi thứ
- Một loạt các lớp khác để thực hiện công việc

Để minh họa chúng làm việc ra sao, chúng ta cần bổ sung thêm một lớp khác vào ứng dụng của mình. Lớp đó sẽ được gọi là "trình điều khiển" (driver).

Tạo ra lớp điều khiển

Lớp điều khiển của chúng ta có thể rất đơn giản:

```
package intro.core;



public class CommunityApplication {

        public static void main(String[] args) {

        }

}
```

Làm theo các bước sau đây để tạo lớp điều khiển và thực sự để nó điều khiển chương trình của chúng ta:

- Tạo lớp trong Eclipse bằng cách dùng các nút trên thanh công cụ New Java Class mà ta đã dùng để xây dựng lớp Adult trong phần Khai báo lớp.

- Đặt tên lớp là CommunityApplication, và đảm bảo là bạn đã đánh dấu tùy chọn để thêm phương thức main() vào lớp này. Eclipse sẽ tạo ra lớp cho bạn, bao gồm cả phương thức main().

- Xóa phương thức main() khỏi lớp Adult.

Tất cả những gì còn phải làm là đặt các thứ vào phương thức main() mới của chúng ta:

```
package intro.core;



public class CommunityApplication {

	public static void main(String[] args) {

		Adult myAdult = new Adult();

		System.out.println(myAdult.walk(10));

	}

}
```

Tạo một cấu hình khởi chạy mới trong Eclipse, giống như ta đã làm đối với lớp Adult trong phần Thực thi mã lệnh trong Eclipse, và chạy cấu hình này. Bạn sẽ thấy rằng đối tượng của chúng ta đã đi được 10 bước.

Bây giờ cái bạn đang có là một ứng dụng đơn giản bắt đầu bằng CommunityApplication.main(), và dùng đối tượng lĩnh vực ứng dụng Adult của chúng ta. Dĩ nhiên, các ứng dụng có thể phức tạp hơn thế, nhưng ý tưởng cơ bản vẫn như vậy. Không có gì là bất thường khi các ứng dụng Java có hàng trăm lớp. Một khi lớp điều khiển chính khởi động mọi thứ, chương trình sẽ chạy nhờ vào việc các lớp cộng tác với nhau để thực hiện công việc. Theo dõi việc thi hành chương trình có thể là khá khó khăn nếu bạn quen với các chương trình hướng thủ tục, khởi động từ điểm đầu và chạy cho đến cuối, nhưng nó sẽ dễ hiểu hơn khi thực hành.

Các tệp JAR

Bạn đóng gói ứng dụng Java thế nào để người khác có thể dùng được hoặc gửi mã lệnh cho người khác để họ có thể sử dụng cho chương trình riêng của họ (như một thư viện các đối tượng hữu ích hay như một khung công tác)? Bạn tạo một tệp Java Archive (JAR) để đóng gói mã lệnh sao cho những lập trình viên khác có thể tích hợp nó vào Java Build Path trong Eclipse, hoặc tích hợp vào đường dẫn lớp nếu họ dùng các công cụ dòng lệnh. Một lần nữa, Eclipse đã khiến mọi việc trở nên dễ dàng hơn. Tạo tệp JAR trong Eclipse (và nhiều IDE khác) chỉ trong chớp mắt:

1. Trong vùng làm việc của bạn, nhấn chuột phải vào gói intro.core và chọn mục **Export**

2. Chọn mục **JAR file** trong hộp thoại **Export**, sau đó nhấn chọn **Next**

3. Chọn vị trí mà bạn muốn đặt tệp JAR, và đặt cho tệp này bất cứ tên gì bạn muốn với phần đuôi mở rộng là .jar

4. Nhấn chọn **Finish**

Bạn có thể thấy tệp JAR vừa tạo ở vị trí mà bạn đã xác định. Khi đã có tệp JAR (của bạn hay từ một nguồn khác), bạn có thể sử dụng các lớp nằm trong tệp này khi viết mã lệnh nếu bạn đặt tệp JAR vào Java Build Path của mình trong Eclipse. Làm việc này cũng không tốn công sức lắm. Hiện tại thì không có mã lệnh nào ta cần thêm vào đường dẫn, nhưng ta hãy làm theo các bước mà bạn cần thực hiện để làm được điều ấy:

1. Nhấn chuột phải vào dự án Intro trong vùng làm việc của bạn, sau đó chọn **Properties**

2. Trong hộp thoại Properties, chọn phiếu Libraries

3. Bạn sẽ thấy các nút **Add JARs...** và **Add External JARs...**, bạn có thể dùng các nút này để đặt các tệp JAR vào Java Build Path của mình.

Một khi mã lệnh (ở đây là các tệp class) trong tệp JAR đã có mặt trong Java Build Path, bạn có thể dùng những lớp này trong mã Java của mình mà không gặp lỗi khi biên dịch. Nếu tệp JAR có tích hợp cả mã nguồn, bạn có thể kết hợp các file

mã nguồn đó với các tệp class trong đường dẫn của mình. Sau đó bạn có thể có các trợ giúp mã lệnh và thậm chí có thể mở mã lệnh ra xem.

Viết mã lệnh Java tốt

Giới thiệu mã lệnh Java

Bây giờ bạn đã biết khá nhiều về cú pháp của Java, nhưng đó không phải là lập trình thực sự chuyên nghiệp. Vậy điều gì tạo nên một chương trình Java "tốt"?

Có lẽ số lượng câu trả lời cho câu hỏi này cũng nhiều như số các lập trình viên Java chuyên nghiệp. Nhưng tôi có một số đề xuất mà tôi tin rằng hầu hết các lập trình viên Java chuyên nghiệp sẽ đồng ý cải tiến chất lượng của mã lệnh Java mà họ xử lý hàng ngày. Với chủ tâm bộc lộ hết, tôi phải nói rằng tôi nghiêng về ủng hộ các phương pháp lanh lẹn (agile) như Lập trình đỉnh cao (Extreme Programming - XP), do đó nhiều quan điểm của tôi về mã lệnh "tốt" nói chung là đã được cộng đồng phát triển theo phương pháp lanh lẹn và đặc biệt là XP công bố. Tôi nghĩ hầu hết các lập trình viên Java chuyên nghiệp giàu kinh nghiệm sẽ đồng ý với những điểm mà tôi sẽ trình bày trong phần này.

Hãy giữ cho các lớp nhỏ gọn

Chúng ta xây dựng lớp Adult đơn giản trong tài liệu này. Thậm chí sau khi chúng ta đã chuyển phương thức main() sang lớp khác, Adult cũng còn hơn 100 dòng mã lệnh. Lớp này có tới hơn hai mươi phương thức, và nó thực sự không làm được gì nhiều nếu so sánh với nhiều lớp và bạn có thể đã thấy (hay tạo ra) trong hoạt động nghề nghiệp. Đây là một lớp nhỏ. Không có gì bất thường khi bạn thấy có những lớp có từ 50 đến 100 phương thức. Điều gì khiến cho bạn nghĩ rằng ít hơn là tệ hơn ? Chẳng có gì cả. Điều quan trọng về các phương thức là bạn có những gì bạn cần. Nếu bạn cần vài phương thức trợ giúp, về bản chất thực hiện cùng một việc nhưng nhận các tham số khác nhau (như phương thức addMoney() chẳng hạn), thì đó là một lựa chọn hay. Hãy đảm bảo là chỉ hạn chế trong danh sách các phương thức bạn cần và đừng thêm nữa.

Thông thường, một lớp có quá nhiều phương thức sẽ có một vài phương thức không thuộc danh sách này vì rằng đối tượng khổng lồ thì cũng làm quá nhiều thứ. Trong cuốn *Tái cấu trúc* (Refactoring, xem Các tài nguyên), Martin Fowler gọi điều này là có mùi mã phương thức ngoại lai (Foreign Method code smell). Nếu bạn có một đối tượng với 100 phương thức, bạn nên suy nghĩ kỹ về việc liệu đối tượng này có phải thực sự là nhiều đối tượng hay không. Trong trường học các lớp đông thường gây phiền toái. Điều này cũng xảy ra đối với mã lệnh Java.

Hãy giữ cho các phương thức nhỏ gọn

Các phương thức nhỏ gọn cũng nên được ưu tiên hơn giống như các lớp nhỏ gọn với lý do tương tự.

Một trong những lời phàn nàn của các lập trình viên hướng đối tượng giàu kinh nghiệm đối với ngôn ngữ Java là nó cung cấp một đống hướng đối tượng nhưng không dạy họ cách thực hiện nó sao cho tốt. Nói cách khác, Java mang lại cho họ đủ rắc rối, dù ít nhất cũng không nhiều như ngôn ngữ C++. Nơi thường thấy điều này chính là trong một lớp với phương thức main() dài dằng dặc, hoặc chỉ một phương thức có tên là doIt(). Nếu chỉ vì bạn có thể nhồi tất cả mã lệnh của mình vào chỉ một phương thức trong một lớp thì điều đó không có nghĩa là bạn nên làm thế. Ngôn ngữ Java có nhiều gia vị cú pháp hơn nhiều ngôn ngữ hướng đối tượng khác nên cũng cần dài dòng đôi chút, nhưng không nên quá đà.

Hãy ngẫm nghĩ chốc lát về những phương thức cực kỳ dài ấy. Phải cuộn đến 10 trang màn hình đầy mã lệnh để luận ra cái gì đang xảy ra khiến cho thật khó khăn để hiểu cái gì đang xảy ra. Phương thức ấy làm gì? Bạn sẽ cần cả một cốc cà phê bự và nghiên cứu vài giờ để hiểu ra. Phương thức nhỏ, thậm chí là tý hon nữa là một bó lệnh dễ tiêu hóa. Hiệu suất chạy thi hành không phải là lý do để viết các phương thức nhỏ gọn. Khả năng dễ đọc hiểu mới là phần thưởng thực sự của nó. Điều này khiến cho mã lệnh của bạn dễ dàng bảo trì hơn và dễ thay đổi hơn khi bạn muốn thêm các đặc tính mới.

Hãy hạn chế sao cho mỗi phương thức thực hiện chỉ một việc.

Hãy đặt tên phương thức phù hợp

Mẫu viết mã lệnh hay nhất mà tôi đã từng xem qua (và tôi đã quên mất nguồn) được gọi là *các tên phương thức biểu lộ mục đích*. Trong hai cái tên phương thức dưới đây, cái nào dễ giải mãi hơn khi chỉ thoáng nhìn qua?

- a()
- computeCommission()

Câu trả lời thật hiển nhiên. Vì một vài lý do, các nhà lập trình dường như không thích đặt tên phương thức dài dòng. Tất nhiên một cái tên dài lố bịch có thể bất tiện, nhưng một cái tên dài đủ rõ ràng thường lại không lố bịch. Tôi chẳng gặp khó

khăn gì với một cái tên phương thức như aReallyLongMethodNameThatIsAbsolutelyClear(). Vào lúc 3:00 giờ sáng khi tôi thử tìm hiểu xem tại sao chương trình của tôi không chạy, nếu tôi gặp phải một phương thức có tên là a() thì tôi chỉ muốn nện cho ai đó một trận.

Hãy dành thêm vài phút để chọn một cái tên rất gợi tả; nếu có thể, bạn hãy cân nhắc việc đặt tên cho các phương thức theo cách thức sao cho mã lệnh của bạn đọc ra giống như lời nói thông thường vậy, thậm chí nếu điều này có nghĩa là cần thêm các phương thức phụ trợ để làm được việc đó. Ví dụ, hãy xem xét việc thêm vào một phương thức phụ trợ khiến cho đoạn mã lệnh này thêm dễ đọc hơn:

```
if (myAdult.getWallet().isEmpty()) {
        do something
}
```

Phương thức isEmpty() của ArrayList tự nó rất có ích, nhưng điều kiện logic trong câu lệnh if của chúng ta có thể được lợi từ phương thức hasMoney() của Adult như sau:

```
public boolean hasMoney() {
        return !getWallet().isEmpty();
}
```

Sau đây là câu lệnh if đọc giống lời nói thông thường hơn:

```
if (myAdult.hasMoney()) {
        do something
```

}

Kỹ thuật này đơn giản và có lẽ là bình thường trong trường hợp này, nhưng nó lại rất hữu hiệu khi mã lệnh trở nên phức tạp hơn.

Giữ cho số lượng các lớp ít nhất

Một trong những nguyên tắc chủ đạo để có được thiết kế đơn giản trong lập trình đỉnh cao (XP) là đạt được mục tiêu với ít lớp nhất có thể, nhưng không ít hơn. Nếu bạn cần một lớp khác, chắc chắn là nên thêm nó vào. Nếu thêm lớp khác làm cho mã lệnh của bạn đơn giản hơn hay làm cho bạn diễn dịch ý định của mình dễ dàng hơn thì hãy cứ tiếp tục thêm lớp vào. Nhưng chẳng có lý do gì để thêm các lớp chỉ để có chúng mà thôi. Thường khi bắt đầu dự án bạn có ít lớp hơn là khi hoàn thành xong xuôi, dĩ nhiên, nhưng cũng thường thì dễ tái cấu trúc mã lệnh của bạn thành nhiều lớp hơn là tích hợp chúng lại. Nếu bạn có một lớp có rất nhiều phương thức, phân tích xem liệu có phải có một lớp khác mắc bẫy vào đây và đang đợi được tách ra hay không. Nếu có, hãy tạo ra một đối tượng mới.

Trong hầu hết các dự án Java của tôi, không ai ngại xây dựng các lớp nhưng chúng tôi cũng luôn cố gắng giảm số lượng các lớp mà không làm cho ý định của mình kém tường minh.

Hãy giữ cho số lượng các chú thích ít nhất

Tôi thường viết các chú thích thừa trong mã lệnh của mình. Đọc chúng hệt như đọc cả một cuốn sách. Bây giờ tôi đã khôn ngoan hơn chút ít.

Tất cả các chương trình học tập về khoa học máy tính, tất cả các cuốn sách dạy lập trình và rất nhiều lập trình viên tôi quen biết đều khuyên bạn chú thích cho các mã lệnh. Trong một vài trường hợp, các chú thích thật sự có ích. Nhưng trong một số

trường hợp khác thì chính chúng lại làm cho việc bảo trì mã lệnh khó thêm. Thử nghĩ xem bạn phải làm gì khi bạn thay đổi mã lệnh. Có chú thích ở đấy không? Nếu có, tốt hơn là bạn phải thay đổi cả chú thích hoặc là nó sẽ trở nên lỗi thời kinh khủng, và thời gian trôi đi, thậm chí nó chả diễn tả gì mã lệnh cả. Theo kinh nghiệm của tôi thì nó chỉ tăng gấp đôi thời gian bảo trì của bạn mà thôi.

Quy tắc chủ đạo của tôi là: Nếu mã lệnh khó đọc và khó hiểu đến nỗi cần phải có chú thích thì tôi cần làm cho nó sáng sủa hơn đủ để không cần chú thích nữa. Có thể là nó quá dài hoặc làm quá nhiều việc. Nếu thế, phải làm cho nó trở nên đơn giản hơn. Có thể nó quá khó hiểu. Nếu thế, tôi sẽ bổ sung thêm các phương thức phụ trợ để làm nó dễ hiểu hơn. Thực tế, trong 3 năm lập trình Java với các thành viên trong đội, tôi có thể đếm số lượng chú thích tôi đã viết trên đầu ngón tay và ngón chân. Hãy làm mã lệnh của bạn sáng sủa hơn! Nếu bạn cần một bức tranh tổng thể về hệ thống, hoặc một thành phần cụ thể nào đó làm cái gì, hãy viết tài liệu ngắn gọn để mô tả nó.

Những chú thích dài dòng thường khó bảo trì, thường chúng không diễn giải ý định của bạn tốt bằng một phương thức được viết chuẩn, nhỏ gọn, và sẽ nhanh chóng trở nên lỗi thời. Đừng phụ thuộc quá nhiều vào các chú thích.

Hãy dùng một phong cách nhất quán

Viết mã lệnh theo phong cách gì thực sự là vấn đề về sự cần thiết và cái có thể chấp nhận được trong môi trường của bạn là gì. Tôi thậm chí không biết đến một phong cách mà tôi có thể gọi là “tiêu biểu”. Nó là chuyện sở thích cá nhân. Ví dụ, những dòng lệnh sau có thể làm tôi giật nảy mình cho đến khi tôi biến đổi đi:

```
public void myMethod()

{

        if (this.a == this.b)

        {

                statements
```

```
        }
}
```

Vì sao điều đó lại làm tôi băn khoăn? Vì cá nhân tôi không ưa kiểu viết mã lệnh thêm cả loạt dòng mới, mà theo ý kiến của tôi, không cần thiết. Trình biên dịch Java ghi nhận những dòng mã lệnh sau đây cũng giống như vậy, và tôi tiết kiệm được vài dòng:

```
public void myMethod() {
        if (this.a == this.b)
                statements
}
```

Không có cách viết nào là "đúng" hay là "sai". Chỉ đơn giản là nó ngắn hơn cách kia. Vậy điều gì xảy ra khi tôi viết mã lệnh với những người thích cách đầu tiên hơn ? Chúng tôi trao đổi về nó, chọn ra kiểu chúng tôi sẽ dùng và sau đó thì trung thành với nó. Chỉ có một quy tắc nghiêm ngặt là *hãy nhất quán*. Nếu những người tham gia thực hiện một dự án áp dụng các phong cách khác nhau, việc đọc mã lệnh sẽ khó khăn. Hãy chọn một kiểu thôi và đừng thay đổi nữa.

Tránh dùng lệnh switch

Một vài lập trình viên Java thích dùng lệnh switch. Tôi thì nghĩ lệnh đó cũng hay, nhưng sau đó nhận ra rằng switch thực ra chỉ là một loạt lệnh if, và như thế có nghĩa là logic điều kiện sẽ xuất hiện ở nhiều nơi trong mã lệnh của tôi. Đó là sự

lặp lại mã lệnh, và đấy là cái không chấp nhận được. Tại sao? Bởi vì có những mã lệnh giống nhau tại nhiều vị trí có thể khiến cho mã lệnh khó thay đổi. Nếu tôi có cùng lệnh switch ở tại 3 nơi và tôi muốn thay đổi cách xử lý một trường hợp cụ thể thì tôi phải thay đổi 3 đoạn mã lệnh.

Bây giờ, nếu như bạn có thể tái cấu trúc mã lệnh sao cho chỉ còn có một lệnh switch thì sao? Tuyệt vời! Tôi không tin có bất kỳ chuyện tệ hại gì khi dùng nó. Trong một vài trường hợp, dùng switch lại khiến mã lệnh sáng tỏ hơn là nhiều lệnh if lồng nhau. Nhưng nếu bạn thấy nó xuất hiện ở nhiều nơi có vấn đề thì bạn nên sửa đi. Cách dễ nhất để khỏi vấp phải vấn đề này là tránh dùng lệnh switch trừ khi nó là công cụ tốt nhất cho công việc đó. Theo kinh nghiệm của tôi thì điều này rất hiếm khi xảy ra.

Hãy để là public

Tôi để dành lời khuyến cáo gây nhiều tranh cãi nhất đến phút cuối cùng. Hãy chuẩn bị tinh thần nào và hít thở thật sâu.

Tôi tin bạn sẽ sai lầm khi đặt toàn bộ các phương thức của bạn ở chế độ truy nhập là public. Các biến cá thể đặt ở chế độ protected.

Dĩ nhiên, nhiều lập trình viên chuyên nghiệp sẽ rùng mình khi nghĩ đến điều đó vì nếu mọi thứ đều là công cộng thì ai cũng có thể biến đổi chúng được, có thể bằng cả những cách bất hợp pháp. Trong một thế giới mà mọi thứ đều có thể truy cập công cộng, bạn phải phụ thuộc vào tính nguyên tắc của người lập trình trong việc đảm bảo rằng mọi người không thể truy nhập vào những thứ mà họ không nên truy nhập khi họ không được phép. Nhưng trong thực tế lập trình, ít có điều gì gây phiền toái hơn là muốn truy cập một biến hay một phương thức mà bạn không nhìn thấy được. Nếu bạn hạn chế truy cập những thứ trong mã lệnh và bạn coi rằng những người khác sẽ không truy cập được, thì có lẽ bạn thực sự là đấng toàn năng. Đó là một giả định nguy hiểm trong mọi thời điểm.

Nỗi phiền toái này thường lộ ra khi bạn dùng mã lệnh của người khác. Bạn có thể thấy một phương thức thực hiện chính xác những gì bạn muốn làm, nhưng nó lại không cho phép truy cập công cộng. Đôi khi có lý do chính đáng để làm việc đó và việc hạn chế các truy nhập là có ý nghĩa. Thế nhưng, đôi khi lý do duy nhất để chế độ truy cập không là public là những người viết mã lệnh đã nghĩ rằng “Chả bao giờ có ai cần truy nhập vào đây cả”. Hoặc có thể họ đã nghĩ “Chẳng ai sẽ cần

truy nhập vì…” và tiếp theo, không có một lý do đủ vững chắc. Nhiều khi người ta sử dụng chế độ private vì nó có sẵn đó. Đừng làm vậy.

Hãy để các phương thức là public và các biến là protected cho đến khi bạn có lý do hợp lý để hạn chế truy nhập.

Theo dấu chân của Fowler

Bây giờ bạn đã biết cách làm thế nào để viết mã lệnh Java tốt và giữ gìn cho nó chuẩn mực.

Cuốn sách hay nhất trong ngành công nghiệp phần mềm là cuốn *“Tái cấu trúc”* của Martin Fowler (xem Các tài nguyên). Nó thậm chí còn rất hài hước nữa. *Tái cấu trúc* có nghĩa là thay đổi thiết kế của mã lệnh hiện có mà không làm biến đổi kết quả của nó. Fowler nói về các “mùi mã lệnh” đang xin được tái cấu trúc, và đi sâu vào chi tiết các kỹ thuật khác nhau (hoặc các kiểu tái cấu trúc khác nhau) để khắc phục chúng. Theo quan điểm của tôi, việc tái cấu trúc và khả năng viết mã lệnh kiểm thử đi trước (code test-first) (xem Các tài nguyên là những kỹ năng quan trọng nhất mà các lập trình viên mới vào nghề cần học. Nếu mọi người đã thạo cả hai, nó sẽ cách mạng hóa ngành công nghiệp này. Nếu bạn trở nên thành thạo ở cả hai kỹ năng, sẽ rất dễ kiếm việc làm, vì bạn sẽ có thể làm ra kết quả tốt hơn so với đa số người tìm việc.

Việc viết mã lệnh Java tương đối đơn giản. Viết mã lệnh Java “tốt” lại là mẹo mực. Bạn hãy tập trung để trở thành những người lành nghề.

Tổng kết

Trong tài liệu này, bạn đã học về lập trình hướng đối tượng, khám phá cú pháp của Java để giúp bạn tạo ra các đối tượng hữu dụng, và hiểu biết về một IDE giúp bạn kiểm soát môi trường phát triển của mình. Bạn có thể tạo ra các đối tượng có khả năng thực hiện tốt một số việc, mặc dù dĩ nhiên là không phải tất cả mọi việc mà bạn có thể tưởng tượng ra. Nhưng bạn có thể mở rộng hiểu biết của mình theo nhiều cách, bao gồm cả việc nghiên cứu kỹ về Java API và khám phá các khả năng khác của ngôn ngữ Java qua các bài hướng dẫn khác của developerWorks. Hãy xem phần Các tài nguyên để tìm tài liệu về tất cả các vấn đề này.

Ngôn ngữ Java chắc chắn không là hoàn hảo; mọi ngôn ngữ đều có các thói tật và mọi các lập trình viên đều có ngôn ngữ ưa thích của mình. Tuy nhiên, nền tảng Java là công cụ tốt có thể giúp bạn viết nên các chương trình chuyên nghiệp chuẩn mực ở mức yêu cầu cao.