

## Mục lục

GIỚI THIỆU .....	5
Chương 1. MỞ ĐẦU .....	7
1.1. KHÁI NIỆM CƠ BẢN .....	12
1.2. ĐỐI TƯỢNG VÀ LỚP.....	13
1.3. CÁC NGUYÊN TẮC TRỤ CỘT .....	15
Chương 2. NGÔN NGỮ LẬP TRÌNH JAVA.....	20
2.1. ĐẶC TÍNH CỦA JAVA.....	20
2.1.1. Máy ảo Java – Java Virtual Machine .....	21
2.1.2. Các nền tảng Java .....	23
2.1.3. Môi trường lập trình Java .....	23
2.1.4. Cấu trúc mã nguồn Java .....	24
2.1.5. Chương trình Java đầu tiên .....	25
2.2. BIẾN.....	27
2.3. CÁC PHÉP TOÁN CƠ BẢN.....	28
2.3.1. Phép gán.....	28
2.3.2. Các phép toán số học.....	28
2.3.3. Các phép toán khác .....	29
2.3.4. Độ ưu tiên của các phép toán.....	30
2.4. CÁC CẤU TRÚC ĐIỀU KHIỂN.....	30
2.4.1. Các cấu trúc rẽ nhánh.....	31
2.4.2. Các cấu trúc lặp .....	37
2.4.3. Biểu thức điều kiện trong các cấu trúc điều khiển .....	43
Chương 3. LỚP VÀ ĐỐI TƯỢNG .....	48
3.1. TẠO VÀ SỬ DỤNG ĐỐI TƯỢNG.....	49
3.2. TƯƠNG TÁC GIỮA CÁC ĐỐI TƯỢNG .....	51
Chương 4. BIẾN VÀ CÁC KIỂU DỮ LIỆU.....	57
4.1. BIẾN VÀ CÁC KIỂU DỮ LIỆU CƠ BẢN .....	58
4.2. THAM CHIẾU ĐỐI TƯỢNG VÀ ĐỐI TƯỢNG.....	59
4.3. PHÉP GÁN.....	62
4.4. CÁC PHÉP SO SÁNH.....	63

4.5. MẢNG .....	64
Chương 5. HÀNH VI CỦA ĐỐI TƯỢNG .....	70
5.1. PHƯƠNG THỨC VÀ TRẠNG THÁI ĐỐI TƯỢNG	70
5.2. TRUYỀN THAM SỐ VÀ GIÁ TRỊ TRẢ VỀ.....	71
5.3. CƠ CHẾ TRUYỀN BẰNG GIÁ TRỊ.....	73
5.4. ĐÓNG GÓI VÀ CÁC PHƯƠNG THỨC TRUY NHẬP	75
5.5. KHAI BÁO VÀ KHỞI TẠO BIẾN THỰC THỂ.....	79
5.6. BIẾN THỰC THỂ VÀ BIẾN ĐỊA PHƯƠNG .....	80
Chương 6. SỬ DỤNG THƯ VIỆN JAVA .....	85
6.1. ArrayList .....	85
6.2. SỬ DỤNG JAVA API.....	87
6.3. MỘT SỐ LỚP THÔNG DỤNG TRONG API.....	88
6.3.1. Math.....	88
6.3.2. Các lớp bọc ngoài kiểu dữ liệu cơ bản.....	89
6.3.3. Các lớp biểu diễn xâu kí tự.....	90
6.4. TRÒ CHƠI BẮN TÀU .....	91
Chương 7. THỪA KẾ VÀ ĐA HÌNH .....	103
7.1. QUAN HỆ THỪA KẾ .....	103
7.2. THIẾT KẾ CÂY THỪA KẾ .....	104
7.3. CÀI ĐỀ – PHƯƠNG THỨC NÀO ĐƯỢC GỌI? ...	107
7.4. CÁC QUAN HỆ IS-A VÀ HAS-A .....	108
7.5. KHI NÀO NÊN DỪNG QUAN HỆ THỪA KẾ?....	110
7.6. LỢI ÍCH CỦA QUAN HỆ THỪA KẾ.....	110
7.7. ĐA HÌNH.....	111
7.8. GỌI PHIÊN BẢN PHƯƠNG THỨC CỦA LỚP CHA	114
7.9. CÁC QUY TẮC CHO VIỆC CÀI ĐỀ.....	115
7.10. CHỒNG PHƯƠNG THỨC.....	116
7.11. CÁC MỨC TRUY NHẬP .....	117
Chương 8. LỚP TRỪU TƯỢNG VÀ INTERFACE.....	124
8.1. MỘT SỐ LỚP KHÔNG NÊN TẠO THỰC THỂ....	124
8.2. LỚP TRỪU TƯỢNG VÀ LỚP CỤ THỂ .....	126

8.3. PHƯƠNG THỨC TRỪU TƯỢNG.....	127
8.4. VÍ DỤ VỀ ĐA HÌNH.....	127
8.5. LỚP Object .....	131
8.6. ĐỐI KIỂU – KHI ĐỐI TƯỢNG MẤT HÀNH VI CỦA MÌNH	132
8.7. ĐA THỪA KẾ VÀ VẤN ĐỀ HÌNH THOI.....	135
8.8. INTERFACE.....	137
Chương 9. VÒNG ĐỜI CỦA ĐỐI TƯỢNG.....	143
9.1. BỘ NHỚ STACK VÀ BỘ NHỚ HEAP .....	143
9.2. KHỞI TẠO ĐỐI TƯỢNG .....	145
9.3. HÀM KHỞI TẠO VÀ VẤN ĐỀ THỪA KẾ.....	149
9.3.1. Gọi hàm khởi tạo của lớp cha.....	150
9.3.2. Truyền đối số cho hàm khởi tạo lớp cha .....	152
9.4. HÀM KHỞI TẠO CHỜNG NHAU .....	153
9.5. TẠO BẢN SAO CỦA ĐỐI TƯỢNG .....	154
9.6. CUỘC ĐỜI CỦA ĐỐI TƯỢNG.....	159
Chương 10. THÀNH VIÊN LỚP VÀ THÀNH VIÊN THỰC THỂ	164
10.1. BIẾN CỦA LỚP .....	164
10.2. PHƯƠNG THỨC CỦA LỚP .....	165
10.3. GIỚI HẠN CỦA PHƯƠNG THỨC LỚP .....	167
10.4. KHỞI TẠO BIẾN LỚP .....	169
10.5. MẪU THIẾT KẾ SINGLETON .....	170
10.6. THÀNH VIÊN BẮT BIẾN – final .....	171
Chương 11. NGOẠI LỆ .....	174
11.1. NGOẠI LỆ LÀ GÌ?.....	175
11.1.1. Tình huống sự cố.....	175
11.1.2. Xử lý ngoại lệ .....	177
11.1.3. Ngoại lệ là đối tượng.....	178
11.2. KHỐI try/catch.....	179
11.2.1. Bắt nhiều ngoại lệ .....	179
11.2.2. Hoạt động của khối try/catch .....	180
11.2.3. Khối finally – những việc dù thế nào cũng phải làm	182

11.2.4. Thứ tự cho các khối catch .....	183
11.3. NÉM NGOẠI LỆ.....	184
11.4. NẾ NGOẠI LỆ .....	185
11.5. NGOẠI LỆ ĐƯỢC KIỂM TRA VÀ KHÔNG ĐƯỢC KIỂM TRA	189
11.6. ĐỊNH NGHĨA KIỂU NGOẠI LỆ MỚI .....	190
11.7. NGOẠI LỆ VÀ CÁC PHƯƠNG THỨC CÀI ĐÈ .	191
Chương 12. CHUỖI HÓA ĐỐI TƯỢNG VÀ VÀO RA FILE	196
12.1. QUY TRÌNH GHI ĐỐI TƯỢNG.....	197
12.2. CHUỖI HÓA ĐỐI TƯỢNG.....	199
12.3. KHÔI PHỤC ĐỐI TƯỢNG.....	202
12.4. GHI CHUỖI KÍ TỰ RA TỆP VĂN BẢN.....	205
12.4.1. Lớp File.....	206
12.4.2. Bộ nhớ đệm .....	207
12.5. ĐỌC TỆP VĂN BẢN.....	207
12.6. CÁC DÒNG VÀO/RA TRONG Java API.....	209
Chương 13. LẬP TRÌNH TỔNG QUÁT VÀ CÁC LỚP COLLECTION	215
13.1. LỚP TỔNG QUÁT .....	217
13.2. PHƯƠNG THỨC TỔNG QUÁT.....	219
13.3. CÁC CẤU TRÚC DỮ LIỆU TỔNG QUÁT TRONG JAVA API	220
13.4. ITERATOR VÀ VÒNG LẶP FOR EACH .....	222
13.5. SO SÁNH NỘI DUNG ĐỐI TƯỢNG .....	224
13.5.1. So sánh bằng .....	224
13.5.2. So sánh lớn hơn/nhỏ hơn .....	226
13.6. KÍ TỰ ĐẠI DIỆN TRONG KHAI BÁO THAM SỐ KIỂU	228
Phụ lục A. DỊCH CHƯƠNG TRÌNH BẰNG JDK .....	233
Phụ lục B. PACKAGE – TỔ CHỨC GÓI CỦA JAVA....	236
Phụ lục C. BẢNG THUẬT NGỮ ANH-VIỆT.....	239
<b>TÀI LIỆU THAM KHẢO .....</b>	<b>241</b>

## GIỚI THIỆU

Phần mềm ngày càng lớn và phức tạp và đòi hỏi được cập nhật liên tục để đáp ứng những yêu cầu mới của người dùng. Phương pháp lập trình thủ tục truyền thống dần trở nên không đáp ứng được những đòi hỏi đó của ngành công nghiệp phần mềm. Lập trình hướng đối tượng đã ra đời trong bối cảnh như vậy để hỗ trợ sử dụng lại và phát triển các phần mềm qui mô lớn.

Giáo trình này cung cấp cho sinh viên các kiến thức từ cơ bản cho đến một số kỹ thuật nâng cao về phương pháp lập trình hướng đối tượng. Giáo trình dùng cho sinh viên ngành Công nghệ thông tin đã có kiến thức căn bản về lập trình. Giáo trình sử dụng ngôn ngữ lập trình Java để minh họa và đồng thời cũng giới thiệu một số kiến thức căn bản của ngôn ngữ này.

Các nội dung chính về phương pháp lập trình hướng đối tượng được trình bày trong giáo trình bao gồm lớp và đối tượng, đóng gói/che giấu thông tin, kế thừa và đa hình, xử lý ngoại lệ và lập trình tổng quát. Ngoài ra, giáo trình cũng trình bày các kiến thức về Java bao gồm các đặc trưng cơ bản của ngôn ngữ, các thư viện cơ bản và cách thức tổ chức vào/ra dữ liệu.

Thay vì cách trình bày theo tính hàn lâm về một chủ đề rộng, để thuận tiện cho giảng dạy, giáo trình chọn cách trình bày theo các bài học cụ thể được sắp xếp theo trình tự kiến thức từ cơ sở đến chuyên sâu. Mỗi chủ đề có thể được giảng dạy với thời lượng 2~3 giờ lý thuyết và giờ thực hành tương ứng. Ch-ương 2 và Ch-ương 6, với nội dung là các kiến thức cơ bản về ngôn ngữ lập trình Java, tuy cần thiết nhưng không phải nội dung trọng tâm của môn học Lập trình hướng đối tượng. Các chương này, do đó, nên để sinh viên tự học. Chương 9 và Chương 10 không nhất thiết phải được dạy thành những chủ đề độc lập mà có thể được tách rời các nội dung kiến thức và giới thiệu kèm theo các khái niệm hướng đối tượng có liên quan, hoặc yêu cầu sinh viên tự đọc khi cần đến các kiến thức này trong quá trình thực hành.

Tuy cuốn giáo trình này không trình bày sâu về lập trình Java, nhưng kiến thức về lập trình Java lại là cần thiết đối với sinh viên, ngay cả với mục đích thực hành môn học. Do đó, ngoài mục đích thực hành các nội dung liên quan đến lập trình hướng đối tượng, các bài tập thực hành của môn học này nên có thêm đóng vai trò định hướng và gợi ý giúp đỡ sinh viên tự học các chủ đề thuần túy Java mà giáo viên cho là cần thiết, chẳng hạn như học về vào ra dữ liệu đơn giản ngay từ tuần đầu tiên của môn học. Các định hướng này có thể được thể hiện ở những bài tập thực hành với những đoạn chương trình mẫu, hoặc yêu cầu tìm hiểu tài liệu API về một số lớp tiện ích. Một số bài tập cuối chương là ví dụ của dạng bài tập này.

Các thuật ngữ hướng đối tượng nguyên gốc tiếng Anh đã được chuyển sang tiếng Việt theo những cách khác nhau tùy các tác giả. Sinh viên cần biết thuật ngữ nguyên gốc tiếng Anh cũng như các cách dịch khác nhau đó để tiện cho việc sử dụng tài liệu tiếng Anh cũng như để liên hệ kiến thức giữa các tài liệu tiếng Việt. Vì lí do đó, giáo trình này cung cấp bảng thuật ngữ Anh-Việt với các cách dịch khác nhau tại Phụ lục C, bên cạnh Phụ lục A về công cụ lập trình JDK và Phụ lục B về tổ chức gói của ngôn ngữ Java.

Các tác giả chân thành cảm ơn PGS. TS. Nguyễn Đình Hóa, TS. Trương Anh Hoàng, TS. Cao Tuấn Dũng, TS. Đặng Đức Hạnh, cũng như các đồng nghiệp và sinh viên tại Khoa Công nghệ thông tin, Trường Đại học Công nghệ đã đọc bản thảo giáo trình và có các góp ý quý báu về nội dung chuyên môn cũng như cách thức trình bày. Tuy vậy, giáo trình vẫn còn nhiều khiếm khuyết, các tác giả mong tiếp tục nhận được góp ý để hoàn thiện trong tương lai.

Đại học Công nghệ - ĐHQGHN

## Chương 1. MỞ ĐẦU

Lập trình là công đoạn quan trọng chủ chốt và không thể thiếu để tạo ra sản phẩm phần mềm. Phần mềm càng trở nên đa dạng và ngành công nghiệp phần mềm càng phát triển thì người ta càng thấy rõ tầm quan trọng của phương pháp lập trình. Phương pháp lập trình tốt không chỉ đảm bảo tạo ra phần mềm tốt mà còn hỗ trợ thiết kế phần mềm có tính mở và hỗ trợ khả năng sử dụng lại các mô đun. Nhờ đó chúng ta có thể dễ dàng bảo trì, nâng cấp phần mềm cũng như giảm chi phí phát triển phần mềm.

Trong những thập kỷ 1970, 1980, phương pháp phát triển phần mềm chủ yếu là lập trình có cấu trúc (*structured programming*). Cách tiếp cận cấu trúc đối với việc thiết kế chương trình dựa trên chiến lược chia để trị: Để giải một bài toán lớn, chúng ta tìm cách chia nó thành vài bài toán nhỏ hơn và giải riêng từng bài; để giải mỗi bài, hãy coi nó như một bài toán mới và có thể tiếp tục chia nó thành các bài toán nhỏ hơn; cuối cùng, ta sẽ đi đến những bài toán có thể giải ngay được mà không cần phải chia tiếp. Cách tiếp cận này được gọi là lập trình từ trên xuống (*top-down programming*).

Lập trình từ trên xuống là một phương pháp tốt và đã được áp dụng thành công cho phát triển rất nhiều phần mềm. Tuy nhiên, cùng với sự đa dạng và phức tạp của phần mềm, phương pháp này bộc lộ những hạn chế. Trước hết, nó hầu như chỉ đáp ứng việc tạo ra các lệnh hay là các quy trình để giải quyết một bài toán. Dần dần, người ta nhận ra rằng thiết kế các cấu trúc dữ liệu cho một chương trình có tầm quan trọng không kém việc thiết kế các hàm/thủ tục và các cấu trúc điều khiển. Lập trình từ trên xuống không quan tâm đủ đến dữ liệu mà chương trình cần xử lý.

Thứ hai, với lập trình từ trên xuống, chúng ta khó có thể tái sử dụng các phần của chương trình này cho các chương trình khác. Bằng việc xuất phát từ một bài toán cụ thể và chia nó thành các mảnh sao cho thuận, cách tiếp cận này có xu hướng tạo ra một thiết kế đặc thù cho chính bài toán đó. Chúng ta khó có khả năng lấy một đoạn mã lớn từ một chương trình cũ lắp vào một dự án mới mà không phải sửa đổi lớn. Việc xây dựng các chương trình chất lượng cao là khó khăn và tốn kém, do đó những nhà phát triển phần mềm luôn luôn muốn tái sử dụng các sản phẩm cũ.

Thứ ba, môi trường hoạt động trong thực tế của các ứng dụng luôn thay đổi. Dẫn đến việc yêu cầu phần mềm cũng phải liên tục thay đổi theo để đáp ứng nhu cầu của người dùng nếu không muốn phần mềm bị đào thải. Do đó, một thiết kế linh hoạt mềm dẻo là cái mà các nhà phát triển phần mềm mong muốn. Phương pháp tiếp cận từ dưới lên (*bottom-up*) hỗ trợ tốt hơn cho linh hoạt mềm dẻo đó.

Trong thực tế, thiết kế và lập trình từ trên xuống thường được kết hợp với thiết kế và lập trình từ dưới lên. Trong tiếp cận từ dưới lên, từ các vấn đề mà ta đã biết

cách giải và có thể đã có sẵn các thành phần tái sử dụng được chúng ta xây dựng dần theo hướng lên trên, hướng đến một giải pháp cho bài toán tổng.

Các thành phần tái sử dụng được nên có tính mô-đun hóa cao nhất có thể. Mỗi mô-đun là một thành phần của một hệ thống lớn hơn, nó tương tác với phần còn lại của hệ thống theo một cách đơn giản và được quy ước chặt chẽ. Ý tưởng ở đây là một mô-đun có thể được "lắp vào" một hệ thống. Chi tiết về những gì xảy ra bên trong mô-đun không cần được xét đến đối với hệ thống nói chung, miễn là mô-đun đó hoàn thành tốt vai trò được giao. Đây gọi là **che giấu thông tin** (*information hiding*), một trong những nguyên lý quan trọng nhất của công nghệ phần mềm.

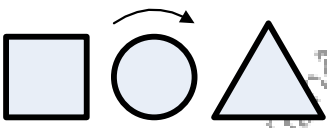
Một dạng thường thấy của các mô-đun phần mềm là nó chứa một số dữ liệu kèm theo một số hàm/thủ tục để xử lý dữ liệu đó. Ví dụ, một mô-đun số địa chỉ có thể chứa một danh sách các tên và địa chỉ, kèm theo là các hàm/thủ tục để thêm một mục tên mới, in nhãn địa chỉ... Với cách này, dữ liệu được bảo vệ vì nó chỉ được xử lý theo các cách đã được biết trước và được định nghĩa chặt chẽ. Ngoài ra, nó cũng tạo thuận lợi cho các chương trình sử dụng mô-đun này, vì các chương trình đó không phải quan tâm đến chi tiết biểu diễn dữ liệu bên trong mô-đun. Thông tin về biểu diễn dữ liệu được che giấu.

Các mô-đun hỗ trợ dạng che giấu thông tin này bắt đầu trở nên phổ biến trong các ngôn ngữ lập trình đầu thập kỷ 1980. Từ đó, một hình thức tiên tiến hơn của chính ý tưởng đó đã lan rộng trong ngành công nghệ phần mềm. Cách tiếp cận đó được gọi là **lập trình hướng đối tượng** (*object-oriented programming*), thường được gọi tắt là **OOP**.

Câu chuyện tưởng tượng sau đây<sup>1</sup> minh họa phần nào sự khác biệt giữa lập trình thủ tục và lập trình hướng đối tượng trong thực tế của ngành công nghệ phần mềm. Có hai lập trình viên nhận được cùng một đặc tả hệ thống và được yêu cầu xây dựng hệ thống đó, thi xem ai là người hoàn thành sớm nhất. Dậu là người chuyên dùng phương pháp lập trình thủ tục, còn Tuất quen dùng lập trình hướng đối tượng. Cả Dậu và Tuất đều cho rằng đây là nhiệm vụ đơn giản.

Đặc tả như sau:

Một phần mềm có giao diện đồ họa, nó vẽ một hình vuông, một hình tròn, và một hình tam giác. Khi người dùng click vào một hình, hình đó sẽ xoay 360° theo chiều kim đồng hồ và chương trình sẽ chơi một đoạn âm thanh AIF đặc thù của hình đó.



<sup>1</sup> Nguồn: Head First Java, 2<sup>nd</sup> Edition.



Dậu tính toán, "Chương trình này phải *làm* những gì? Ta cần đến những thủ tục nào?" Anh tự trả lời, "**xoay** và **chơi nhạc**." Và anh bắt tay vào viết các thủ tục đó. Chương trình không phải là một loạt các thủ tục thì nó là cái gì?

Trong khi đó, Tuất nghĩ, "Trong chương trình này có những *thứ* gì...đâu là những *nhân tố* chính?" Đầu tiên, anh ta nghĩ đến những **Hình vẽ**. Ngoài ra, anh còn nghĩ đến những đối tượng khác như *người dùng*, *âm thanh*, và sự kiện *click chuột*. Nhưng anh đã có sẵn thư viện mã cho mấy đối tượng đó, nên anh tập trung vào việc xây dựng các *Hình vẽ*.

Dậu đã quá thạo với công việc kiểu này rồi, anh ra bắt tay vào viết các thủ tục quan trọng và nhanh chóng hoàn thành hai thủ tục **xoay** (rotate) và **chơi nhạc** (playSound):

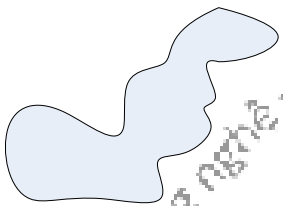
```
rotate(shapeNum) {  
    // cho hình xoay 360°  
}  
playSound(shapeNum) {  
    // dùng shapeNum để tra xem cần chơi file AIF nào  
    // và chơi file đó  
}
```

Còn Tuất ngồi viết ba lớp, mỗi lớp dành cho một hình.

Square	Circle	Triangle
<pre>rotate() {     // xoay hình vuông } playSound() {     // chơi file AIF cho     // hình vuông }</pre>	<pre>rotate() {     // xoay hình tròn } playSound() {     // chơi file AIF cho     // hình tròn }</pre>	<pre>rotate() {     // xoay hình tam giác } playSound() {     // chơi file AIF cho     // hình tam giác }</pre>

Dậu vừa nghĩ rằng mình đã thắng cuộc thì sếp nói "Về mặt kỹ thuật thì Dậu xong trước, nhưng ta phải bổ sung một chút xíu nữa vào chương trình." Hai người đã quá quen với chuyện đặc tả thay đổi – chuyện thường ngày trong ngành.

Đặc tả được bổ sung nội dung sau:

<p>Sẽ có một hình kiểu trùng biến hình (amoeba) trên màn hình cùng với các hình khác. Khi người dùng click vào hình amoeba, nó sẽ xoay và chơi một file nhạc dạng .hif.</p>	
---	--

Đối với Dậu, thủ tục rotate vẫn ổn, mã dùng một bảng tra cứu để khớp giá trị shapeNum với một hình đồ họa cụ thể. Nhưng *playSound* thì phải sửa.

Rất cực không phải sửa nghiêm trọng, nhưng Dậu vẫn thấy *không thoải mái khi phải động vào sửa phần mã đã được test xong từ trước*. Anh biết, dù quản lý dự án có nói gì đi chăng nữa, *đặc tả thay đổi suốt*.

Còn Tuất thì thần nhiên vừa nhâm nhi cà phê vừa viết một lớp mới. Điều anh thích nhất về OOP là anh không phải sửa gì ở phần mã đã được test và bàn giao. Anh nghĩ về những ích lợi của OOP và lẩm bẩm "Tính linh hoạt, khả năng mở rộng,...".

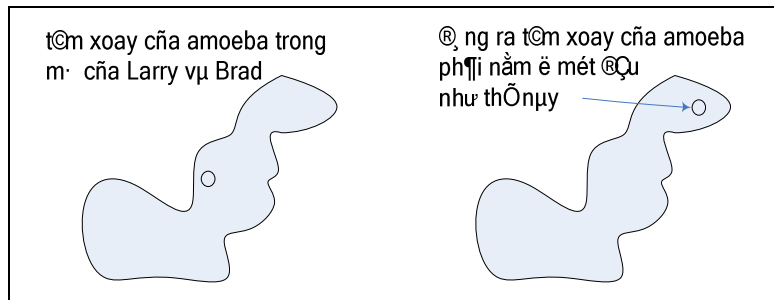
```

Amoeba
rotate() {
  // xoay hình amoeba
}
playSound() {
  // chơi file .hif cho
  // hình amoeba
}

```

Dậu cũng vừa kịp hoàn thành chỉ một lát trước Tuất. Nhưng nụ cười của anh vụt tắt khi nhìn thấy bộ mặt của sếp và nghe thấy giọng sếp vẻ thất vọng "không được rồi, amoeba thực ra không xoay kiểu này..."

Thì ra cả hai lập trình viên đều đã viết đoạn xoay hình theo cách: (1) xác định hình chữ nhật bao hình; (2) xác định tâm của hình chữ nhật đó và xoay hình quanh điểm đó. Nhưng hình trùng biến hình thì lại cần xoay quanh một điểm ở một *đầu mút*, như kiểu kim đồng hồ.



"Mình tèo rồi." Dậu ngán ngẩm. "Tuy là, ừm, có thể thêm một lệnh if/else nữa vào thủ tục rotate, rồi hard-code tâm xoay cho amoeba. Làm vậy chắc là sẽ không làm hỏng đoạn nào khác." Nhưng một giọng nói trong đầu Dậu thì thào, "*Nhằm to! Cậu có chắc là đặc tả sẽ không thay đổi lần nữa không đấy?*"

Cuối cùng Dậu chọn cách bổ sung tham số về tâm xoay vào cho thủ tục rotate. **Rất nhiều đoạn mã đã bị ảnh hưởng.** Phải test lại, dịch lại cả đồng mã. Có những đoạn trước chạy tốt thì nay không chạy được nữa.

```

rotate(shapeNum, xPt, yPt) {

```

Đại học Công nghệ - HOGHN

```

//nếu hình không phải amoeba,
// tính tâm xoay
// dựa trên một hình chữ nhật
// rồi xoay hình
//nếu không
// dựng xPt và yPt làm offset tâm xoay
// rồi xoay hình
}

```

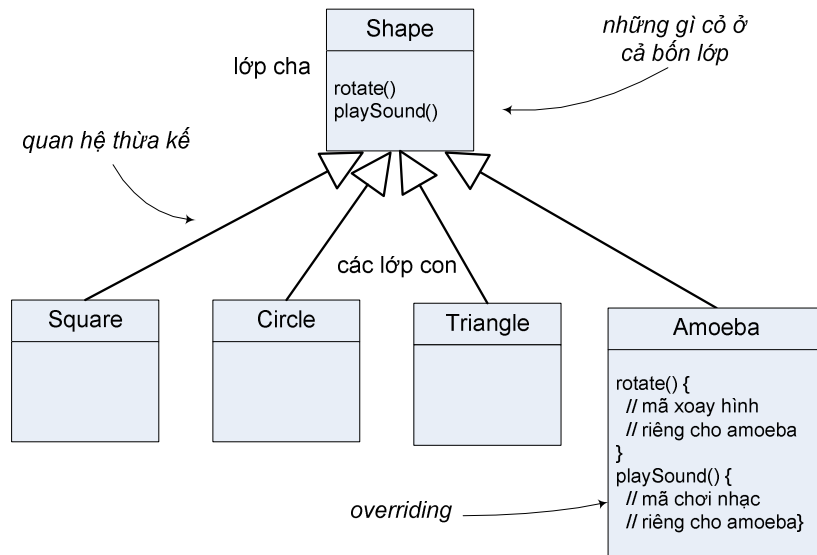
Còn Tuất, không chần chừ chút nào, anh sửa luôn phương thức rotate, nhưng chỉ sửa ở lớp Amoeba mà thôi. Tuất *không hề động đến các đoạn mã đã dịch, đã chạy và đã test* tại các phần khác trong chương trình. Để cho Amoeba một tâm xoay, anh thêm một **thuộc tính** mà tất cả các hình trùng biến hình sẽ có. Anh nhanh chóng sửa, test, và bàn giao mã cho sếp.

Amoeba
<pre> int x int y rotate() { // xoay hình amoeba // theo x và y của nó } playSound() { // chơi file .hif cho // hình amoeba } </pre>

"Không nhanh thế được!" Dậu tìm thấy một nhược điểm trong cách tiếp cận của Tuất, và anh chắc chắn nó sẽ giúp anh chuyển bại thành thắng. Dậu thấy mã của Tuất bị lặp, rotate có mặt ở cả bốn thứ hình, thiết kế này có gì hay ho khi phải bảo trì cả bốn phương thức rotate khác nhau?

Tuất giải thích: Dậu chưa nhìn thấy đặc điểm quan trọng của thiết kế, đó là quan hệ thừa kế. Bốn lớp có những đặc điểm chung, những đặc điểm đó được tách ra và đặt trong một lớp mới tên là Shape. Các lớp kia, mỗi lớp đều được xem là "thừa kế từ lớp Shape". Nói cách khác, nếu lớp Shape có những chức năng gì thì các lớp kia tự động có các chức năng đó.

Đại học Công nghệ - HHOUGHIN



Tuy nhiên, Amoeba có tâm xoay khác và chơi file nhạc khác. Lớp Amoeba **cài đặt** các hoạt động rotate và playSound đã được thừa kế từ Shape bằng cách định nghĩa lại các thủ tục này. Và khi chạy, hệ thống tự biết là cần dùng phiên bản được viết tại Amoeba thay vì dùng phiên bản thừa kế từ Shape. Đó là đặc điểm thú vị của phương pháp hướng đối tượng.

Khi ta cần yêu cầu một hình nào đó xoay, tam giác hay amoeba, ta chỉ việc gọi phương thức rotate cho đối tượng đó, và hệ thống sẽ tự biết phải làm gì, trong khi phần còn lại của chương trình không biết hoặc không quan tâm đến việc đối tượng đó xoay kiểu gì. Và khi ta cần bổ sung một cái gì đó mới vào chương trình, ta chỉ phải viết một lớp mới cho loại đối tượng mới, từ đó, các đối tượng mới sẽ có cách hành xử của riêng chúng.

### 1.1. KHÁI NIỆM CƠ BẢN

Hướng đối tượng là kĩ thuật mô hình hóa một hệ thống thế giới thực trong phần mềm dựa trên các đối tượng. **Đối tượng (object)** là khái niệm trung tâm của OOP, nó là một mô hình của một thực thể hay khái niệm trong thế giới thực. Việc mô hình hóa này bao gồm xác định các đối tượng tham gia bài toán – những cái làm nhiệm vụ gì đó hoặc bị làm gì đó. Lập trình theo kiểu hướng đối tượng là hoạt động định nghĩa các thể loại của các đối tượng đó ở hình thức các khuôn mẫu để tạo ra chúng.

Trong thời gian chạy, một chương trình OOP chính là một tập các đối tượng gửi thông điệp cho nhau để yêu cầu dịch vụ và thực hiện dịch vụ khi được yêu cầu. Việc một đối tượng thực hiện một dịch vụ có thể dẫn đến việc nó thay đổi trạng thái của bản thân. Một ví dụ có tính chất gần với thế giới thực: ông A đến rút tiền tại máy ATM. Ta có các đối tượng: ông A, máy ATM, cơ sở dữ liệu ngân hàng, và tài khoản của ông A. Trình tự diễn ra như sau: Ông A cho thẻ ngân hàng vào khe máy ATM;

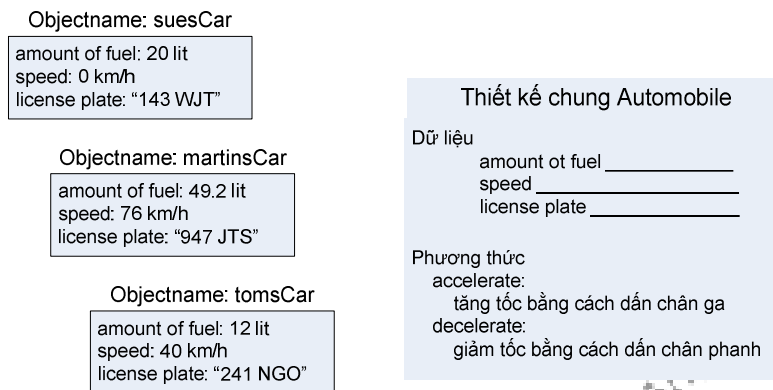
đối tượng ATM yêu cầu cơ sở dữ liệu ngân hàng cung cấp đối tượng tài khoản của ông A; ông A yêu cầu rút 100.000 đồng; đối tượng ATM yêu cầu đối tượng tài khoản trừ đi 100.000 đồng. Như vậy giao dịch này bao gồm chuỗi các yêu cầu dịch vụ và việc các đối tượng thực hiện các yêu cầu đó, đồng thời thay đổi trạng thái của mình (tài khoản ông A bị bớt tiền, ông A có thêm tiền, dữ liệu nhật trình ATM có thêm thông tin về một giao dịch).

## 1.2. ĐỐI TƯỢNG VÀ LỚP

Gần như bất cứ thứ gì cũng có thể được mô hình hóa bằng một đối tượng. Chẳng hạn, một màu, một hình vẽ, một cái nhiệt kế.

Mỗi đối tượng có một tập các **thuộc tính** (*attribute*) như các giá trị hay trạng thái để mô hình hóa đối tượng đó. Chẳng hạn, một cái nhiệt kế có thể có thuộc tính là vị trí hiện tại của nó và trạng thái hiện tại tắt hay bật, các thuộc tính một màu có thể là giá trị của ba thành phần RGB của nó. Một cái ô tô có các thuộc tính như: lượng xăng hiện có, tốc độ hiện tại, biển số.

Mỗi đối tượng có một tập các trách nhiệm mà nó thực hiện bằng cách cung cấp dịch vụ cho các đối tượng khác. Các dịch vụ này có thể cho phép truy vấn thông tin hoặc làm thay đổi trạng thái của đối tượng. Ví dụ, nhiệt kế cho phép truy vấn về tình trạng tắt/bật của nó; đáp ứng các yêu cầu về nhiệt độ hiện hành mà nó đo được, yêu cầu tắt/bật. Một cái ô tô cho phép tăng ga, giảm ga để tăng/giảm tốc độ di chuyển. Đối với thiết kế tốt, các đối tượng bên ngoài không phải quan tâm xem một đối tượng nào đó cài đặt một dịch vụ như thế nào, mà chỉ cần biết đối tượng đó cung cấp những dịch vụ nào (hay nó có những trách nhiệm gì). Chẳng hạn, người lái xe không cần biết cơ chế chuyển đổi từ lực nhấn lên chân đạp ga sang sự thay đổi về tốc độ của ô tô.



Hình 1.1: Các đối tượng ô tô và đặc điểm chung của chúng.

Trong mỗi ứng dụng, các đối tượng có đặc điểm tương tự nhau, chẳng hạn các tài khoản ngân hàng, các sinh viên, các máy ATM, những chiếc ô tô được xếp vào cùng một nhóm, đó là **lớp** (*class*). Mỗi lớp là đặc tả các đặc điểm của các đối tượng

thuộc lớp đó. Cụ thể, một định nghĩa lớp mô tả tất cả các thuộc tính của các đối tượng thành viên của lớp đó và các phương thức thực thi hành vi của các đối tượng đó. Ví dụ, ta có thể có nhiều đối tượng ô tô với thông số khác nhau về lượng xăng hiện có, tốc độ hiện tại, và biển số xe; định nghĩa lớp ô tô mô tả đặc điểm chung của các thông số đó cùng với các phương thức thực hiện các hoạt động tăng tốc, giảm tốc.

Automobile
- fuel: double - speed: double - license: String
+ accelerate (double pedalPressure): void + decelerate (double pedalPressure): void

Hình 1.2: Lớp Automobile vẽ bằng kí pháp UML

Quan hệ giữa lớp và đối tượng gần giống như quan hệ giữa kiểu dữ liệu và các biến thuộc kiểu dữ liệu đó. Các đối tượng được tạo ra khi chương trình chạy, và lớp là khuôn mẫu mà từ đó có thể tạo ra các đối tượng thuộc lớp đó. Mỗi đối tượng được tạo ra từ một lớp được gọi là một **thực thể** (*instance*) của lớp đó. Một chương trình khi được viết là sự kết hợp của các lớp khác nhau. Còn khi chạy, nó là một tập hợp các đối tượng hoạt động và tương tác với nhau, các đối tượng này được sinh ra từ các lớp cấu thành nên chương trình đó.

Mỗi đối tượng đều có một **thời gian sống**. Trong khi chương trình chạy, đối tượng được tạo và khởi tạo giá trị theo yêu cầu. Ngay khi một đối tượng được tạo ra, hệ thống tự động gọi một **hàm khởi tạo** (*constructor*) để khởi tạo giá trị cho các thuộc tính của đối tượng. Kể từ đó, đối tượng bắt đầu tồn tại, nó gửi và nhận các thông điệp, và cuối cùng thì nó bị hủy đi khi không còn cần đến nữa. Trong khi đối tượng tồn tại, nó giữ định danh và trạng thái của mình. Mỗi đối tượng có một định danh riêng và có bộ thuộc tính riêng, độc lập với các đối tượng khác thuộc cùng một lớp. Trong thực tế, mỗi đối tượng có vị trí riêng trong bộ nhớ.

Các đối tượng dùng các **thông điệp** (*message*) để liên lạc với nhau. Nhìn từ phương diện lập trình, việc gửi một thông điệp tới một đối tượng chính là gọi một phương thức của đối tượng đó, còn việc một đối tượng nhận được một thông điệp chính là việc một phương thức của nó được một đối tượng khác gọi. Chương trình khi chạy là một tập các đối tượng, mỗi đối tượng gửi thông điệp cho các đối tượng khác trong hệ thống và đáp ứng các thông điệp mà mình nhận được. Thông thường, một thông điệp được gửi bằng một lời gọi phương thức trong chương trình. Tuy nhiên, các thông điệp có thể xuất phát từ hệ điều hành hoặc môi trường chạy chương trình. Chẳng hạn khi người dùng click chuột vào một nút bấm tại một cửa sổ chương trình, một thông điệp sẽ được gửi đến đối tượng điều khiển nút bấm đó thông báo rằng cái nút đó đã bị nhấn.

### 1.3. CÁC NGUYÊN TẮC TRỤ CỘT

Lập trình hướng đối tượng có ba nguyên tắc trụ cột: đóng gói, thừa kế và đa hình, còn trừu tượng hóa là khái niệm nền tảng.

**Trừu tượng hóa** (*abstraction*) là một cơ chế cho phép biểu diễn một tình huống phức tạp trong thế giới thực bằng một mô hình được đơn giản hóa. Nó bao gồm việc tập trung vào các tính chất quan trọng của một đối tượng khi phải làm việc với lượng lớn thông tin. Ví dụ, đối với một con mèo trong ngữ cảnh một cửa hàng bán thú cảnh, ta có thể tập trung vào giống mèo, màu lông, cân nặng, tuổi, đã tiêm phòng đại hay chưa, và bỏ qua các thông tin khác như dung tích phổi, nồng độ đường trong máu, huyết áp, còn đối với một con mèo trong ngữ cảnh bệnh viện thú y thì lại là một chuyện khác. Các đối tượng ta thiết kế trong chương trình OOP sẽ là các trừu tượng hóa theo nghĩa đó, ta bỏ qua nhiều đặc điểm của đối tượng thực và chỉ tập trung vào các thuộc tính quan trọng cho việc giải một bài toán cụ thể. Người ta gọi một trừu tượng hóa là một mô hình của một đối tượng hoặc khái niệm trong thế giới thực.

Trừu tượng hóa là một trong những công cụ cơ bản của tất cả các phương pháp lập trình, không chỉ lập trình hướng đối tượng. Khi viết một chương trình giải một bài toán của thế giới thực, trừu tượng hóa là một cách để mô hình hóa bài toán đó. Ví dụ, khi ta viết một chương trình quản lý số địa chỉ, ta sẽ dùng các trừu tượng hóa như tên, địa chỉ, số điện thoại, thứ tự bảng chữ cái, và các khái niệm liên quan tới một số địa chỉ. Ta sẽ định nghĩa các thao tác để xử lý dữ liệu chẳng hạn như thêm một mục tên mới hoặc sửa một địa chỉ. Trong ngữ cảnh lập trình, trừu tượng hóa là mô hình hóa thế giới thực theo cách mà nó có thể được cài đặt dưới dạng một chương trình máy tính.

Phương pháp hướng đối tượng trừu tượng hóa thế giới thực thành các đối tượng và tương tác giữa chúng với các đối tượng khác. Việc mô hình hóa trở thành mô hình hóa các đối tượng tham gia bài toán – một cái nhiệt kế, một người chủ tài khoản ngân hàng, một số địa chỉ... mỗi đối tượng cần có đủ các thuộc tính và phương thức để thực hiện được tất cả các dịch vụ mà nó được yêu cầu.

**Đóng gói** (*encapsulation*): Các trừu tượng hóa của những gì có liên quan đến nhau được đóng gói vào trong một đơn vị duy nhất. Các trạng thái và hành vi của các trừu tượng hóa được bọc lại trong một khối gọi là lớp. Cụ thể, sau khi đã xác định được các đối tượng, rồi đến các thuộc tính và hành động của mỗi đối tượng, mục tiêu là đóng gói trong mỗi đối tượng các tính năng cần thiết để nó có thể thực hiện được vai trò của mình trong chương trình. Thí dụ, một đối tượng nhiệt kế cần có những gì cần thiết để có thể đo nhiệt độ, lưu trữ số liệu của các lần đo nhiệt độ trước và cho phép truy vấn các số liệu này.

Định nghĩa lớp là công cụ lập trình chính yếu cho việc thực hiện nguyên tắc đóng gói. Một lớp là mô tả về một tập hợp các đối tượng có cùng các thuộc tính, hành vi.

**Thuộc tính** (*attribute*) dùng để lưu trữ thông tin trạng thái của một đối tượng. Một thuộc tính có thể chỉ đơn giản là một biến Boolean lưu trữ trạng thái tắt hoặc bật, hay phức tạp hơn khi chính nó lại là một đối tượng khác. Các thuộc tính được khai báo trong định nghĩa lớp và được gọi là các **biến của thực thể** (*instance variable*), gọi tắt là **biến thực thể**. Chúng còn được gọi là các **thành viên dữ liệu** (*data member*), hay **trường** (*field*).

**Trạng thái** (*state*) phản ánh các giá trị hiện tại của các thuộc tính của một đối tượng và là kết quả của hành vi của đối tượng đó theo thời gian.

**Hành vi** (*behavior*) là hoạt động của một đối tượng mà có thể nhìn thấy được từ bên ngoài. Trong đó có việc đối tượng thay đổi trạng thái ra sao hoặc việc nó trả về thông tin trạng thái khi nó được thông điệp yêu cầu.

**Phương thức** (*method*) là một thao tác hay dịch vụ được thực hiện đối với đối tượng khi nó nhận thông điệp tương ứng. Các phương thức cài đặt hành vi của đối tượng và được định nghĩa trong định nghĩa lớp. Phương thức còn được gọi bằng các cái tên khác như: **hàm thành viên** (*member function*) – gọi tắt là 'hàm', **thao tác** (*operation*), **dịch vụ** (*service*).

Khái niệm đóng gói còn đi kèm với khái niệm **che giấu thông tin** (*information hiding*) nghĩa là che giấu các chi tiết bên trong của một đối tượng khỏi thế giới bên ngoài. Chẳng hạn khi dùng một cái cầu dao điện, đối với người sử dụng, nó chỉ là một cái hộp mà khi gạt cần sẽ có tác dụng ngắt và nối điện và cái hộp có khả năng tự ngắt điện khi quá tải. Người dùng không biết và không cần biết các mạch điện bên trong được thiết kế ra sao, cơ chế phát hiện quá tải như thế nào. Những chi tiết đó được giấu bên trong, còn từ bên ngoài ta chỉ nhìn thấy cầu dao là một cái hộp có cần gạt.

Nói theo phương diện lập trình, nhìn từ bên ngoài một mô-đun chỉ thấy được các giao diện. Các lập trình viên tự do cài đặt chi tiết bên trong, với ràng buộc duy nhất là tuân theo giao diện đã được quy ước từ trước. Ta có thể thực hiện nguyên tắc đóng gói với tất cả các ngôn ngữ lập trình hướng đối tượng cũng như các ngôn ngữ thủ tục. Tuy nhiên, chỉ các ngôn ngữ hướng đối tượng mới cung cấp cơ chế cho phép che giấu thông tin, ngăn không cho bên ngoài truy nhập vào chi tiết bên trong của mô-đun.

**Thừa kế** (*inheritance*) là quan hệ mang tính phân cấp mà trong đó các thành viên của một lớp được kế thừa bởi các lớp được dẫn xuất trực tiếp hoặc gián tiếp từ lớp đó. Đây là cơ chế cho phép định nghĩa một lớp mới dựa trên định nghĩa của một lớp có sẵn, sao cho tất cả các thành viên của lớp "cũ" (lớp cơ sở hay lớp cha) cũng có mặt trong lớp mới (lớp dẫn xuất hay lớp con) và các đối tượng thuộc lớp mới có thể được sử dụng thay cho đối tượng của lớp cũ ở bất cứ đâu. Thừa kế là một hình thức tái sử dụng phần mềm, trong đó một lớp mới được xây dựng bằng cách hấp thụ các thành viên của một lớp có sẵn và bổ sung những tính năng mới hoặc sửa tính năng có sẵn. Nói cách khác, xuất phát từ một lớp mô hình hóa một khái niệm tổng quát hơn,



chẳng hạn Shape, ta có thể dùng quan hệ thừa kế để xây dựng các lớp mô hình hóa các khái niệm cụ thể hơn, chẳng hạn Circle, Triangle. Bằng cách này, ta có thể sử dụng giao diện cũng như cài đặt của lớp cũ cho lớp mới.

**Đa hình** (*polymorphism*), theo nghĩa tổng quát, là khả năng tồn tại ở nhiều hình thức. Trong hướng đối tượng, đa hình đi kèm với quan hệ thừa kế và nó có nghĩa rằng cùng một cái tên có thể được hiểu theo các cách khác nhau tùy từng tình huống. Các đối tượng thuộc các lớp dẫn xuất khác nhau có thể được đối xử như nhau, như thể chúng là các đối tượng thuộc lớp cơ sở, chẳng hạn có thể đặt các đối tượng Triangle và Circle trong cùng một cấu trúc dữ liệu dành cho Shape, hoặc dùng cùng một lời gọi hàm rotate cho các đối tượng Triangle hay Circle. Và khi nhận được cùng một thông điệp đó, các đối tượng thuộc các lớp khác nhau hiểu nó theo những cách khác nhau. Ví dụ, khi nhận được thông điệp "rotate", các đối tượng Triangle và Amoeba thực hiện các phương thức rotate() khác nhau.

## Bài tập

- Điền từ thích hợp vào chỗ trống trong mỗi câu sau:
  - Quan hệ giữa một ngôi nhà và một bản thiết kế tương tự như quan hệ giữa một \_\_\_\_\_ với một lớp.
  - Khi mỗi đối tượng của một lớp giữ một bản riêng của một thuộc tính, trường dữ liệu đại diện cho thuộc tính đó được gọi là \_\_\_\_\_
- Chú trọng đến các tính chất quan trọng trong khi bỏ qua các chi tiết ít quan trọng được gọi là
  - Trừu tượng hóa
  - Đa hình
  - Đóng gói
  - Che giấu thông tin
- "Cùng một thông điệp được hiểu theo các cách khác nhau tùy theo đối tượng nhận được thông điệp đó thuộc lớp nào" là đặc điểm của khái niệm nào?
  - Đóng gói
  - Đa hình
  - Thừa kế
  - Tái sử dụng
- "Đối tượng thuộc lớp con có thể được đối xử như đối tượng thuộc lớp cha" là đặc điểm của khái niệm nào?
  - Trừu tượng hóa
  - Đa hình
  - Đóng gói
  - Che giấu thông tin
  - Thừa kế
- "Che đi các chi tiết cài đặt và chỉ cho thấy giao diện của mô-đun" là đặc điểm của khái niệm nào?
  - Trừu tượng hóa
  - Đa hình
  - Đóng gói
  - Tái sử dụng

Đại học Công nghệ - ĐHQGHN

## Chương 2. NGÔN NGỮ LẬP TRÌNH JAVA

Java được hãng Sun Microsystems<sup>2</sup> thiết kế năm 1991 như là một ngôn ngữ dành cho các chương trình nhúng (*embedded program*) chạy trên các thiết bị điện tử gia dụng như lò vi sóng và các hệ thống an ninh gia đình. Tuy nhiên, sự phát triển và lan rộng của Internet và World Wide Web (WWW) đã khiến Sun chuyển hướng Java từ một ngôn ngữ cho lập trình nhúng sang ngôn ngữ lập trình ứng dụng Web. Đến nay, Java đã trở thành ngôn ngữ lập trình ứng dụng phổ thông và là một trong những ngôn ngữ quan trọng nhất để phát triển các ứng dụng Web và Internet.

### 2.1. ĐẶC TÍNH CỦA JAVA

Java là ngôn ngữ hướng đối tượng. Các ngôn ngữ hướng đối tượng chia chương trình thành các mô-đun riêng biệt, được gọi là các đối tượng, chúng đóng gói dữ liệu và các thao tác của chương trình. Các khái niệm lập trình hướng đối tượng và thiết kế hướng đối tượng nói về phong cách tổ chức chương trình đang ngày càng được lựa chọn cho việc xây dựng các hệ thống phần mềm phức tạp. Không như ngôn ngữ C++, trong đó các đặc điểm hướng đối tượng được gắn thêm vào ngôn ngữ C, ngay từ đầu Java được thiết kế là một ngôn ngữ hướng đối tượng.

Java là ngôn ngữ có tính chắc chắn. Không như nhiều ngôn ngữ lập trình khác, lỗi trong các chương trình Java không gây sự cố hệ thống (*system crash*). Một số đặc tính của ngôn ngữ còn cho phép phát hiện nhiều lỗi tiềm tàng trước khi chương trình chạy.

Java có tính độc lập nền tảng (*platform independent*). Một nền tảng (*platform*) ở đây có nghĩa một hệ thống máy tính với hệ điều hành cụ thể, chẳng hạn như một hệ thống Windows hay Macintosh. Thương hiệu của Java là "Write once, run anywhere" (*Viết một lần, chạy bất cứ đâu*). Có nghĩa là một chương trình Java có thể chạy trên các nền tảng khác nhau mà không phải dịch lại. Một số ngôn ngữ bậc cao khác không có được đặc tính này. Tính khả chuyển, hay khả năng chạy trên hầu như tất cả các nền tảng, còn là nguyên do cho việc Java rất phù hợp cho các ứng dụng Web.

Java là ngôn ngữ phân tán. Các chương trình có thể được thiết kế để chạy trên mạng máy tính, một chương trình bao gồm những lớp đặt rải rác tại các máy khác nhau trong mạng. Bên cạnh ngôn ngữ, Java còn có một bộ sưu tập phong phú các thư viện mã đã được thiết kế để dùng trực tiếp cho các loại ứng dụng cụ thể, tạo

---

<sup>2</sup> Sun Microsystems đã nhập vào hãng Oracle từ năm 2010.

điều kiện thuận lợi cho việc xây dựng các hệ thống phần mềm cho Internet và WWW.

Java là một ngôn ngữ an toàn. Được thiết kế để dùng cho các mạng máy tính, Java có những đặc tính tự bảo vệ trước những phần mã không được tin cậy – những phần có thể đưa virus vào hệ thống hoặc gây rối hệ thống bằng cách nào đó. Ví dụ, khi một chương trình Web viết bằng Java đã được tải xuống trình duyệt máy tính, chúng bị cấm đọc và ghi thông tin tại máy tính.

### 2.1.1. Máy ảo Java – Java Virtual Machine

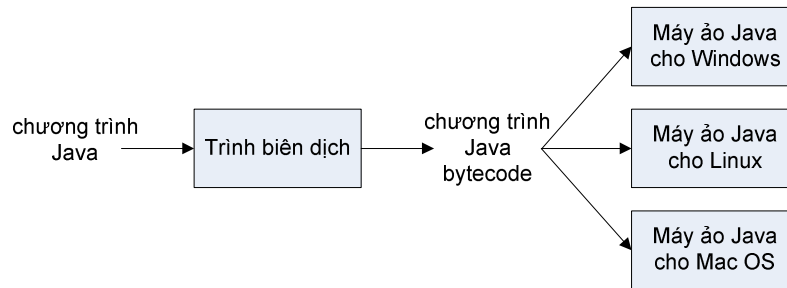
Ngôn ngữ máy bao gồm những **chỉ thị** (*instruction*) rất đơn giản mà CPU máy tính có thể thực hiện trực tiếp. Tuy nhiên, hầu hết các chương trình đều được viết bằng các ngôn ngữ lập trình bậc cao như Java hay C++. Một chương trình viết bằng ngôn ngữ bậc cao cần được dịch sang ngôn ngữ máy trước khi có thể được chạy trên máy tính. Việc dịch này do trình biên dịch thực hiện. Để chạy trên các loại máy tính với các ngôn ngữ máy khác nhau, cần đến các trình biên dịch phù hợp với loại ngôn ngữ máy đó.

Có một lựa chọn khác thay vì biên dịch chương trình viết bằng ngôn ngữ bậc cao. Thay vì dùng một trình biên dịch để dịch thẳng toàn bộ chương trình, ta có thể dùng một trình thông dịch, nó dịch từng chỉ thị một và chỉ dịch khi cần đến. Một trình thông dịch là một chương trình hoạt động gần như một CPU với một dạng chu trình nạp-và-thực-thi (*fetch-and-execute*). Để thực thi một chương trình, trình thông dịch lặp đi lặp lại chuỗi công việc: đọc một chỉ thị từ trong chương trình, xác định xem cần làm gì để thực hiện chỉ thị đó, và rồi thực hiện các lệnh mã máy thích hợp để thực hiện chỉ thị đó.

Một công dụng của trình thông dịch là để thực thi các chương trình viết bằng ngôn ngữ bậc cao, chẳng hạn như ngôn ngữ Lisp. Công dụng thứ hai là chúng cho phép ta chạy một chương trình ngôn ngữ máy dành cho một loại máy tính này trên một loại máy tính hoàn toàn khác. Ví dụ, có một chương trình tên là "Virtual PC" chạy trên các máy tính cài hệ điều hành Mac OS, đó là một trình thông dịch thực thi các chương trình mã máy viết cho các máy tính tương thích IBM PC. Nếu ta chạy "Virtual PC" trên một máy Mac OS, ta có thể chạy bất cứ chương trình PC nào, trong đó có cả các chương trình viết cho Windows.

Những người thiết kế Java chọn cách tổ hợp giữa trình biên dịch và trình thông dịch. Các chương trình viết bằng Java được biên dịch thành mã máy, nhưng đây là loại ngôn ngữ máy dành cho loại máy tính không tồn tại – loại máy "ảo" này được gọi là **Máy ảo Java** (*Java Virtual Machine – JVM*). Ngôn ngữ máy dành cho máy ảo Java được gọi là Java bytecode, hay ngắn gọn là bytecode. Để chạy được các chương trình Java trên một loại máy tính bất kì, người ta chỉ cần một trình thông dịch dành cho Java bytecode, trình thông dịch này giả lập máy ảo Java theo kiểu mà Virtual PC giả lập một máy tính PC. Máy ảo Java cũng chính là tên gọi dành cho trình thông

dịch bytecode thực hiện nhiệm vụ giả lập, do đó ta nói rằng một máy tính cần một máy ảo Java để chạy các chương trình Java.



Hình 2.1: Biên dịch và thông dịch đối với các chương trình Java

Tất nhiên, mỗi loại máy tính cần một trình thông dịch Java bytecode khác, nhưng một khi đã có một trình thông dịch như vậy, nó có thể chạy một chương trình Java bytecode bất kỳ. Và cũng chính chương trình Java bytecode đó có thể chạy trên bất cứ máy tính nào có một trình thông dịch Java bytecode. Đây chính là một trong các đặc điểm quan trọng của Java: một chương trình sau khi biên dịch có thể chạy trên nhiều loại máy tính khác nhau.

Có nhiều lý do tại sao nên dùng mã trung gian là Java bytecode thay cho việc phân phát mã nguồn chương trình Java và để cho mỗi người tự biên dịch nó sang mã máy của máy tính họ đang dùng. Thứ nhất, trình biên dịch là một chương trình phức tạp trong khi trình thông dịch chỉ là một chương trình nhỏ và đơn giản. Viết một trình thông dịch cho một loại máy tính mới dễ hơn là viết một trình biên dịch. Thứ hai, nhiều chương trình Java cần được tải xuống từ mạng máy tính. Việc này dẫn đến các mối quan tâm về bảo mật: ta không muốn tải về và chạy một chương trình sẽ phá hoại máy tính hoặc các file trong máy tính của ta. Trình thông dịch bytecode hoạt động với vai trò bộ đệm giữa máy tính của ta và chương trình ta tải về. Nó có thể bảo vệ ta khỏi các hành động nguy hiểm tiềm tàng của chương trình đó.

Khi Java còn là một ngôn ngữ mới, nó đã bị chỉ trích là chạy chậm. Do Java bytecode được thực thi bởi một trình thông dịch, có vẻ như các chương trình bytecode không bao giờ có thể chạy nhanh bằng các chương trình đã được biên dịch ra ngôn ngữ máy của chính máy tính mà chương trình đang chạy trên đó. Tuy nhiên, vấn đề này đã được giải quyết gần như toàn bộ bằng việc sử dụng trình biên dịch JIT (*just-in-time compiler*) cho việc thực thi Java bytecode. Trình biên dịch JIT dịch Java bytecode thành mã máy. Nó làm việc này trong khi thực thi chương trình. Cũng như một trình thông dịch thông thường, đầu vào cho một trình biên dịch JIT là một chương trình Java bytecode, và nhiệm vụ của nó là thực thi chương trình đó. Nhưng trong khi thực thi chương trình, nó dịch một phần của chương trình ra mã máy. Những phần được biên dịch này khi đó có thể được thực thi nhanh hơn là so

với khi chúng được thông dịch. Do một phần của chương trình thường được thực thi nhiều lần trong khi chương trình chạy, một trình biên dịch JIT có thể cải thiện đáng kể tổng thời gian chạy của chương trình.

### 2.1.2. Các nền tảng Java

Hãng Sun đã định nghĩa và hỗ trợ bốn bản Java hướng đến các môi trường ứng dụng khác nhau. Nhiều API (giao diện lập trình ứng dụng) của Java cũng được phân ra thành nhóm theo từng nền tảng. Bốn nền tảng đó là:

1. **Java Card** dành cho thẻ thông minh (*smartcard*) và các thiết bị nhỏ nhỏ tương tự. Thẻ SIM và thẻ ATM có sử dụng nền tảng này.
2. **Java Platform, Micro Edition** (Java ME) dành cho các môi trường hệ thống nhúng, chẳng hạn như điện thoại di động.
3. **Java Platform, Standard Edition** (Java SE) là nền tảng tiêu chuẩn, dành cho môi trường máy trạm, thường được dùng để phát triển Java application và Java applet. Đây là nền tảng được sử dụng rộng rãi, dùng để triển khai các ứng dụng nhẹ cho mục đích sử dụng tổng quát. Java SE bao gồm một máy ảo Java và một bộ các thư viện cần thiết cho việc sử dụng hệ thống file, mạng, giao diện đồ họa, v.v.. trong chương trình.
4. **Java Platform, Enterprise Edition** (Java EE) dành cho môi trường lớn và phân tán của doanh nghiệp hoặc Internet, thường dùng để phát triển các server. Nền tảng này khác với Java SE ở chỗ nó có thêm các thư viện với chức năng triển khai các phần mềm phân tán đa tầng có khả năng chịu lỗi.

Cuốn sách này sẽ chỉ dùng Java làm ngôn ngữ minh họa cho lập trình hướng đối tượng, nên chỉ giới hạn trong phạm vi Java SE và Java application.

### 2.1.3. Môi trường lập trình Java

Một môi trường lập trình Java thường bao gồm một số chương trình thực hiện các nhiệm vụ khác nhau để phục vụ công việc soạn, dịch, và chạy một chương trình Java.

Có thể sử dụng một chương trình soạn thảo văn bản dạng text bất kỳ để viết mã nguồn Java. Một chương trình Java bao gồm một hoặc nhiều định nghĩa lớp. Theo quy ước, mỗi định nghĩa lớp được đặt trong một file riêng. Theo quy tắc một file mã nguồn Java chỉ được chứa nhiều nhất một định nghĩa lớp với từ khóa **public** – ý nghĩa của từ khóa này sẽ được nói đến sau. File chứa định nghĩa lớp phải có tên trùng với tên của lớp public đặt trong file đó, ví dụ file HelloWorld.java chứa lớp public có tên HelloWorld, file HelloWorldApplet.java chứa lớp public có tên HelloWorldApplet.

Java là ngôn ngữ phân biệt chữ hoa chữ thường. Do đó nếu lớp HelloWorld được đặt trong file helloworld.java thì sẽ gây lỗi khi biên dịch.

Những người mới bắt đầu sử dụng Java nên bắt đầu từ việc viết chương trình bằng một phần mềm soạn thảo đơn giản và sử dụng các công cụ dòng lệnh trong bộ JDK để dịch và chạy chương trình. Ngay cả những lập trình viên thành thạo đôi khi cũng sử dụng cách này.

Các bước cơ bản để xây dựng và thực thi một chương trình Java:

- **Soạn thảo:** Mã nguồn chương trình được viết bằng một phần mềm soạn thảo văn bản dạng text và lưu trên ổ đĩa. Ta có thể dùng những phần mềm soạn thảo văn bản đơn giản nhất như Notepad (trong môi trường Windows) hay emacs (trong môi trường Unix/Linux), hoặc các công cụ soạn thảo trong môi trường tích hợp để viết mã nguồn chương trình. Mã nguồn Java đặt trong các file với tên có phần mở rộng là .java.
- **Dịch:** Trình biên dịch Java (javac) lấy file mã nguồn và dịch thành các lệnh bằng bytecode mà máy ảo Java hiểu được, kết quả là các file có đuôi .class.
- **Nạp và chạy:** Trình nạp Java (java) sẽ dùng máy ảo Java để chạy chương trình đã được dịch ra dạng bytecode.

Để thuận tiện và tăng năng suất cho việc lập trình, người ta dùng các **môi trường lập trình tích hợp** (IDE – *integrated development environment*). Trong đó, các bước dịch và chạy thường được kết hợp và thực hiện tự động, tất cả các công đoạn đối với người dùng chỉ còn là việc chạy các tính năng trong một phần mềm duy nhất. Trong số các IDE phổ biến nhất cho Java có Eclipse, NetBean và JBuilder.

Tuy IDE rất hữu ích cho các lập trình viên, những người mới làm quen với ngôn ngữ nên tự thực hiện các bước dịch và chạy chương trình thay vì thông qua các chức năng của IDE. Như vậy, người học mới có thể nắm được bản chất các bước của quá trình xây dựng chương trình, hiểu được bản chất và đặc điểm chung của các IDE, tránh tình trạng bị phụ thuộc vào một IDE cụ thể. Do đó, cuốn sách này không hướng dẫn về một IDE nào mà chỉ dùng công cụ chạy từ dòng lệnh trong bộ JDK.

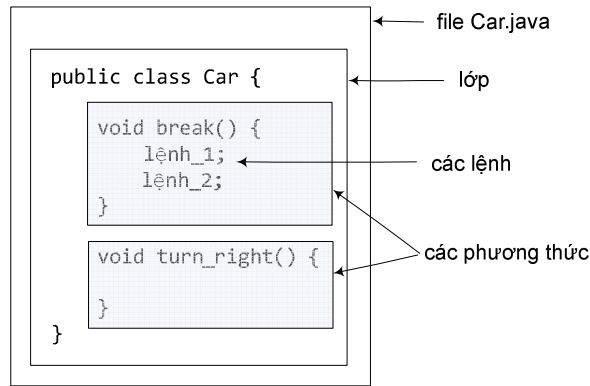
#### 2.1.4. Cấu trúc mã nguồn Java

Mỗi file mã nguồn (tên file có đuôi .java) chứa một định nghĩa lớp (class). Mỗi lớp đại diện cho một mảnh của chương trình, một chương trình nhỏ có thể chỉ bao gồm một lớp. Định nghĩa lớp phải được bọc trong một cặp ngoặc { }.

Mỗi lớp có một vài phương thức. Trong lớp Car, phương thức break chứa các lệnh mô tả chiếc xe con cần phanh như thế nào. Các phương thức của lớp nào phải được khai báo ở bên trong định nghĩa lớp đó.

Bên trong cặp ngoặc { } của một phương thức, ta viết một chuỗi các lệnh quy định hoạt động của phương thức đó. Có thể tạm coi phương thức của Java gần giống như hàm hay chương trình con.





Hình 2.2: Cấu trúc mã Java.

### 2.1.5. Chương trình Java đầu tiên

Chương trình đơn giản trong Hình 2.3 sẽ hiện ra màn hình dòng chữ “Hello, world!”. Trong chương trình có những chi tiết mà tại thời điểm này ta chưa cần hiểu rõ và có thể để đến vài chương sau. Ta sẽ xem xét từng dòng.

---

```
// A program to display the message
// "Hello, world!" on standard output

public class HelloWorld {

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

} // end of class HelloWorld
```

---

Hình 2.3: Chương trình Java đầu tiên.

Hai dòng đầu tiên bắt đầu bằng chuỗi // là các dòng chú thích chương trình. Đó là kiểu chú thích dòng đơn. Các dòng chú thích không gây ra hoạt động gì của chương trình khi chạy, trình biên dịch bỏ qua các dòng này. Ngoài ra còn có dạng chú thích kéo dài trên nhiều dòng, sử dụng /\* và \*/ để đánh dấu điểm bắt đầu và điểm kết thúc đoạn chú thích.

Dòng thứ ba, `public class HelloWorld {` tuyên bố rằng đây là định nghĩa về một lớp có tên HelloWorld. "HelloWorld" là tên của lớp và cũng là tên của chương trình, tuy rằng không phải lớp nào cũng là một chương trình như trong ví dụ này.

Để một lớp là một chương trình, ta cần viết cho lớp đó một phương thức có tên main với định nghĩa có dạng sau. Đây là cú pháp bắt buộc của phương thức main():

```
public static void main(String[] args) {
    <các lệnh>
}
```

Khi ta yêu cầu trình thông dịch Java chạy chương trình HelloWorld, máy ảo Java sẽ tìm lớp có tên HelloWorld, rồi nó tìm phương thức main() với cú pháp bắt buộc như trên. Đây là nơi chương trình bắt đầu thực hiện và kết thúc, máy ảo lần lượt chạy các lệnh ở bên trong cặp ngoặc { } của phương thức main(). Phương thức main() có thể gọi các phương thức khác được định nghĩa trong lớp hiện tại hoặc trong các lớp khác, nó quyết định chuỗi công việc mà máy tính sẽ thực hiện khi chương trình chạy. Mỗi ứng dụng Java phải có ít nhất một lớp, và có một phương thức main() trong một lớp nào đó.

Từ khóa **public** tại dòng đầu tiên của main() có nghĩa rằng đây là phương thức có mức truy nhập public (*công khai*) – phương thức có thể được gọi từ bất cứ đâu trong mã chương trình. Thực tế là main() được gọi từ trình thông dịch – một thứ nằm ngoài chương trình. Từ khóa **static** sẽ được giải thích trong các chương sau. Từ khóa **void** có nghĩa rằng phương thức main() không có kết quả trả về. Tham số String[] args của hàm main() là mảng chứa các chuỗi ký tự được nhập vào dưới hình thức tham số dòng lệnh khi ta chạy chương trình từ cửa sổ lệnh (*console*).

Thân phương thức main(), cũng như bất kỳ một hàm nào khác, được bắt đầu và kết thúc bởi cặp ngoặc { }, bên trong đó là chuỗi các lệnh mà khi chương trình chạy chúng sẽ được thực hiện tuần tự từ lệnh đầu tiên cho đến lệnh cuối cùng. Mỗi lệnh Java đều kết thúc bằng một dấu chấm phẩy. Phương thức main() trong ví dụ đang xét có chứa đúng một lệnh. Lệnh này có tác dụng hiển thị thông điệp ra đầu ra chuẩn (*standard output*). Đó là ví dụ về một lệnh gọi hàm. Lệnh này gọi hàm System.out.println(), một hàm có sẵn trong thư viện chuẩn Java, yêu cầu hàm này thực hiện việc hiển thị thông điệp. Nói theo cách của lập trình hướng đối tượng, lệnh đó chính là một thông điệp gửi tới đối tượng có tên System.out yêu cầu in ra đầu ra chuẩn một chuỗi ký tự. Khi chạy chương trình, thông điệp "Hello, world!" (không có nháy kép) sẽ được hiển thị ra đầu ra chuẩn. Đầu ra chuẩn là cái gì thì tùy vào việc chương trình đang chạy ở loại thiết bị nào, platform nào.

Lưu ý rằng trong Java, một hàm không thể tồn tại độc lập. Nó phải thuộc về một lớp nào đó. Một chương trình được định nghĩa bởi một lớp public có dạng

```
public class <program-name> {
    <định nghĩa các thành viên dữ liệu hoặc phương thức>
    public static void main(String[] args) {
        <các lệnh>
    }
    <định nghĩa các thành viên dữ liệu hoặc phương thức>
}
```

Trong đó, <program-name> là tên lớp, tên chương trình, và cũng là tên file mã nguồn. public là từ khóa cần được đặt đầu khai báo các lớp chương trình. Những

lớp được khai báo với từ khóa này cần được đặt tại một file có tên file trùng với tên lớp, chính xác đến cả chữ hoa hay chữ thường. Ví dụ, lớp HelloWorld ở trên nằm trong file mã nguồn có tên HelloWorld.java. Sau khi biên dịch file mã nguồn HelloWorld.java, ta sẽ được file bytecode HelloWorld.class – file có thể chạy bằng trình thông dịch Java.

Phụ lục A hướng dẫn chi tiết về cách sử dụng công cụ dòng lệnh JDK để dịch và chạy chương trình. Đây là bộ phần mềm miễn phí, có thể được tải về từ trang web của Oracle<sup>3</sup>.

## 2.2. BIẾN

Trong một chương trình, biến là tên của một vùng bộ nhớ được dùng để lưu dữ liệu trong khi chương trình chạy. Dữ liệu lưu trong một biến được gọi là giá trị của biến đó. Chúng ta có thể truy nhập, gán hay thay đổi giá trị của các biến, khi biến được gán một giá trị mới, giá trị cũ sẽ bị ghi đè lên.

Java yêu cầu mỗi biến trước khi dùng phải được khai báo. Ví dụ:

```
int x;    // khai báo  
x = 10;  // sử dụng
```

Các biến được khai báo ở trong một hàm là **biến địa phương**. Nên khai báo biến địa phương ngay trước khi sử dụng hoặc ở đầu khối mã chương trình được đóng khung trong cặp ngoặc { }. Biến địa phương được khai báo tại hàm nào thì có hiệu lực ở bên trong hàm đó, chẳng hạn numberOfBaskets và applePerBasket trong Hình 2.4 là các biến địa phương của hàm main và chỉ có hiệu lực ở bên trong hàm main(). Ngoài biến địa phương, Java còn có loại biến thực thể với phạm vi nằm trong một đối tượng và biến lớp với phạm vi lớp. Chương 4 và Chương 10 sẽ mô tả chi tiết về hai loại biến này.

---

<sup>3</sup> Địa chỉ <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

---

```

public class AppleProgram {
    public static void main() {
        int totalApples;
        int numberOfBaskets = 5;
        int applePerBasket = 10;

        totalApples = numberOfBaskets * applePerBasket;
        System.out.print("Number of apples is " + totalApples);
    }
}

```

*Khai báo biến địa phương totalApples kiểu int, không kèm theo khởi tạo giá trị.*

*Khai báo các biến địa phương numberOfBaskets, applePerBasket và khởi tạo giá trị của chúng.*

---

Hình 2.4: Sử dụng biến địa phương.

Một biến địa phương đã được khai báo nhưng chưa được gán một giá trị nào được gọi là biến chưa được khởi tạo và nó có giá trị không xác định. Trình biên dịch sẽ báo lỗi đối với mã sử dụng biến địa phương chưa được khởi tạo. Có thể khởi tạo giá trị của biến ngay tại lệnh khai báo để tránh tình huống quên khởi tạo biến, ví dụ:

```
char grade = 'A';
```

Vùng hiệu lực của một biến có thể còn nhỏ hơn phạm vi phương thức. Trong các phương thức, ta thường tạo các khối lệnh. Thông thường, các khối được giới hạn bởi cặp ngoặc {}. Ví dụ về một số khối thường gặp là các lệnh có cấu trúc (for, while) và các lệnh điều kiện (if) được trình bày chi tiết tại Mục 2.4. Nếu một biến được khai báo bên trong một khối lệnh thì nó chỉ có phạm vi cho đến hết khối lệnh đó.

## 2.3. CÁC PHÉP TOÁN CƠ BẢN

### 2.3.1. Phép gán

Phép gán là cách gán một giá trị cho một biến hoặc thay đổi giá trị của một biến. Lệnh gán trong Java có công thức:

*biến = biểu thức;*

Trong đó, dấu bằng (=) được gọi là dấu gán hay toán tử gán, biểu thức ở vế phải dấu gán được tính rồi lấy kết quả gán cho biến nằm ở vế trái.

Biểu thức tại vế phải có thể là một giá trị trực tiếp, một biến, hoặc một biểu thức phức tạp.

### 2.3.2. Các phép toán số học

Java hỗ trợ năm phép toán số học sau: + (cộng), - (trừ), \* (nhân), / (chia), % (modulo – lấy phần dư của phép chia). Các phép toán này chỉ áp dụng được cho các biến kiểu cơ bản như int, long và không áp dụng được cho các kiểu tham chiếu.

Phép chia được thực hiện cho hai giá trị kiểu nguyên sẽ cho kết quả là thương nguyên. Ví dụ biểu thức 4 / 3 cho kết quả bằng 1, còn 3 / 5 cho kết quả bằng 0.

Một số phép gán kèm theo biểu thức xuất hiện nhiều lần trong một chương trình, vì vậy Java cho phép viết các phép gán biểu thức đó một cách ngắn gọn hơn, sử dụng các phép gán phức hợp (`+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, `|=`).

Cách sử dụng phép gán phức hợp `+=` như sau:

*biến += biểu thức; tương đương biến = biến + biểu thức;*

Ví dụ:

```
apples += 2; tương đương apples = apples + 2;
```

Các phép gán phức hợp khác được sử dụng tương tự.

Java còn cung cấp các phép toán `++` (hay `--`) để tăng (giảm) giá trị của biến lên một đơn vị. Ví dụ:

`apples++` hay `++apples` có tác dụng tăng `apples` thêm 1 đơn vị

`apples--` hay `--apples` có tác dụng giảm `apples` đi 1 đơn vị

Khác biệt giữa việc viết phép tăng/giảm ở trước biến (tăng/giảm trước) và viết phép tăng/giảm ở sau biến (tăng/giảm sau) là thời điểm thực hiện phép tăng/giảm, thể hiện ở giá trị của biểu thức. Phép tăng/giảm trước được thực hiện trước khi biểu thức được tính giá trị, còn phép tăng/giảm sau được thực hiện sau khi biểu thức được tính giá trị. Ví dụ, nếu `apples` vốn có giá trị 1 thì các biểu thức `++apples` hay `apples++` đều có hiệu ứng là `apples` được tăng từ 1 lên 2. Tuy nhiên, `++apples` là biểu thức có giá trị bằng 2 (tăng `apples` trước tính giá trị), trong khi `apples++` là biểu thức có giá trị bằng 1 (tăng `apples` sau khi tính giá trị biểu thức). Nếu ta chỉ quan tâm đến hiệu ứng tăng hay giảm của các phép `++` hay `--` thì việc phép toán được đặt trước hay đặt sau không quan trọng. Đó cũng là cách dùng phổ biến nhất của các phép toán này.

### 2.3.3. Các phép toán khác

Các phép toán so sánh được sử dụng để so sánh giá trị hai biểu thức. Các phép toán này cho kết quả kiểu boolean bằng `true` nếu đúng và `false` nếu sai. Ví dụ:

```
boolean enoughApples = (totalApples > 10);
```

Các phép toán so sánh trong Java được liệt kê trong Bảng 2.1.

Cần lưu ý rằng mặc dù tất cả các phép toán này đều dùng được cho các kiểu dữ liệu cơ bản, chỉ có `==` và `!=` là dùng được cho kiểu tham chiếu. Tuy nhiên, hai phép toán này cũng không có ý nghĩa so sánh giá trị của các đối tượng. Chi tiết sẽ được nói đến tại Chương 3.

Ký hiệu toán học	Toán tử	Ví dụ	Ý nghĩa
>	>	$x > y$	x lớn hơn y
<	<	$x < y$	x nhỏ hơn y
$\geq$	$\geq$	$x \geq y$	x lớn hơn hoặc bằng y
$\leq$	$\leq$	$x \leq y$	x nhỏ hơn hoặc bằng y
=	==	$x == y$	x bằng y
$\neq$	!=	$x != y$	x khác y

Bảng 2.1: Các phép toán so sánh.

Toán tử	Ý nghĩa	Ví dụ	Ý nghĩa của ví dụ
&&	And	$x \&\& y$	Cho giá trị đúng khi cả x và y đúng, ngược lại cho giá trị sai.
	Or	$x \ \  y$	Cho giá trị đúng khi x đúng hoặc y đúng, ngược lại cho giá trị sai
!	Not	!x	Phủ định của x. Cho giá trị đúng khi x sai; cho giá trị sai khi x đúng

Bảng 2.2: Các phép toán logic.

Các phép toán logic dành cho các toán hạng là các biểu thức quan hệ hoặc các giá trị boolean. Kết quả của biểu thức logic là giá trị boolean.

Ví dụ:

```
bool enoughApples = (apples > 3) && (apples < 10);
```

có kết quả là biến enoughApples nhận giá trị là câu trả lời của câu hỏi "biến apples có giá trị lớn hơn 3 và nhỏ hơn 10 hay không?".

### 2.3.4. Độ ưu tiên của các phép toán

Mức độ ưu tiên của một số phép toán thường gặp có thứ tự của chúng như sau: Các toán tử đơn, +, -, !, ++ và -- có độ ưu tiên cao nhất. Tiếp theo là các phép toán đôi \*, / và %. Cuối cùng là các phép toán đôi +, -. Cuối cùng là các phép toán so sánh <, >, <=, >=. Ví dụ:  $3 + 4 < 2 + 6$  cho kết quả true.

Có thể dùng các cặp ngoặc ( ) để định rõ thứ tự ưu tiên trong biểu thức. Ví dụ:  $2 * (1 + 3)$  cho kết quả bằng 8.

## 2.4. CÁC CẤU TRÚC ĐIỀU KHIỂN

Java cung cấp hai loại lệnh để kiểm soát luồng điều khiển:

- **lệnh rẽ nhánh** (*branching*) chọn một hành động từ danh sách gồm nhiều hành động.

- **lệnh lặp** (*loop*) thực hiện lặp đi lặp lại một hành động cho đến khi một điều kiện dừng nào đó được thỏa mãn.

Hai loại lệnh đó tạo thành các **cấu trúc điều khiển** (*control structure*) bên trong chương trình.

### 2.4.1. Các cấu trúc rẽ nhánh

#### Lệnh if-else

Lệnh **if-else** (hay gọi tắt là lệnh **if**) cho phép rẽ nhánh bằng cách lựa chọn thực hiện một trong hai hành động. Ví dụ, trong một chương trình xếp loại điểm thi, nếu điểm của sinh viên nhỏ hơn 4.0, sinh viên đó được coi là trượt, nếu không thì được coi là đỗ. Thể hiện nội dung đó bằng một lệnh if-else của Java, ta có đoạn mã:

```
if (score < 4.0)
    System.out.print("Failed");
else
    System.out.print("Passed");
```

Khi chương trình chạy một lệnh if-else, đầu tiên nó kiểm tra biểu thức điều kiện nằm trong cặp ngoặc đơn sau từ khóa if. Nếu biểu thức có giá trị bằng true thì lệnh nằm sau từ khóa if sẽ được thực hiện. Ngược lại, lệnh nằm sau else sẽ được thực hiện. Chú ý là biểu thức điều kiện phải được đặt trong một cặp ngoặc đơn.

---

```
import java.util.Scanner;

public class IfElseExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        double score;
        System.out.print("Enter your score: ");
        score = input.nextDouble();

        if (score < 4.0)
            System.out.println("Sorry. You've failed the course.");
        else
            System.out.println(
                "Congratulations! You've passed the course.");
    }
}
```

```
% java IfElseExample
Enter your score: 2.0
Sorry. You've failed the course.

% java IfElseExample
Enter your score: 7.0
Congratulations! You've passed the course.
```

Hình 2.5: Ví dụ về cấu trúc *if-else*.

Chương trình ví dụ trong Hình 2.5 yêu cầu người dùng nhập điểm rồi in ra các thông báo khác nhau tùy theo điểm số đủ đỗ hoặc trượt.

Trong cấu trúc rẽ nhánh if-else, ta có thể bỏ phần else nếu không muốn chương trình thực hiện hành động nào nếu điều kiện không thỏa mãn. Chẳng hạn, nếu muốn thêm một lời khen đặc biệt cho điểm số xuất sắc từ 9.0 trở lên, ta có thể thêm lệnh if sau vào trong chương trình tại Hình 2.5.

```
if (score >= 9.0)
    System.out.print("Excellent!");
```

Ta có thể dùng các cấu trúc if-else lồng nhau để tạo ra điều kiện rẽ nhánh phức tạp. Lấy một ví dụ phức tạp hơn: cho trước điểm số (lưu tại biến score kiểu double), xác định xếp loại học lực A, B, C, D, F tùy theo điểm đó. Quy tắc xếp loại là: nếu điểm từ 8.5 trở lên thì đạt loại A, điểm từ 7.0 tới dưới 8.5 đạt loại B, v.v.. Tại đoạn mã xét các trường hợp của xếp loại điểm, ta có thể dùng cấu trúc if-else lồng nhau như sau:

```
if (score >= 8.5)
    grade = 'A';
else if (score >= 7.0)
    grade = 'B';
else if (score >= 5.5)
    grade = 'C';
else if (score >= 4.0)
    grade = 'D';
else
    grade = 'F';
```

Một điều cần đặc biệt lưu ý là nếu muốn thực hiện nhiều hơn một lệnh trong mỗi trường hợp của lệnh if-else, ta cần dùng cặp ngoặc { } bọc tập lệnh đó thành một **khối lệnh**. Ví dụ, phiên bản phức tạp hơn của lệnh if trong Hình 2.5:

Đại học Công nghệ - ĐHQGHN



```

if (score < 4.0) {
    System.out.print("Failed");
    System.out.println
        ("You have to take this course again");
}
else {
    System.out.print("Congratulations!!!");
    System.out.println
        ("You passed this course.");
}

```

### Lệnh switch

Khi chúng ta muốn viết một cấu trúc rẽ nhánh có nhiều lựa chọn, ta có thể sử dụng nhiều lệnh if-else lồng nhau. Tuy nhiên, trong trường hợp việc lựa chọn rẽ nhánh phụ thuộc vào giá trị (kiểu số nguyên hoặc kí tự, hoặc xâu kí tự kể từ JDK 7.0) của một biến hay biểu thức, ta có thể sử dụng cấu trúc **switch** để chương trình dễ hiểu hơn. Lệnh switch điển hình có dạng như sau:

```

switch (biểu_thức) {
    case hằng_1:
        tập_lệnh_1; break;
    case hằng_2:
        tập_lệnh_2; break;
    ...
    default:
        tập_lệnh_mặc_định;
}

```

Khi lệnh switch được chạy, *biểu\_thức* được tính giá trị và so sánh với *hằng\_1*. Nếu bằng nhau, chuỗi lệnh kể từ *tập\_lệnh\_1* được thực thi cho đến khi gặp lệnh break đầu tiên, đến đây chương trình sẽ nhảy tới điểm kết thúc cấu trúc switch. Nếu *biểu\_thức* không có giá trị bằng *hằng\_1*, nó sẽ được so sánh với *hằng\_2*, nếu bằng nhau, chương trình sẽ thực thi chuỗi lệnh kể từ *tập\_lệnh\_2* tới khi gặp lệnh break đầu tiên thì nhảy tới cuối cấu trúc switch. Quy trình cứ tiếp diễn như vậy. Cuối cùng, nếu *biểu\_thức* có giá trị khác với tất cả các giá trị đã được liệt kê (*hằng\_1*, *hằng\_2*, ...), chương trình sẽ thực thi *tập\_lệnh\_mặc\_định* nằm sau nhãn default: nếu như có nhãn này (không bắt buộc).

Ví dụ, lệnh sau so sánh giá trị của biến grade với các hằng kí tự 'A', 'B', 'C' và in ra các thông báo khác nhau cho từng trường hợp.

Đại học Công nghệ - CHUONG 2

```

switch (grade) {
    case 'A':
        System.out.print("Grade = A"); break;
    case 'B':
        System.out.print("Grade = B"); break;
    case 'C':
        System.out.print("Grade = C"); break;
    default:
        System.out.print("Grade's not A, B or C");
}

```

Nó tương đương với khối lệnh if-else lồng nhau sau:

```

if (grade == 'A')
    System.out.print("Grade = A");
else if (grade == 'B')
    System.out.print( "Grade = B");
else if (grade == 'C')
    System.out.print("Grade = C");
else
    System.out.print("Grade's not A, B, or C");

```

Lưu ý, các nhãn case trong cấu trúc switch phải là hằng chứ không thể là biến hay biểu thức. Nếu cần so sánh với biến hay biểu thức, ta nên dùng khối lệnh if-else lồng nhau.

Vấn đề đặc biệt của cấu trúc switch là các lệnh break. Nếu ta không tự gắn một lệnh break vào cuối chuỗi lệnh cần thực hiện cho mỗi trường hợp, chương trình sẽ chạy tiếp chuỗi lệnh của trường hợp sau chứ không tự động nhảy tới cuối cấu trúc switch. Ví dụ, đoạn chương trình sau sẽ chạy lệnh in thứ nhất nếu grade nhận một trong ba giá trị 'A', 'B', 'C' và chạy lệnh in thứ hai trong trường hợp còn lại:

```

switch (grade) {
    case 'A':
    case 'B':
    case 'C':
        cout << "Grade is A, B or C."; break;
    default:
        cout << "Grade is not A, B or C.";
}

```

Chương trình trong Hình 2.6 là một ví dụ hoàn chỉnh sử dụng cấu trúc switch để in ra các thông báo khác nhau tùy theo xếp loại học lực (grade) mà người dùng nhập từ bàn phím. Trong đó, case 'A' kết thúc với break sau chỉ một lệnh, còn case 'B' chạy tiếp qua case 'C', 'D' rồi mới gặp break và thoát khỏi lệnh switch. Nhãn default được dùng để xử lý trường hợp biến grade giữ giá trị không hợp lệ đối với xếp loại học lực. Trong nhiều chương trình, phần default thường được dùng để xử lý các trường hợp không mong đợi, chẳng hạn như để bắt lỗi các kí hiệu học lực không hợp lệ mà người dùng có thể nhập sai.

Có một lưu ý nhỏ là Scanner không hỗ trợ việc đọc từng kí tự một. Do đó, để đọc giá trị của grade do người dùng nhập, ta dùng phương thức next() để đọc một chuỗi (không chứa kí tự trắng), rồi lấy kí tự đầu tiên bằng hàm charAt(0) (mà kiểu String cung cấp) làm giá trị của grade.

---

```

import java.util.Scanner;

public class SwitchExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter your grade: ");
        String userInput = input.next();
        char grade = userInput.charAt(0);

        switch (grade) {
            case 'A':
                System.out.println("Excellent!"); break;
            case 'B':
                System.out.println("Great!");
            case 'C':
            case 'D':
                System.out.println("Well done!"); break;
            case 'F':
                System.out.println("Sorry, you failed."); break;
            default:
                System.out.println("Error! Invalid grade.");
        }
    }
}

```

#### Kết quả chạy chương trình

```

Enter your grade: A
Excellent!

```

```

Enter your grade: B
Great!
Well done!

```

```

Enter your grade: D
Well done!

```

```

Enter your grade: F
Sorry, you failed.

```

---

Hình 2.6: Ví dụ sử dụng cấu trúc switch.

Kể từ Java SE 7, ta có thể dùng các đối tượng String làm nhãn cho các lệnh case. Ví dụ:

```

switch (answer) {
    case "yes":
        System.out.print("You said 'yes'"); break;
    case "no":
        System.out.print("You said 'no'"); break;
    default:
        System.out.print("I don't get what you mean.");
}

```

## 2.4.2. Các cấu trúc lặp

Các chương trình thường cần phải lặp đi lặp lại một hoạt động nào đó. Ví dụ, một chương trình xếp loại học lực sẽ chứa các lệnh rẽ nhánh để gán xếp loại A, B, C... cho một sinh viên tùy theo điểm số của sinh viên này. Để xếp loại cho cả một lớp, chương trình sẽ phải lặp lại thao tác đó cho từng sinh viên trong lớp. Phần chương trình lặp đi lặp lại một lệnh hoặc một khối lệnh được gọi là một **vòng lặp**. Lệnh hoặc khối lệnh được lặp đi lặp lại được gọi là thân của vòng lặp. Cấu trúc lặp cho phép lập trình viên chỉ thị cho chương trình lặp đi lặp lại một hoạt động trong khi một điều kiện nào đó vẫn được thỏa mãn.

Khi thiết kế một vòng lặp, ta cần xác định thân vòng lặp thực hiện hành động gì. Ngoài ra, ta còn cần một cơ chế để quyết định khi nào vòng lặp sẽ kết thúc.

Mục này sẽ giới thiệu về các lệnh lặp mà Java cung cấp.

### Vòng while

Vòng **while** lặp đi lặp lại chuỗi hành động, gọi là thân vòng lặp, nếu như điều kiện lặp vẫn còn được thỏa mãn. Cú pháp của vòng lặp while như sau:

```
while (điều_kiện_lặp)
    thân_vòng_lặp
```

Cấu trúc này bắt đầu bằng từ khóa while, tiếp theo là điều kiện lặp đặt trong một cặp ngoặc đơn, cuối cùng là thân vòng lặp. Thân vòng lặp hay chứa nhiều hơn một lệnh và khi đó thì phải được gói trong một cặp ngoặc { }.

Khi thực thi một cấu trúc while, đầu tiên chương trình kiểm tra giá trị của biểu thức điều kiện, nếu biểu thức cho giá trị false thì nhảy đến điểm kết thúc lệnh while, còn nếu điều kiện lặp có giá trị true thì tiến hành thực hiện tập lệnh trong thân vòng lặp rồi quay trở lại kiểm tra điều kiện lặp, nếu không thỏa mãn thì kết thúc, nếu thỏa mãn thì lại thực thi thân vòng lặp rồi quay lại... Tập lệnh ở thân vòng lặp có thể làm thay đổi giá trị của biểu thức điều kiện từ true sang false để dừng vòng lặp.

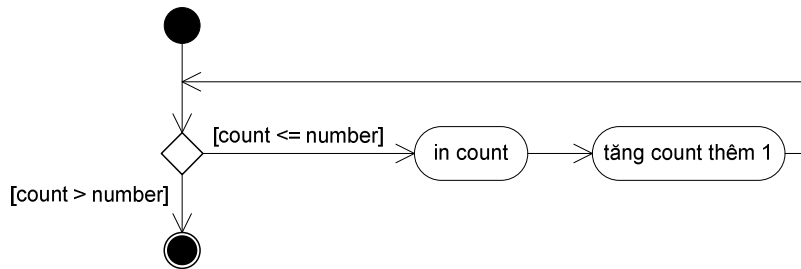
Ví dụ, xét một chương trình có nhiệm vụ đếm từ 1 đến một ngưỡng number cho trước. Đoạn mã đếm từ 1 đến number có thể được viết như sau:

```
count = 1;
while (count <= number) {
    System.out.print(count + ", ");
    count++;
}
```

Giả sử biến number có giá trị bằng 2, đoạn mã trên hoạt động như sau: Đầu tiên, biến count được khởi tạo bằng 1. Vòng while bắt đầu bằng việc kiểm tra điều kiện (count <= number), nghĩa là  $1 \leq 2$ , điều kiện thỏa mãn. Thân vòng lặp được thực thi lần thứ nhất: giá trị 1 của count được in ra màn hình kèm theo dấu phẩy, sau đó count được tăng lên 2. Vòng lặp quay về điểm xuất phát: kiểm tra điều kiện lặp, giờ là  $2 \leq 2$ , vẫn thỏa mãn. Thân vòng lặp được chạy lần thứ hai (in giá trị 2 của count và

tăng count lên 3) trước khi quay lại điểm xuất phát của vòng lặp. Tại lần kiểm tra điều kiện lặp này, biểu thức  $3 \leq 2$  cho giá trị false, vòng lặp kết thúc do điều kiện lặp không còn được thỏa mãn, chương trình chạy tiếp ở lệnh nằm sau cấu trúc while đang xét.

Cấu trúc while trong đoạn mã trên có thể được biểu diễn bằng sơ đồ trong Hình 2.7.



Hình 2.7: Sơ đồ một vòng lặp while.

Chương trình hoàn chỉnh trong Hình 2.8 minh họa cách sử dụng vòng lặp while để in ra các số nguyên (biến count) từ 1 cho đến một ngưỡng giá trị do người dùng nhập vào từ bàn phím (lưu tại biến number). Kèm theo là kết quả của các lần chạy khác nhau với các giá trị khác nhau của number. Đặc biệt, khi người dùng nhập giá trị 0 cho number, thân vòng while không chạy một lần nào, thể hiện ở việc không một số nào được in ra màn hình. Lí do là vì nếu number bằng 0 thì biểu thức  $count \leq number$  ngay từ đầu vòng while đã có giá trị false.

Đại học Công nghệ - ĐHQGHN

---

```

import java.util.Scanner;

public class WhileExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int count, number;
        System.out.print("Enter a number: ");
        number = input.nextInt();

        count = 1;
        while (count <= number) {
            System.out.print(count + ", ");
            count++;
        }
        System.out.println("BOOOOOM!");
    }
}

```

#### Kết quả chạy chương trình

```

Enter a number: 2
1, 2, BOOOOOM!

```

```

Enter a number: 1
1, BOOOOOM!

```

```

Enter a number: 0
BOOOOOM!

```

---

Hình 2.8: Ví dụ về vòng lặp while.

### Vòng do-while

Vòng **do-while** rất giống với vòng while, khác biệt là ở chỗ thân vòng lặp sẽ được thực hiện trước, sau đó mới kiểm tra điều kiện lặp, nếu đúng thì quay lại chạy thân vòng lặp, nếu sai thì dừng vòng lặp. Khác biệt đó có nghĩa rằng thân của vòng do-while luôn được chạy ít nhất một lần, trong khi thân vòng while có thể không được chạy lần nào.

Công thức của vòng do-while tổng quát là:

```

do
    thân_vòng_lặp
while (điều_kiện_lặp);

```

Tương tự như ở vòng while, *thân\_vòng\_lặp* của vòng do-while có thể chỉ gồm một lệnh hoặc thường gặp hơn là một chuỗi lệnh được bọc trong một cặp ngoặc { }. Lưu ý dấu chấm phẩy đặt cuối toàn bộ khối do-while.

Đại học Công nghệ - HHOUGHN

Công thức tổng quát của vòng do-while ở trên tương đương với công thức sau nếu dùng vòng while:

```
thân_vòng_lặp
while (điều_kiện_lặp)
    thân_vòng_lặp
```

Để minh họa hoạt động của hai cấu trúc lặp while và do-while, ta so sánh hai đoạn mã dưới đây:

<pre>count = 1; while (count &lt;= number) {     System.out.print(         count + ", ");     count++; }</pre>	<pre>count = 1; do {     System.out.print(         count + ", ");     count++; } while (count &lt;= number);</pre>
--	--

Hai đoạn mã chỉ khác nhau ở chỗ một bên trái dùng vòng while, bên phải dùng vòng do-while, còn lại, các phần thân vòng lặp, điều kiện, khởi tạo đều giống hệt nhau. Đoạn bên trái được lấy từ ví dụ trong mục trước, nó in ra các số từ 1 đến number. Đoạn mã dùng vòng do-while bên phải cũng thực hiện công việc giống hệt đoạn bên trái, ngoại trừ một điểm: khi number nhỏ hơn 1 thì nó vẫn đếm 1 trước khi dùng vòng lặp – thân vòng lặp chạy một lần trước khi kiểm tra điều kiện.

### Vòng for

Vòng **for** là cấu trúc hỗ trợ việc viết các vòng lặp mà số lần lặp được kiểm soát bằng biến đếm. Chẳng hạn, đoạn mã giả sau đây mô tả thuật toán in ra các số từ 1 đến number:

*Làm nhiệm vụ sau đây đối với mỗi giá trị của count từ 1 đến number:*

*In count ra màn hình*

Đoạn mã giả đó có thể được viết bằng vòng for như sau:

```
for (count = 1; count <= number; count++)
    cout << count << ", ";
```

Với number có giá trị bằng 3, đoạn trình trên cho kết quả in ra màn hình là:

1, 2, 3,

Cấu trúc tổng quát của vòng lặp for là:

```
for ( khởi_tạo; điều_kiện_lặp; cập_nhật )
    thân_vòng_lặp
```

Trong đó, biểu thức *khởi\_tạo* thường khởi tạo con đếm điều khiển vòng lặp, *điều\_kiện\_lặp* xác định xem thân vòng lặp có nên chạy tiếp hay không (điều kiện này thường chứa ngưỡng cuối cùng của con đếm), và biểu thức *cập\_nhật* làm tăng hay giảm con đếm. Cũng tương tự như ở các cấu trúc if, while..., nếu *thân\_vòng\_lặp* có nhiều hơn một lệnh thì cần phải bọc nó trong một cặp ngoặc { }. Lưu ý rằng cặp ngoặc đơn bao quanh bộ ba *khởi\_tạo, điều\_kiện\_lặp, cập\_nhật*, cũng như hai dấu chấm phẩy ngăn cách ba thành phần đó, là các thành bắt buộc của cú pháp cấu trúc for. Ba



thành phần đó cũng có thể là biểu thức rỗng nếu cần thiết, nhưng kể cả khi đó vẫn phải có đủ hai dấu chấm phẩy.

Ta có thể khai báo biến ngay trong phần *khởi\_tạo* của vòng for, chẳng hạn đối với biến con đếm. Nhưng các biến được khai báo tại đó chỉ có hiệu lực ở bên trong cấu trúc lặp. Ví dụ:

```
for (int count = 1; count <= number; count++)
    cout << count << ", ";
```

---

```
import java.util.Scanner;

public class ForExample {
    public static void main(String[] args) {
        float sum = 0;
        int subjects = 10;
        Scanner input = new Scanner(System.in);
        System.out.print( "Enter the marks for "
            + subjects + " subjects: ");
        for (int count = 0; count < subjects; count++) {
            float mark;
            mark = input.nextFloat();
            sum += mark;
        }
        System.out.print("Average mark = "+sum/subjects);
    }
}
```

---

Hình 2.9: Ví dụ về vòng lặp for.

Hình 2.9 minh họa cách sử dụng vòng lặp for để tính điểm trung bình từ điểm của 10 môn học (số môn học lưu trong biến subjects). Người dùng sẽ được yêu cầu nhập từ bàn phím điểm số của 10 môn học trong khi chương trình cộng dồn tổng của 10 điểm số này. Công việc mà chương trình cần lặp đi lặp lại 10 lần là: nhập điểm của một môn học, cộng dồn điểm đó vào tổng điểm. Đầu tiên vòng for sẽ tiến hành bước khởi tạo với mục đích chính là khởi tạo biến đếm. Việc khởi tạo chỉ được tiến hành duy nhất một lần. Trong ví dụ này, biến count được khai báo ngay tại vòng for và khởi tạo giá trị bằng 0. Tiếp theo vòng for sẽ tiến hành kiểm tra điều kiện lặp count < subjects. Nếu điều kiện sai, vòng lặp for sẽ kết thúc. Nếu điều kiện đúng, thân vòng lặp for sẽ được thực hiện (nhập một giá trị kiểu float rồi cộng dồn vào biến sum). Sau đó là bước cập nhật với nhiệm vụ tăng biến đếm thêm 1. Kết quả là vòng lặp sẽ chạy 10 lần với các giá trị count bằng 0, 1, ..., 9 (khi count nhận giá trị 10 thì điều kiện lặp không còn đúng và vòng lặp kết thúc).

### Các lệnh break và continue

Như đã giới thiệu ở các mục trước, các vòng lặp while, do-while, và for đều kết thúc khi kiểm tra biểu thức điều kiện được giá trị false và chạy tiếp thân vòng lặp trong trường hợp còn lại. Các lệnh break và continue là các lệnh nhảy cho phép thay đổi luồng điều khiển đó.

Lệnh **break** khi được thực thi bên trong một cấu trúc lặp hay một cấu trúc switch có tác dụng lập tức chấm dứt cấu trúc đó, chương trình sẽ chạy tiếp ở lệnh nằm tiếp sau cấu trúc đó. Lệnh break thường được dùng để kết thúc sớm vòng lặp (thay vì đợi đến lượt kiểm tra điều kiện lặp) hoặc để bỏ qua phần còn lại của cấu trúc switch.

Về ví dụ sử dụng lệnh break trong vòng lặp. Chẳng hạn, nếu ta sửa ví dụ trong Hình 2.9 để vòng for ngừng lại khi người dùng nhập điểm số có giá trị âm, ta có chương trình trong Hình 2.10. Với cài đặt này, khi người dùng nhập một điểm số có giá trị âm, điều kiện ( $mark < 0$ ) sẽ cho kết quả true, chương trình thoát khỏi vòng for và chạy tiếp từ lệnh if nằm sau đó. Trong trường hợp đó, biến count chưa kịp tăng đến ngưỡng subjects (điều kiện lặp của vòng for chưa kịp bị phá vỡ). Do đó, biểu thức ( $count \geq subjects$ ) trong lệnh if sau đó có nghĩa "vòng for có chạy đủ subjects lần hay không?" hoặc "vòng for có bị ngắt giữa chừng bởi lệnh break hay không?", hay là "dữ liệu nhập vào có thành công hay không?".

---

```
import java.util.Scanner;

public class BreakTest {
    public static void main(String[] args) {
        float sum = 0;
        int count, subjects = 10;
        Scanner input = new Scanner(System.in);
        System.out.print( "Enter the marks for "
            + subjects + " subjects: ");
        for (count = 0; count < subjects; count++) {
            float mark;
            mark = input.nextFloat();
            if (mark < 0) break;
            sum += mark;
        }

        if (count >= subjects) // successful
            System.out.print("Average mark = "+sum/subjects);
        else
            System.out.print( "Error: Invalid mark!");
    }
}
```

---

Hình 2.10: Ví dụ về lệnh break.

Lệnh **continue** nằm trong một vòng lặp có tác dụng kết thúc lần lặp hiện hành của vòng lặp đó. Hình 2.11 là một bản sửa đổi khác của chương trình trong Hình 2.9. Trong phiên bản này, chương trình không ghi nhận điểm số có giá trị âm, cũng không kết thúc chương trình sau khi báo lỗi như bản trong Hình 2.10, mà yêu cầu nhập lại cho đến khi nào thành công. Khi gặp điểm số âm được nhập vào (biến mark), lệnh continue được thực thi có tác dụng bỏ qua đoạn lệnh ghi nhận điểm ở nửa sau của thân vòng while (đoạn cộng dồn vào tổng sum và tăng biến đếm count).

Lần lặp được thực hiện sau đó sẽ yêu cầu nhập lại điểm cho môn học đang nhập dở (xem kết quả chạy chương trình trong Hình 2.11).

Một điểm cần lưu ý là các lệnh break hay continue chỉ có tác dụng đối với vòng lặp trong cùng chứa nó. Chẳng hạn, nếu có hai vòng lặp lồng nhau và lệnh break nằm trong vòng lặp bên trong, thì khi được thực thi, lệnh break đó chỉ có tác dụng kết thúc vòng lặp bên trong.

```
import java.util.Scanner;

public class ContinueTest {
    public static void main(String[] args) {
        float sum = 0;
        int count=0, subjects = 3;
        Scanner input = new Scanner(System.in);
        System.out.print( "Enter the marks for "
            + subjects + " subjects: ");
        while (count < subjects) {
            System.out.print("#" + (count+1) + ": ");
            float mark = input.nextFloat();
            if (mark < 0) {
                System.out.println(mark + " ignored");
                continue;
            }
            sum += mark;
            count++;
        }
        System.out.print("Average mark = "+sum/subjects);
    }
}
```

```
% java ContinueTest
Enter the marks of 3 subjects.
#1: 8.0
#2: 7.2
#3: -5
-5 ignored
#3: 10.0
Average mark = 8.400001
```

Hình 2.11: Ví dụ về lệnh continue.

### 2.4.3. Biểu thức điều kiện trong các cấu trúc điều khiển

Hầu hết các cấu trúc điều khiển mà ta nói đến trong chương này đều dùng đến một thành phần quan trọng: biểu thức điều kiện. Trong các ví dụ trước, ta mới chỉ dùng đến các điều kiện đơn giản, chẳng hạn `count <= number` hay `grade == 'A'`, với duy nhất một phép so sánh. Khi cần viết những điều kiện phức tạp hơn, cần đến nhiều điều kiện nhỏ, ta có thể kết hợp chúng bằng các phép toán logic `&&` (AND – và), `||` (OR – hoặc) và `!` (NOT – phủ định). Ví dụ:

Khi kiểm tra điều kiện `80 <= score < 90`, bất đẳng thức toán học này cần được tách thành hai điều kiện đơn. Bất đẳng đúng khi cả hai điều kiện đơn đều thỏa mãn. Đó là khi ta cần dùng phép toán logic `&&` (AND).

```
if (score >= 80 && score < 90)
    grade = 'B';
```

Khi một trong hai điều kiện xảy ra, hoặc tiền đã hết hoặc túi đã đầy, thì không thể mua thêm hàng. Trường hợp này, ta cần dùng phép toán logic `||` (OR).

```
if (moneyLeft <= 0 || bagIsFull)
    cout << "Can't buy anything more!";
```

Tiếp tục lặp trong khi dữ liệu vào chưa có giá trị bằng giá trị canh – đánh dấu điểm cuối của chuỗi dữ liệu:

```
while ( !(input == 0))
    ...
```

Trong trường hợp này, ta có thể dùng phép phủ định, hoặc chọn cách đơn giản hơn là dùng phép so sánh khác (`!=`) như sau:

```
while (input != 0)
    ...
```

## Đọc thêm

Chương này chỉ giới thiệu các nét cơ bản về ngôn ngữ Java. Bạn đọc có thể tìm hiểu sâu hơn tại các tài liệu như:

1. Language Basics, The Java™ Tutorials,  
<http://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html>
2. Chương 4, 5, Deitel & Deitel, *Java How to Program*, 6<sup>th</sup> edition, Prentice Hall, 2005.

Đại học Công nghệ - ĐHQGHN

## Bài tập

- Điền từ thích hợp vào chỗ trống trong mỗi câu sau:
  - Mỗi khai báo lớp mà bắt đầu bằng từ khóa \_\_\_\_\_ phải được lưu trong một file có tên trùng với tên lớp và kết thúc bằng phần mở rộng .java
  - Từ khóa \_\_\_\_\_ tạo một đối tượng thuộc lớp có tên đặt ở bên phải từ khóa.
  - Mỗi biến khi khai báo phải được chỉ rõ một \_\_\_\_\_ và một \_\_\_\_\_.
- Các phát biểu sau đây đúng hay sai? Nếu sai, hãy giải thích.
  - Lớp nào có chứa phương thức public static void main(String [] args) thì có thể được dùng để chạy ứng dụng.
  - Một tập lệnh chứa trong một cặp ngoặc {} được gọi là một khối lệnh
  - Một lệnh có điều kiện cho phép một hành động được lặp đi lặp lại trong khi một điều kiện nào đó vẫn giữ giá trị true.
  - Java cung cấp các phép toán phức hợp như +=, -= để viết tắt các lệnh gán.
  - Luồng điều khiển quy định thứ tự các lệnh được thực thi trong chương trình.
  - Phép toán đổi kiểu (double) trả về một giá trị nguyên là bản sao của toán hạng của nó.
  - Biến địa phương kiểu boolean nhận giá trị mặc định là false.
  - Biến địa phương kiểu boolean nhận giá trị mặc định là true.
- Cài đặt bộ công cụ JDK, dịch và chạy thử các chương trình ví dụ đã cho trong chương này.
- Viết các lệnh Java để thực hiện từng nhiệm vụ sau đây:
  - Dùng một lệnh để gán tổng của x và y cho z và tăng x thêm 1 sau phép tính trên.
  - Kiểm tra xem giá trị biến count có lớn hơn 10 hay không, nếu có thì in ra dòng text "Count is greater than 10".
  - Giảm x đi 1 đơn vị, sau đó gán cho biến total giá trị là hiệu của total và x. Chỉ dùng một lệnh.
  - Tính phần dư của phép chia q cho d rồi gán kết quả đó cho q.
  - Khai báo các biến sum và x thuộc kiểu int.
  - Cộng dồn giá trị của x vào biến sum
  - In ra dòng "The sum is ", tiếp theo là giá trị của biến sum.
  - Tính tổng các số chẵn trong khoảng từ 1 đến 99.

- i) Sử dụng vòng lặp để in ra các số từ 1 đến 10 trên một dòng, dùng kí tự tab ('\t') để ngăn cách giữa các số.
- Viết một chương trình tính tổng các số nguyên từ 1 đến 10, sử dụng vòng while cho nhiệm vụ lặp.
  - Viết một chương trình tính tổng các số nguyên từ 1 đến 10, sử dụng vòng for cho nhiệm vụ lặp.
  - Viết một chương trình tính tổng các số nguyên từ 1 đến 10, sử dụng vòng do-while cho nhiệm vụ lặp.
  - Tìm kết quả hiển thị của chương trình sau:

```
class Output {
    public static void main(String [] args) {
        Output o = new Output();
        o.go() ;
    }
    void go() {
        int y = 7;
        for (int x = 1; x < 8; x++) {
            y++;
            if (x > 4) {
                System.out.print(++y + " ");
            }
            if (y > 14) {
                System.out.println(" x = " + x);
                break;
            }
        }
    }
}
```

- Sắp xếp lại các dòng mã sau đây thành chương trình có kết quả hiển thị như hình dưới. Tự bổ sung các ngoặc đóng } vào những nơi thích hợp.

```
x++;
if (x == 1) {
for(int x = 0; x < 4; x++) {
class MultiFor {
for(int y = 4; y > 2; y--) {
system.out.println(x + " " + y);
public static void main(String [] args) {
```

```
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

- Viết chương trình thử nghiệm việc in dữ liệu ra màn hình bằng lệnh System.out.printf(), ví dụ:

```
System.out.printf(
```

```
    "Hello, I am %s, I am %d years old.\n", "Bob", 20 );
```

trong đó, %s là kí hiệu định dạng đại diện cho một chuỗi kí tự (trong ví dụ là "Bob") còn %d đại diện cho một số kiểu int (trong ví dụ là 20). Tính năng này được Java 5.0 mượn từ ngôn ngữ lập trình C. Chi tiết về tính năng này xem lại Phụ lục G – Formatted Output của tài liệu [1].

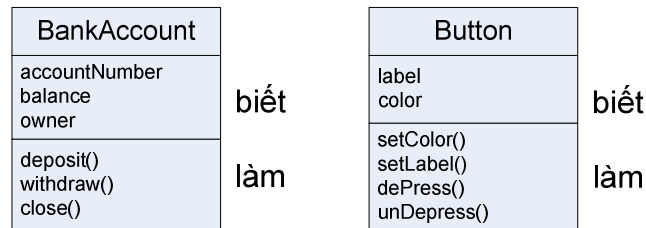
## Chương 3. LỚP VÀ ĐỐI TƯỢNG

Trong chương này, chúng ta sẽ bàn sâu hơn về lớp và đối tượng. Các khái niệm hướng đối tượng khác sẽ lần lượt là trọng tâm của các chương sau. Chương này cũng nói về các vấn đề căn bản mà ta cần xem xét mỗi khi thiết kế một lớp.

Như đã giới thiệu sơ lược trong chương trước, chương trình Java khi chạy là một tập hợp các đối tượng, chúng được yêu cầu thực hiện dịch vụ và yêu cầu dịch vụ của các đối tượng khác. Một đối tượng được tạo ra từ một lớp được gọi là một **thực thể** (*instance*) của lớp đó. Ta có thể coi "thực thể" là một cách gọi khác của "đối tượng".

Lớp là khuôn mẫu để từ đó tạo ra các thực thể. Vậy nên, khi thiết kế một lớp, ta cần nghĩ đến những đối tượng sẽ được tạo ra từ lớp đó. Có hai loại thông tin quan trọng về mỗi đối tượng:

- Những thông tin mà đối tượng đó **biết**.
- Những việc mà đối tượng đó **làm**.



Hình 3.1: Hai loại thông tin quan trọng về đối tượng.

Những gì mà một đối tượng **biết** về bản thân nó được gọi là các **biến thực thể** (*instance variable*) hay **thuộc tính thực thể** (*instance attribute*). Chúng biểu diễn **trạng thái** (*state*) của đối tượng hay còn gọi là dữ liệu của đối tượng, các đối tượng khác nhau thuộc cùng loại có thể có các giá trị khác nhau cho các biến thực thể. Các biến thực thể có thể là biến thuộc một trong những kiểu dữ liệu cơ bản (int, boolean, float...) hoặc là tham chiếu tới đối tượng thuộc một lớp nào đó.

Những gì một đối tượng có thể **làm** được gọi là các phương thức (method). Các phương thức được thiết kế để thao tác trên dữ liệu của đối tượng. Một đối tượng thường có các phương thức đọc và ghi giá trị cho các biến thực thể. Ví dụ, một đối tượng đồng hồ báo thức có một biến thực thể lưu thời gian cần báo thức, và hai phương thức để đặt và lấy giờ báo thức.

Tóm lại, đối tượng có các biến thực thể và các phương thức, các biến thực thể và các phương thức này được định nghĩa trong thiết kế của lớp. Công việc viết chương trình là viết các định nghĩa lớp. Định nghĩa lớp mô tả về các thành phần mà mỗi

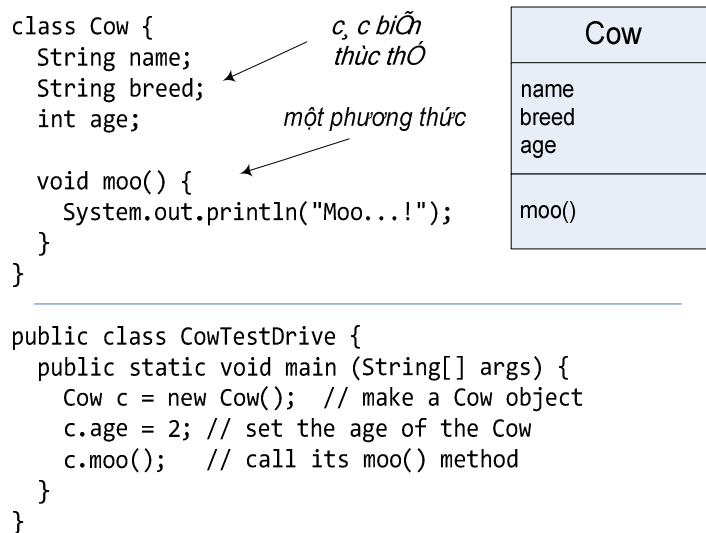


thực thể của nó sẽ chứa, cụ thể là dữ liệu của mỗi thực thể và các phương thức cho phép truy nhập và sửa đổi dữ liệu đó.

Một lớp không phải là một đối tượng, nó là một khuôn mẫu dùng để tạo nên đối tượng. Nó mô tả cách tạo một đối tượng thuộc kiểu cụ thể đó. Mỗi đối tượng tạo ra từ một lớp có thể có các giá trị riêng cho các biến thực thể. Ví dụ, ta có thể dùng lớp BankAccount để tạo ra nhiều đối tượng tài khoản ngân hàng, mỗi tài khoản có một chủ tài khoản, một số tài khoản, và một số dư riêng; mỗi tài khoản đều có thể làm những việc giống nhau (rút tiền, gửi tiền, đóng tài khoản), tuy chỉ biết những gì chỉ có ở tài khoản cụ thể đó.

### 3.1. TẠO VÀ SỬ DỤNG ĐỐI TƯỢNG

Vậy làm thế nào để tạo và sử dụng một đối tượng? Ta cần đến hai lớp. Một lớp dành cho kiểu đối tượng mà ta muốn tạo (BankAccount, Dog, Cow, AlarmClock, AddressBookEntry,...) và một lớp khác để thử nghiệm lớp đó. Lớp thử nghiệm là chương trình, nơi ta đặt phương thức main, và tại phương thức main đó, ta tạo và sử dụng các đối tượng thuộc lớp vừa xây dựng. Lớp thử nghiệm chỉ có một nhiệm vụ duy nhất: chạy thử các biến và phương thức của lớp đối tượng mới.



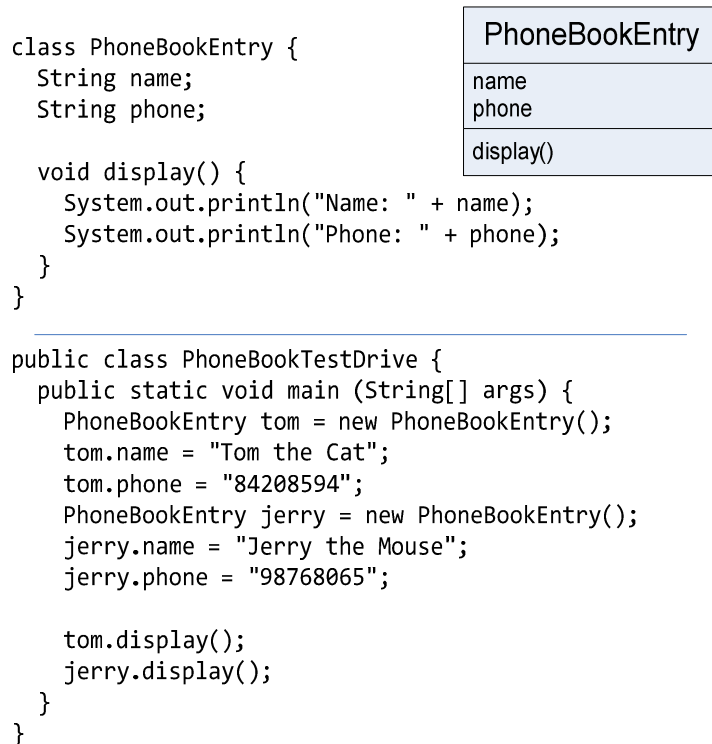
Hình 3.2. Lớp Cow và lớp thử nghiệm CowTestDrive

Từ đây, trong nhiều ví dụ, ta sẽ dùng hai lớp. Một là lớp định nghĩa các đối tượng ta muốn dùng, lớp kia là lớp thử nghiệm với tên là <Cái\_gì\_đó>TestDrive. Chẳng hạn, khi ta xây dựng lớp Cow, ta sẽ cần thêm lớp CowTestDrive là chương trình thử nghiệm lớp Cow. Chỉ có <Cái\_gì\_đó>TestDrive mới có chứa một phương thức main, với mục đích tạo các đối tượng thuộc lớp <Cái\_gì\_đó>, rồi truy nhập các phương thức và biến của các đối tượng đó. Ví dụ trong Hình 3.2 dùng hai lớp Cow

và CowTestDrive để minh họa cách xây dựng một lớp mới và thử nghiệm các đối tượng thuộc lớp đó.

Chương trình CowTestDrive thử nghiệm lớp Cow bằng cách tạo một đối tượng <sup>4</sup> thuộc lớp này (lệnh `Cow c = new Cow()`), sau đó dùng toán tử dấu chấm (.) để truy cập các biến thực thể và gọi phương thức của đối tượng. Cụ thể, lệnh `c.age = 2` gán giá trị 2 cho biến thực thể `age` của `c`, còn `c.moo()` kích hoạt phương thức `moo()` của `c`.

Để cài đặt một lớp, việc viết một lớp thử nghiệm kèm theo không phải là bước bắt buộc. Tuy nhiên, đó là công việc cần thiết để đảm bảo rằng lớp đó đã được cài đặt đúng và hoạt động như mong muốn của người thiết kế.



Hình 3.3. Lớp PhoneBookEntry và lớp thử nghiệm.

Ví dụ trong Hình 3.3 tương tự với ví dụ trong Hình 3.2. Phương thức `main` ở đây tạo hai đối tượng thuộc lớp `PhoneBookEntry`, gán giá trị cho các biến thực thể của chúng và gọi phương thức `display()` cho từng đối tượng. Để ý rằng hai đối tượng `tom` và `jerry` có các bộ biến `name` và `phone` độc lập với nhau, tuy rằng chúng trùng tên.

---

<sup>4</sup> Ở đây ta tạm gọi là "đối tượng `c`". Trong nhiều ngữ cảnh, người ta cũng quen gọi là "đối tượng `c`". Tuy nhiên, `c` bản chất không phải là đối tượng mà chỉ là tham chiếu đối tượng. Chương 4 sẽ bàn chi tiết về khái niệm tham chiếu đối tượng.

### 3.2. TƯƠNG TÁC GIỮA CÁC ĐỐI TƯỢNG

Như đã nói đến trong các phần trước, phương thức main phục vụ hai mục tiêu sử dụng: (1) để thử nghiệm các lớp đã cài; (2) để khởi động ứng dụng Java. Khi ở trong các phương thức main nói trên, ta không thực sự ở môi trường hướng đối tượng, main chỉ tạo và chạy thử các đối tượng. Trong khi đó, ở một ứng dụng hướng đối tượng thực thụ, các đối tượng phải "nói chuyện" với nhau.

Một ứng dụng hướng đối tượng nói chung và ứng dụng Java nói riêng thực chất là các đối tượng nói chuyện với nhau. "Nói chuyện" ở đây có nghĩa rằng các đối tượng gọi các phương thức của nhau. Tại các ví dụ trước, ta có các lớp TestDrive tạo đối tượng các lớp khác và chạy thử các phương thức của chúng. Tại Chương 5, ta sẽ có ví dụ mà phương thức main tạo các đối tượng rồi thả cho chúng tương tác với nhau.

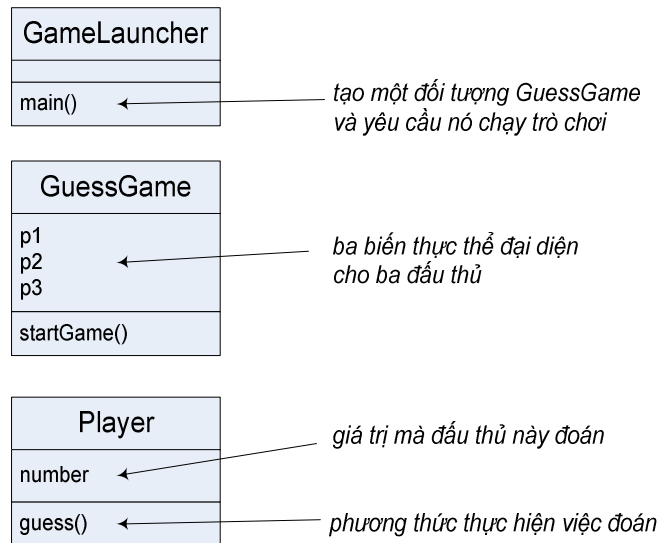
Tạm thời, ta dùng một ví dụ nhỏ về trò chơi đoán số để có một chút phác họa về hoạt động của một ứng dụng hướng đối tượng thực thụ. Do ta vẫn đang ở giai đoạn làm quen với Java, chương trình ví dụ này hơi lộn xộn và không hiệu quả, ta sẽ cải tiến nó ở những chương sau. Nếu có những đoạn mã khó hiểu, ta hãy tạm bỏ qua, vì điểm quan trọng của ví dụ này là các đối tượng nói chuyện với nhau.

Trò chơi đoán số bao gồm một đối tượng game và ba đối tượng player. Đối tượng game sinh ngẫu nhiên một số trong đoạn từ 0 đến 9, ba player lần lượt thử đoán số đó. Chương trình bao gồm ba lớp: GameLauncher, GuessGame, và Player

*Lô-gic chương trình:*

- GameLauncher là nơi ứng dụng bắt đầu chạy. Lớp này có phương thức main().
- Phương thức main() tạo một đối tượng GuessGame được tạo và chạy phương thức startGame() của nó.
- Phương thức startGame() của đối tượng GuessGame là nơi toàn bộ ván chơi xảy ra. Nó tạo ra ba đấu thủ (player), rồi "bịa" ra một số ngẫu nhiên (cái mà các đấu thủ cần đoán). Sau đó, nó yêu cầu từng đấu thủ đoán, kiểm tra kết quả, và in ra thông tin về (các) đấu thủ thắng cuộc hoặc yêu cầu cả ba đoán lại.

Đại học Công nghệ - HHOCHIM



Hình 3.4. Ba lớp của chương trình đoán số.

Nội dung đầy đủ của mã nguồn các lớp GameLauncher, GuessGame và Player được cho trong Hình 3.5 và Hình 3.6.

Đại học Công nghệ - ĐHQGHN

```

public class GuessGame {
    Player p1;
    Player p2;
    Player p3;

    public void startGame() {
        p1 = new Player();
        p2 = new Player();
        p3 = new Player();
        int guessp1 = 0;
        int guessp2 = 0;
        int guessp3 = 0;
        boolean p1isRight = false;
        boolean p2isRight = false;
        boolean p3isRight = false;
        int targetNumber = (int) (Math.random() * 10);
        System.out.println("I'm thinking of a number between 0 and 9...");
        while(true) {
            System.out.println("Number to guess is " + targetNumber);

            p1.guess();
            p2.guess();
            p3.guess();

            guessp1 = p1.number;
            System.out.println("Player one guessed " + guessp1);
            guessp2 = p2.number;
            System.out.println("Player two guessed " + guessp2);
            guessp3 = p3.number;
            System.out.println("Player three guessed " + guessp3);

            if (guessp1 == targetNumber) {
                p1isRight = true;
            }
            if (guessp2 == targetNumber) {
                p2isRight = true;
            }
            if (guessp3 == targetNumber) {
                p3isRight = true;
            }
        }
        if (p1isRight || p2isRight || p3isRight) {
            System.out.println("We have a winner!");
            System.out.println("Player one got it right? " + p1isRight);
            System.out.println("Player two got it right? " + p2isRight);
            System.out.println("Player three got it right? " + p3isRight);
            System.out.println("Game is over");
            break;
        }
        else {
            System.out.println("Players will have to try again.");
        }
    }
}

```

*GuessGame có 3 biến thực thể dành cho 3 đối tượng Player*  
*tạo 3 đối tượng Player và gán cho 3 biến thực thể*  
*khai báo 3 biến để lưu 3 giá trị mà 3 đấu thủ đoán*  
*khai báo 3 biến để lưu giá trị đúng/sai tùy theo câu trả lời của các đấu thủ*  
*sinh 1 số để 3 đấu thủ đoán*  
*yêu cầu từng đấu thủ đoán (gọi phương thức guess())*  
*lấy kết quả đoán của từng đấu thủ*  
*Kiểm tra từng người xem đoán đúng không, nếu đúng thì đặt biến của người đó về true. Nhớ rằng ta đã đặt giá trị mặc định của các biến đó là false*  
*nếu có ít nhất 1 người đoán đúng (|| là toán tử HOẶC)*  
*nếu không thì lặp lại việc yêu cầu đoán số*

Hình 3.5: Mã nguồn GuessGame.java.

Đại học Công nghệ - ĐHQGHN

```

public class Player {
    int number = 0;
    public void guess()
    {
        number = (int) (Math.random() * 10);
        System.out.println("I'm guessing " + number);
    }
}

public class GameLauncher {
    public static void main (String[] args) {
        GuessGame game = new GuessGame();
        game.startGame();
    }
}

```

Hình 3.6: Player.java và GameLauncher.java

### Những điểm quan trọng:

- Tất cả mã Java đều nằm trong một **lớp** nào đó.
- Một lớp đặc tả cách tạo một đối tượng thuộc lớp đó. Một lớp giống như một bản thiết kế
- Một đối tượng có thể tự lo cho bản thân, ta không phải cần biết hay quan tâm một đối tượng làm việc đó *như thế nào*.
- Một đối tượng **biết** về một số thứ và có thể **làm** một số việc.
- Những gì một đối tượng biết về chính nó được gọi là các **biến thực thể (thuộc tính)** của nó. Chúng đại diện cho trạng thái của đối tượng đó.
- Những gì một đối tượng có thể làm được gọi là các **phương thức**. Chúng đại diện cho hành vi của đối tượng đó.
- Khi viết một lớp, ta có thể muốn viết một lớp khác để test. Tại đó ta tạo các đối tượng thuộc lớp kia và thử nghiệm với chúng.
- Tại thời gian chạy, một chương trình Java chính là một nhóm các đối tượng đang "nói chuyện" với nhau.

## Bài tập

1. Điền vào chỗ trống các từ thích hợp (lớp, đối tượng, phương thức, biến thực thể):

- \_\_\_\_\_ được biên dịch từ một file .java.
- \_\_\_\_\_ đóng vai trò như một khuôn mẫu.
- \_\_\_\_\_ thực hiện các công việc.
- \_\_\_\_\_ có thể có nhiều phương thức.
- \_\_\_\_\_ biểu diễn 'trạng thái'
- \_\_\_\_\_ có các hành vi.
- \_\_\_\_\_ được đặt trong các đối tượng.
- \_\_\_\_\_ được dùng để tạo các thực thể đối tượng.
- \_\_\_\_\_ có thể thay đổi khi chương trình chạy
- \_\_\_\_\_ có các phương thức.

2. Tìm và sửa lỗi của các chương trình sau (mỗi phần là nội dung của một file mã nguồn hoàn chỉnh).

a)

```
class SoundDeck {
    boolean canRecord = false;

    void play() {
        System.out.println("playing");
    }

    void record() {
        System.out.println("recording");
    }
}

class SoundDeckTestDrive {
    public static void main(String [] args) {
        t.canRecord = true;
        t.play();
        if (t.canRecord == true) {
            t.record();
        }
    }
}
```

b)

Đại học Công nghệ - ĐHQGHN

```
class DVDPlayer {
    boolean canRecord = false;

    void recordDVD() {
        System.out.println("DVD recording");
    }
}

class DVDPlayerTestDrive {
    public static void main(String [] args) {
        DVDPlayer d = new DVDPlayer();
        d.canRecord = true;
        d.playDVD();
        if (d.canRecord == true) {
            d.recordDVD();
        }
    }
}
```

Đại học Công nghệ - ĐHQGHN



## Chương 4. BIẾN VÀ CÁC KIỂU DỮ LIỆU

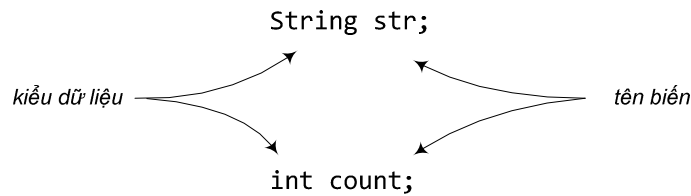
Trong các ví dụ ở các chương trước, ta đã gặp các biến được sử dụng ở hai môi trường: (1) biến thực thể là trạng thái của đối tượng, và (2) biến địa phương là biến được khai báo bên trong một phương thức. Sau này, ta sẽ dùng biến ở dạng đối số (các giá trị được truyền vào trong phương thức bởi lời gọi phương thức), và ở dạng giá trị trả về (giá trị do phương thức trả về cho nơi gọi nó). Ta đã gặp các biến được khai báo với kiểu dữ liệu cơ bản, ví dụ kiểu `int`, và các biến được khai báo thuộc kiểu đối tượng như `String`, `Cow`, `PhoneBookAddress`. Trong chương này, ta sẽ mô tả kĩ về các loại biến của Java, cách khai báo và sử dụng biến.

Java là **ngôn ngữ định kiểu mạnh** (*strongly-typed language*). Nghĩa là, biến nào cũng có kiểu dữ liệu xác định và phải được khai báo trước khi sử dụng. Trình biên dịch không cho phép gán một giá trị kiểu `Cow` vào một biến kiểu `String`, chuyện gì xảy ra nếu ta gọi phương thức `length()` của biến `String` đó để lấy độ dài chuỗi ký tự? Java cũng không cho phép gán một giá trị kiểu số thực với dấu chấm động (chẳng hạn `float`) vào một biến kiểu số nguyên (chẳng hạn `int`), trình biên dịch sẽ phát hiện và báo lỗi. Ta phải dùng phép đổi kiểu một cách tường minh để làm việc này, biết rằng việc đó có thể làm giảm độ chính xác của giá trị.

Các kiểu dữ liệu của Java được chia thành hai loại: **dữ liệu cơ bản** (*primitive*) và **tham chiếu đối tượng** (*object reference*).

Các kiểu dữ liệu cơ bản dành cho các giá trị cơ bản như các số hay các ký tự. Ví dụ như các kiểu `char` (ký tự), `int`. Còn các tham chiếu đối tượng là các tham chiếu tới đối tượng. Nghe có vẻ không rõ ràng hơn được chút nào, nhưng ta sẽ quay lại khái niệm "tham chiếu" này sau (nếu ta đã biết về C/C++ thì khái niệm này gần giống với con trỏ tới đối tượng). Nhưng dù thuộc loại dữ liệu nào, mỗi biến đều cần có một cái tên và thuộc một kiểu dữ liệu cụ thể. Khi ta nói một đối tượng thuộc lớp `X`, điều đó cũng có ý rằng đối tượng đó thuộc kiểu dữ liệu `X`.

Đại học Công nghệ - HHOUGHN



Hình 4.1. Mỗi biến cần có một kiểu dữ liệu và một cái tên

## 4.1. BIẾN VÀ CÁC KIỂU DỮ LIỆU CƠ BẢN

Trước hết, ta bàn về các kiểu dữ liệu cơ bản. Biến thuộc một kiểu dữ liệu cơ bản có kích thước cố định tùy theo đó là kiểu dữ liệu gì (xem Bảng 4.1 liệt kê các kiểu dữ liệu cơ bản của Java).

Kiểu	Mô tả	Kích thước	Khoảng giá trị
char	ký tự đơn (Unicode)	2 byte	tất cả các giá trị Unicode từ 0 đến 65.535
boolean	giá trị boolean	1 bit	true hoặc false
short	số nguyên	2 byte	-32.767 đến 32.767
int	số nguyên	4 byte	-2.147.483.648 tới 2.147.483.647
long	số nguyên	8 byte	-9.223.372.036.854.775.808 tới 9.223.372.036.854.775.808
float	số thực dấu phẩy động	4	+/- 1,4023x10 <sup>-45</sup> tới 3,4028x10 <sup>38</sup>
double	số thực dấu phẩy động	8	+/- 4,9406x10 <sup>-324</sup> tới 1,7977x10 <sup>308</sup>

Bảng 4.1: Các kiểu dữ liệu cơ bản của Java.

Tại mỗi thời điểm, biến đó lưu trữ một giá trị. Khi gán một giá trị khác cho biến đó, giá trị mới sẽ thay thế cho giá trị cũ (bị ghi đè). Ta có thể dùng phép gán để ghi giá trị mới cho một biến theo nhiều cách, trong đó có:

- dùng một giá trị trực tiếp sau dấu gán. Ví dụ:

```
x = 10; isCrazy = true; bloodType = 'A';
```

- lấy giá trị của biến khác. Ví dụ:

```
x = y;
```

- kết hợp hai cách trên trong một biểu thức. Ví dụ:

```
x = y + 1;
```

Thông thường, ta không thể ghi một giá trị kích thước lớn vào một biến thuộc kiểu dữ liệu nhỏ. Trình biên dịch sẽ báo lỗi nếu phát hiện ra. Ví dụ:

```
int x = 10;
byte b = x;    // compile error!
```

Tuy rằng rõ ràng 10 là một giá trị đủ bé để lưu trong một biến kiểu byte, nhưng trình biên dịch không quan tâm đến giá trị, nó chỉ biết rằng ta đang cố lấy nội dung của một biến kiểu int với kích thước lớn hơn để ghi vào một biến kiểu byte với kích thước nhỏ hơn.

Như đã thấy tại các ví dụ trước, biến thuộc các kiểu dữ liệu cơ bản được gọi đến bằng tên của nó. Ví dụ sau lệnh khai báo `int a`; ta có một biến kiểu int có tên là `a`, mỗi khi cần thao tác với biến này, ta dùng tên `a` để chỉ định biến đó, ví dụ `a = 5`;. Vậy có những quy tắc gì liên quan đến tên biến?

**Định danh** (*identifier*) là thuật ngữ chỉ tên (tên biến, tên hàm, tên lớp...). Java quy định định danh là một chuỗi kí tự viết liền nhau, (bao gồm các chữ cái a..z, A..Z, chữ số 0..9, dấu gạch chân '\_'). Định danh không được bắt đầu bằng chữ số và không được trùng với các **từ khóa** (*keyword*). Từ khóa là từ mang ý nghĩa đặc biệt của ngôn ngữ lập trình, chẳng hạn ta đã gặp các từ khóa của Java như `public`, `static`, `void`, `int`, `byte`... Lưu ý, Java phân biệt chữ cái hoa và chữ cái thường.

Cách đặt tên biến tuân thủ theo cách đặt tên định danh. Tên biến nên dễ đọc, và gọi nhớ đến công dụng của biến hay kiểu dữ liệu mà biến sẽ lưu trữ. Ví dụ, nếu cần dùng một biến để lưu số lượng quả táo, ta có thể đặt tên là `totalApples`. Không nên sử dụng các tên biến chỉ gồm một kí tự và không có ý nghĩa như `a` hay `b`. Theo thông lệ, tên lớp bắt đầu bằng một chữ viết hoa (ví dụ `String`), tên biến bắt đầu bằng chữ viết thường (ví dụ `totalApples`); ở các tên cấu tạo từ nhiều từ đơn, các từ từ thứ hai trở đi được viết hoa để "tách" nhau.

## 4.2. THAM CHIẾU ĐỐI TƯỢNG VÀ ĐỐI TƯỢNG

Biến kiểu cơ bản chỉ lưu các giá trị cơ bản. Vậy còn các đối tượng thì sao?

Thực ra, trong Java không có khái niệm biến **đối tượng**, mà chỉ có biến **tham chiếu** đối tượng. Một biến tham chiếu đối tượng lưu trữ các bit đại diện cho một cách truy nhập tới một đối tượng. Biến tham chiếu không lưu trữ chính đối tượng đó. Có thể nói rằng nó lưu cái gì đó giống như một con trỏ, hay địa chỉ của đối tượng trong bộ nhớ máy tính. Ta không biết chính xác giá trị đó là cái gì. Chỉ cần biết rằng giá trị đó đại diện cho một và chỉ một đối tượng, và rằng máy ảo Java biết cách dùng tham chiếu đó để truy nhập đối tượng.

Nói cách khác, về bản chất, các biến kiểu cơ bản hay các biến tham chiếu đều là các ô nhớ chứa đầy các bit 0 và 1. Sự phân biệt giữa hai loại biến này nằm ở ý nghĩa của các bit đó. Đối với một biến kiểu cơ bản, các bit của nó biểu diễn giá trị thực của biến. Còn các bit của biến tham chiếu biểu diễn cách truy nhập tới một đối tượng.

Nhớ lại ví dụ trong Hình 3.2, với các lệnh

```
Cow c = new Cow();
```

```
c.moo();
```

Ta có thể coi biến tham chiếu c như là một cái điều khiển từ xa của đối tượng bò được sinh ra từ lệnh new Cow(). Ta dùng cái điều khiển đó kèm với toán tử dấu chấm (.) để yêu cầu con bò rống lên một hồi (bấm nút "moo" của cái điều khiển từ xa để kích hoạt phương thức moo() của đối tượng).

Tương tự như vậy, ta lấy ví dụ:

```
String s1 = new String("Hello, ");  
System.out.println(s1.length());
```

Ta có s1 là biến tham chiếu kiểu String. Nó được chiếu tới đối tượng kiểu String được tạo ra bởi biểu thức new String("Hello, "). Tại đây, đối tượng kiểu String vừa tạo không có tên, s1 không phải tên của nó mà là tham chiếu hiện đang chiếu tới đối tượng đó và là cách duy nhất để tương tác với nó. Ta gọi hàm length() của đối tượng đó để lấy độ dài của nó bằng cách dùng tham chiếu s1 trong biểu thức s1.length().

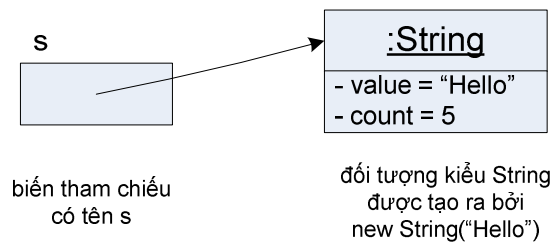
Nhấn mạnh, **một biến tham chiếu đối tượng không phải là một đối tượng, nó chỉ đóng vai trò như một con trỏ tới một đối tượng nào đó**. Tuy rằng, trong ngôn ngữ thông thường, ta hay dùng các cách nói như "*Ta truyền đối tượng kiểu String s1 vào cho phương thức System.out.println()*" hay "*Ta tạo một đối tượng Cow mới với tên c*", s1 hay c không phải tên của các đối tượng đó, chúng chỉ là các tham chiếu. Thực chất, các đối tượng không có tên, chúng cũng không nằm trong biến nào. Trong Java, các đối tượng được tạo ra đều nằm trong bộ nhớ heap.

Hình 4.2 minh họa quan hệ giữa biến s và đối tượng kiểu String<sup>5</sup> mà nó chiếu tới. Cụ thể, tại ví dụ đang xét, s và đối tượng nó chiếu tới nằm tại hai loại bộ nhớ khác nhau: đối tượng xâu "Hello" nằm trong heap, còn biến s nằm trong vùng bộ nhớ stack dành cho các biến địa phương của hàm main(). Sự khác biệt về vị trí của hai ô dữ liệu này dẫn đến độ dài cuộc đời của chúng. Một biến tham chiếu là biến địa phương của một hàm sẽ kết thúc sự tồn tại của mình sau khi hàm kết thúc. Còn đối tượng được tạo ra từ bên trong hàm đó vẫn tiếp tục tồn tại cho đến khi nào được máy ảo Java giải phóng – sau khi đối tượng đó không còn được dùng đến nữa.

---

<sup>5</sup> Từ nay, "đối tượng kiểu X", với X là một lớp, sẽ được viết ngắn gọn thành "đối tượng X".

```
String s = new String("Hello");
```



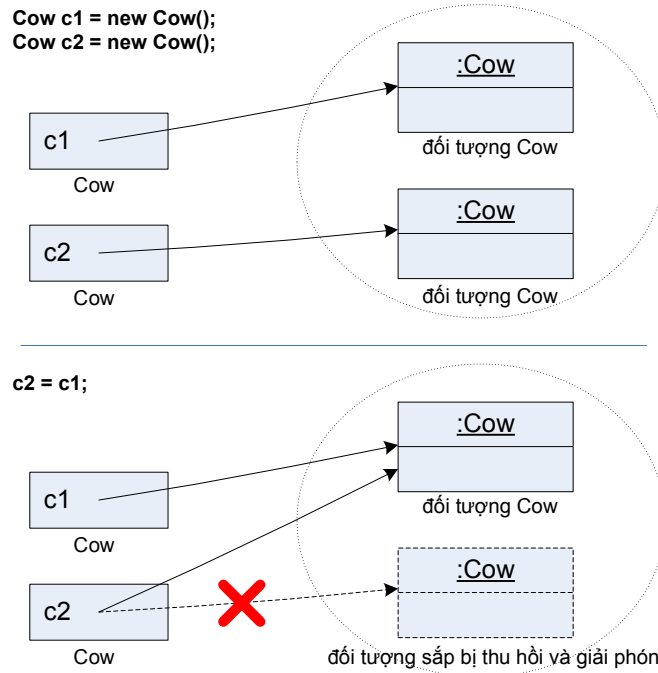
Hình 4.2. Biến tham chiếu s và đối tượng kiểu String

Với dòng lệnh `String s = new String("Hello");` như trong Hình 4.2, có ba bước khai báo, tạo và gán đối tượng và tham chiếu đối tượng. Bước 1, `String s`, khai báo một biến tham chiếu có kiểu cố định là `String` và được đặt tên là `s`. Bước 2, `new String("Hello")`, yêu cầu máy ảo Java cấp phát bộ nhớ cho một đối tượng `String` mới, đặt tại heap, với dữ liệu khởi tạo là chuỗi "Hello". Bước 3, `=`, là phép gán gắn biến tham chiếu `s` với đối tượng `String` vừa tạo, từ nay có thể dùng `s` làm một cái điều khiển từ xa đối với đối tượng đó.

**Tham chiếu null** là tham chiếu đang nhận giá trị `null` – không chiếu tới một đối tượng nào hết. Nếu chương trình truy nhập biến thực thể hoặc gọi phương thức từ một tham chiếu `null`, nghĩa là không có đối tượng nào để truy nhập các biến thực thể hoặc gọi phương thức của nó, khi thực thi đến lệnh đó, chương trình sẽ sập vì gặp lỗi `NullPointerException` (con trỏ null). Cần cẩn thận tránh lỗi này bằng cách kiểm tra tham chiếu `null` trước khi truy nhập đối tượng qua tham chiếu đó.

Đối với một đối tượng, lời gọi lệnh `new` như trong bước 2 là giai đoạn mở đầu. Trước khi ta có thể làm bất cứ việc gì đối với một đối tượng mới, nó phải được khởi tạo, nghĩa là các biến thực thể của nó phải được gán giá trị ban đầu. Khi ta dùng lệnh `new`, Java thực hiện tự động công việc này bằng cách gọi một phương thức đặc biệt được gọi là **hàm khởi tạo** (*constructor*). Phương thức này không trả về giá trị nào và có tên trùng với tên lớp. Một lớp có thể có nhiều hơn một hàm khởi tạo với danh sách tham số khác nhau. Trình biên dịch sẽ dựa vào danh sách đối số tại lời gọi `new` để gọi hàm khởi tạo tương ứng. Chi tiết về hàm khởi tạo được nói đến trong mục 9.2.

Đối tượng mới được tạo sẽ tồn tại trong bộ nhớ chừng nào ta còn có một tham chiếu nào đó chiếu tới nó. Khi một đối tượng không còn một tham chiếu nào chiếu tới, ta không có cách nào sử dụng đối tượng đó nữa. Ví dụ như trong Hình 4.3, sau khi ta chiếu biến `c2` đến chỗ khác, ta mất hoàn toàn 'liên lạc' đối với đối tượng `Cow` thứ hai. Nói cách khác, nó đã bị bỏ rơi và do đó sẽ được bộ phận dọn rác (*garbage collector*) của máy ảo Java thu hồi để tái sử dụng vùng bộ nhớ mà nó đã chiếm giữ. Chi tiết về nội dung này được nói đến trong Chương 9.



Hình 4.3. Đối tượng sẽ bị thu hồi khi không còn biến tham chiếu nào gắn với nó.

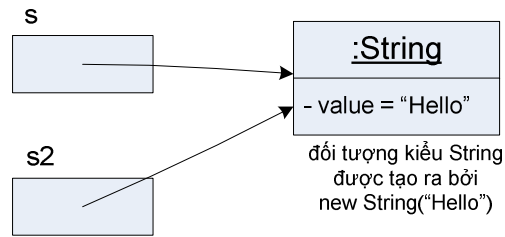
### 4.3. PHÉP GÁN

Cũng như ta có thể gán một giá trị mới cho một biến kiểu cơ bản, ta cũng có thể dùng phép gán để chiếu một biến tham chiếu tới một đối tượng khác khi cần, miễn là đối tượng đó phải thuộc cùng kiểu.

Thể hiện đúng bản chất của tham chiếu đối tượng, và hoạt động sao chép nội dung của phép gán, phép gán xảy ra giữa hai biến tham chiếu không có tác dụng sao chép nội dung của đối tượng này sang đối tượng khác. Phép gán chỉ sao chép chuỗi bit của biến tham chiếu này sang biến tham chiếu kia. Kết quả là biến tham chiếu ở vế trái được trỏ tới đối tượng mà biến/biểu thức tham chiếu tại vế bên phải đang chiếu tới.

Hình 4.4 minh họa kết quả của một phép gán biến tham chiếu. Biến `String s2` sau khi nhận giá trị của `s` thì chiếu tới cùng một đối tượng `String` mà `s` khi đó đang chiếu tới. Phép gán đối với các biến tham chiếu không tạo ra một bản sao của đối tượng. Vậy nếu ta muốn sao chép nội dung đối tượng thì làm thế nào? Vấn đề này sẽ được nói đến trong Chương 9.

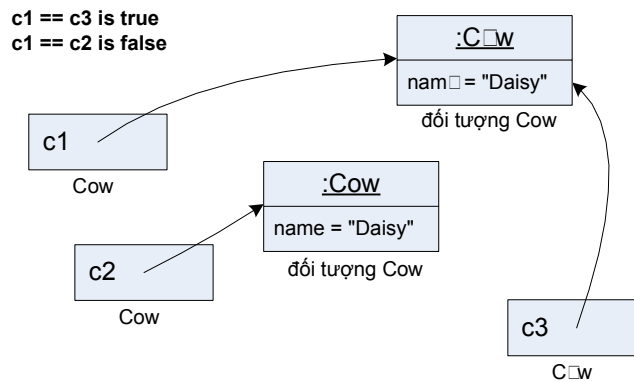
```
String s = new String("Hello");
String s2 = s;
```



Hình 4.4. Phép gán đối với biến tham chiếu.

#### 4.4. CÁC PHÉP SO SÁNH

Cũng tương tự như phép gán, các phép so sánh `==` và `!=` đối với các biến tham chiếu so sánh chuỗi bit nằm trong các biến đó. Ta biết rằng chuỗi bit của hai tham chiếu sẽ giống hệt nhau nếu chúng cùng chiếu tới một đối tượng. Nói cách khác, so sánh hai biến tham chiếu là kiểm tra xem chúng có trở tới cùng một đối tượng hay không. Các phép so sánh tham chiếu không hề so sánh nội dung đối tượng mà tham chiếu chiếu tới. Trong ví dụ Hình 4.5, `c1` và `c3` bằng nhau vì chúng chiếu tới cùng một đối tượng. Còn `c1` và `c2` khác nhau vì chúng chiếu tới hai đối tượng nằm tại hai chỗ khác nhau trong bộ nhớ, bất kể hai đối tượng đó có "giống nhau" về nội dung hay không.



Hình 4.5. So sánh tham chiếu.

Các phép so sánh lớn hơn, nhỏ hơn không có ý nghĩa và không thể dùng cho các kiểu tham chiếu đối tượng.

Để so sánh nội dung của các đối tượng, ta có những cách khác sẽ được bàn đến trong những chương sau (các mục 8.5 và 13.5).

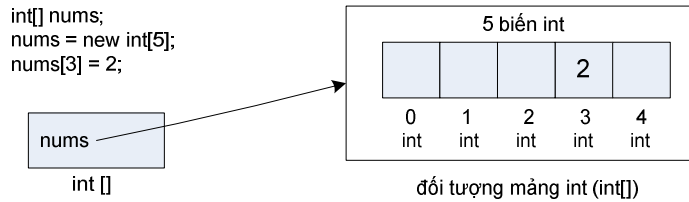
## 4.5. MẢNG

Về mặt hình tượng, **mảng** (*array*) là một chuỗi các biến thuộc cùng một loại được đánh số thứ tự. Ví dụ một mảng `int` kích thước 5 là một chuỗi liên tục 5 biến kiểu `int` được đánh số thứ tự từ 0 tới 4. Một mảng Java thực chất là một đối tượng. Một biến mảng là tham chiếu tới một đối tượng mảng.

Ví dụ:

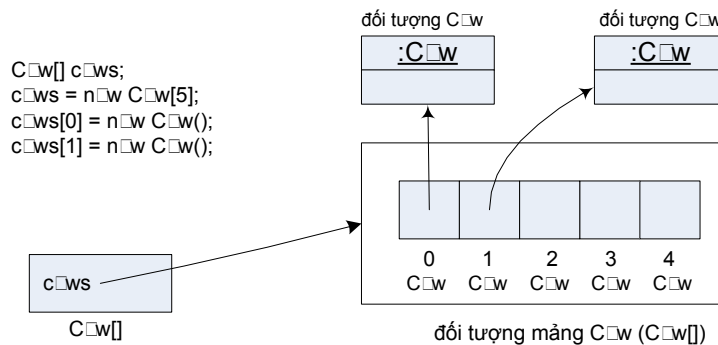
```
int[] nums;  
nums = new int[5];  
nums[3] = 2;
```

Lệnh thứ nhất khai báo biến tham chiếu `nums` kiểu mảng `int` (`int[]`). Nó sẽ là cái điều khiển từ xa của một đối tượng mảng. Lệnh thứ hai tạo một mảng `int` với độ dài 5 và gán nó với biến `nums` đã được khai báo trước đó. Lệnh thứ ba gán giá trị 2 cho phần tử có chỉ số 3 trong mảng.



Hình 4.6. Tham chiếu và đối tượng mảng `int`.

Ví dụ trên minh họa mảng gồm các phần tử kiểu cơ bản. Mỗi phần tử mảng kiểu `int` là một biến kiểu `int`. Vậy còn mảng `Cow` hay mảng `String` thì sao? Cũng y hệt như vậy, mảng `Cow` chứa các biến kiểu `Cow`, nghĩa là các tham chiếu đối tượng `Cow` (cái điều khiển từ xa chứ không phải bản thân đối tượng `Cow`).



Hình 4.7. Tham chiếu và đối tượng mảng `Cow`.

Tóm lại, mảng có thể được khai báo để chứa các phần tử thuộc kiểu cơ bản hoặc kiểu tham chiếu đối tượng. Tùy theo mảng được khai báo kiểu dữ liệu gì thì chứa các phần tử là biến thuộc kiểu dữ liệu đó. Tuy nhiên, dù các phần tử thuộc kiểu cơ bản hay tham chiếu đối tượng thì bản thân mỗi mảng là một đối tượng, và biến mảng là tham chiếu tới đối tượng mảng.



Thao tác đối với các phần tử mảng kiểu Cow có khác gì với việc thao tác một biến kiểu Cow? Ta cũng dùng toán tử (.) như bình thường, nhưng vì phần tử mảng không có tên biến, thay vào đó, ta dùng kí hiệu phần tử của mảng. Ví dụ, với lớp Cow được định nghĩa như trong Hình 4.8, ta dùng các tham chiếu mảng để thao tác với các phần tử mảng Cow như trong Hình 4.9.

<pre>public class Cow {     String name;     public void eat() {}     public void produceMilk() {}     public void moo() {         System.out.println(name + " says Moooo...");     } }</pre>	Cow
	name
	eat() produceMilk() moo()

Hình 4.8: Cow.java

```
public class CowArrayDemo {
    public static void main(String[] args) {
        Cow cow1 = new Cow();
        cow1.moo();
        cow1.name = "Lazy";

        Cow[] myCows = new Cow[3];
        myCows[0] = new Cow();
        myCows[1] = new Cow();
        myCows[2] = cow1;

        myCows[0].name = "Daisy";
        myCows[1].name = "Lady";

        System.out.print("last cow's name is ");
        System.out.println(myCows[2].name);

        int x = 0;
        while (x < myCows.length) {
            myCows[x].moo();
            x = x+1;
        }
    }
}
```

```
%> java CowArrayDemo
null says Moooo...
last cow's name is Lazy
Daisy says Moooo...
Lady says Moooo...
Lazy says Moooo...
```

*Mảng có một biến 'length'  
cho ta biết số phần tử của mảng*

Hình 4.9: CowArrayTest.java

Ta thường dùng vòng for để duyệt các phần tử của một mảng. Ví dụ, đoạn mã duyệt mảng myCows và in ra tên của từng con bò trong đó có thể được viết như sau:

```
for (int x = 0; x < myCows.length; x++) {
    System.out.println(myCows[x].name);
}
```

Ngoài cú pháp thông dụng như ở trên, vòng for duyệt mảng còn có một cách viết ngắn gọn hơn, đó là vòng **for-each**. Ví dụ, ta có thể viết lại vòng for trên như dưới đây:

```
for (Cow aCow : myCows) {  
    System.out.println(aCow.name);  
}
```

Trong đó, ta khai báo biến chạy aCow là biến kiểu Cow, biến chạy sẽ chạy từ đầu đến cuối mảng myCows, mỗi lần lại lấy giá trị bằng giá trị của phần tử hiện tại trong mảng (trong ví dụ này, giá trị đó là một tham chiếu tới một đối tượng Cow).

Vòng for-each có thể áp dụng cho mảng thuộc kiểu dữ liệu tham chiếu cũng như kiểu cơ bản, ngoài ra còn dùng được cho các cấu trúc collection của thư viện Java mà ta sẽ nói đến trong Chương 13.

Đại học Công nghệ - ĐHQGHN

## Bài tập

1. Điền từ thích hợp vào chỗ trống trong mỗi câu sau:
  - a) Biến thực thể thuộc các kiểu char, byte, short, int, long, float, và double đều có giá trị mặc định là \_\_\_\_\_.
  - b) Biến thực thể thuộc kiểu boolean có giá trị mặc định là \_\_\_\_\_.
  - c) Các kiểu dữ liệu trong Java được phân thành hai loại: các kiểu \_\_\_\_\_ và các kiểu \_\_\_\_\_.
2. Các phát biểu sau đây đúng hay sai?
  - a) Có thể gọi phương thức từ một biến kiểu cơ bản.
  - b) Các đối tượng được tạo ra đều tồn tại trong bộ nhớ heap cho đến khi chương trình kết thúc
  - c) Lúc nào một đối tượng thuộc diện dùng được cũng cần phải có một tham chiếu chiếu tới nó.
  - d) Các giá trị có dạng dấu chấm động trong mã nguồn được hiểu mặc định là các giá trị trực tiếp dấu chấm động thuộc kiểu float.
3. Biến thực thể dùng để làm gì?
4. Tham số của phương thức main() là một mảng String. Mảng này là danh sách các tham số dòng lệnh khi ta chạy chương trình. Ví dụ, khi chạy lệnh java CowArrayDemo foo bar từ dấu nhắc cửa sổ lệnh. Mảng args[] sẽ chứa các chuỗi ký tự foo và bar. Hãy viết một chương trình in ra màn hình tất cả các tham số dòng lệnh đã nhận được.
5. Tìm và sửa lỗi của các chương trình sau (mỗi phần là một file mã nguồn hoàn chỉnh).

a)

```
class Books {
    String title;
    String author;
}

class BooksTestDrive {
    public static void main(String [] args)
        Books [] myBooks = new Books(3);
        int x = 0;
        myBooks[0].title = "The Grapes of Java";
        myBooks[1].title = "The Java Gatsby";
        myBooks[2].title = "The Java Cookbook";
        myBooks[0].author = "bob";
        myBooks[1].author = "sue";
        myBooks[2].author = "ian";

        while (x < 3) {
            System.out.print(myBooks[x].title);
            System.out.print(" by ");
            System.out.println(rnyBooks[x].author) ;
            x = x + 1;
        }
    }
}
```

b)

```
class Hobbits {
    String name;

    public static void main(String [] args) {
        Hobbits [] h = new Hobbits[3];
        int z = 0;

        while (z < 4) {
            z = z + 1;
            h[z] = new Hobbits();
            h[z].name = "bilbo";
            if (z == 1) {
                h[z].name= "frodo";
            }
            if (z==2){
                h[z].name = "sam";
            }
            System.out.print (h[z].name + " is a ");
            System.out.println("good Hobbit name");
        }
    }
}
```

6. Cho chương trình sau, liệt kê các đối tượng HeapQuiz được tạo ra; hỏi đến đoạn //do stuff thì các phần tử mảng hq[0] cho tới hq[4] chiếu tới các đối tượng nào.

```

class HeapQuiz {
    int id == 0;
    public static void main(String [] args) {
        int x = 0;
        HeapQuiz [] hq = new HeapQuiz[5];
        while ( x < 3 ) {
            hq[x] = new HeapQuiz();
            hq[x].id = x;
            x = x + 1;
        }
        hq[3] = hq[1];
        hq[4] = hq[1];
        hq[3] = null;
        hq[4] = hq[0];
        hq[0] = hq[3];
        hq[3] = hq[2];
        hq[2] = hq[0] ;
        // do stuff
    }
}

```

7. Dành nhờ Tí và Sửu giúp viết nhanh một đoạn mã xử lý danh bạ điện thoại cho điện thoại di động, người nào có giải pháp tốt hơn sẽ được trả công là một túi bông ngô. Sau khi nghe Dành mô tả, Sửu viết lên bảng đoạn mã sau:

```

Contact[] a = new Contact[10];
while (x < 10) { //tạo 10 đối tượng Contact
    a[x] = new Contact();
    x = x + 1;
}
// dùng mảng a để cập nhật danh bạ

```

Tí nhìn qua rồi cười "Điện thoại di động bộ nhớ bé tí mà cậu hoang phí quá!". Nói đoạn, Tí viết:

```

Contact ref;
while (x < 10) { //tạo 10 đối tượng Contact
    ref = new Contact();
    x = x + 1;
}
// dùng ref để cập nhật danh bạ

```

Viết xong, Tí hề hả "Bông ngô là của tớ rồi!". Dành cười "Tiết kiệm bộ nhớ hơn thật, nhưng cậu phải ăn ké Sửu thôi."

Tại sao Dành lại quyết định như vậy?

Đại học Công nghệ - QHOGHN

## Chương 5. HÀNH VI CỦA ĐỐI TƯỢNG

Trạng thái ảnh hưởng đến hành vi, hành vi ảnh hưởng đến trạng thái. Ta đã biết rằng đối tượng có trạng thái và hành vi, chúng được biểu diễn bởi các biến thực thể và các phương thức. Ta cũng đã biết rằng mỗi thực thể của một lớp (mỗi đối tượng thuộc một lớp) có các giá trị riêng cho các biến thực thể. Chẳng hạn đối tượng Cow này có tên (name) là "Lady" và nặng (weight) 80 kg, trong khi một đối tượng Cow khác tên là "Daisy" và nặng 150kg. Hai đối tượng đó thực hiện phương thức moo() có khác nhau hay không? Có thể, vì mỗi đối tượng có hành vi thể hiện tùy theo trạng thái của nó. Nói cách khác, phương thức gọi từ đối tượng nào sẽ sử dụng giá trị của các biến thực thể của đối tượng đó. Chương này sẽ xem xét mối quan hệ tương hỗ này.

### 5.1. PHƯƠNG THỨC VÀ TRẠNG THÁI ĐỐI TƯỢNG

Nhớ lại rằng lớp là khuôn mẫu để tạo ra các đối tượng thuộc lớp đó. Khi ta viết một lớp, ta mô tả cách xây dựng một đối tượng thuộc lớp đó. Ta đã biết rằng giá trị của cùng một biến thực thể của các đối tượng khác nhau có thể khác nhau. Nhưng còn các phương thức thì sao? Chúng có hoạt động khác nhau hay không? Đại loại là có. Mỗi thực thể của một lớp đều có chung các phương thức, nhưng các phương thức này có thể hoạt động khác nhau tùy theo giá trị cụ thể của các biến thực thể.

Ví dụ, lớp PhoneBookEntry có hai biến thực thể, name và phone. Phương thức display() hiển thị nội dung của đối tượng PhoneBookEntry, cụ thể là giá trị của name và phone của đối tượng đó. Các đối tượng khác nhau có các giá trị khác nhau cho hai biến đó, nên nội dung được display() hiển thị cho các đối tượng đó cũng khác nhau.

tom.display() → Name: Tom the Cat  
Phone: 84208594

jerry.display() → Name: Jerry the Mouse  
Phone: 98768065

Xem lại ví dụ trong Hình 3.3, ta sẽ thấy các lời gọi phương thức display() từ tom và jerry hiện ra kết quả khác nhau trên màn hình, tuy rằng mã nguồn của display() cho tom hay jerry đều là một:

```
void display() {  
    System.out.println("Name: " + name);  
    System.out.println("Phone: " + phone);  
}
```

Thực chất, nội dung trên của `display()` tương đương cách viết như sau:

```
void display() {
    System.out.println("Name: " + this.name);
    System.out.println("Phone: " + this.phone);
}
```

Trong đó, **this** là từ khóa có ý nghĩa là một tham chiếu đặc biệt chiếu tới đối tượng chủ của phương thức hiện hành. Chẳng hạn, đối với lời gọi `tom.display()`, `this` có giá trị bằng giá trị của tham chiếu `tom`; đối với lời gọi `jerry.display()`, `this` có giá trị bằng `jerry`. Có thể nói rằng khi gọi một phương thức đối với một đối tượng, tham chiếu tới đối tượng đó được truyền vào phương thức tới một tham số ẩn: tham chiếu `this`.

Tham chiếu `this` có thể được dùng để truy cập biến thực thể hoặc gọi phương thức đối với đối tượng hiện hành. Thông thường, công dụng này của `this` chỉ có ích khi tên biến thực thể bị trùng với một biến địa phương hoặc tham số của phương thức. Chẳng hạn, giả sử phương thức `setName()` của lớp `PhoneBookEntry` lấy một tham số `name` kiểu `String` trùng tên với biến thực thể `name` của lớp đó. Từ trong phương thức `setName()`, nếu dùng tên `'name'` thì trình biên dịch sẽ hiểu là ta đang nói đến tham số `name`. Để gọi đến biến thực thể `name`, cách duy nhất là sử dụng tham chiếu `this` để gọi một cách tường minh. Ví dụ như sau:

```
void setName(String name) {
    this.name = name;
}
```

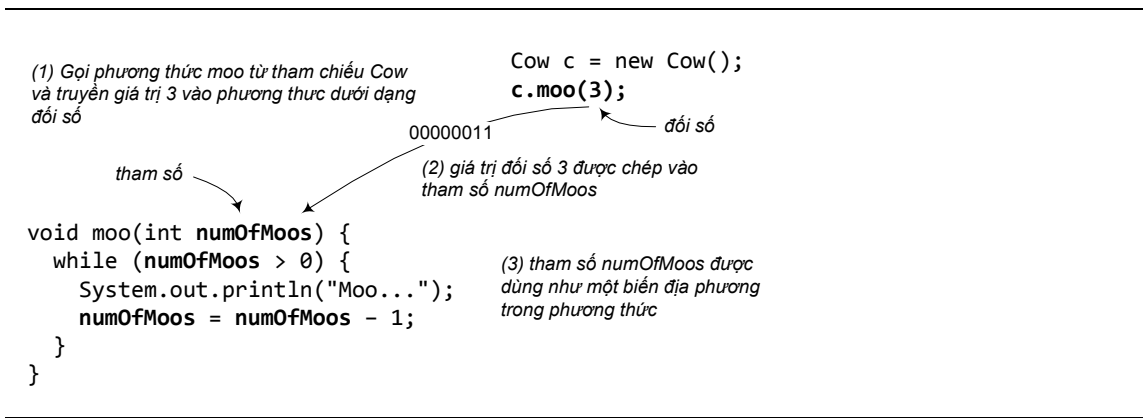
*biến thực thể name*      *tham số name*

## 5.2. TRUYỀN THAM SỐ VÀ GIÁ TRỊ TRẢ VỀ

Cũng như trong các ngôn ngữ lập trình khác, ta có thể truyền các giá trị vào trong phương thức. Ví dụ, ta muốn chỉ thị cho một đối tượng `Cow` về số lần rống cần thực hiện bằng cách gọi phương thức như sau:

```
c.moo(3);
```

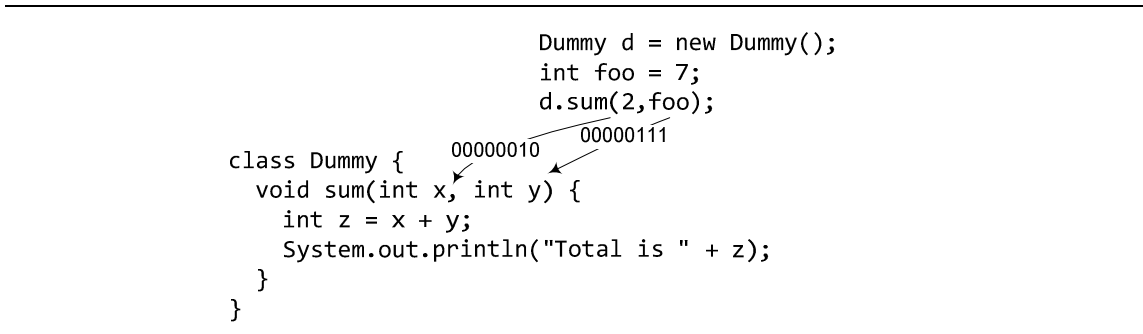
Ta gọi **đối số** (*argument*) là những gì ta truyền vào trong phương thức. Đối với Java, **đối số là một giá trị**, chẳng hạn 3 như trong lời gọi ở trên, hoặc "Hello" như trong `System.out.println("Hello")`, hoặc giá trị của một tham chiếu tới một đối tượng `Cow`. Khi lời gọi phương thức được thực thi, giá trị đối số đó được chép vào một **tham số**. Tham số (*parameter*) thực chất chỉ là một biến địa phương của phương thức – một biến có một cái tên và một kiểu dữ liệu, nó có thể được sử dụng bên trong thân của phương thức.



Hình 5.1: Đối số và tham số.

Điều quan trọng cần nhớ: Nếu một phương thức yêu cầu một tham số, ta phải truyền cho nó một giá trị nào đó, và giá trị đó phải thuộc đúng kiểu được khai báo của tham số.

Phương thức có thể có nhiều tham số. Khi khai báo, ta dùng dấu phẩy để tách giữa chúng. Và khi gọi hàm, ta phải truyền các đối số thuộc đúng kiểu dữ liệu và theo đúng thứ tự đã khai báo.



Hình 5.2: Phương thức có thể có nhiều tham số.

Phương thức có thể trả về giá trị. Mỗi phương thức được khai báo với một kiểu trả về, nhưng cho đến nay, các phương thức ví dụ của ta vẫn dùng kiểu trả về là void, nghĩa là chúng không trả về cái gì.

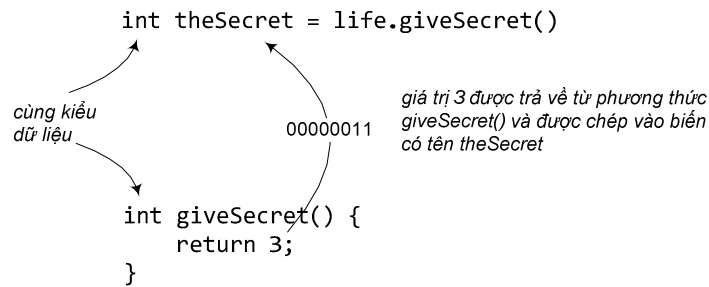
```
void doSomething() {
}
```

Ta có thể khai báo để phương thức trả về cho nơi gọi nó một loại giá trị cụ thể, chẳng hạn:

```
int giveSecret() {
    return 3;
}
```

Phương thức đã khai báo sẽ trả về giá trị thuộc kiểu dữ liệu gì thì phải trả về giá trị thuộc kiểu đó. (Hoặc một giá trị thuộc một kiểu tương thích với kiểu đã khai báo. Ta sẽ bàn chi tiết về điểm này khi nói về đa hình ở Chương 5.)





Hình 5.3: Ví dụ về giá trị trả về từ phương thức

Như đã nói đến ở mục trước, **this** là tham chiếu tới đối tượng hiện hành. Do đó, nếu một phương thức cần trả về tham chiếu tới đối tượng hiện hành, nó dùng lệnh `return this`; Tham chiếu `this` cũng có thể được dùng làm đối số nếu ta cần truyền cho một phương thức một tham chiếu tới đối tượng hiện hành. Chẳng hạn, từ bên trong một phương thức của lớp `Square`, đối tượng hình vuông hiện hành yêu cầu một đối tượng đồ họa `myGraphics` dùng lời gọi `myGraphics.draw(this)`; để vẽ chính hình vuông đó, trong đó, `this` là phương tiện để đối tượng lớp `Square` truyền tham chiếu tới chính mình vào cho phương thức `draw()`.

Hay một ví dụ khác là lớp `MyInteger` trong Hình 5.4. Ví dụ này minh họa các công dụng của tham chiếu `this`. Một điểm đáng chú ý trong ví dụ này là phương thức `increment()` trả về tham chiếu tới chính đối tượng chủ, điều này cho phép gọi phương thức này thành chuỗi như trong phần mã ví dụ sử dụng lớp `MyInteger`.

---

```
class MyInteger {
    private int value;
    public boolean greaterThan(MyInteger other) {
        return (this.value > other.value);
    }
    public boolean lessThan(MyInteger other) {
        return (other.greaterThan(this));
    }
    public MyInteger increment() {
        value++;
        return this;
    }
}
```

---

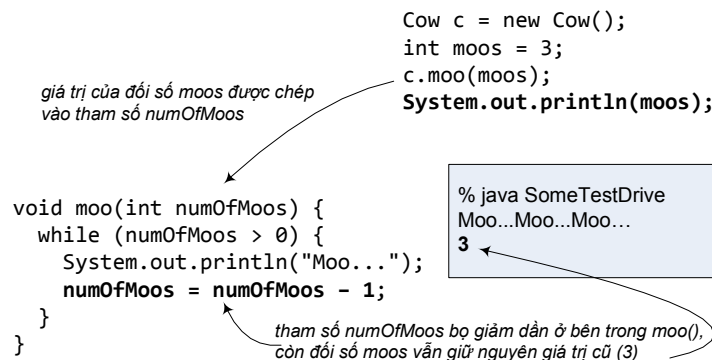
```
MyInteger counter = new MyInteger();
counter.increment().increment(); // tăng hai lần
```

Hình 5.4: Các công dụng của tham chiếu `this` trong phương thức.

### 5.3. CƠ CHẾ TRUYỀN BẰNG GIÁ TRỊ

Ngôn ngữ lập trình sử dụng duy nhất một cơ chế truyền tham số: **truyền bằng giá trị** (*pass-by-value*). Khi một đối số được truyền vào một phương thức, chỉ có giá

trị của nó được chép vào tham số tương ứng. Kể từ đó, các thao tác liên quan của phương thức chỉ được thực hiện trên tham số đó – thực chất là biến địa phương của phương thức. Còn bản thân đối số đó không chịu ảnh hưởng gì của phương thức được gọi.



Hình 5.5: Đối số không chịu ảnh hưởng của tham số.

Cơ chế truyền bằng giá trị hoạt động như thế nào khi đối số là tham chiếu đối tượng? Cũng vậy thôi, giá trị của đối số được chép vào tham số. Và giá trị ở đây, như ta đã nói về bản chất của tham chiếu, là chuỗi bit biểu diễn cách truy nhập đối tượng đang được chiếu tới. Kết quả của việc truyền đối số là ta được tham số cũng là một tham chiếu chiếu tới cùng một đối tượng mà đối số đang chiếu tới. Ta sẽ gặp nhiều ví dụ về việc này trong các chương sau.

### Những điểm quan trọng:

- Lớp định nghĩa những gì mà một đối tượng biết và những gì nó có thể làm.
- Những gì mà một đối tượng biết là các biến thực thể của nó (trạng thái của đối tượng)
- Những gì một đối tượng có thể làm là các phương thức của nó (hành vi của đối tượng)
- Các phương thức có thể sử dụng các biến thực thể của đối tượng, nhờ đó các đối tượng thuộc cùng một lớp có thể có hành xử không giống nhau.
- Một phương thức có thể có các tham số. Ta có thể truyền các giá trị vào phương thức qua các tham số của phương thức.
- Số lượng và kiểu dữ liệu của các giá trị ta truyền vào phương thức (đối số) phải khớp với thứ tự và kiểu dữ liệu của các tham số được khai báo của phương thức.
- Các giá trị truyền vào phương thức hoặc được trả về từ phương thức có thể được ngầm đổi từ kiểu hẹp hơn sang kiểu rộng hơn, hoặc phải được đổi từ mình sang kiểu hẹp hơn.
- Các giá trị dùng làm đối số có thể là một giá trị trực tiếp (1, 'd', v.v..) hoặc một biến hay biểu thức có giá trị thuộc kiểu đã được khai báo cho tham số.

- Một phương thức phải có kiểu trả về. Kiểu trả về void có nghĩa phương thức không trả về giá trị gì. Nếu không, phương thức phải trả về một giá trị tương thích với kiểu trả về đã khai báo.

#### 5.4. ĐÓNG GÓI VÀ CÁC PHƯƠNG THỨC TRUY NHẬP

Các tham số và giá trị trả về được sử dụng đặc lực nhất trong các phương thức có nhiệm vụ truy nhập dữ liệu của đối tượng. Có hai loại phương thức truy nhập:

- Các phương thức đọc dữ liệu của đối tượng và trả về dữ liệu đọc được. Chúng thường được đặt tên là `getDữLiệuGiĐó`, nên còn được gọi là các **phương thức get**.
- Các phương thức ghi dữ liệu vào các biến thực thể của đối tượng, chúng nhận dữ liệu mới qua các tham số rồi ghi vào các biến liên quan. Chúng thường được đặt tên là `setDữLiệuGiĐó`, nên còn được gọi là các **phương thức set**.

Ví dụ như trong Hình 5.6

<pre>class Cow {     String name;     int age;      void setName(String aName) {         name = aName;     }     String getName() {         return name;     }     void setAge(int anAge) {         age = anAge;     }     int getAge() {         return age;     } }</pre>	<table border="1"> <tr> <th style="text-align: center;">COW</th> </tr> <tr> <td>n<sub>ame</sub> i<sub>age</sub></td> </tr> <tr> <td>getN<sub>ame</sub>() setN<sub>ame</sub>() getA<sub>ge</sub>() setA<sub>ge</sub>()</td> </tr> </table>	COW	n <sub>ame</sub> i <sub>age</sub>	getN <sub>ame</sub> () setN <sub>ame</sub> () getA <sub>ge</sub> () setA <sub>ge</sub> ()
COW				
n <sub>ame</sub> i <sub>age</sub>				
getN <sub>ame</sub> () setN <sub>ame</sub> () getA <sub>ge</sub> () setA <sub>ge</sub> ()				

Hình 5.6: Lớp Cow với các hàm đọc/ghi

Cho đến nay, ta đã lò đi một trong những nguyên tắc quan trọng nhất của lập trình hướng đối tượng, đó là đóng gói và che giấu thông tin. Nguyên tắc này nói rằng "**Đừng để lộ cấu trúc dữ liệu bên trong**". Trong tất cả các ví dụ từ đầu cuốn sách đến giờ, ta đã để lộ tất cả dữ liệu. 'Để lộ' ở đây có nghĩa là từ bên ngoài lớp có thể dùng một tham chiếu tới đối tượng kèm theo toán tử dấu chấm (.) để truy nhập biến thực thể của đối tượng đó. Ví dụ:

```
theCow.age = 2;
```

Nói cách khác là ta đang cho phép dùng tham chiếu để trực tiếp sửa biến thực thể của đối tượng. Đây là công cụ nguy hiểm nếu đặt trong tay những ai muốn phá hoại hoặc không biết dùng đúng cách. Nó cho phép người ta làm những việc chẳng hạn như cho một đối tượng Cow có tuổi là số âm:

```
theCow.age = -2;
```

Để ngăn chặn nguy cơ này, ta cần cài các phương thức set cho các biến thực thể và tìm cách buộc các đoạn mã khác phải gọi các phương thức set thay vì truy nhập trực tiếp đến dữ liệu. Khi đã đảm bảo được rằng gọi một phương thức set là cách duy nhất để sửa một biến thực thể, ta có thể kiểm tra tính hợp lệ của dữ liệu mới và bảo vệ không cho phép bất cứ ai gán một giá trị không hợp lệ cho biến thực thể đó. Ví dụ, trong lớp Cow, phương thức setAge() có thể bảo vệ tính hợp lệ của biến thực thể age như sau:

```
void setAge(int a) {  
    if (a >= 0) {  
        age = a;  
    }  
}
```

Nửa công việc còn lại, cần làm gì để che giấu dữ liệu, không cho phép các đoạn mã khác dùng tham chiếu trực tiếp sửa biến thực thể? Làm cách nào để *che giấu* dữ liệu? Quy tắc khởi đầu cho việc thực hiện đóng gói là: đánh dấu các biến thực thể với từ khóa **private** và cung cấp các phương thức **public** set và get cho biến đó. Các từ khóa private và public quy định quyền truy nhập của biến thực thể, phương thức, hay lớp được khai báo với từ khóa đó. (Ta đã quen với từ khóa public, nó đi kèm khai báo của tất cả các phương thức main.) Từ khóa private có nghĩa là riêng tư, cá nhân. Trong một lớp, biến thực thể / phương thức nào được khai báo với từ khóa private thì chỉ có mã chương trình ở bên trong lớp đó mới có quyền truy nhập biến / phương thức đó. Từ nay ta sẽ gọi các biến / phương thức được khai báo với từ khóa private là biến private / phương thức private. Còn public có nghĩa là mã ở bất cứ đâu đều có thể truy nhập biến / phương thức đó.

Minh họa ở lớp ProtectedCow trong Hình 5.7. Tại đó, biến thực thể age được khai báo là biến private, còn hai phương thức get và set tương ứng, setAge() và getAge(), được khai báo là phương thức public. Khi ta thành thạo hơn trong việc thiết kế và cài đặt bằng Java, ta có thể sẽ làm hơi khác, nhưng tại thời điểm này, quy tắc đơn giản "*biến thực thể private, get và set public*" là lựa chọn an toàn.

Đại học Công nghệ - HHOUGHN

```

class SecuredCow {
    private int age;

    public void setAge(int a) {
        if (a>0) {
            age = a;
        }
    }
    public int getAge() {
        return age;
    }
    void moo() {
        if (age <= 5) {
            System.out.println("Mooooooooo...");
        } else {
            System.out.println("Moo.");
        }
    }
}

```

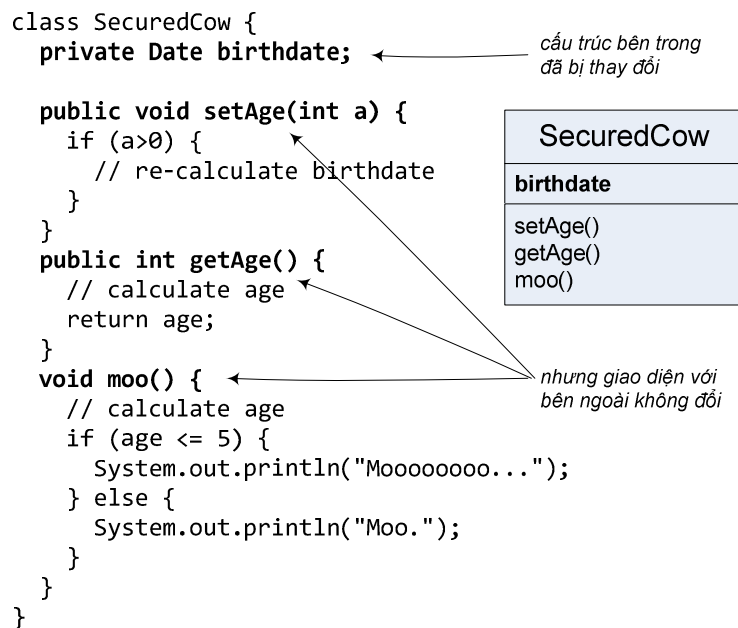
SecuredCow
age
setAge() getAge() moo()

Hình 5.7: Lớp SecuredCow và nguyên tắc đóng gói

Ngoài việc bảo vệ dữ liệu, đóng gói và che giấu dữ liệu còn mang lại một lợi ích khác. Đó là khả năng thay đổi cấu trúc bên trong của một lớp mà không làm ảnh hưởng đến những phần mã bên ngoài có sử dụng đến lớp đó.

Tại ví dụ trong Hình 5.8, cấu trúc bên trong của lớp SecuredCow đã bị sửa đổi. Tuổi của bò không được đại diện bởi biến thực thể age như trước mà thay vào đó là biến birthdate lưu ngày sinh của con bò. Tuổi của bò có thể được tính từ ngày sinh và ngày tháng năm hiện tại. Nội dung các phương thức dùng đến giá trị tuổi bò cũng thay đổi một cách tương xứng. Trong khi đó, giao diện của lớp SecuredCow với bên ngoài không thay đổi. Cụ thể là các phương thức public vẫn giữ nguyên tên, kiểu trả về, và danh sách tham số. Điều đó có nghĩa rằng các đoạn mã dùng đến SecuredCow từ bên ngoài sẽ không bị thay đổi.

Đại học Công nghệ - HHOUGHN



Hình 5.8: Lớp SecuredCow với cấu trúc bên trong đã được sửa.

Chương trình ClientProgram dưới đây đã chạy được với phiên bản trước của SecuredCow và cũng chạy được với phiên bản mới mà không cần sửa đổi. Bất kì chương trình nào khác dùng đến SecuredCow cũng đều tiếp tục hoạt động như không có thay đổi gì đã xảy ra.

```

class ClientProgram {
    public static void main(String [] args) {
        SecuredCow cow = new SecuredCow();
        cow.setAge(2);
        cow.moo();
        System.out.println(cow.getAge());
    }
}

```

Tình huống tương tự không xảy ra đối với lớp Cow khi ta muốn đổi age thành birthdate hay một thay đổi tương tự. Các đoạn mã trực tiếp truy nhập biến age từ bên ngoài sẽ không thể chạy được sau sửa đổi.

Khả năng thay đổi cấu trúc bên trong của một lớp mà không làm ảnh hưởng đến những phần mã bên ngoài có sử dụng đến lớp đó cho phép ta giảm mạnh số lỗi phát sinh do sửa chương trình. Điều đó rất có giá trị cho việc phát triển chương trình một cách hiệu quả.

Việc che giấu chi tiết bên trong của một mô-đun nếu được thực hiện càng tốt thì càng làm giảm sự phụ thuộc lẫn nhau giữa mô-đun này và phần còn lại của hệ thống. Mô-đun đó không phải phụ thuộc vào việc nó phải được bên ngoài sử dụng đúng cách, vì nó có thể tự đảm bảo là nó không thể bị dùng sai cách. Ví dụ, từ bên ngoài lớp Cow chỉ có thể sửa tuổi bò thông qua setAge(), trong khi setAge() đảm bảo bò không thể có tuổi là số âm. Ngược lại, phần còn lại của hệ thống không phải biết

quá nhiều về mô-đun đó để có thể sử dụng nó đúng cách. Ví dụ, chỉ cần gọi `setAge()` chứ không trực tiếp gán giá trị cho biến thực thể của Cow nên không cần biết Cow dùng cách gì để lưu trữ tuổi bò, quy tắc cho giá trị đó như thế nào. Sự ít phụ thuộc lẫn nhau giữa các mô-đun chương trình là một trong những đặc điểm của thiết kế có chất lượng tốt.

## 5.5. KHAI BÁO VÀ KHỞI TẠO BIẾN THỰC THỂ

Ta đã biết rằng một lệnh khai báo biến thực thể có ít nhất hai phần: tên biến và kiểu dữ liệu. Ví dụ:

```
int age;  
String name;
```

Ta còn có thể khởi tạo (gán một giá trị đầu tiên) cho biến ngay tại lệnh khởi tạo:

```
int age = 2;  
String name = "Fido";
```

Nhưng nếu ta không khởi tạo một biến thực thể, chuyện gì sẽ xảy ra khi ta gọi một phương thức `get`? Nói cách khác, một biến thực thể có giá trị gì *trước* khi nó được khởi tạo? Xem lại ví dụ trong Hình 5.6, `age` và `name` được khai báo nhưng không được khởi tạo, vậy `getAge()` và `getName()` sẽ trả về giá trị gì?

Các biến thực thể luôn có một giá trị mặc định. Nếu ta không gán giá trị cho một biến thực thể, hoặc không gọi một phương thức `set` để gán trị cho nó, nó vẫn có một giá trị mặc định: 0 nếu biến thuộc kiểu số nguyên, 0.0 nếu biến thuộc kiểu số thực dấu chấm động, `false` nếu biến thuộc kiểu boolean, `null` nếu biến là tham chiếu.

```

class Cow {
    private String name;
    private int age;

    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}

public class CowTestDrive {
    public static void main (String[] args) {
        Cow c = new Cow();
        System.out.println("Cow age: " + c.getAge());
        System.out.println("Cow name: " + c.getName());
    }
}

```

```

% java CowTestDrive
Cow age: 0
Cow name: null

```

Hình 5.9: Giá trị mặc định của biến thực thể

Ví dụ trong Hình 5.9 minh họa giá trị mặc định của hai biến thực thể `name` và `age` của lớp `Cow`. Hai biến này không được khởi tạo, và giá trị mặc định của biến `age` kiểu `int` là 0, còn giá trị mặc định của `name` kiểu tham chiếu là `null`. Nhớ rằng `null` có nghĩa là một tham chiếu không chiếu tới một đối tượng nào, hay một cái điều khiển từ xa không điều khiển cái gì cả. Ví dụ trong Hình 4.9 ở chương trước cũng đã minh họa việc đọc biến tham chiếu `name` của đối tượng `Cow` trước khi nó được khởi tạo.

## 5.6. BIẾN THỰC THỂ VÀ BIẾN ĐỊA PHƯƠNG

Ta đã gặp cả biến thực thể và biến địa phương trong các ví dụ trước. Mục này tổng kết lại các đặc điểm phân biệt giữa hai loại biến này.

- Biến thực thể được khai báo bên trong một lớp nhưng không nằm trong một phương thức nào. Ví dụ `a` và `b` trong Hình 5.10 là biến thực thể của lớp `Foo`.
- Biến địa phương được khai báo bên trong một phương thức. Ví dụ `sum` và `dummy` trong Hình 5.10.
- Biến địa phương phải được khởi tạo trước khi sử dụng. Ví dụ `dummy` chưa được khởi tạo nhưng đã được dùng trong lệnh `sum = a + dummy`; sẽ gây lỗi khi biên dịch.



---

```

class Foo {
    int a = 1;
    int b;

    public int add() {
        int sum = a + b;
        return sum;
    }
    public int addThatWontCompile() {
        int dummy;
        int sum = a + dummy;
        return sum;
    }
}

```

*a là biến thực thể chưa được khởi tạo nhưng đã có giá trị mặc định*

*lỗi biên dịch do dùng biến địa phương dummy chưa được khởi tạo*

---

Hình 5.10: Biến thực thể và biến địa phương.

Như đã nói, tham số của một phương thức cũng là biến địa phương của phương thức đó. Nó đã được khởi tạo bằng giá trị của đối số được truyền vào phương thức.

Đó là các đặc điểm mang tính chất cú pháp và đặc thù ngôn ngữ. Còn về bản chất khái niệm, hai loại biến này khác hẳn nhau theo nghĩa sau:

- Biến địa phương thuộc về một phương thức – nơi khai báo nó. Nó được sinh ra khi phương thức được gọi và dòng lệnh khai báo nó được thực thi. Nó hết hiệu lực khi ra ngoài phạm vi – kết thúc khối lệnh khai báo nó hoặc khi phương thức kết thúc.
- Biến thực thể thuộc về một thực thể – đối tượng chủ của nó. Nó được tạo ra khi đối tượng được tạo ra và hết hiệu lực khi đối tượng đó bị hủy.

Đại học Công nghệ - HHOUGHN

## Bài tập

1. Điền vào mỗi chỗ trống một hoặc vài từ trong các từ sau: biến thực thể, đối số, giá trị trả về, phương thức get, phương thức set, đóng gói, public, private, truyền bằng giá trị, phương thức.

Một lớp có thể có số lượng tùy ý các \_\_\_\_\_.

Một phương thức chỉ có thể có một \_\_\_\_\_.

\_\_\_\_\_ có thể được ngầm đòi kiểu dữ liệu.

\_\_\_\_\_ có nghĩa là "tôi muốn biến thực thể của tôi ở dạng private".

\_\_\_\_\_ thực chất có nghĩa là "tạo một bản sao".

\_\_\_\_\_ chỉ nên được cập nhật bởi các phương thức setter.

Một phương thức có thể có nhiều \_\_\_\_\_.

\_\_\_\_\_ trả về giá trị gì đó.

\_\_\_\_\_ không nên được dùng cho các biến thực thể.

\_\_\_\_\_ có thể có nhiều đối số.

\_\_\_\_\_ giúp thực hiện nguyên tắc đóng gói.

\_\_\_\_\_ lúc nào cũng chỉ có một.

2. Điền từ thích hợp vào chỗ trống trong mỗi câu sau:

a) Mỗi tham số phải được chỉ rõ một \_\_\_\_\_ và một \_\_\_\_\_

b) Từ khóa \_\_\_\_\_ đặt tại khai báo kiểu trả về quy định rằng một phương thức sẽ không trả về giá trị gì sau khi nó hoàn thành nhiệm vụ.

3. Các phát biểu sau đây đúng hay sai? Nếu sai, hãy giải thích.

a) Cặp ngoặc rỗng() đứng sau tên phương thức tại một khai báo phương thức cho biết phương thức đó không yêu cầu tham số nào.

b) Các biến thực thể hoặc phương thức được khai báo với từ khóa private chỉ được truy cập từ các phương thức nằm trong lớp nơi chúng được khai báo.

c) Thân phương thức được giới hạn trong một cặp ngoặc {}.

d) Có thể gọi phương thức từ một biến kiểu cơ bản.

e) Các biến địa phương kiểu cơ bản về mặc định là được khởi tạo sẵn.

f) Số các đối số chứa trong lời gọi phương thức phải khớp với số tham số trong danh sách tham số của khai báo phương thức đó.

4. Phân biệt giữa biến thực thể và biến địa phương.

5. Giải thích mục đích của tham số phương thức. Phân biệt giữa tham số và đối số.

6. Tại sao một lớp có thể cần cung cấp phương thức set và phương thức get cho một biến thực thể?
7. Viết class Employee chứa ba mẫu thông tin dưới dạng các thành viên dữ liệu: tên (first name, kiểu String), họ (last name, kiểu String) và lương tháng (salary, kiểu double). Class Employee cần có một hàm khởi tạo có nhiệm vụ khởi tạo ba thành viên dữ liệu này. Hãy viết một hàm set và một hàm get cho mỗi thành viên dữ liệu. Nếu lương tháng có giá trị âm thì hãy gán cho nó giá trị 0.0. Viết một chương trình thử nghiệm EmployeeTest để chạy thử các tính năng của class Employee. Tạo hai đối tượng Employee và in ra màn hình tổng lương hàng năm của mỗi người. Sau đó cho tăng lương cho mỗi người thêm 10% và hiển thị lại lương của họ theo năm.
8. Tạo một lớp có tên Invoice (hóa đơn) mà một cửa hàng có thể dùng để biểu diễn một hóa đơn cho một món hàng được bán ra tại cửa hàng. Mỗi đối tượng Invoice cần có 4 thông tin chứa trong các thành viên dữ liệu: số hiệu của mặt hàng (partNumber kiểu String), miêu tả mặt hàng (partDescription kiểu String), số lượng bán ra (quantity kiểu int) và đơn giá (unitPrice kiểu double). Lớp Invoice cần có một hàm khởi tạo có nhiệm vụ khởi tạo 4 thành viên dữ liệu đó. Hãy viết một phương thức set và một phương thức get cho mỗi thành viên dữ liệu. Ngoài ra, hãy viết một phương thức có tên getInvoiceAmount với nhiệm vụ tính tiền hóa đơn (nghĩa là số lượng nhân với đơn giá), rồi trả về giá trị hóa đơn dưới dạng một giá trị kiểu double. Nếu số lượng không phải số dương thì cần gán cho nó giá trị 0. Nếu đơn giá có giá trị âm, nó cũng cần được gán giá trị 0.0. Viết một ứng dụng thử nghiệm tên là InvoiceTest để chạy thử các tính năng của class Invoice.
9. Tìm và sửa lỗi của các chương trình sau (mỗi phần là một file mã nguồn hoàn chỉnh).

a)

```
class XCopy {
    public static void main(String [] args) {
        int orig = 42;
        Xcopy x = new xCopy();
        int y = x.go(orig);
        System.out.println(orig + " " + y);
    }
    int go(int arg) {
        arg = arg * 2;
        return arg;
    }
}
```

Đại học Công nghệ - HHOUGHN

b)

```
class Clock {
    String time;
    void setTime(String t) {
        time = t;
    }
    void getTime() {
        return time;
    }
}
class ClockTestDrive {
    public static void main(String [] args) {
        Clock c = new Clock();
        c.setTime("1245");
        String tod = c.getTime();
        System.out.println("time: " + tod);
    }
}
```

Đại học Công nghệ - ĐHQGHN

## Chương 6. SỬ DỤNG THƯ VIỆN JAVA

Khả năng hỗ trợ tái sử dụng của lập trình hướng đối tượng thể hiện ở thư viện đồ sộ của Java bao gồm hàng trăm lớp được xây dựng sẵn. Đó là các khối cơ bản để cho ta lắp ghép thành chương trình lớn. Chương này giới thiệu về các khối cơ bản đó.

### 6.1. ArrayList

Đầu tiên, ta lấy một ví dụ về một lớp trong thư viện: ArrayList. Ta đã biết về cấu trúc mảng của Java. Cũng như mảng của nhiều ngôn ngữ khác, mảng của Java có những hạn chế chẳng hạn như ta phải biết kích thước khi tạo mảng; việc xóa một phần tử ở giữa mảng không đơn giản; mảng không thể lưu nhiều phần tử hơn kích thước đã khai báo. Lớp ArrayList là một cấu trúc dạng mảng khắc phục được các nhược điểm của cấu trúc mảng. Ta không cần biết một ArrayList cần có kích thước bao nhiêu khi tạo nó, nó sẽ tự giãn ra hoặc co vào khi các đối tượng được đưa vào hoặc lấy ra. Thêm vào đó, ArrayList còn là cấu trúc có tham số kiểu, ta có thể tạo ArrayList<String> để lưu các phần tử kiểu String, ArrayList<Cow> để lưu các phần tử kiểu Cow, v.v..

ArrayList cho ta các tiện ích sau:

#### **add(Object item)**

gắn đối tượng vào cuối danh sách

#### **add(int i, Object item)**

chèn đối tượng vào vị trí i trong danh sách

#### **get(int i)**

trả về đối tượng tại vị trí i trong danh sách

#### **remove(int index)**

xóa đối tượng tại vị trí có chỉ số index

#### **remove(Object item)**

xóa đối tượng nếu nó nằm trong danh sách

#### **contains(Object item)**

trả về true nếu danh sách chứa đối tượng item

#### **isEmpty()**

trả về true nếu danh sách rỗng

Đại học Công nghệ - HHOUGHN

## size()

trả về số phần tử hiện đang có trong danh sách

## get(int index)

trả về đối tượng hiện đang nằm tại vị trí index

Ví dụ sử dụng ArrayList được cho trong Hình 6.1. Trong đó, lệnh khởi tạo new ArrayList<String> tạo một đối tượng danh sách dành cho kiểu String, tạm thời danh sách rỗng. Lần gọi add thứ nhất làm kích thước danh sách tăng từ 0 lên 1. Lần thứ hai add xâu "Goodbye" vào vị trí số 1 trong danh sách và làm cho kích thước danh sách tăng lên 2. Sau khi remove(a), kích thước danh sách lại giảm về 1. Bản chất một đối tượng ArrayList lưu trữ một danh sách các tham chiếu tới các đối tượng thuộc kiểu được khai báo. Như trong ví dụ này, ở thời điểm sau khi gọi add(0,b), đối tượng ArrayList của ta chứa một danh sách gồm hai tham chiếu kiểu String, một chiếu tới đối tượng String "Goodbye" mà b đang chiếu tới, tham chiếu còn lại chiếu tới đối tượng String "Hello".

```
import java.util.ArrayList;

public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        String a = new String("Hello");
        list.add(a);

        String b = new String("Goodbye");
        list.add(0,b);
        System.out.println("List size is " + list.size());

        if (list.contains("Hello"))
            System.out.println("Hello is in");

        System.out.println("Index of "+ a +" is "+ list.indexOf(a));

        System.out.println("List is empty? " + list.isEmpty());

        list.remove(a);
    }
}
```

```
% java ArrayListTest
List size is 2
Hello is in
Index of Hello is 1
List is empty? false
```

Hình 6.1: Ví dụ sử dụng ArrayList.

Cú pháp <String> tại dòng khai báo ArrayList sẽ được giải thích chi tiết tại Chương 13. Tạm thời, ta tạm chấp nhận ArrayList<String> là kiểu danh sách của các đối tượng String, ArrayList<Cow> là kiểu danh sách của các đối tượng Cow.

## 6.2. SỬ DỤNG JAVA API

Trong Java API, các lớp được nhóm thành các **gói** (*package*). Để dùng một lớp trong thư viện, ta phải biết nó nằm trong gói nào. Mỗi gói đã được đặt một cái tên, chẳng hạn java.util. Scanner nằm trong gói java.util này. Nó chứa rất nhiều lớp tiện ích. Ta cũng đã dùng đến lớp System (System.out.println), String, và Math là các lớp nằm trong gói java.lang.

Chi tiết về gói, trong đó có cách đặt các lớp của chính mình vào gói của riêng mình, được trình bày trong Phụ lục B. Trong chương này, ta chỉ giới thiệu qua về việc sử dụng một số lớp trong thư viện Java.

Ta sẽ lấy ví dụ về ArrayList trong mục trước để minh họa cho các nội dung trong mục này.

Đầu tiên, ta cần biết tên đầy đủ của lớp mà ta muốn sử dụng trong chương trình. Tên đầy đủ của ArrayList không phải ArrayList mà là java.util.ArrayList. Trong đó java.util là tên gói, còn ArrayList là tên lớp.

Ta phải cho máy ảo Java biết ta định dùng ArrayList nào. Ta có hai lựa chọn:

1. Dùng lệnh import ở đầu file mã nguồn. Ví dụ dòng đầu tiên trong file chương trình ArrayListTest trong mục trước là:

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

2. Gọi thẳng tên đầy đủ của lớp đó mỗi khi gọi đến tên nó. Ví dụ:

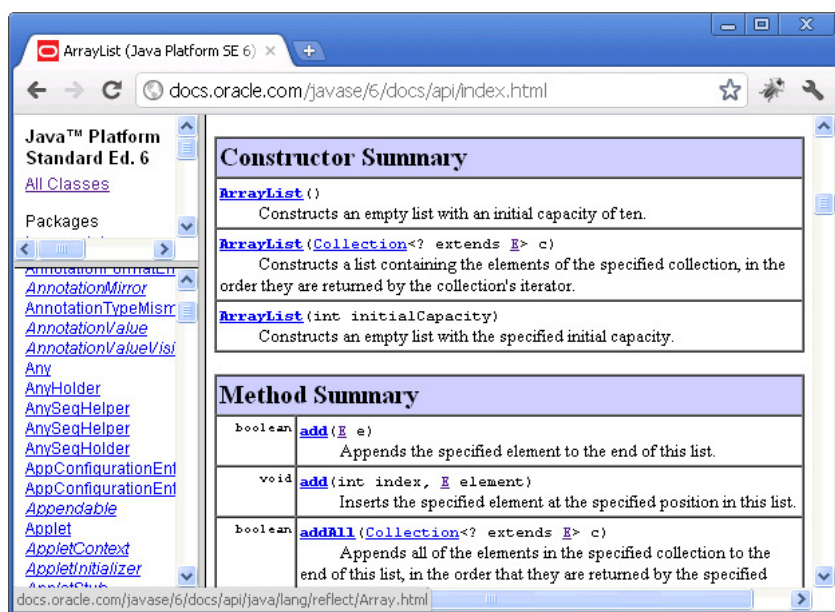
```
java.util.ArrayList<Cow> =  
    new java.util.ArrayList<Cow>
```

Gói java.lang thuộc dạng đã được nạp sẵn. Do đó ta đã không phải import java.lang hay dùng tên đầy đủ để có thể sử dụng các lớp String và System.

Có ba lí do cho việc tổ chức các lớp vào các gói: Thứ nhất, gói giúp ích cho việc tổ chức project hay thư viện. Thay cho một lô các lớp đặt cùng một chỗ, các lớp được đặt vào các gói khác nhau tùy theo chức năng, chẳng hạn GUI, cấu trúc dữ liệu, hay cơ sở dữ liệu. Thứ hai, cấu trúc gói cho ta một không gian tên, giúp tránh trùng tên. Nếu một loạt lập trình viên tạo các lớp có tên giống nhau nhưng đặt tại các gói khác nhau thì máy ảo Java vẫn có thể được các lớp đó. Thứ ba, tổ chức gói cho ta một mức bảo mật (mức gói), ta có thể hạn chế mã ta viết trong một gói để chỉ có các lớp nằm trong gói đó mới có thể truy nhập. Ta sẽ nói kĩ hơn về vấn đề này sau.

Sử dụng API bằng cách nào?

Ta cần biết hai điều: (1) trong thư viện có những lớp nào, (2) khi đã tìm thấy một lớp, làm thế nào để biết nó có thể làm được gì. Để trả lời cho hai câu hỏi đó, ta có thể tra cứu một cuốn sách về Java hoặc tài liệu API.



Hình 6.2: Tài liệu API phiên bản Java 6, trang về ArrayList.

Tài liệu API là nguồn tài liệu tốt nhất để tìm chi tiết về từng lớp và các phương thức của nó. Tại đó, ta có thể tìm và duyệt theo gói, tìm và tra cứu theo tên lớp. Với mỗi lớp, ta có đầy đủ thông tin mô tả lớp, các lớp liên quan, danh sách các phương thức, và đặc tả chi tiết của từng phương thức.

## 6.3. MỘT SỐ LỚP THÔNG DỤNG TRONG API

### 6.3.1. Math

Math là lớp cung cấp các hàm toán học thông dụng.

- **Math.random()** : trả về một giá trị kiểu double trong khoảng [0.0,..,1.0).
- **Math.abs()** : trả về một giá trị double là giá trị tuyệt đối của đối số kiểu double, tương tự đối với đối số và giá trị trả về kiểu int.
- **Math.round()** : trả về một giá trị int hoặc long (tùy theo đối số là kiểu float hay double) là giá trị làm tròn của đối số tới giá trị nguyên gần nhất. Lưu ý rằng các hằng kiểu float được Java hiểu là thuộc kiểu double trừ khi thêm kí tự f vào cuối, ví dụ 1.2f.
- **Math.min()** : trả về giá trị nhỏ hơn trong hai đối số. Đối số có thể là int, long, float, hoặc double.
- **Math.max()**: trả về giá trị lớn hơn trong hai đối số. Đối số có thể là int, long, float, hoặc double.

Ngoài ra, Math còn các phương thức khác như sqrt(), tan(), ceil(), floor(), và sin(). Ta nên tra cứu chi tiết tại tài liệu API.



### 6.3.2. Các lớp bọc ngoài kiểu dữ liệu cơ bản

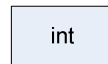
Đôi khi, ta muốn đối xử với một giá trị kiểu cơ bản như là một đối tượng. Ví dụ, ở các phiên bản Java trước 5.0, ta không thể chèn thẳng một giá trị kiểu cơ bản vào trong một cấu trúc kiểu ArrayList. Các lời gọi tương tự như list.add(2) sẽ bị trình biên dịch báo lỗi do phương thức add lấy đối số là tham chiếu đối tượng.

Trong những trường hợp như vậy, ta có các lớp bọc ngoài mỗi kiểu cơ bản (*wrapper class*). Các lớp bọc ngoài này có tên gần trùng với tên kiểu cơ bản tương ứng: Boolean, Character, Byte, Short, Integer, Long, Float, Double. Mỗi đối tượng thuộc các lớp trên bao bọc một giá trị kiểu cơ bản tương ứng, kèm theo các phương thức để thao tác với giá trị đó. Ví dụ:

---

#### Tạo đối tượng bọc ngoài một giá trị

```
int i = 10;  
Integer iWrap = new Integer(i);
```



đối tượng Integer

#### Lấy giá trị bên trong đối tượng

```
int unwrapped = iWrap.intValue();
```

#### Lấy giá trị int từ biểu diễn chuỗi kí tự

```
i = Integer.parseInt("10");
```

#### Lấy biểu diễn chuỗi kí tự của giá trị int

```
String iString = Integer.toString(i);  
hoặc  
String iString = iWrap.toString();
```

#### Các hằng giá trị

```
Integer.MAX_VALUE, Integer.MIN_VALUE
```

---

Hình 6.3: Sử dụng lớp Integer.

Các lớp bọc ngoài khác cũng có cách sử dụng và các phương thức tiện ích tương tự như Integer. Chẳng hạn mỗi đối tượng Boolean có phương thức booleanValue() trả về giá trị boolean chứa trong nó.

Tóm lại, nếu dùng phiên bản Java trước 5.0 hay từ 5.0 trở đi, ta sẽ sử dụng ArrayList cho các giá trị int theo kiểu như sau:

---

#### Trước Java 5.0

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(new Integer(10));  
Integer item = (Integer) list.get(0);  
int intItem = item.intValue();
```

#### Từ Java 5.0 trở đi

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(10);  
int intItem = list.get(0);
```

---

Với các phiên bản Java từ 5.0 trở đi, trình biên dịch tự động làm hộ ta các công việc bọc và gỡ các đối tượng bọc ngoài thuộc kiểu tương ứng. Nói cách khác,

ArrayList thực sự là danh sách của các đối tượng Integer, nhưng ta có thể coi như ArrayList lấy vào và trả về các giá trị int. Trình biên dịch không chỉ tự động bọc và gỡ bọc trong các tình huống sử dụng các cấu trúc dữ liệu tương tự ArrayList. Việc này còn xảy ra ở hầu hết các tình huống khác:

- **Đối số của phương thức:** dù một phương thức khai báo tham số kiểu cơ bản hay kiểu lớp bọc ngoài thì nó vẫn chấp nhận đối số ở cả dạng cơ bản cũng như kiểu lớp bọc ngoài.
- **Giá trị trả về:** dù một phương thức khai báo kiểu trả về kiểu cơ bản hay bọc ngoài thì lệnh return trong phương thức dùng giá trị ở cả dạng cơ bản cũng như bọc ngoài đều được.
- **Biểu thức boolean:** ở những vị trí yêu cầu một biểu thức boolean, ta có thể dùng biểu thức cho giá trị boolean (chẳng hạn  $2 < a$ ), hoặc một biến boolean, hoặc một tham chiếu kiểu Boolean đều được.
- **Phép toán số học:** ta có thể dùng tham chiếu kiểu bọc ngoài làm toán hạng của các phép toán số học, kể cả phép ++.
- **Phép gán:** ta có thể dùng một tham chiếu kiểu bọc ngoài để gán trị cho một biến kiểu cơ bản và ngược lại. Ví dụ: `Double d = 10.0;`

### 6.3.3. Các lớp biểu diễn chuỗi ký tự

String và StringBuffer là hai lớp thông dụng để biểu diễn dữ liệu dạng chuỗi ký tự. String dành cho các chuỗi ký tự không thể sửa đổi nội dung. Tất cả các hằng chuỗi ký tự như "abc" đều được Java coi như các thực thể của lớp String. StringBuffer và StringBuilder cho phép sửa đổi nội dung chuỗi, sử dụng một trong hai lớp này sẽ hiệu quả hơn String nếu ta cần dùng nhiều thao tác sửa chuỗi. Từ Java 5.0, ta nên dùng StringBuilder thay vì String Buffer cho mục đích này, trừ khi ta cần chú ý tránh xung đột giữa các thao tác xử lý chuỗi tại các luồng khác nhau.

String và StringBuffer/StringBuilder đều có các phương thức sau:

- **charAt (int index)** trả về ký tự tại một vị trí
- **compareTo()** so sánh giá trị với một đối tượng cùng loại.
- các phương thức **indexOf()** tìm vị trí của một ký tự/xâu con theo chiều từ trái sang phải.
- các phương thức **lastIndexOf()** tìm vị trí của một ký tự/xâu con theo chiều từ phải sang trái.
- **length()** trả về độ dài của chuỗi.
- **substring(int start, int end)** trả về đối tượng String là chuỗi con.

Để nối chuỗi, ta dùng **concat()** cho String và **append()** cho StringBuffer/StringBuilder.

Ngoài ra, String còn có thêm các tiện ích :

- **valueOf()** trả về biểu diễn kiểu String của một giá trị thuộc kiểu cơ bản,
- **split()** để tách chuỗi thành các từ con theo một cú pháp cho trước,
- **replace(char old, char new)** trả về một String mới là kết quả của việc thay thế hết các ký tự old bằng ký tự new
- **trim()** trả về một String mới là kết quả của việc xóa các ký tự trắng ở đầu và cuối String hiện tại.

StringBuffer và StringBuilder có các phương thức cung cấp các phương thức để chèn (**insert**), thay (**replace**), xóa một phần (**delete**), đảo chuỗi (**reverse**) tại đối tượng StringBuffer/StringBuilder hiện tại.

Ta đã biết những cách đơn giản để lấy biểu diễn bằng chuỗi ký tự cho các giá trị số:

```
int n = 302044;
String s1 = "" + n;
String s2 = Integer.toString(n);
```

Đôi khi, ta cần biểu diễn các giá trị số một cách cầu kì hơn, chẳng hạn 302,044, hay quy định số chữ số nằm sau dấu phẩy thập phân sẽ được in ra, biểu diễn dạng nhị phân, hệ cơ số 16... Phương thức `format()` của lớp String giúp chúng ta làm được việc này. Ví dụ:

```
String.format("n = %,d",1234567)
    ───────────────────────────▶ "n = 1,234,567"

String.format("n = %, .1f",12345.67)
    ───────────────────────────▶ "n = 12,345.7"

String.format(Locale.GERMAN, "n = %, .1f",12345.67)
    ───────────────────────────▶ "n = 12.345,7"
```

## 6.4. TRÒ CHƠI BẮN TÀU

Trong mục này, ta sẽ làm một chương trình ví dụ: trò chơi bắn tàu SinkAShip<sup>6</sup>. Đây sẽ là một ứng dụng hoàn chỉnh minh họa việc sử dụng Java API, và cũng là một ứng dụng đủ lớn để minh họa rõ hơn sự tương tác giữa các đối tượng trong chương trình.

Trò chơi bắn tàu được mô tả như sau: Máy tính có một số con tàu kích thước 1 x 3 trên một vùng là lưới vuông 7 x 7, cho phép người chơi bắn một lần một viên đạn, mỗi viên trúng ô nào sẽ làm cháy phần tàu nằm trong ô đó, nếu như ở đó có tàu. Người chơi không biết các con tàu đó ở đâu, nhưng có mục tiêu là bắn cháy hết tàu, nên phải đoán xem nên bắn vào đâu để tổn càn ít đạn càn tốt.

---

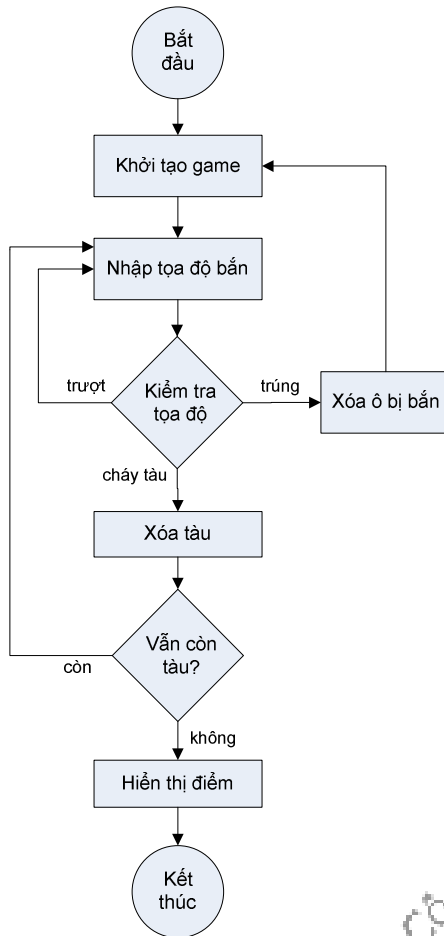
<sup>6</sup> Chính sửa từ ví dụ DotComBust của cuốn *Head First Java*, 2<sup>nd</sup> Edition.

Khi bắt đầu một ván chơi, chương trình sẽ đặt ngẫu nhiên ba con tàu vào một lưới ảo kích thước 7x7, sau đó mời người chơi bắn phát đầu tiên.

Ta chưa học lập trình giao diện đồ họa, do đó chương trình của chúng ta sẽ sử dụng giao diện dòng lệnh. Mỗi lần, chương trình sẽ mời người chơi nhập tọa độ một phát bắn, người chơi nhập một tọa độ có dạng "A5" hay "B1". Chương trình xử lý phát bắn, kiểm tra xem có trúng hay không rồi in ra màn hình một thông báo thuộc một trong các loại: "hit" (trúng), "miss" (trượt), hoặc "You sunk a ship" (khi một tàu vừa bị bắn cháy hết). Khi cả ba con tàu đều bị cháy hết, ván chơi kết thúc, chương trình thông báo điểm của người chơi.

Tọa độ trong trò chơi có dạng "A4", trong đó kí tự thứ nhất là một chữ cái trong đoạn từ A đến G đại diện cho tọa độ dòng, kí tự thứ hai là một chữ số trong đoạn từ 0 đến 6 đại diện cho tọa độ cột trong lưới vuông 7x7.

Thiết kế mức cao cho hoạt động của chương trình:



Bước tiếp theo là xác định ta cần đến các đối tượng nào. Ít nhất, ta sẽ cần đến ván chơi và các mô hình tàu, tương ứng với hai lớp SinkAShip và Ship. Khi viết một lớp, quy trình chung được gợi ý như sau:

- Xác định các nhiệm vụ và hoạt động của lớp
- Liệt kê các biến thực thể và phương thức
- Viết mã giả cho các phương thức để mô tả thuật toán/quy trình công việc của chúng.
- Viết chương trình test cho các phương thức.
- Cài đặt lớp
- Test các phương thức
- Tìm lỗi và cài lại nếu cần
- Test với người dùng thực.

Ta sẽ bỏ qua bước cuối cùng.

Đầu tiên là lớp Ship, ta cần lưu hai thông tin chính: tọa độ các ô của tàu và tàu đã bị bắn cháy hết hay chưa. Dưới đây là thiết kế mà ta dễ dàng nghĩ đến.

**locationCells:** Mảng lưu vị trí của các ô tọa độ của tàu

Ship
String [] locationCells int numOfHits String name
String check(String guess) void setLocationCells(String[] loc)

**numOfHits:** số phát đạn mà tàu đã bị bắn trúng

**name:** tên của tàu để khi cần có thể in ra màn hình

**METHOD: String check(String shot)**

LOOP: với mỗi ô trong mảng locationCells

IF shot khớp với ô đó:

tăng biến đếm numOfHits

IF số hit là 3 thì RETURN "kill"

ELSE RETURN "hit"

END LOOP

shot không khớp ô nào, RETURN "miss"

**METHOD: void setLocationCells(String[] loc)**

gán từng ô trong mảng loc

cho các ô trong locationCells[]

Nhưng thiết kế trên chưa tính đến trường hợp người chơi bắn hai phát vào cùng một ô, chưa phân biệt một phát đạn bắn vào ô chưa bị cháy với một phát đạn bắn vào ô đã cháy. Nếu người chơi bắn ba lần vào cùng một ô thì thuật toán trên sẽ cho là tàu đã bị bắn cháy, mặc dù thực tế vẫn còn hai ô chưa bị bắn. Ta có thể giải quyết vấn đề này bằng một mảng phụ chứa các giá trị boolean để đánh dấu các ô đã bị bắn, hoặc dùng giá trị int sẵn có tại mảng locationCells để mã hóa các trạng thái chưa bị bắn / đã bị bắn. Tuy nhiên, để có giải pháp vừa gọn gàng, vừa tận dụng thư viện Java, ta chọn cách dùng ArrayList để lưu danh sách các ô chưa bị bắn của con tàu. Mỗi khi ô nào bị bắn trúng, phần tử tương ứng sẽ bị xóa khỏi danh sách. Khi danh sách rỗng là khi tàu đã bị bắn cháy. Như vậy ta chỉ cần một đối tượng ArrayList là đủ dùng thay cho cả mảng int locationCells và biến đếm numOfHits. Ta có thiết kế như sau:

**locationCells:** danh sách các ô chưa bị bắn của tàu

Ship
ArrayList<String> locationCells String name
String check(String guess) void setLocationCells (ArrayList<String> loc)

**METHOD: String check(String shot)**

Tìm phần tử của locationCells khớp với shot  
IF có ô đó:  
xóa nó ra khỏi locationCells  
IF locationCells rỗng, RETURN "kill"  
ELSE RETURN "hit"  
ELSE, shot không khớp ô nào, RETURN "miss"

**METHOD: void setLocationCells  
(ArrayList<String> loc)**

Gán loc cho locationCells

Cài đặt lớp Ship theo thiết kế trên:

```
import java.util.ArrayList;

public class Ship {
    private ArrayList<String> locationCells;
    private String name;

    public void setName(String string) {
        name = string;
    }

    public void setLocationCells(
        ArrayList<String> loc) {
        locationCells = loc;
    }

    public String check(String userInput) {
        String result = "miss";
        int index = locationCells.indexOf(userInput);
        if (index >= 0) {
            locationCells.remove(index);
            if (locationCells.isEmpty())
                result = "kill";
            else result = "hit";
        }
        return result;
    }
}
```

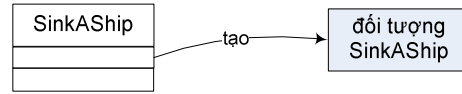
Lớp SinkAShip có các nhiệm vụ sau:

- tạo ra ba con tàu,
- cho mỗi con tàu một cái tên,
- đặt ba con tàu vào lưới. Ở đây ta cần tính vị trí tàu một cách ngẫu nhiên, ta tạo một lớp GameHelper để cung cấp tiện ích này (sẽ nói đến Helper sau).
- hỏi tọa độ bắn của người chơi, kiểm tra với cả ba con tàu rồi in kết quả. Lặp cho đến khi nào cả ba con tàu đều đã bị cháy.

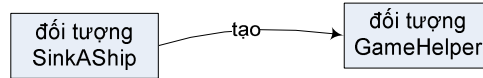
Như vậy, ta cần ba lớp: SinkAShip vận hành trò chơi, Ship đại diện cho tàu, và GameHelper cung cấp cho Sink các tiện ích trợ giúp như nhận input từ người chơi và sinh vị trí cho các con tàu. Ta cần một đối tượng SinkAShip, ba đối tượng Ship, và một đối tượng GameHelper. Ngoài ra còn có các đối tượng ArrayList chứa trong ba đối tượng Ship.

Vậy ai làm gì trong một ván SinkAShip? Các đối tượng trong chương trình bắn tàu hoạt động và tương tác với nhau theo từng giai đoạn như sau:

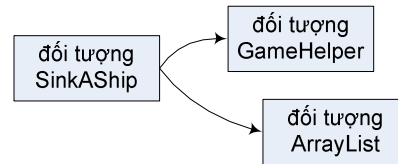
1. Phương thức main() của lớp SinkAShip tạo một đối tượng SinkAShip, đối tượng này sẽ vận hành trò chơi.



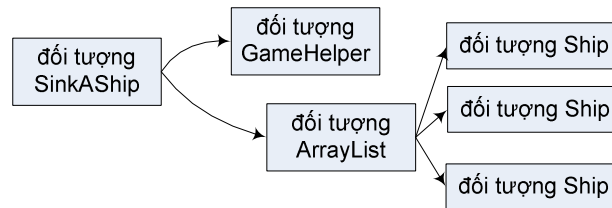
2. Đối tượng SinkAShip tạo một đối tượng GameHelper để nó làm 'trợ lí'.



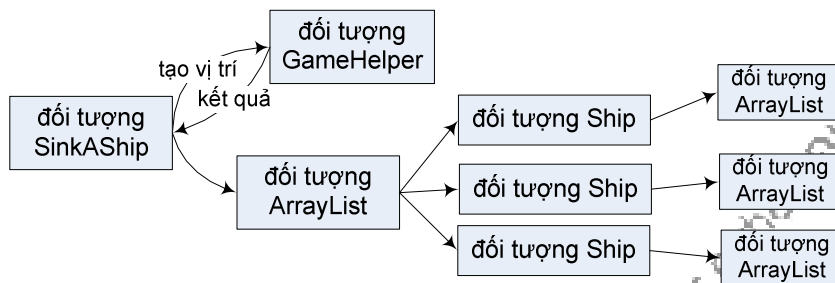
3. Đối tượng SinkAShip tạo một ArrayList để chuẩn bị lưu trữ ba đối tượng Ship.



4. Đối tượng SinkAShip tạo ba đối tượng Ship và gắn vào ArrayList nói trên.

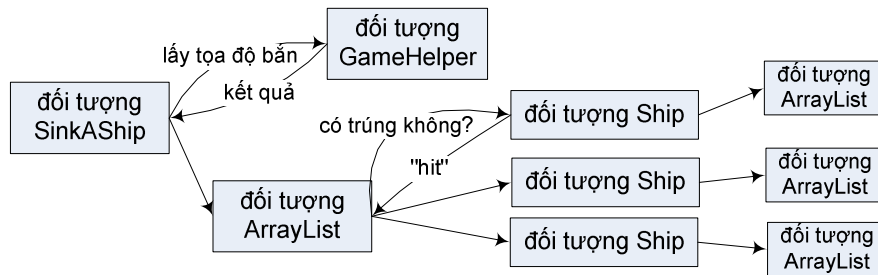


5. Đối tượng SinkAShip yêu cầu 'trợ lí' sinh tọa độ cho từng đối tượng Ship, chuyển dữ liệu tọa độ nhận được cho các đối tượng Ship. Các đối tượng Ship cập nhật danh sách tọa độ tại ArrayList của mình.



6. Đối tượng SinkAShip yêu cầu 'trợ lí' lấy tọa độ bắn của người chơi, 'trợ lí' hiển thị lời mời nhập tại giao diện dòng lệnh và nhận input của người chơi). Nhận được kết quả do 'trợ lí' cung cấp, đối tượng SinkAShip yêu cầu từng đối tượng Ship tự kiểm tra xem có bị bắn trúng hay không. Mỗi đối tượng Ship kiểm tra từng vị trí trong ArrayList của mình và trả về kết quả tương ứng

("miss", "hit", ...). Bước này lặp đi lặp lại cho đến khi tất cả các con tàu đều bị bắn cháy.



Như đã nói ở Chương 1, chương trình hướng đối tượng là một nhóm các đối tượng tương tác với nhau. Các ví dụ trước trong cuốn sách này đều nhỏ nên khó thấy rõ sự tương tác giữa các đối tượng. Ví dụ trò chơi bắn tàu này đủ lớn để minh họa được khía cạnh đó.

Với hoạt động như đã mô tả, lớp SinkAShip được thiết kế như sau:

**helper:** đối tượng GameHelper để trợ giúp

**numOfShots:** số phát đạn mà người chơi đã bắn

**shipList:** danh sách tàu

**METHOD: void setUpGame()**

```

// tạo ba con tàu và đặt tên cho chúng
Tạo 3 đối tượng Ship, đặt tên, và gắn chúng vào shipList
LOOP với mỗi tàu trong shipList
  gọi placeShip() của helper để lấy một tọa độ tàu
  gắn tọa độ cho con tàu đó
END LOOP
  
```

**METHOD: void startPlaying()**

```

LOOP trong khi vẫn còn tàu trong danh sách shipList
  gọi getUserInput() của helper để lấy tọa độ bắn của người dùng
  kiểm tra kết quả bắn bằng phương thức checkUserShot()
END LOOP
  
```

**METHOD: void checkUserShot(string userShot)**

```

//kiểm tra xem phát đạn có trúng tàu nào không.
Tăng số đếm số phát đạn lưu tại biến numOfShots
Gán giá trị "miss" cho biến địa phương result
LOOP với mỗi tàu trong danh sách shipList
  Gọi check() của tàu đó để xem nó có bị bắn trúng hay không
  Gán "hit" hoặc "kill" cho biến result nếu thích hợp
  Nếu kết quả là "kill" thì xóa tàu hiện hành khỏi shipList
END LOOP
Hiển thị kết quả cho người dùng xem
  
```

**METHOD: void finishGame()**

```

Hiển thị thông báo "Game over".
IF số phát đạn đã bắn nhỏ thì
  Hiển thị một thông điệp chúc mừng
ELSE
  Hiển thị một thông điệp chia buồn
END IF
  
```

SinkAShip
GameHelper helper ArrayList shipList int numOfShots
setUpGame() startPlaying() checkUserShot() finishGame()

Đại học Công nghệ - HHOUGHIN



Lớp SinkAShip được cài đặt như sau:

Đại học Công nghệ - ĐHQGHN

---

```

import java.util.*;

public class SinkAShip {
    private GameHelper helper = new GameHelper();
    private ArrayList<Ship> shiplist = new ArrayList<Ship>();
    private int numOfGuesses = 0;

    private void setUpGame() {
        Ship one = new Ship();
        one.setName("Endeavour");
        Ship two = new Ship();
        two.setName("Black Pearl");
        Ship three = new Ship();
        three.setName("Flying Dutchman");
        shiplist.add(one);
        shiplist.add(two);
        shiplist.add(three);

        System.out.println("Your goal is to sink three ships.");
        System.out.println("Endeavour, Black Pearl, Flying Dutchman");
        System.out.println("Try to sink them all in the fewest number of guesses");

        for (Ship ShipSet : shiplist) {
            ArrayList<String> newLocation = helper.placeShip(3);
            ShipSet.setLocationCells(newLocation);
        }
    }

    private void startPlaying() {
        while (! shiplist.isEmpty()) {
            String userGuess = helper.getUserInput("Enter a guess");
            checkUserGuess(userGuess);
        }
        finishGame();
    }

    private void checkUserGuess(String userGuess) {
        numOfGuesses++;
        String result = "miss";

        for (Ship shipToTest : shiplist) {
            result = shipToTest.check(userGuess);
            if (result.equals("hit")) {
                break;
            }
            if (result.equals("kill")) {
                shiplist.remove(shipToTest);
                break;
            }
        }
        System.out.println(result);
    }
}

```

Đại học Công nghệ - ĐHQGHN

```

private void finishGame() {
    System.out.println("All the ships are sunk! You're now the king of the sea!");
    if (numOfGuesses <= 18) {
        System.out.println("It only took you " + numOfGuesses + " guesses");
        System.out.println("You got out before your options sank.");
    } else {
        System.out.println("Took you long enough. " + numOfGuesses + " guesses.");
        System.out.println("Fish are dancing with your options.");
    }
}

public static void main(String[] args) {
    SinkAShip game = new SinkAShip();
    game.setUpGame();
    game.startPlaying();
}
}

```

---

Cuối cùng là lớp GameHelper chứa các phương thức tiện ích cho SinkAShip sử dụng. Lớp này cung cấp hai phương thức. Phương thức getUserInput() nhận input của người chơi bằng cách hiển thị lời mời nhập tọa độ bắn và đọc chuỗi kí tự người dùng gõ vào từ dòng lệnh. Phương thức thứ hai, placeShip(), sinh tự động vị trí cho các con tàu. Trong mã nguồn, có một số lệnh System.out.print(ln) trong phương thức placeShip() đã được chuyển thành dòng chú thích. Đó là các lệnh hiển thị tọa độ của các con tàu. Nếu cho các lệnh này chạy, chúng sẽ cho phép ta biết tọa độ của tàu để chơi "ăn gian" hoặc để test chương trình.

Do chỉ là một ví dụ minh họa, chương trình này tuy hoàn chỉnh nhưng được viết ở mức độ vắn tắt tối đa với giao diện tối thiểu. Bạn đọc có thể sửa để cải thiện phần giao diện đối với người dùng, chẳng hạn như hiển thị bản đồ vùng biển cùng với các thông tin về các tọa độ đã bắn trúng hoặc trượt để hỗ trợ người chơi, hoặc có thể sử dụng thư viện giao diện đồ họa của Java để tăng tính thẩm mỹ và tính thân thiện người dùng.

Đại học Công nghệ - ĐHQGHN

---

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

public class GameHelper {

    private static final String alphabet = "abcdefg";
    private int gridLength = 7;
    private int gridSize = 49;
    private int [] grid = new int[gridSize];
    private int shipCount = 0;

    public String getUserInput(String prompt) {
        String inputLine = null;
        System.out.print(prompt + " ");
        try {
            BufferedReader is = new BufferedReader(
                new InputStreamReader(System.in));
            inputLine = is.readLine();
            if (inputLine.length() == 0 ) return null;
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
        return inputLine.toLowerCase();
    }
}
```

---

Hình 6.4: GameHelper, phần 1/2.

Đại học Công nghệ - ĐHQGHN

---

```

public ArrayList<String> placeShip(int size) {
    ArrayList<String> alphaCells = new ArrayList<String>();
    String [] alphacoords = new String [size]; // holds 'f6' type coords
    String temp = null; // temporary String for concat
    int [] coords = new int[size]; // current candidate coords
    int attempts = 0; // current attempts counter
    boolean success = false; // flag = found a good location ?
    int location = 0; // current starting location
    shipCount++; // nth ship to place
    int incr = 1; // set horizontal increment
    if ((shipCount % 2) == 1) { // if odd ship (place vertically)
        incr = gridLength; // set vertical increment
    }
    while ( !success & attempts++ < 200 ) { // main search loop (32)
        location = (int) (Math.random() * gridSize); // get random starting point
        //System.out.print(" try " + location);
        int x = 0; // nth position in ship to place
        success = true; // assume success
        while (success && x < size) { // look for adjacent unused spots
            if (grid[location] == 0) { // if not already used
                coords[x++] = location; // save location
                location += incr; // try 'next' adjacent
                if (location >= gridSize) { // out of bounds - 'bottom'
                    success = false; // failure
                }
                if (x>0 & (location % gridLength == 0)) { // out of bounds - right edge
                    success = false; // failure
                }
            } else { // found already used location
                // System.out.print(" used " + location);
                success = false; // failure
            }
        }
    }
    // end while
    int x = 0; // turn good location into alpha coords
    int row = 0;
    int column = 0;
    // System.out.println("\n");
    while (x < size) {
        grid[coords[x]] = 1; // mark master grid pts. as 'used'
        row = (int) (coords[x] / gridLength); // get row value
        column = coords[x] % gridLength; // get numeric column value
        temp = String.valueOf(alphabet.charAt(column)); // convert to alpha
        alphaCells.add(temp.concat(Integer.toString(row)));
        x++;
        // System.out.print(" coord " + x + " = " + alphaCells.get(x-1));
    }
    // System.out.println("\n");
    return alphaCells;
}
}

```

---

Hình 6.5: GameHelper, phần 2/2.

Đại học Công nghệ - ĐHQGHN

## Bài tập

1. Viết lớp Dice mô hình hóa xúc xắc và việc tung xúc xắc. Mỗi đối tượng Dice có một biến int lưu trạng thái hiện tại là mặt ngửa của lần gieo gần nhất (một giá trị trong khoảng từ 1 đến 6), một phương thức public roll() giả lập việc gieo xúc xắc và trả về giá trị của mặt ngửa vừa gieo được. Hãy sử dụng thư viện Math cho việc sinh số ngẫu nhiên.
2. Viết lớp Card mô hình hóa các quân bài tứ-lo-kho. Sử dụng ArrayList để xây dựng lớp CardSet mô hình hóa một xấp bài có quân không xác định. Cài phương thức shuffle() của lớp CardSet với nhiệm vụ tráo ngẫu nhiên các quân bài trong xấp bài. Viết lớp CardTestDrive để thử nghiệm hai lớp Card và CardSet nói trên.
3. Có thể dùng một đối tượng thuộc lớp Scanner để đọc dữ liệu từ một file text tương tự như đọc dữ liệu từ bàn phím. Ví dụ:

```
try {  
    Scanner input =  
        new Scanner (new File("C:\\Tmp\\test.txt"));  
    // đọc dữ liệu  
    int n = input.nextInt();  
    ...  
} catch (java.io.FileNotFoundException e) { }
```

a) Hãy viết một chương trình Java đọc dữ liệu từ một file text và in từng từ ra màn hình.

b) Sửa chương trình tại phần a để bỏ qua các dấu .,:...khi đọc các từ trong văn bản.

Gợi ý:

Lệnh sau đây đặt chế độ cho đối tượng Scanner coi tất cả các kí tự không phải a..z hay A..Z như các kí tự phân tách giữa các từ khi thực hiện lệnh đọc từng từ

```
input.useDelimiter(Pattern.compile("[^a-zA-Z]"));
```

Lệnh sau đây bỏ qua tất cả các kí tự không phải a..z hay A..Z cho đến khi gặp một kí tự trong khoản a..z hay A..Z

```
input.skip("[^a-zA-Z]*");
```

Đại học Công nghệ - HHOUGHIN

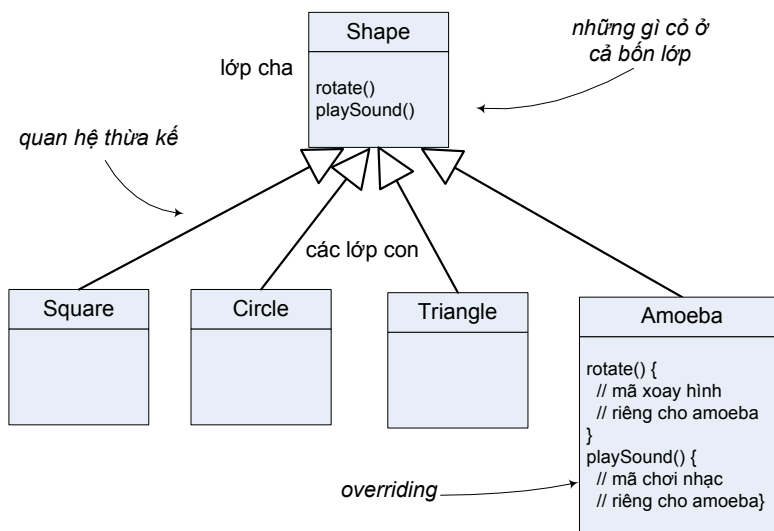
## Chương 7. THỪA KẾ VÀ ĐA HÌNH

Hai nguyên lý thừa kế và đa hình của lập trình hướng đối tượng giúp ta có thể xây dựng chương trình một cách nhanh chóng và hiệu quả hơn, thu được kết quả là những mô-đun chương trình mà các lập trình viên khác dễ mở rộng hơn, có khả năng đáp ứng tốt hơn đối với sự thay đổi liên tục của các yêu cầu của khách hàng.

### 7.1. QUAN HỆ THỪA KẾ

Nhớ lại ví dụ đầu tiên về lập trình hướng đối tượng tại Chương 1. Trong đó, Dâu xây dựng 4 lớp: Square (hình vuông), Circle (đường tròn), Triangle (hình tam giác), và Amoeba (hình trùng biến hình). Cả bốn đều là các hình với hai phương thức rotate() và playSound(). Do đó, anh ta dùng tư duy trừu tượng hóa để tách ra các đặc điểm chung và đưa chúng vào một lớp mới có tên Shape (hình nói chung). Sau đó, kết nối các lớp hình vẽ kia với lớp Shape bởi một quan hệ gọi là thừa kế.

Ta nói rằng "Square thừa kế từ Shape", "Circle thừa kế từ Shape", v.v.. Ta tháo gỡ rotate() và playSound ra khỏi 4 loại hình, và giờ thì chỉ còn phải quản lý một bản đặt tại lớp Shape. Shape được gọi là **lớp cha** (*superclass*) hay **lớp cơ sở** (*base class*) của bốn lớp kia. Còn bốn lớp đó là các **lớp con** (*subclass*) hay **lớp dẫn xuất** (*derived class*) của lớp Shape. Các lớp con thừa kế các phương thức của lớp cha. Nói cách khác, nếu lớp Shape có chức năng gì thì các lớp con của nó tự động có các chức năng đó.



Vậy thế nào là quan hệ thừa kế? Nếu ta cần xây dựng các lớp đại diện cho hai loài mèo nhà và hổ, mèo nhà nên thừa kế từ hổ, hay hổ nên thừa kế từ mèo, hay cả hai cùng thừa kế từ một lớp thứ ba?

Khi ta dùng quan hệ thừa kế trong thiết kế, ta đặt các phần mã dùng chung tại một lớp và coi đó là lớp cha – lớp dùng chung trừu tượng hơn, các lớp cụ thể hơn là các lớp con. Các **lớp con được thừa kế từ lớp cha** đó. Quan hệ thừa kế có nghĩa rằng lớp con được thừa hưởng các **thành viên** (*member*) của lớp cha. Thành viên của một lớp là các biến thực thể và phương thức của lớp đó. Ví dụ, Shape trong ví dụ trên có hai thành viên rotate() và playSound(), Cow trong Hình 5.6 có các thành viên name, age, getName(), getAge(), setName(), setAge().

Ta còn nói rằng lớp con **chuyên biệt hóa** (*specialize*) lớp cha. Nghĩa của "chuyên biệt hóa" ở đây gồm có hai phần: (1) lớp con là một loại con của lớp cha – thể hiện ở chỗ lớp con tự động thừa hưởng các thành viên của lớp cha, (2) lớp con có những đặc điểm của riêng nó - thể hiện ở chỗ lớp con có thể bổ sung các phương thức và biến thực thể mới của riêng mình, và nó có thể **cài đặt** (*override*) các phương thức thừa kế từ lớp cha. Ví dụ, hình trùng biến hình (Amoeba) cũng là một hình (Shape), do đó lớp con Amoeba có tất cả những gì mà Shape có. Ngoài ra, Amoeba có thêm những đặc điểm riêng của thể loại hình trùng biến hình: các biến thực thể đại diện cho tâm xoay để phục vụ cách xoay của riêng nó, và nó định nghĩa lại các phương thức rotate để xoay theo cách riêng, định nghĩa lại playSound để chơi loại âm thanh riêng. Theo thuật ngữ, và cũng là từ khóa, của Java, lớp con "nối dài" (*extends*) lớp cha.

Các **biến thực thể không bị cài đặt** vì việc đó là không cần thiết. Biến thực thể không quy định một hành vi đặc biệt nào và lớp con chỉ việc gán giá trị tùy chọn cho biến được thừa kế.

## 7.2. THIẾT KẾ CÂY THỪA KẾ

Giả sử ta cần thiết kế một chương trình giả lập cho phép người dùng thả một đám các con động vật thuộc các loài khác nhau vào một môi trường để xem chuyện gì xảy ra. Ta hiện chưa phải viết mã mà mới chỉ ở giai đoạn thiết kế.

Ta biết rằng mỗi con vật sẽ được đại diện bởi một đối tượng, và các đối tượng sẽ di chuyển loanh quanh trong môi trường, thực hiện các hành vi được lập trình cho loài vật đó. Ta được giao một danh sách các loài vật sẽ được đưa vào chương trình: sư tử, hà mã, hổ, chó, mèo, sói.

Và ta muốn rằng, *khi cần, các lập trình viên khác cũng có thể bổ sung các loài vật mới vào chương trình.*

**Bước 1**, ta xác định các đặc điểm chung và trừu tượng mà tất cả các loài động vật đều có.

Các đặc điểm chung đó bao gồm:

*năm biến thực thể:*

*picture* – tên file ảnh đại diện cho con vật này



*food* – loại thức ăn mà con vật thích. Hiện giờ, biến này chỉ có hai giá trị: cỏ (grass) hoặc thịt (meat).

*hunger* – một biến int biểu diễn mức độ đói của con vật. Biến này thay đổi tùy theo khi nào con vật ăn và nó ăn bao nhiêu.

*boundaries* – các giá trị biểu diễn chiều dọc và chiều ngang (ví dụ 640 x 480) của khu vực mà các con vật sẽ đi lại hoạt động trong đó.

*location* – các tọa độ X và Y của con vật trong khu vực của nó.

và bốn *phương thức*:

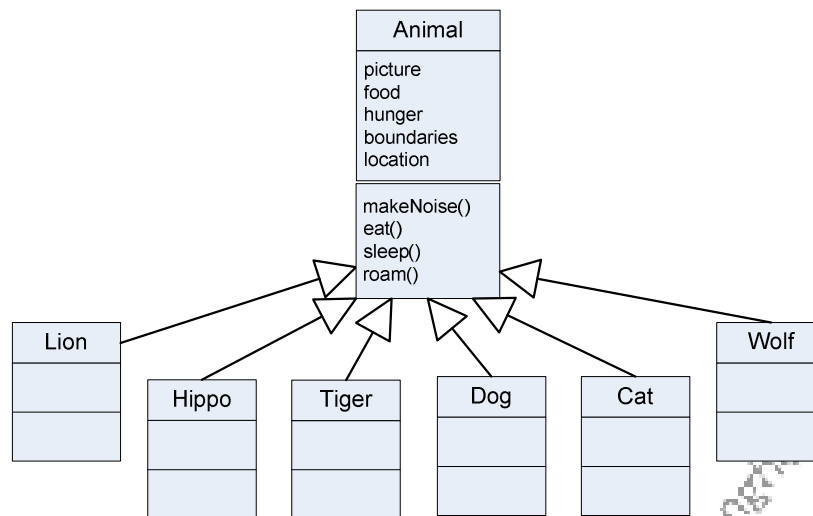
*makeNoise()* – hành vi khi con vật phát ra tiếng kêu

*eat()* – hành vi khi con vật gặp nguồn thức ăn ưa thích, thịt hoặc cỏ.

*sleep()* – hành vi khi con vật được coi là đang ngủ.

*roam()* – hành vi khi con vật không phải đang ăn hay đang ngủ, có thể chỉ đi lang thang đợi gặp món gì ăn được hoặc gặp biên giới lãnh địa.

**Bước 2**, thiết kế một lớp với tất cả các thuộc tính và hành vi chung kể trên. Đây sẽ là lớp mà tất cả các lớp động vật đều có thể chuyên biệt hóa. Các đối tượng trong ứng dụng đều là các con vật (animal), do đó, ta sẽ gọi tên lớp cha chung của chúng là Animal. Ta đưa vào đó các phương thức và biến thực thể mà tất cả các con vật đều có thể cần. Kết quả là ta được lớp cha là lớp tổng quát hơn, hay nói cách khác là trừu tượng hơn, còn các lớp con mang tính đặc thù hơn, chuyên biệt hơn lớp cha.



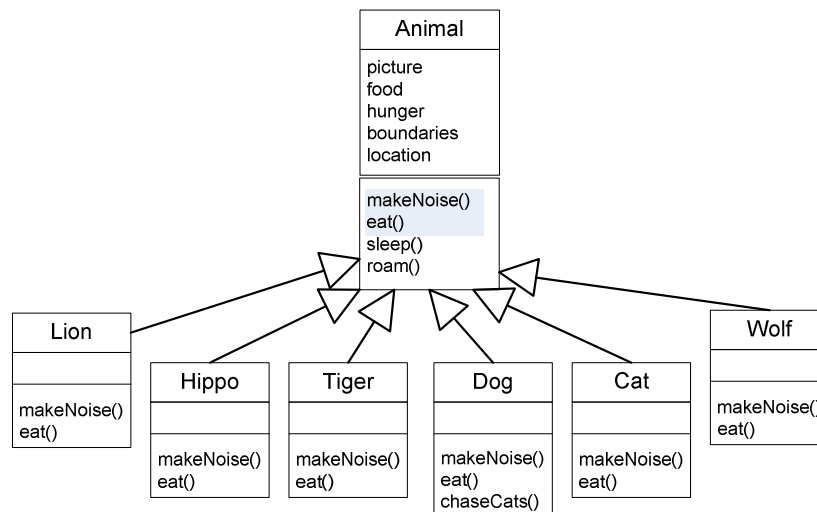
*Các con vật hoạt động có giống nhau không?*

Ta đã biết rằng mỗi loại Animal đều có tất cả các biến thực thể đã khai báo cho Animal. Một con sư tử sẽ có các giá trị riêng cho *picture*, *food*, *hunger*, *boundaries*, và *location*. Một con hà mã sẽ có những giá trị khác cho bộ biến thực thể tương tự. Cũng như vậy đối với chó, hổ... Thế còn các *hành vi* của chúng thì sao?

**Bước 3:** Xác định xem các lớp con có cần các hành vi (cài đặt của các phương thức) đặc thù của thể loại con cụ thể đó hay không?

Để ý lớp Animal. Chắc chắn sư tử không ăn giống hà mã. Còn về tiếng kêu, ta có thể viết duy nhất một phương thức makeNoise tại Animal trong đó chơi một file âm thanh có tên là giá trị của một biến thực thể mà có giá trị khác nhau tùy loài, để con vật này kêu khác con vật khác. Nhưng làm vậy có vẻ chưa đủ vì tùy từng tình huống mà các loài khác nhau phát ra các tiếng kêu khác nhau, chẳng hạn tiếng kêu khi đang ăn và tiếng kêu khi gặp kẻ thù, v.v..

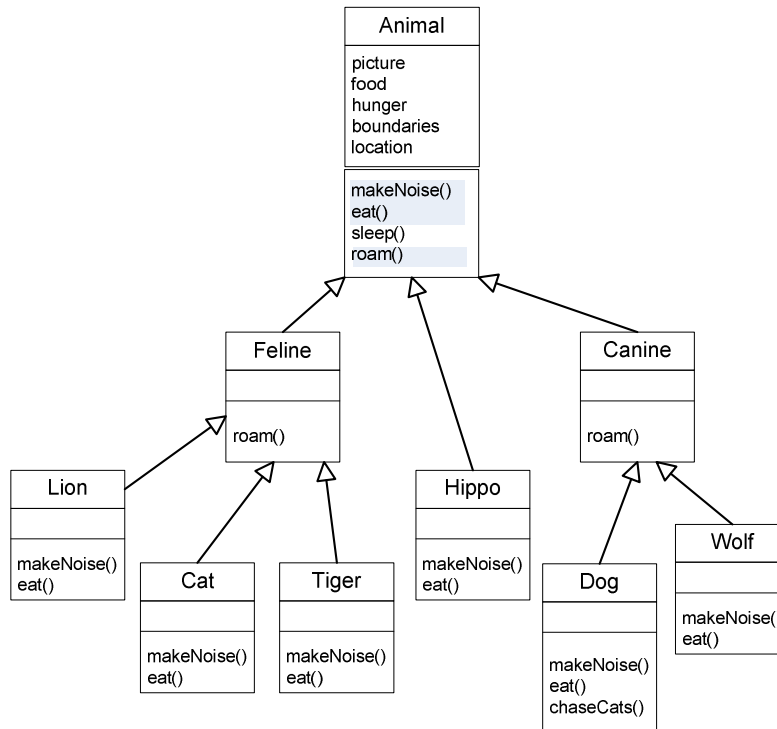
Do đó, ta quyết định rằng eat() và makeNoise() nên được cài đặt tại từng lớp con. Tạm coi các con vật sleep và roam như nhau và không cần cài đặt hai phương thức này. Ngoài ra, một số loài có những hành vi riêng đặc trưng của loài đó, chẳng hạn chó có thêm hành vi đuổi mèo (chaseCats()) bên cạnh các hành vi mà các loài động vật khác cũng có.



**Bước 4:** Tiếp tục dùng trừu tượng hóa tìm các lớp con có thể còn có hành vi giống nhau, với mục đích phân nhóm mịn hơn nếu cần.

Ví dụ, sói và chó có họ hàng gần, cùng thuộc họ Chó (*canine*) trong phân loại động vật học, chúng cùng có xu hướng di chuyển theo bầy đàn nên có thể dùng chung một phương thức roam(). Mèo, hổ và sư tử cùng thuộc họ Mèo (*feline*). Ba loài này có thể chung phương thức roam() vì khi di chuyển chúng cùng có xu hướng tránh đồng loại. Ta sẽ để cho hà mã tiếp tục dùng phương thức roam() tổng quát được thừa kế từ Animal.

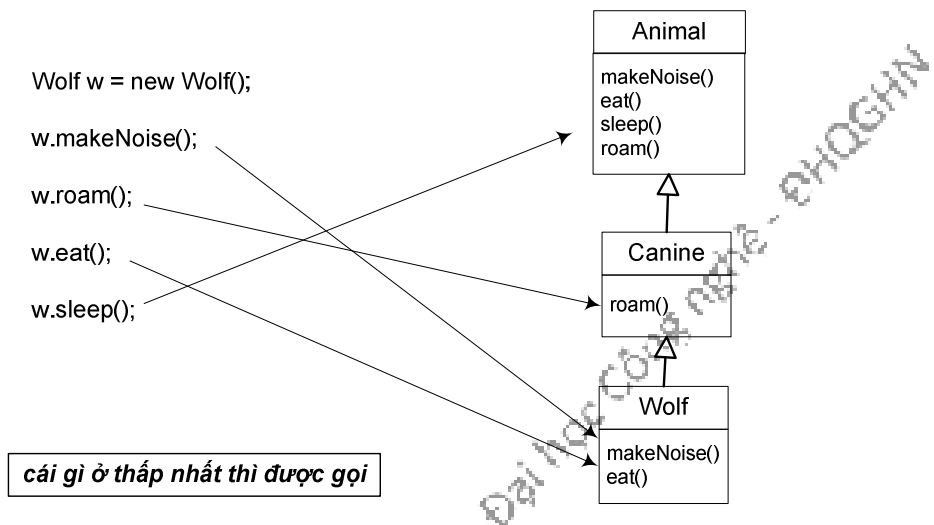
Ta tạm hoàn thành thiết kế như trong Hình 7.1 và sẽ quay lại bài toán này trong chương sau.



Hình 7.1: Cây thừa kế của các loài động vật.

### 7.3. CÀI ĐỀ – PHƯƠNG THỨC NÀO ĐƯỢC GỌI?

Lớp Wolf có bốn phương thức: sleep() được thừa kế từ Animal, roam() được thừa kế từ Canine (thực ra là phiên bản đề bản của Animal), và hai phương thức mà Wolf cài đề bản của Animal - makeNoise() và eat(). Khi ta tạo một đối tượng Wolf và gán một biến tham chiếu tới nó, ta có thể dùng biến đó để gọi cả bốn phương thức trên. Nhưng *phiên bản* nào của chúng đó sẽ được gọi?



Khi gọi phương thức từ một tham chiếu đối tượng, ta đang gọi phiên bản đặc thù nhất của phương thức đó đối với lớp của đối tượng cụ thể đó. Nếu hình dung cây thừa kế theo kiểu các lớp cha ở phía trên còn các lớp con ở phía dưới, thì quy tắc ở đây là: phiên bản thấp nhất sẽ được gọi. Trong ví dụ dùng biến `w` để gọi phương thức cho một đối tượng `Wolf` ở trên, thứ tự từ thấp lên cao lần lượt là `Wolf`, `Canine`, `Animal`. Khi gọi một phương thức cho một đối tượng `Wolf`, máy ảo Java bắt đầu tìm từ lớp `Wolf` lên, nếu nó không tìm được một phiên bản của phương thức đó tại `Wolf` thì nó chuyển lên tìm tại lớp tiếp theo bên trên `Wolf` ở cây thừa kế, cứ như vậy cho đến khi tìm thấy một phiên bản khớp với lời gọi phương thức. Với ví dụ đang xét, như được minh họa trong hình vẽ, `w.makeNoise()` sẽ dẫn đến việc kích hoạt phiên bản của `Wolf`, `w.roam()` gọi phiên bản của `Canine`, v.v..

#### 7.4. CÁC QUAN HỆ IS-A VÀ HAS-A

Như đã trình bày trong các chương trước, khi một lớp kế thừa từ một lớp khác, ta nói rằng lớp con *chuyên biệt hóa* lớp cha. Nhưng liệu khi nào thì nên chuyên biệt hóa một lớp khác?

Nhớ lại rằng lớp cha là loại tổng quát, còn lớp con là loại cụ thể và chuyên biệt, là loại con của lớp cha. Nhìn từ khía cạnh khác, tập hợp các đối tượng mà lớp con đại diện là một tập con của các đối tượng mà lớp cha đại diện. Do đó, để đưa ra lựa chọn đúng đắn cho vấn đề nên hay không nên để lớp X là lớp chuyên biệt hóa lớp Y, ta có một phương pháp hiệu quả: kiểm tra quan hệ IS-A, nghĩa là xem thứ này có là thứ kia hay không.

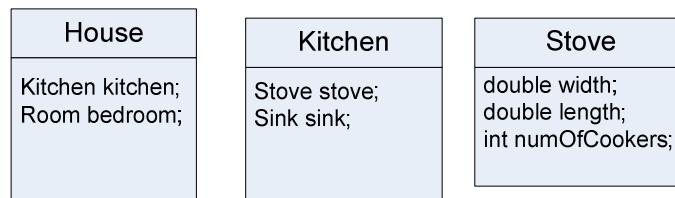
Để xem X có nên là lớp con của Y hay không, ta đặt câu hỏi theo dạng "Nếu phát biểu một cách tổng quát rằng loại X là một dạng/thứ/kiểu của loại Y thì có lý hay không?". Nếu câu trả lời là "Có", thì X có thể là lớp con của Y.

Ví dụ: Tam giác là một hình (Triangle IS-A Shape)? Đúng. Mèo là một động vật họ Mèo (Cat IS-A Feline)? Đúng. Xe tải là một phương tiện giao thông (Truck IS-A Vehicle)? Đúng. Nghĩa là, Triangle có thể là lớp con của Shape, Cat có thể là lớp con của Feline, Truck có thể là lớp con của Vehicle.

Ta xét tiếp: Phòng bếp là một cái nhà (Kitchen IS-A House)? Chắc chắn sai. Ngược lại thì sao? Nhà là một phòng bếp (House IS-A Kitchen)? Đúng là có một số người vì phong tục hay điều kiện sống mà ngôi nhà của họ chỉ có một phòng duy nhất nên đó vừa là nơi nấu bếp vừa là phòng cho nhiều chức năng khác. Tuy nhiên, các trường hợp đó chỉ là "một số", nên câu trả lời tổng quát vẫn là "Sai". Cho nên, Kitchen không thể là lớp con của House hay ngược lại.

Phòng bếp và nhà rõ ràng có liên quan đến nhau, nhưng không phải qua quan hệ thừa kế mà là một quan hệ chứa – HAS-A. Câu hỏi ở đây là: Nhà có chứa một phòng bếp hay không (House HAS-A Kitchen)? Nếu câu trả lời là "Có", điều đó có nghĩa House có một biến thực thể kiểu Kitchen. Nói cách khác, House có một *tham*

chiều tới một đối tượng Kitchen, chứ House không *chuyên biệt hóa* Kitchen hay ngược lại.



*House HAS-A Kitchen, Kitchen HAS-A Stove.  
Nhà có một phòng bếp, phòng bếp có một cái bếp.  
Nhưng không có lớp nào thừa kế một lớp khác.*

Quan hệ HAS-A trong Java được cài đặt bằng tham chiếu đặt tại đối tượng chứa chiều tới đối tượng thành phần. Quan hệ HAS-A giữa hai lớp thể hiện một trong ba quan hệ: **kết hợp** (*association*), **tụ hợp** (*aggregation*) và **hợp thành** (*composition*) mà các tài liệu về thiết kế hướng đối tượng thường nói đến. Giữa hai lớp có quan hệ kết hợp nếu như các đối tượng thuộc lớp này cần biết đến đối tượng thuộc lớp kia để có thể thực hiện được công việc của mình. Chẳng hạn, một người nhân viên chịu sự quản lý của một người quản lý, ta có quan hệ kết hợp nối từ Employee tới Manager, thể hiện ở việc mỗi đối tượng Employee có một tham chiếu boss kiểu Manager. Hợp thành và tụ hợp là các quan hệ giữa một đối tượng và thành phần của nó (cũng là đối tượng). Khác nhau ở chỗ, với quan hệ hợp thành, đối tượng thành phần là phần không thể thiếu được của đối tượng chứa nó, còn với quan hệ tụ hợp thì ngược lại. Ví dụ, một cuốn sách bao gồm nhiều trang sách và một cuốn sách không thể tồn tại nếu không có trang nào. Do đó giữa Book (sách) và Page (trang) có quan hệ hợp thành. Thư viện có nhiều sách, nhưng thư viện không có cuốn sách nào vẫn là một thư viện, nên quan hệ giữa Library (thư viện) và Book là quan hệ tụ hợp. Java không có cấu trúc nào dành riêng để cài đặt các quan hệ tụ hợp hay hợp thành. Ta chỉ cài đặt đơn giản bằng cách đặt vào đối tượng chủ các tham chiếu tới đối tượng thành phần, hay nói cách khác là phân rã thành các quan hệ HAS-A, chẳng hạn quan hệ hợp thành giữa Book và Page có thể được phân rã thành 'Book HAS-A ArrayList<Page>' và nhiều quan hệ 'ArrayList<Page> HAS-A Page'. Các ràng buộc khác được đảm bảo bởi các phương thức có nhiệm vụ khởi tạo hay sửa các tham chiếu đó.

Quay lại quan hệ IS-A, có một điểm cần lưu ý: **quan hệ thừa kế IS-A chỉ có một chiều**. Ví dụ: "Tam giác là một hình" là phát biểu có lý, nhưng khẳng định theo chiều ngược lại, "Hình là một tam giác", thì không đúng. Có nhiều hình là hình tam giác, nhưng cũng có vô số hình không phải hình tam giác.

Thực ra, lưu ý trên là hiển nhiên, nếu ta nhớ đến mô tả về lớp con tại mục trước: **Lớp con chuyên biệt hóa lớp cha**.

Đến đây, chúng ta chưa kết thúc câu chuyện về quan hệ thừa kế. Chương sau sẽ tiếp tục trình bày về các vấn đề hướng đối tượng. Một số giải pháp thiết kế trong chương này sẽ được xem lại và cải tiến.

## 7.5. KHI NÀO NÊN DÙNG QUAN HỆ THỪA KẾ?

Mục này liệt kê một số quy tắc hướng dẫn việc sử dụng quan hệ thừa kế trong thiết kế. Tại thời điểm này, ta tạm bằng lòng với việc *biết* quy tắc. Việc *hiểu* quy tắc nếu chưa trọn vẹn thì sẽ được bồi đắp dần trong những phần sau của cuốn sách.

**NÊN** dùng quan hệ thừa kế khi một lớp là một loại cụ thể hơn của một lớp cha. Ví dụ, tài khoản tiết kiệm (saving account) là một loại tài khoản ngân hàng (bank account), nên SavingAccount là lớp con của BankAccount là hợp lí.

**NÊN** cân nhắc việc thừa kế khi ta có một hành vi (mã đã được viết) nên được dùng chung giữa nhiều lớp thuộc cùng một kiểu tổng quát nào đó. Ví dụ, Square, Circle và Triangle trong bài toán của Dâu và Tuất cùng cần xoay và chơi nhạc, nên việc đặt các chức năng đó tại một lớp cha Shape là hợp lí. Tuy vậy, cần lưu ý rằng mặc dù thừa kế là một trong những đặc điểm quan trọng của lập trình hướng đối tượng nhưng nó không nhất thiết là cách tốt nhất cho việc tái sử dụng hành vi. Quan hệ thừa kế giúp ta khởi động việc tái sử dụng, và nó thường là lựa chọn đúng khi thiết kế, nhưng các mẫu thiết kế sẽ giúp ta nhận ra những lựa chọn khác tinh tế và linh hoạt hơn.

**KHÔNG NÊN** dùng thừa kế chỉ nhằm mục đích tái sử dụng mã của một lớp khác, trong khi quan hệ giữa lớp cha và lớp con vi phạm một trong hai quy tắc ở trên. Ví dụ, giả sử ta đã viết cho lớp DoorBell (chuông cửa) một đoạn mã dành riêng cho việc in, và giờ ta cần viết mã cho chức năng in của lớp Piano. Không nên vì nhu cầu đó mà cho Piano làm lớp con của DoorBell. Đàn piano không phải là một loại chuông gọi cửa. (Giải pháp nên chọn cho tình huống này là: phần mã cho chức năng in nên được đặt trong một lớp Printer, và các lớp cần có chức năng in sẽ hưởng lợi từ lớp Printer đó qua một quan hệ HAS-A.)

**KHÔNG NÊN** dùng quan hệ thừa kế nếu lớp con và lớp cha không qua được thử nghiệm IS-A. Hãy tự kiểm tra xem lớp con có phải là một kiểu chuyên biệt của lớp cha hay không. Ví dụ: *Bike IS-A Vehicle* (xe đạp là một phương tiện giao thông) hợp lí. Nhưng *Vehicle IS-A Bike* (phương tiện giao thông là một loại xe đạp) thì không được.

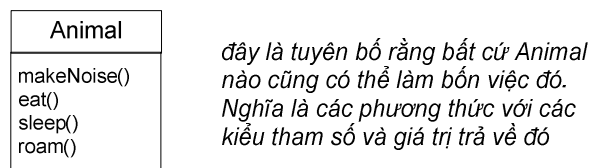
## 7.6. LỢI ÍCH CỦA QUAN HỆ THỪA KẾ

Quan hệ thừa kế trong thiết kế mang lại cho ta rất nhiều điều.

Lợi ích thứ nhất: tránh lặp các đoạn mã bị trùng lặp. Ta có thể loại bỏ được những đoạn mã trùng lặp bằng cách tách ra các hành vi chung của một nhóm các lớp đối tượng và đưa phần mã đó vào một lớp cha. Nhờ đó, khi ta cần sửa nó, ta chỉ cần cập nhật mã ở duy nhất một nơi, và *sửa đổi đó có hiệu lực tại tất cả các lớp kế thừa hành vi đó*. Công việc gói gọn trong việc sửa và dịch lớp cha. Tóm lại: **ta không phải động đến các lớp con!**

Với ngôn ngữ Java, chương trình là một tập các lớp. Do đó, ta không cần phải dịch lại các lớp con để có thể dùng được phiên bản mới của lớp cha. Điều duy nhất là phiên bản mới của lớp cha không *phá vỡ* cái gì của lớp con. Nghĩa cụ thể của từ "phá vỡ" trong ngữ cảnh trên sẽ được trình bày chi tiết sau. Tạm thời, ta chỉ cần hiểu rằng hành động đó có nghĩa là sửa cái gì đó tại lớp cha mà lớp con bị phụ thuộc vào, chẳng hạn như sửa kiểu tham số, hay kiểu trả về, hoặc tên của một phương thức nào đó.

Lợi ích thứ hai: ta định nghĩa được một giao thức chung cho tập các lớp gắn kết với nhau bởi quan hệ thừa kế. Quan hệ thừa kế cho phép ta đảm bảo rằng tất cả các lớp con của một lớp đều có tất cả các phương thức<sup>7</sup> mà lớp đó có. Đó là một dạng giao thức mà lớp đó tuyên bố với tất cả các phần mã khác rằng: "Tất cả các thể loại con của tôi (nghĩa là các lớp con) đều có thể làm những việc này, với các phương thức trông như thế này...". Nói cách khác, ta thiết lập một **hợp đồng** (*contract*).

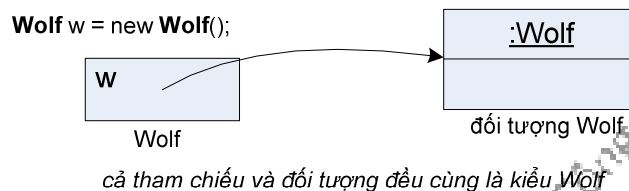


Lưu ý rằng, khi nói về *Animal* bất kì, ý ta đang nói về đối tượng *Animal* hay đối tượng thuộc *bất cứ lớp nào có Animal là tổ tiên trong cây phả hệ*. Khi ta định nghĩa một kiểu tổng quát (lớp cha) cho một nhóm các lớp, bất cứ lớp con nào trong nhóm đó đều có thể dùng thay cho vị trí của lớp cha. Ta đã có *Wolf* là một loại con của *Animal*; một đối tượng *Wolf* có tất cả các thành viên mà một đối tượng *Animal* có. Vậy thì lô-gic hiển nhiên: một đối tượng *Wolf* có thể được coi là thuộc loại *Animal*; nơi nào dùng được *Animal* thì cũng dùng được *Wolf*.

Ta bắt đầu chạm đến phần thú vị nhất của lập trình hướng đối tượng: đa hình.

## 7.7. ĐA HÌNH

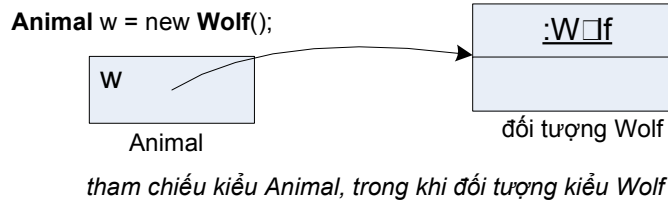
Trước khi trình bày về đa hình, ta nhắc lại một chút về cách khai báo một tham chiếu và tạo một đối tượng.



<sup>7</sup> Nếu muốn nói thật chính xác thì phải là "tất cả các phương thức thừa kế được". Tạm thời, nó có nghĩa là "các phương thức public", nhưng ta sẽ tinh chỉnh định nghĩa này sau.

Trong ví dụ trên, tham chiếu `w` được khai báo bằng lệnh `Wolf w`, đối tượng lớp `Wolf` được khai báo bằng lệnh `new Wolf`. Điểm đáng chú ý là kiểu của biến tham chiếu và kiểu của đối tượng cùng là `Wolf`.

Với đa hình thì sao? Đây là ví dụ: `w` được khai báo thuộc kiểu `Animal`, trong khi đối tượng vẫn được tạo theo kiểu `Wolf`:



Với đa hình, tham chiếu có thể thuộc kiểu lớp cha của lớp của đối tượng được tạo. Khi ta khai báo một biến tham chiếu thuộc kiểu lớp cha, nó có thể được gán với bất cứ đối tượng nào thuộc một trong các lớp con.

Đặc tính này cho phép ta có những thứ thú vị kiểu như mảng đa hình. Ví dụ, trong Hình 7.2, ta khai báo một mảng kiểu `Animal`, nghĩa là một mảng để chứa các đối tượng thuộc loại `Animal`. Nhưng sau đó ta lại gán vào mảng các đối tượng thuộc các lớp con tùy ý của `Animal`. Và vòng lặp duyệt mảng sau đó là phần thú vị nhất liên quan đến đa hình – ý trọng tâm của ví dụ. Tại đó, ta duyệt từ đầu đến cuối mảng, với mỗi phần tử mảng, ta gọi một trong các phương thức `Animal` từ tham chiếu kiểu `Animal`. Khi `i` chạy từ 0 tới 4, `animals[i]` lần lượt chiếu tới một đối tượng `Dog`, `Cat`, `Wolf`, `Hippo`, `Lion`. Kết quả của `animals[i].eat()` hay `animals[i].roam()` đều là: mỗi đối tượng thực hiện đúng phiên bản thích hợp với loại của chính mình.

```
Animal[] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Wolf();
animals[3] = new Hippo();
animals[4] = new Lion();

for (int i = 0; i < animals.length; i++) {
    animals[i].eat();
    animals[i].roam();
}
```

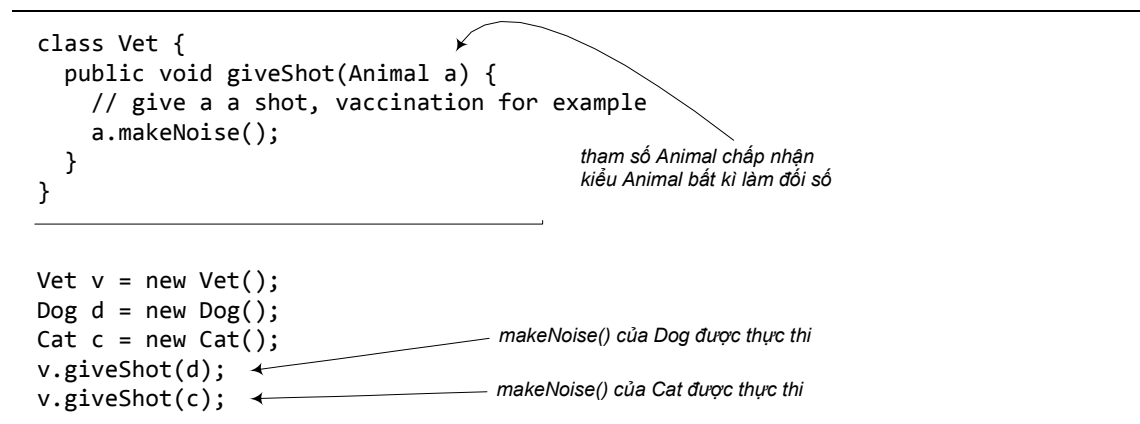
*mảng được khai báo kiểu Animal nhưng lại chứa đối tượng thuộc đủ loại lớp con của Animal*

*vòng lặp gọi phương thức cho từng Animal, nhưng mỗi đối tượng lại thực hiện đúng công việc của mình*

Hình 7.2: Mảng đa hình

Tính đa hình còn có thể thể hiện ở kiểu dữ liệu của đối số và giá trị trả về.





Hình 7.3: Tham số đa hình

Trong ví dụ Hình 7.3, tại phương thức `giveShot()`, tham số `Animal` chấp nhận đối số thuộc kiểu `Animal` bất kì. Đoạn mã bên dưới đã gọi `giveShot()` lần lượt với đối số là các đối tượng `Dog` và `Cat`. Sau khi bác sĩ thú y (`Vet`) tiêm xong, `makeNoise()` được gọi từ trong phương thức `giveShot()` cho đối tượng `Animal` mà `a` đang chiếu tới. Mặc dù `a` là tham chiếu thuộc kiểu `Animal`, nhưng đối tượng nó chiếu tới thuộc lớp nào quyết định phiên bản `makeNoise()` nào được chạy. Kết quả là phiên bản của `Dog` được chạy cho đối tượng `Dog`, và phiên bản của `Cat` được chạy cho đối tượng `Cat`.

Như vậy, với đa hình, ta có thể viết những đoạn mã không phải sửa đổi khi ta bổ sung lớp con mới vào chương trình. Lấy ví dụ lớp `Vet` trong ví dụ vừa rồi, do sử dụng tham số kiểu `Animal`, phần mã này có thể dùng cho lớp con bất kì của `Animal`. Bên cạnh các lớp `Lion`, `Tiger`...sẵn có, nếu ta muốn bổ sung loài động vật mới, chẳng hạn `Cow`, trong khi vẫn muốn tận dụng lớp `Vet`, ta chỉ cần cho lớp mới đó là lớp con của `Animal`. Khi đó, các phương thức của `Vet` vẫn tiếp tục hoạt động được với lớp mới, mặc dù khi viết `Vet` ta không có chút thông tin gì về các loại con của `Animal` mà nó sẽ hoạt động cùng.

Tóm lại, đa hình là gì? Theo nghĩa tổng quát, đa hình là khả năng tồn tại ở nhiều hình thức. Trong hướng đối tượng, đa hình đi kèm với quan hệ thừa kế và có hai đặc điểm sau: (1) các đối tượng thuộc các lớp dẫn xuất khác nhau có thể được đối xử như nhau, như thể chúng là các đối tượng thuộc lớp cơ sở, chẳng hạn có thể gửi cùng một thông điệp tới đối tượng; (2) khi nhận được cùng một thông điệp đó, các đối tượng thuộc các lớp dẫn xuất khác nhau hiểu nó theo những cách khác nhau.

Ta đã thấy đặc điểm thứ nhất thể hiện ở việc ta có thể dùng tham chiếu kiểu lớp cha để chiếu tới các đối tượng thuộc lớp con như thể chúng đều là các đối tượng thuộc lớp cha, trong các ví dụ gần đây là tham số `Animal` chấp nhận các đối số kiểu `Dog` và `Cat`, `Vet` đối xử với các loại con của `Animal` một cách thống nhất như thể chúng đều thuộc loại `Animal`. Đặc điểm thứ hai thể hiện ở việc khi ta gọi phương thức của đối tượng từ tham chiếu kiểu cha, phiên bản được gọi tùy theo đối tượng thuộc loại cụ thể gì. Kết quả của cùng một lệnh `a.makeNoise()` là `makeNoise()` của

Dog được gọi nếu a đang chiếu tới đối tượng Dog, makeNoise() của Cat được gọi nếu a đang chiếu tới đối tượng Cat.

## 7.8. GỌI PHIÊN BẢN PHƯƠNG THỨC CỦA LỚP CHA

Đôi khi, tại một lớp con, ta cài đặt một hành vi của lớp cha, nhưng ta không muốn thay thế hoàn toàn mà chỉ muốn bổ sung một số chi tiết. Chẳng hạn, lớp Account đại diện cho tài khoản ngân hàng chung chung. Nó cung cấp phương thức withdraw(double) với chức năng rút tiền, phương thức này thực hiện quy trình rút tiền cơ bản: trừ số tiền rút khỏi số dư tài khoản (balance). FeeBasedAccount là loại tài khoản ngân hàng thu phí đối với mỗi lần rút tiền, nghĩa là bên cạnh quy trình rút tiền cơ bản, nó còn làm thêm một việc là trừ phí rút tiền khỏi số dư tài khoản. Như vậy, FeeBasedAccount có cần đến nội dung của bản withdraw() được Account cung cấp sẵn, nhưng vẫn phải cài đặt vì nội dung đó không đủ dùng. Ta cũng không muốn chép nội dung bản withdraw() của Account vào bản của FeeBasedAccount. Thay vào đó, ta muốn có cách gọi phương thức withdraw() của Account từ trong phiên bản cài đặt tại FeeBasedAccount.

Tóm lại, từ trong phiên bản cài đặt tại lớp con, ta muốn gọi đến chính phương thức đó của lớp cha, ta phải làm như thế nào? Từ khóa super cho phép gọi đến cách thành viên được thừa kế. Phương thức withdraw() của FeeBasedAccount có thể được cài đặt đại loại như trong Hình 7.4

```
public class Account {
    private double balance = 0;
    public void deposit(double money) {
        balance += money;
    }
    public void withdraw(double money) {
        balance -= money;
    }
}

public class FeeBasedAccount extends Account {
    double fee = 10;
    public void withdraw(double money) {
        super.withdraw(money); ← gọi phương thức
        balance -= fee;                                     của lớp cha
    }
}
```

Hình 7.4: Gọi phiên bản phương thức của lớp cha.

Một tham chiếu tới đối tượng thuộc lớp con sẽ luôn luôn gọi phiên bản mới nhất – chính là phiên bản của lớp con nếu có. Đó là cách hoạt động của đa hình. Tuy nhiên, từ khóa super cho phép gọi phiên bản cũ hơn – phiên bản mà lớp con được thừa kế.

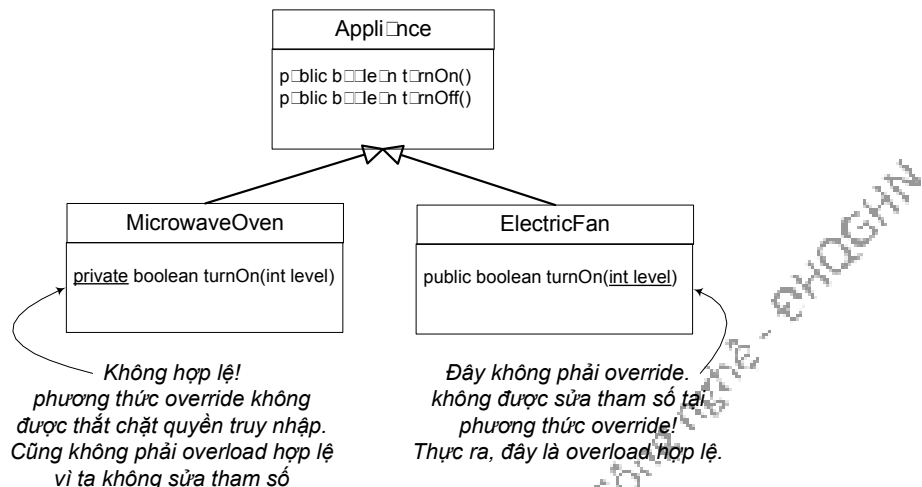
Từ khóa **super** của Java thực chất là một tham chiếu tới phần được thừa kế của một đối tượng. Khi mã của lớp con dùng **super**, chẳng hạn như trong lời gọi phương thức, phiên bản được thừa kế sẽ chạy.

## 7.9. CÁC QUY TẮC CHO VIỆC CÀI ĐỀ

Khi ta cài đề một phương thức của lớp cha, ta đồng ý tuân thủ hợp đồng mà lớp cha đã cam kết. Chẳng hạn, hợp đồng nói rằng "tôi không lấy đối số và tôi trả về một giá trị boolean". Nói cách khác, các kiểu đối số và kiểu trả về của phiên bản mới của phương thức phải trông giống hệt với bản của lớp cha.

Các phương thức chính là hợp đồng.

Nhớ lại rằng, với mỗi lời gọi phương thức, trình biên dịch dùng kiểu tham chiếu để xác định xem ta có thể gọi phương thức đó từ tham chiếu đó hay không. Với một tham chiếu kiểu `Appliance` (thiết bị điện) chiếu tới một đối tượng `ElectricFan` (quạt điện), trình biên dịch chỉ quan tâm xem lớp `Appliance` có phương thức mà ta đang gọi từ tham chiếu `Appliance` hay không. Còn khi chương trình chạy, máy ảo Java không để ý đến kiểu tham chiếu (`Appliance`) và chỉ quan tâm đến đối tượng `ElectricFan` thực tế đang nằm trong bộ nhớ heap. Do đó, nếu trình biên dịch đã chấp thuận lời gọi phương thức, lời gọi đó chỉ có thể hoạt động được nếu như phiên bản cài đề cũng có các tham số và kiểu trả về giống như phiên bản của `Appliance`. Khi ai đó dùng một tham chiếu `Appliance` gọi `turnOn()` không có đối số, phiên bản `turnOn()` của `Appliance` sẽ được chạy, ngay cả khi `ElectricFan` có một bản `turnOn()` với một tham số `int`. Nói cách khác, đơn giản là phương thức `turnOn(int level)` tại `ElectricFan` không đề phiên bản `turnOn()` không tham số tại `Appliance`!



Hình 7.5: Ví dụ về cài đề sai

Việc cài đề phải tuân thủ các quy tắc sau:

1. **Danh sách tham số phải trùng nhau, kiểu giá trị trả về phải tương thích.** Hợp đồng của lớp cha quy định quy cách mà các phần mã khác sử dụng các phương thức của nó. Phương thức của lớp cha có thể được gọi với danh sách đối số như thế nào thì cũng có thể gọi phương thức của lớp con với danh sách đối số đó. Phương thức của lớp cha tuyên bố kiểu trả về là gì, thì phương thức của lớp con cũng phải khai báo chính kiểu trả về đó hoặc một kiểu lớp con của kiểu đó. Nhớ lại rằng một đối tượng thuộc lớp con phải được đảm bảo có thể làm được bất cứ thứ gì mà lớp cha đã tuyên bố, do đó, việc trả về đối tượng lớp con ở vị trí của đối tượng lớp cha là việc an toàn.
2. **Phương thức đề không được giảm quyền truy nhập so với phiên bản của lớp cha.** Nói cách khác, quyền truy nhập mà phiên bản của lớp con cho phép phải bằng hoặc rộng hơn phiên bản của lớp cha. Ta không thể cài đề một phương thức public bằng một phiên bản private. Nếu không, tình huống xảy ra là một lời gọi phương thức đã được trình biên dịch chấp nhận vì tưởng là phương thức public nhưng đến khi nó chạy lại bị máy ảo từ chối vì phiên bản được gọi lại là private.

Như vậy, ta đã hiểu thêm về hai mức quyền truy nhập: private và public. Còn hai mức quyền truy nhập khác sẽ được nói đến trong Mục 7.11. Ngoài ra còn có một quy tắc khác về cài đề liên quan đến xử lý ngoại lệ, ta sẽ nói về quy tắc này tại Chương 10.

## 7.10. CHỒNG PHƯƠNG THỨC

Các ví dụ về cài đề sai trong mục trước đã nói đến khái niệm **cài chồng phương thức** (*method overload*).

Cài chồng phương thức chỉ đơn giản là có một vài phương thức trùng tên nhưng khác danh sách đối số. Phương thức chồng không liên quan đến đa hình hay thừa kế. Một phương thức cài chồng không phải phương thức cài đề.

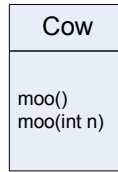
Cài chồng phương thức cho phép ta tạo nhiều phiên bản của một phương thức, mỗi phiên bản chấp nhận một danh sách đối số khác nhau, nhằm tạo thuận lợi cho việc gọi phương thức.

Đại học Công nghệ - HHOGR

```

public class Cow {
    public void moo() {
        System.out.println(name + " says Moooo...");
    }
    public void moo(int n) {
        System.out.print(name + " says");
        for (int i = 0; i < n; i++)
            System.out.print(" Moooo...");
        System.out.println("");
    }
}

```



Hình 7.6: Ví dụ về phương thức chồng

Ta sẽ còn quay lại các trường hợp áp dụng cài chồng khi nói về các hàm khởi tạo (constructor) trong Chương 9.

Do cơ chế cài chồng phương thức không phải tuân thủ hợp đồng đa hình do lớp cha quy định, các phương thức chồng có tính linh hoạt cao hơn.

- Kiểu trả về có thể khác nhau. Ta có thể tùy ý thay đổi kiểu trả về tại các phương thức chồng, miễn là danh sách đối số khác nhau.
- Khác biệt duy nhất ở kiểu trả về là không đủ. Nếu không, đó không phải là việc cài chồng hợp lệ, trình biên dịch sẽ cho rằng ta đang định cài đè phương thức. Để overload, ta nhất định phải sửa danh sách tham số.
- Có thể nói rộng hoặc hạn chế quyền truy nhập tùy ý. Ta có thể tùy ý thay đổi quyền truy nhập của phương thức chồng vì phương thức mới không bị buộc phải tuân theo hợp đồng đa hình, nếu có, của phương thức cũ.

## 7.11. CÁC MỨC TRUY NHẬP

Đến đây, ngoài hai từ khóa public và private quy định mức truy nhập, ta đã có thể học thêm về loại **protected** (được bảo vệ). Mục này tổng kết các kiến thức về các loại quyền truy nhập mà Java quy định.

Ta có bốn **mức truy nhập** (*access level*) và ba từ khóa tương ứng private, protected và public, mức còn lại là mức mặc định không cần từ khóa. Các mức truy nhập được liệt kê theo thứ tự từ chặt tới lỏng như sau:

- **mức private**: chỉ có mã bên trong cùng một lớp mới có thể truy nhập được những thứ private. private ở đây có nghĩa "của riêng lớp" chứ không phải "của riêng đối tượng". Một đối tượng Dog có thể sửa các biến private hay gọi phương thức private của một đối tượng Dog khác, nhưng một đối tượng Cat thì thậm chí không 'nhìn thấy' các thứ private của Dog. Các đối tượng Dog cũng không thể 'nhìn thấy' các biến / phương thức private của các đối tượng Animal mà nó thừa kế. Vậy nên người ta nói rằng lớp con không thừa kế các biến / phương thức private của lớp cha.

- **mức truy nhập mặc định:** các biến/phương thức với mức truy nhập mặc định của một lớp chỉ có thể được truy nhập bởi mã nằm bên trong cùng một gói với lớp đó.
- **mức protected:** các biến/phương thức với mức protected của một lớp chỉ có thể được thừa kế bởi các lớp con cháu của lớp đó, kể cả nếu lớp con đó không nằm trong cùng một gói với lớp cha.
- **mức public:** mã ở bất cứ đâu cũng có thể truy nhập các thứ public (lớp, biến thực thể, biến lớp, phương thức, hàm khởi tạo...)

public và private là hai mức được sử dụng nhiều nhất. Mức public thường dùng cho các lớp, hằng (biến static final, xem chi tiết tại Mục 10.6), các phương thức dành cho mục đích tương tác với bên ngoài (ví dụ các phương thức get và set), và hầu hết các hàm khởi tạo. private được dùng cho hầu hết các biến thực thể và cho các phương thức mà ta không muốn được gọi từ bên ngoài lớp (các phương thức dành riêng cho các phương thức public của lớp đó sử dụng).

Mức mặc định được dùng để giới hạn phạm vi trong một gói (xem thêm về gói tại Phụ lục B). Người ta dùng giới hạn này vì gói được thiết kế là một nhóm các lớp cộng tác với nhau như là một tập hợp gắn bó với nhau. Trong khi tất cả các lớp bên trong cùng một gói thường cần truy nhập lẫn nhau, chỉ có một nhóm trong số đó cần phải để lộ ra ngoài gói, nhóm này sẽ dùng các mức public hay protected một cách thích hợp. Lưu ý rằng nếu lớp có mức protected, thì các phương thức bên trong nó dù có thuộc mức public thì bên ngoài cũng không thể 'nhìn thấy', do không thể nhìn thấy lớp chứa các phương thức đó.

Mức protected gần như giống hệt với mức mặc định, chỉ khác ở chỗ: nó cho phép các lớp con thừa kế các thứ protected của lớp cha, kể cả khi lớp con nằm ngoài gói chứa lớp cha. Như vậy, mức này chỉ áp dụng cho quan hệ thừa kế. Nếu một lớp con nằm ngoài gói có một tham chiếu tới một đối tượng thuộc lớp cha, và giả sử lớp cha này có một phương thức protected, lớp con cũng không thể gọi phương thức đó từ tham chiếu đó. Cách duy nhất để một lớp con có khả năng truy nhập một phương thức protected là thừa kế phương thức đó. Nói cách khác, lớp con ngoài gói không thể *truy nhập* phương thức protected, nó chỉ *sở hữu* phương thức đó qua quan hệ thừa kế.

#### **Những điểm quan trọng:**

- Lớp con chuyên biệt hóa lớp cha của nó.
- Lớp con thừa kế tất cả các biến thực thể và phương thức public của lớp cha, nhưng không thừa kế các biến thực thể và phương thức private của lớp cha.
- Có thể cài đặt các phương thức được thừa kế; không thể cài đặt các biến thực thể được thừa kế (tuy có thể gán trị lại tại lớp con, nhưng đây là hai việc khác nhau)

- Dùng thử nghiệm IS-A để kiểm tra xem cấu trúc thừa kế của ta có hợp lý hay không. Nếu X là lớp con của Y thì khẳng định X IS-A Y phải hợp lý.
- Quan hệ IS-A chỉ có một chiều. Con sói nào cũng là động vật, nhưng không phải con vật nào cũng là chó sói.
- Khi một phương thức được cài đặt tại một lớp con, và phương thức đó được kích hoạt cho một đối tượng của lớp đó, thì phiên bản tại lớp con sẽ được chạy (cái gì ở thấp nhất thì được gọi).
- Nếu lớp B là lớp con của A, lớp C là lớp con của B, thì mỗi đối tượng B thuộc loại A, mỗi đối tượng C thuộc loại B, và mỗi đối tượng C cũng thuộc loại A. (quan hệ IS-A)
- Để gọi phiên bản phương thức của lớp cha từ trong lớp con, sử dụng từ khóa super làm tham chiếu tới lớp cha.

## Bài tập

- Điền từ thích hợp vào các chỗ trống dưới đây
  - Các thành viên có mức truy nhập \_\_\_\_\_ của lớp cha có thể được truy nhập từ trong lớp cha và lớp con.
  - Trong quan hệ \_\_\_\_\_, một đối tượng của một lớp con có thể được đối xử như một đối tượng thuộc lớp cha.
  - Trong quan hệ \_\_\_\_\_ giữa hai lớp, đối tượng của một lớp này có biến thực thể là tham chiếu tới đối tượng thuộc lớp kia.
- Các phát biểu sau đây đúng hay sai:
  - Quan hệ HAS-A được cài đặt bằng cơ chế thừa kế.
  - Lớp Ô tô có quan hệ IS-A đối với các lớp Bánh lái và Phan.
  - Khi lớp con định nghĩa lại một phương thức của lớp cha trong khi giữ nguyên danh sách tham số của phương thức đó, lớp con được gọi là đã cài chồng phương thức của lớp cha.
  - Có thể đối xử với các đối tượng lớp cha và các đối tượng lớp con như nhau.
- Hoàn chỉnh cài đặt sau để có kết quả hiển thị như trong hình

```
public class MonsterTestDrive {
    public static void main(String[] args) {
        Monster[] ma = new Monster[3];
        ma[0] = new Vampire();
        ma[1] = new Dragon();
        ma[2] = new Monster();
        for (int i = 0; i < 3; i++) {
            ma[i].frighten(x);
        }
    }
}

class Monster {
    // insert code here
}

class Vampire extends Monster {
    // insert code here
}

class Dragon extends Monster {
    boolean frighten(int degree) {
        System.out.println("breath fire");
        return true;
    }
}
```

```
% java MonsterTestDrive
a bite?
breath fire
arrrgh
```



4. Cho chương trình sau với một ô trống.

```
class A {
    int iVar = 7;
    void m1() {
        System.out.print("A's m1, ");
    }
    void m2() {
        System.out.print("A's m2, ");
    }
    void m3() {
        System.out.print("A's m3, ");
    }
}

class B extends A {
    void m1() {
        System.out.print("B's m1, ");
    }
}

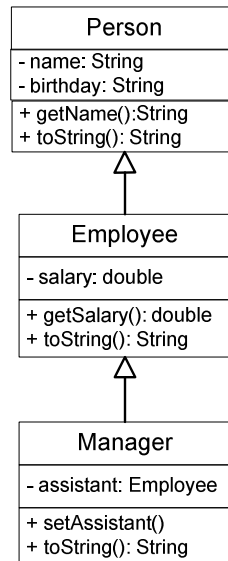
class C extends B {
    void m3() {
        System.out.print("C's m3, " + (ivar + 1));
    }
}

public class MixedUp {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        A a2 = new C();
        
    }
}
```

Nếu điền vào ô đó các lệnh ở dưới đây thì kết quả của chương trình là gì?

- a) b.m1(); c.m2(); a.m3();
- b) c.m1(); c.m2(); c.m3();
- c) a.m1(); b.m2(); c.m3();
- d) a2.m1(); a2.m2(); a2.m3();

5. Viết các lớp Person, Employee, Manager như thiết kế trong sơ đồ sau. Bổ sung các phương thức thích hợp nếu thấy cần. Định nghĩa lại các phương thức toString() cho phù hợp với dữ liệu tại mỗi lớp.



Viết lớp PeopleTest để chạy thử các lớp trên: tạo một vài đối tượng và in thông tin của chúng ra màn hình. Trong hàm main của lớp PeopleTest, tạo một mảng kiểu Person, gán ba đối tượng ở trên vào mảng, rồi dùng vòng lặp để in ra thông tin về các đối tượng trong mảng.

Đọc Phụ lục B. Tách các lớp Person, Employee vào trong gói peoples. Đặt Manager và PeopleTest ở gói mặc định (nằm ngoài gói peoples). Chỉnh lại các khai báo quyền truy nhập tại các lớp để chương trình viết ở trên lại chạy được.

Đại học Công nghệ - ĐHQGHN

6. Viết các lớp Account, NormalAccount, NickelNDime, Gambler về các loại tài khoản ngân hàng theo mô tả sau: Thông tin về mỗi tài khoản ngân hàng gồm có số dư hiện tại (int balance), số giao dịch đã thực hiện kể từ đầu tháng (int transactions). Mỗi tài khoản cần đáp ứng các thao tác sau:

- a) Một hàm khởi tạo cho phép mở một tài khoản mới với một số dư ban đầu cho trước;
- b) Các phương thức boolean deposit(int) cho phép gửi tiền vào tài khoản, boolean withdraw(int) cho phép rút tiền từ tài khoản. Các phương thức này trả về true nếu giao dịch thành công, nếu không thì trả về false, tương tự cập nhật số đếm giao dịch.
- c) Phương thức void endMonth() thực hiện tất toán, sẽ được mô đun quản lý tài khoản (nằm ngoài phạm vi bài này) gọi định kỳ vào các thời điểm cuối tháng. Phương thức này tính phí hàng tháng nếu có bằng cách gọi phương thức int endMonthCharge(), trừ phí, in thông tin tài khoản (số dư, số giao dịch, phí), và đặt lại số giao dịch về 0 để sẵn sàng cho tháng sau.
- d) phương thức endMonthCharge() trả về phí tài khoản trong tháng vừa qua.

Phí tài khoản được tính tùy theo từng loại tài khoản. Loại NormalAccount tính phí hàng tháng là 10.000 đồng. Loại NickelNDime tính phí theo số lần rút tiền, phí cho mỗi lần rút là 2000 đồng, cuối tháng mới thu. Loại Gambler không tính phí cuối tháng nhưng thu phí tại từng lần rút tiền theo xác suất như sau: Với xác suất 49%, tài khoản không bị hụt đi đồng nào và giao dịch thành công miễn phí. Với xác suất 51%, phí rút tiền bằng đúng số tiền rút được.

Account là lớp cha của NormalAccount, NickelNDime, và Gambler. Cần thiết kế sao cho tái sử dụng và tránh lặp code được càng nhiều càng tốt.

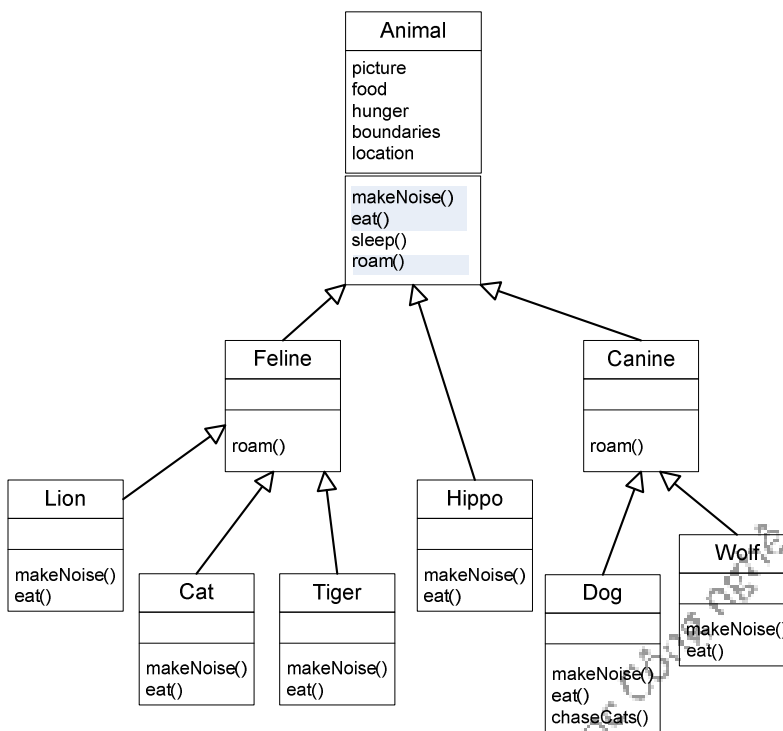
Đại học Công nghệ - HHOUGHN

## Chương 8. LỚP TRỪ TƯỢNG VÀ INTERFACE

Thừa kế mới chỉ là khởi đầu. Để khai thác cơ chế đa hình, các ngôn ngữ lập trình hướng đối tượng cung cấp các cơ chế **kiểu trừ tượng** (*abstract type*). Các kiểu trừ tượng có cài đặt không đầy đủ hoặc không có cài đặt. Nhiệm vụ chính của chúng là giữ vai trò kiểu tổng quát hơn của một số các kiểu khác. Kiểu trừ tượng không hề có cài đặt là các interface (không phải khái niệm giao diện đồ họa người dùng GUI). Kiểu trừ tượng có cài đặt một phần là các lớp trừ tượng. Chúng mang lại sự linh hoạt và khả năng mở rộng cho thiết kế hướng đối tượng. Ví dụ cuối chương trước về lớp Vet có thể hoạt động với loại Animal bất kì đã chạm vào bề mặt của vấn đề. Ta sẽ bàn về các kiểu trừ tượng trong chương này.

### 8.1. MỘT SỐ LỚP KHÔNG NÊN TẠO THỰC THỂ

Nhớ lại thiết kế cây phả hệ các loài động vật mà ta đã làm trong chương trước. Đó là giải pháp không tồi. Ta đã thiết kế sao cho các đoạn mã bị trùng lặp là tối thiểu, và ta đã cài đặt những phương thức mà ta cho là nên có cài đặt cụ thể cho các lớp con.



Đó là giải pháp tốt nếu nhìn từ góc độ đa hình, bởi vì ta có thể thiết kế các chương trình dùng Animal với các đối số kiểu Animal (kể cả khai báo mảng Animal), sao cho kiểu Animal bất kì - *kể cả những kiểu ta chưa bao giờ nghĩ tới* - có

thể được truyền vào và sử dụng tại thời gian chạy. Ta đã đặt vào Animal giao thức chung cho tất cả các loại Animal (bốn phương thức mà ta tuyên bố rằng loại Animal nào cũng có), và ta sẵn sàng xây dựng các đối tượng mới loại Lion, Tiger và Hippo.

Từ ví dụ của các chương trước, ta đã quen thuộc với việc tạo và dùng đối tượng Dog, Cat, Wolf, việc tạo đối tượng mới kiểu Lion hay Tiger cũng không có gì đặc biệt. Nhưng nếu ta tạo một đối tượng Animal thì sao? Một con động vật chung chung trông nó như thế nào? Nó có hình gì? màu gì? to cỡ nào? có mấy chi? mấy mắt? Đối tượng Animal chứa các giá trị gì tại các biến thực thể? Ta dùng một đối tượng Animal cho việc gì nếu không thể trả lời các câu hỏi trên?

Tuy nhiên, ta lại cần một lớp Animal cho cơ chế thừa kế và đa hình. Và ta muốn rằng các lập trình viên chỉ tạo các đối tượng thuộc các lớp con ít trừu tượng hơn của Animal, chứ không bao giờ tạo đối tượng của chính lớp Animal. Ta muốn các đối tượng Tiger, Lion, Dog, Cat, ta không muốn các đối tượng Animal.

Ta lấy một ví dụ khác. Một thư viện đồ họa cho phép vẽ (draw), xóa (erase), di chuyển (move) các hình đồ họa. Trong đó thư viện có các lớp Circle (hình tròn), Rectangle (hình chữ nhật)... và để có thể tận dụng quan hệ thừa kế và khi cần có thể xử lý đồng loạt các thành phần của một bản vẽ chẳng hạn, thư viện có thêm lớp tổng quát Shape (hình) là lớp cha chung của các hình đồ họa đó. Liệu có khi nào ta cần tạo một đối tượng thuộc lớp Shape? Nó có hình dạng như thế nào? Làm thế nào để vẽ/xóa nó? Ta viết nội dung gì cho các phương thức draw và erase của lớp Shape? Chẳng lẽ để trống hoặc thông báo gì đó? Lỡ có ai tạo một đối tượng Shape rồi gọi phương thức mà đáng ra nó không nên làm gì?

Một lớp cha không bao giờ được dùng để tạo đối tượng được gọi là **lớp cơ sở trừu tượng**, hay ngắn gọn là **lớp trừu tượng** (*abstract class*). Với những lớp thuộc diện này, trình biên dịch sẽ báo lỗi bất cứ đoạn mã nào định tạo thực thể của lớp đó. Tất nhiên, ta vẫn có thể dùng tham chiếu thuộc kiểu lớp trừu tượng. Thực ra đây là mục đích quan trọng nhất của việc sử dụng lớp trừu tượng - để có đa hình cho đối số, kiểu trả về, và mảng. Bên cạnh đó là mục đích sử dụng lớp trừu tượng làm nơi đặt các phương thức dùng chung để các lớp con thừa kế.

Khi ta thiết kế cấu trúc thừa kế, ta cần quyết định lớp nào *trừu tượng*, lớp nào *cụ thể*. Các lớp cụ thể (*concrete*) là các lớp đủ đặc trưng để có thể tạo thực thể. Trong phạm vi lập trình, một lớp cụ thể có nghĩa đơn giản là: ta được phép tạo đối tượng thuộc loại đó.

Các lớp ta vẫn thấy trong các ví dụ từ đầu cuốn sách này đều là các lớp được khai báo là lớp cụ thể. Để quy định một lớp là trừu tượng, ta đặt từ khóa `abstract` vào đầu khai báo lớp. Ví dụ:

```
abstract class Canine extends Animal {
    public void roam() { }
}
```

Kết quả là trình biên dịch sẽ không cho phép ta tạo thực thể của lớp đó nữa.

```

public class CanineTestDrive {
    public static void main(String [] args) {
        Canine c;
        c = new Dog(); ← ok, có thể dùng tham chiếu kiểu trừu tượng
        c = new Canine(); ← trình biên dịch sẽ báo lỗi,
                          lớp Canine trừu tượng nên không thể
                          tạo đối tượng Canine
        c.roam();
    }
}

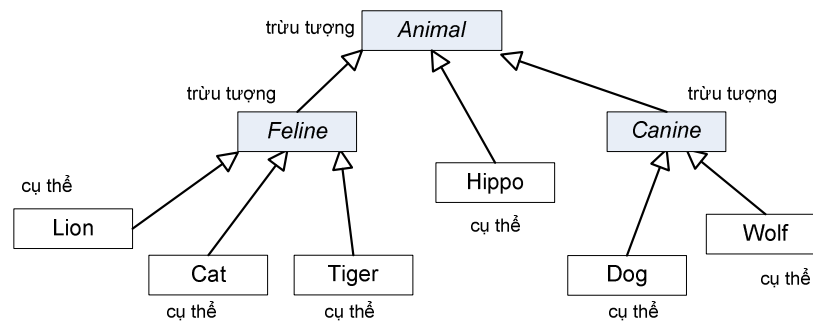
```

Một lớp trừu tượng gần như<sup>8</sup> vô dụng, vô giá trị, trừ khi nó có lớp con.

## 8.2. LỚP TRỪU TƯỢNG VÀ LỚP CỤ THỂ

Một lớp không phải là lớp trừu tượng thì nó là lớp cụ thể

Trong cây phả hệ Animal, nếu ta cho Animal, Feline, và Canine là các lớp trừu tượng, thì còn lại sẽ là các lớp cụ thể.



Xem qua bộ thư viện chuẩn của Java, ta sẽ thấy có rất nhiều lớp trừu tượng, đặc biệt trong thư viện giao diện đồ họa GUI. Một thành phần giao diện đồ họa chung chung (GUI Component) có hình dạng như thế nào? Lớp Component là lớp cha của các lớp liên quan đến giao diện đồ họa cho những thứ như nút bấm, cửa sổ soạn thảo, thanh cuộn, hộp hội thoại, v.v.. Ta không muốn tạo một đối tượng Component tổng quát và đặt nó vào màn hình, ta muốn tạo những thứ chẳng hạn như JButton để làm một nút bấm. Nói cách khác, ta chỉ tạo thực thể từ các lớp con cụ thể của Component nhưng không bao giờ từ chính Component.

Vậy khi nào một lớp nên là lớp trừu tượng, khi nào thì nên là lớp cụ thể? *Bút* chắc là lớp trừu tượng. *Bút bi* và *Bút máy* có lẽ cũng nên là các lớp trừu tượng. Vậy đến khi nào thì các lớp trở thành lớp cụ thể? *Bút máy Parker* liệu có thành lớp cụ thể hay vẫn là lớp trừu tượng? Có vẻ như *Bút máy Hồng Hà nét hoa 2008* chắc chắn là lớp cụ thể. Nhưng làm thế nào để chắc chắn?

<sup>8</sup> Có một ngoại lệ: một lớp trừu tượng có thể có các thành viên static hữu dụng (xem Ch-ương 10)

### 8.3. PHƯƠNG THỨC TRỪ TƯỢNG

Không chỉ lớp, ta còn có thể khai báo các phương thức trừ tượng. Một lớp trừ tượng có nghĩa phải tạo lớp con cho nó; còn một phương thức trừ tượng có nghĩa rằng nó phải được cài đặt.

Ta có thể quy định rằng một vài (hoặc tất cả) các hành vi của một lớp trừ tượng phải được cài đặt bởi một lớp con có tính đặc trưng hơn, nếu không các hành vi đó là vô nghĩa. Nói cách khác, ta không thể nghĩ ra một cài đặt tổng quát nào cho phương thức đó mà có thể hữu ích cho các lớp con. Một phương thức `makeNoise()` tổng quát sẽ làm gì?

Cú pháp Java quy định rằng phương thức trừ tượng không có thân phương thức. Dòng khai báo phương thức kết thúc tại dấu chấm phẩy và không có cặp ngoặc `{}`.

```
public abstract void makeNoise();
```

Nếu ta khai báo một phương thức là `abstract`, ta phải đánh dấu lớp đó cũng là `abstract`. Ta không thể đặt một phương thức trừ tượng ở bên trong một lớp cụ thể. Tuy nhiên, ta có thể có phương thức không trừ tượng bên trong một lớp trừ tượng.

Các phương thức trừ tượng phải được cài đặt tại một lớp con. Các phương thức trừ tượng không có nội dung, nó tồn tại chỉ để phục vụ cơ chế đa hình. Điều đó có nghĩa rằng lớp cụ thể đầu tiên nằm dưới nó trên cây phả hệ bắt buộc phải cài tất cả các phương thức trừ tượng; các lớp con trừ tượng có thể bỏ qua việc này.

Ví dụ, nếu cả `Animal` và `Canine` đều trừ tượng và cùng có các phương thức trừ tượng, lớp `Canine` không buộc phải cài các phương thức trừ tượng của `Animal`. Nhưng ngay khi ta đi xuống đến lớp con cụ thể đầu tiên, chẳng hạn `Dog`, lớp đó sẽ phải cài *tất cả* các phương thức trừ tượng thừa kế từ `Animal` và `Canine`.

Tuy nhiên, nhớ lại rằng một lớp trừ tượng có thể chứa cả các phương thức trừ tượng cũng như cụ thể, cho nên `Canine` chẳng hạn có thể cài một phương thức trừ tượng thừa kế từ `Animal`, dẫn tới `Dog` không phải làm việc này nữa. Còn nếu `Canine` không cài phương thức trừ tượng nào từ `Animal`, `Dog` sẽ phải cài tất cả các phương thức trừ tượng của `Animal` cũng như những phương thức trừ tượng mà `Canine` bổ sung. Khi ta nói "cài đặt phương thức trừ tượng", điều đó có nghĩa ta cài đặt phương thức đó với một thân hàm để có một phiên bản cụ thể của phương thức đó (tất nhiên ở phiên bản mới không có từ khóa `abstract` trong khai báo).

### 8.4. VÍ DỤ VỀ ĐA HÌNH

Giả sử ta muốn viết một lớp danh sách để quản lý các đối tượng `Dog` mà không dùng đến các cấu trúc danh sách có sẵn trong thư viện Java. Bước đầu, ta chỉ cần

một phương thức add() để đưa các đối tượng Dog vào danh sách. Ta dùng một mảng Dog đơn giản với kích thước 5 để lưu các đối tượng Dog được đưa vào danh sách. Khi trong danh sách đã đủ 5 đối tượng, ta vẫn có thể tiếp tục gọi phương thức add() nhưng nó sẽ không làm gì. Nếu chưa đủ 5, phương thức add() sẽ gắn đối tượng tiếp theo vào vị trí tiếp theo còn trống rồi tăng chỉ số của vị trí tiếp theo còn trống (nextIndex) thêm 1.

---

```
public class MyDogList {  
    private Dog[] dogs = new Dog[5];  
    private int nextIndex = 0;  
  
    public void add(Dog d) {  
        if (nextIndex < dogs.length) {  
            dogs[nextIndex] = d;  
            System.out.print("Dog added at " + nextIndex);  
            nextIndex++;  
        }  
    }  
}
```

MyDogList
Dog[] dogs int nextIndex
add(Dog d)

---

Nhưng nếu ta còn muốn quản lí cả mèo lẫn chó trong danh sách? Có một vài lựa chọn. Thứ nhất: viết thêm lớp MyCatList dành riêng cho các đối tượng Cat. Thứ hai: viết một lớp DogAndCatList chung, trong đó có hai mảng, một dành cho các đối tượng Dog, một dành cho các đối tượng Cat. Thứ ba: viết một lớp AnimalList trong đó có thể chấp nhận các đối tượng thuộc lớp con bất kì của Animal (phòng trường hợp đặc tả lại thay đổi để yêu cầu nhận thêm các loài vật khác). Lựa chọn thứ ba gọn gàng và có khả năng mở rộng cao nhất nên ta sẽ dùng cho phiên bản thứ hai. Ta sẽ sửa lớp MyDogList, tổng quát hóa nó để chấp nhận các lớp con bất kì của Animal thay vì chỉ Dog. Lô-gic chương trình vẫn giữ nguyên như cũ, chỉ có các thay đổi được đánh đậm trong đoạn mã dưới đây:

Đại học Công nghệ - HHOUGHVN



```

public class AnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.print("Animal added at " + nextIndex);
            nextIndex++;
        }
    }
}

```

AnimalList	
Animal[] animals	int nextIndex
add(Animal a)	

```

public class AnimalTestDrive {
    public static void main(String [] args) {
        AnimalList list = new AnimalList();
        d = new Dog();
        c = new Cat();
        list.add(d);
        list.add(c);
    }
}

```

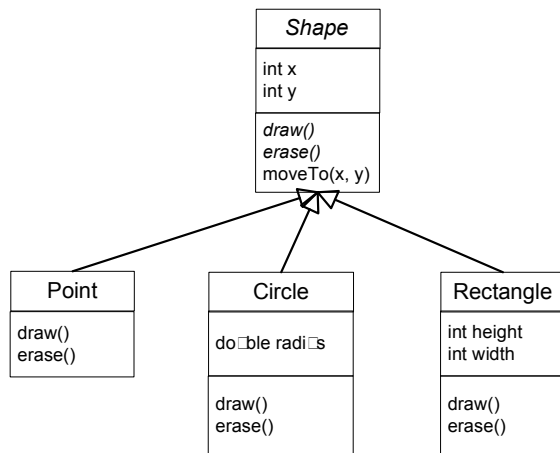
```

% java AnimalTestDrive
Animal added at 0
Animal added at 1

```

Hình 8.1: Ví dụ đa hình với các lớp Animal.

Ta lại lấy ví dụ Shape đã nói đến ở đầu chương. Lớp cha tổng quát Shape nên là lớp trừu tượng do ứng dụng không cần và không nên tạo đối tượng Shape. Ngoài ra, các phương thức draw và erase của lớp này cũng nên là phương thức trừu tượng do ta không thể nghĩ ra nội dung gì hữu ích cho chúng. Các lớp con cụ thể, Point, Circle, Rectangle, và các lớp mà sau này sẽ bổ sung vào thư viện khi cần, sẽ định nghĩa các phiên bản với nội dung riêng cụ thể phù hợp với chính mình. Chẳng hạn như ví dụ trong Hình 8.2.



Đại học Công nghệ - ĐHQGHN

```

abstract public class Shape {
    protected int x, y;
    protected Shape (int _x, int _y) { x = _x; y = _y; }

    abstract public void draw();
    abstract public void erase();

    public void moveTo(int _x, int _y) {
        erase();
        x = _x;
        y = _y;
        draw();
    }
}

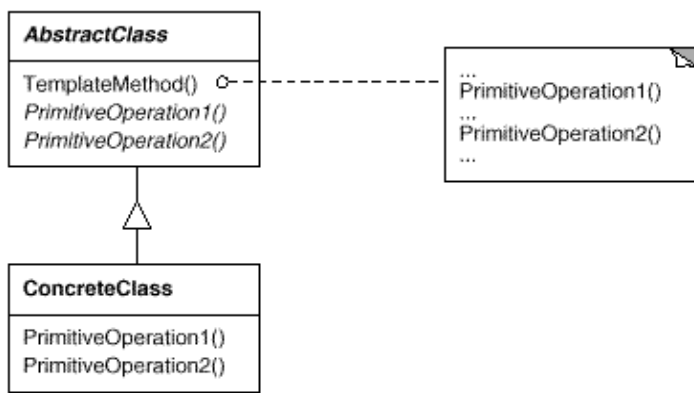
public class Circle extends Shape {
    private double radius;
    public Circle(int _x, int _y, double _r) {
        super(_x, _y);
        radius = _r;
    }
    public void draw() {
        System.out.println("Draw circle");
    }
    public void erase() {
        System.out.println("Erase circle");
    }
}

```

Hình 8.2: Ví dụ đa hình với các lớp Shape.

Khác với draw và erase, moveTo lại là phương thức có thể định nghĩa ngay tại lớp Shape. Thuật toán ba bước cho moveTo là như nhau cho mọi hình: (1) xóa tại vị trí hiện hành, (2) sửa tọa độ hình, (3) vẽ tại vị trí mới, mặc dù xóa như thế nào và vẽ như thế nào là tùy theo từng loại hình cụ thể. Hiệu ứng đa hình cho phép moveTo dùng đến các phiên bản draw và erase khác nhau tùy theo nó được gọi cho đối tượng thuộc loại hình nào. Khi thư viện được bổ sung thêm các lớp đặc tả các loại hình khác, ta chỉ phải cài draw và erase cho loại hình đó mà không phải làm thêm gì cho các phương thức biến đổi hình có quy trình chung đã được định nghĩa sẵn tương tự như moveTo.

Ví dụ này cũng minh họa một mẫu thiết kế có tên **Template Method** (*phương thức khuôn mẫu*). Xem Hình 8.3. Ở đây, Shape là lớp trừu tượng (AbstractClass) định nghĩa một phương thức khuôn mẫu moveTo, và quy định hai thao tác cơ bản (PrimitiveOperation) là erase và draw mà phương thức khuôn mẫu dùng đến. Circle là lớp con cụ thể (ConcreteClass), nó cài đặt các thao tác cơ bản này. Đây là một trong những mẫu thiết kế thông dụng nhất.



Hình 8.3: Mẫu thiết kế Template Method.

## 8.5. LỚP Object

Thêm một bước nữa, nếu ta muốn có danh sách lưu được cả những đối tượng không phải động vật thì sao? Ta có thể tiếp tục thay đổi theo kiểu sửa kiểu mảng, kiểu đối số phương thức add() thành cái gì đó tổng quát hơn và trừu tượng hơn Animal? Nhưng ta không viết lớp cha cho Animal.

Thực ra Animal đã có lớp cha. Đối với Java, tất cả các lớp đều là lớp con của lớp Object. Object là tổ tiên của tất cả. Ngay từ đầu, ta đã viết các lớp con của Object mà không biết, ta viết lớp con của Object mà không cần phải khai báo quan hệ thừa kế đó bằng từ khóa extends.

Bất kì lớp nào không được khai báo tường minh là lớp con của một lớp khác thì đều được khai báo ẩn là lớp con của Object. Vậy nên, ta có Dog không phải là lớp con trực tiếp của Object, còn Animal là lớp con trực tiếp của Object, và tất cả Dog, Cat, Canine, Animal... đều nằm trong cây phả hệ có gốc là Object.

Với tất cả các lớp đều nằm trong cây thừa kế có Object tại gốc, cơ chế đa hình cho phép ta tạo các cấu trúc dữ liệu dành cho đối tượng thuộc tất cả các lớp. Chẳng hạn một mảng kiểu Object có thể lưu đối tượng thuộc đủ loại Animal, Cow, Dog, Cat, PhoneBook, String... Trong thư viện chuẩn của Java có lớp ArrayList được định nghĩa để quản lý các đối tượng thuộc kiểu Object. ArrayList có thể dùng để quản lý đối tượng thuộc tất cả các kiểu.

Lớp Object cho các lớp khác thừa kế những gì? Trong các phương thức được thừa kế của Object có bốn phương thức thông dụng:

- **boolean equals(Object o)** kiểm tra xem hai đối tượng hiện hành có 'bằng nhau' hay không, xem thêm về ý nghĩa của khái niệm 'bằng nhau' này tại Chương 13.

```

Cow c1 = new Cow();
Cow c2 = new Cow();
System.out.println(c1.equals(c2));
  
```

```

% java TestObject
false
  
```

- **Class getClass()** trả về lớp mà đối tượng hiện hành đã được tạo từ đó,

```
d = new Dog();
System.out.println(d.getClass());
```

```
% java TestObject
class Dog
```

- **int hashCode()** trả về mã băm của đối tượng hiện hành, ta tạm thời xem mã này như là một định danh của đối tượng, và

```
c = new Cat();
System.out.println(c.hashCode());
```

```
% java TestObject
80202111
```

- **String toString()** trả về biểu diễn dạng String của đối tượng, ta thường cài đặt phương thức này để trả về biểu diễn String theo ý muốn của ta thay vì trả về chuỗi kí tự được kết xuất một cách tổng quát như ví dụ bên dưới.

```
c = new Cat();
System.out.println(c.toString());
```

```
% java TestObject
Cat@7d277F
```

## 8.6. ĐỐI KIỂU – KHI ĐỐI TƯỢNG MẤT HÀNH VI CỦA MÌNH

Rắc rối của việc dùng cơ chế đa hình coi mọi thứ như là một Object hay coi các đối tượng động vật như là một Animal là đôi khi các đối tượng có vẻ như đánh mất (tạm thời) các đặc trưng của mình. Dog có vẻ mất các đặc điểm của chó. Ta hãy xem chuyện gì xảy ra khi một phương thức trả về một tham chiếu tới một đối tượng Dog nhưng khai báo kiểu trả về là Animal.

Nhớ lại lớp AnimalList ta đã tạo để quản lý danh sách các con vật. Giả sử AnimalList đã có thêm phương thức get(int index) trả về tham chiếu tới đối tượng đứng tại vị trí index trong danh sách.

```
public class AnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public Animal get(int index) {
        return animals[index];
    }

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.print("Animal added at " + nextIndex);
            nextIndex++;
        }
    }
}
```

```
AnimalList
```

```
Animal[] animals
int nextIndex
```

```
add(Animal a)
get(int index)
```

Ta thử nghiệm bằng chương trình DogTestDrive, trong đó một đối tượng Dog được tạo và đưa vào một danh sách AnimalList. Sau đó ta gọi phương thức get() của danh sách đó để lấy lại chính đối tượng vừa đưa vào.

```

public class DogTestDrive {
    public static void main(String [] args) {
        AnimalList list = new AnimalList();
        Dog d = new Dog();
        list.add(d);
        d = list.get(0);
    }
}

```

← lỗi biên dịch!

```

% javac DogTestDrive.java
DogTestDrive.java:6: incompatible types
found   : Animal
required: Dog
    d = list.get(0);
           ^
1 error

```

Để ý rằng phương thức `get()` gọi từ `list` trả về một tham chiếu tới chính đối tượng `Dog` nói trên, nhưng dưới dạng một tham chiếu kiểu `Animal`. Việc này hoàn toàn hợp lệ. Nhưng trình biên dịch không biết rằng thứ được trả về từ đó thực chất đang chiếu tới một đối tượng `Dog`, cho nên nó không cho phép ta gán giá trị trả về đó cho một tham chiếu kiểu `Dog`.

Nếu ta gán giá trị đó cho một tham số kiểu `Animal`, chẳng hạn, `Animal a = list.get(0)`, thì trình biên dịch sẽ không phàn nàn gì. Tuy nhiên, khi đó ta sẽ chỉ có thể gọi các phương thức mà `Dog` thừa kế từ `Animal`, chẳng hạn `roam()`, chứ không thể gọi phương thức mà chỉ `Dog` mới có, như `chaseCats()` chẳng hạn.

```

public class DogTestDrive {
    public static void main(String [] args) {
        AnimalList list = new AnimalList();
        Dog d = new Dog();
        list.add(d);

        Animal a = list.get(0);
        a.roam();
        a.chaseCats();
    }
}

```

← lỗi biên dịch! Animal không có phương thức chaseCats()!

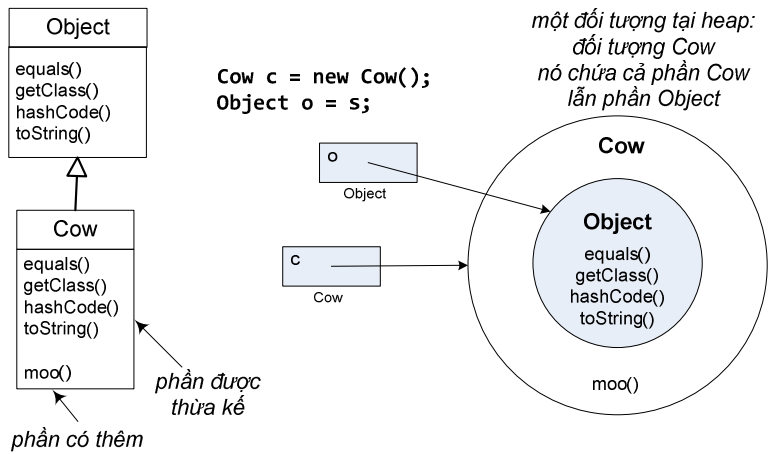
Ngay cả khi ta biết chắc chắn đối tượng có hành vi `chaseCats` (nó thực sự là một đối tượng `Dog`!), trình biên dịch chỉ nhìn thấy nó như là một thứ kiểu `Animal`, mà `Animal` thì không có `chaseCats()`.

Vấn đề ở đây giống như ta đã nói đến ở Mục 7.9. Để xác định xem ta có thể gọi một phương thức nào đó hay không, trình biên dịch dựa trên kiểu *tham chiếu* chứ không dựa trên kiểu *đối tượng* thực tế.

Vậy cơ chế thừa kế có bản chất như thế nào?

Mỗi đối tượng chứa tất cả những gì nó thừa kế từ tất cả các lớp cha, ông, tổ tiên của nó, trong đó có cả lớp `Object`. Vậy nên nó có thể được coi là một thực thể của mỗi lớp cha ông đó. Lấy ví dụ lớp `Cow` đơn giản. Một đối tượng `Cow` có thể được đối xử không chỉ như một đối tượng `Cow`, nó còn có thể được xem như một `Object`. Khi ta gọi `new Cow()`, ta được một đối tượng tại heap – một đối tượng `Cow` – nhưng

đối tượng đó có một cái lõi là phần Object (chữ cái O viết hoa) của nó. Một tham chiếu kiểu Cow tới đối tượng này có thể 'nhìn thấy' toàn bộ đối tượng Cow, do đó có thể truy nhập toàn bộ các phương thức của Cow, bao gồm cả các phương thức được thừa kế. Trong khi đó, một tham chiếu kiểu Object chỉ tới cùng một đối tượng chỉ có thể 'nhìn thấy' phần Object của đối tượng đó, do đó chỉ có thể truy cập phần đó.



Hình 8.4: Cấu trúc lớp con và phần được thừa kế.

Như vậy ta đã giải thích được tại sao khi dùng một tham chiếu kiểu lớp cha cho đối tượng thuộc lớp con thì lớp con có vẻ như mất bản sắc riêng.

Nhưng ta vẫn chưa giải quyết xong vấn đề của chương trình DogTestDrive. Đối tượng mà ta lấy ra từ danh sách list thực sự là Dog, vậy làm cách nào để gọi được phương thức của Dog? Ta phải dùng một tham chiếu được khai báo kiểu Dog. Sao chép tham chiếu kiểu Animal mà ta đang có và ép sang kiểu Dog để ghi vào một tham chiếu kiểu Dog. Sau đó, ta có thể dùng tham chiếu Dog để gọi phương thức của Dog như bình thường.

```
public class DogTestDrive {
    public static void main(String [] args) {
        AnimalList list = new AnimalList();
        Dog d = new Dog();
        list.add(d);

        Animal a = list.get(0);
        Dog d2 = (Dog) a;
        d2.roam();
        d2.chaseCats();
    }
}
```

*ép kiểu từ Object trở về kiểu Dog*

Nếu hành động ép kiểu của ta là sai, nghĩa là đối tượng đang quan tâm thực ra không phải kiểu Dog, thì khi chạy, chương trình của ta sẽ bị ngắt giữa chừng do lỗi run-time ClassCastException. Do đó, trong những trường hợp mà ta không chắc chắn về kiểu của đối tượng, ta có thể dùng toán tử instanceof để kiểm tra.

```
if (o instanceof Dog) {
    Dog d = (Dog) o;
```

}

## 8.7. ĐA THỪA KẾ VÀ VẤN ĐỀ HÌNH THOI

Cây thừa kế động vật vốn được thiết kế để dùng cho bài toán giả lập môi trường sống của động vật. Nếu cần xây dựng phần mềm dạy học cho môn động vật học, ta sẽ tái sử dụng được các lớp trong cây thừa kế đó. Giả sử bây giờ ta mới nhận được yêu cầu xây dựng phần mềm PetShop cho cửa hàng thú cảnh, và ta muốn dùng lớp Dog cho phần mềm mới. Hiện tại các lớp động vật *chưa có các hành vi của thú cảnh* (Pet) như *play()* và *beFriendly()*. Với vai trò lập trình viên cho lớp Dog, ta sẽ làm gì? Chỉ việc *thêm* những phương thức cần thiết? Làm vậy ta sẽ không phá vỡ mã của bất kì ai khác vì ta không động đến các phương thức đã *có sẵn* mà mã của người khác có thể gọi cho các đối tượng Dog.

Đúng nhưng chưa đủ. Lưu ý rằng đây là phần mềm cho cửa hàng thú cảnh, ở đó không chỉ có chó, ta sẽ không chỉ cần đến lớp Dog. Việc bổ sung các phương thức mới vào Dog, do đó, có những nhược điểm gì?

Ta lần lượt xét từng phương án:

**Phương án 1:** đặt các hành vi thú cảnh tại lớp Animal.

*Ưu điểm:* Tất cả các lớp động vật lập tức có các hành vi thú cảnh. Ta không phải sửa các lớp khác, và các lớp con sẽ được tạo trong tương lai cũng được thừa kế. Lớp Animal có thể dùng làm kiểu đa hình trong chương trình muốn đối xử đồng loạt các đối tượng Animal như là thú cảnh.

*Nhược điểm:* Hà mã, sư tử, chó sói hầu như chắc chắn không phải thú cảnh nên Hippo, Lion, và Wolf không nên có các hành vi thú cảnh. Kể cả nếu cài đặt các hành vi thú cảnh tại các lớp này để chúng 'không làm gì' thì vẫn không ổn, vì khi đó hợp đồng của các lớp Hippo, Lion,... cho những đối tượng không bao giờ là thú cảnh vẫn có những hành vi của thú cảnh.

Đây là cách tiếp cận tồi. Ta không nên đưa vào lớp Animal những thứ không áp dụng cho tất cả các lớp con của nó.

**Phương án 2:** chỉ đặt các hành vi thú cảnh tại các lớp cần đến nó.

*Ưu điểm:* Không còn rắc rối về chuyện hà mã làm thú cảnh. Dog và Cat có thể cài các phương thức đó và các lớp khác không bị liên lụy.

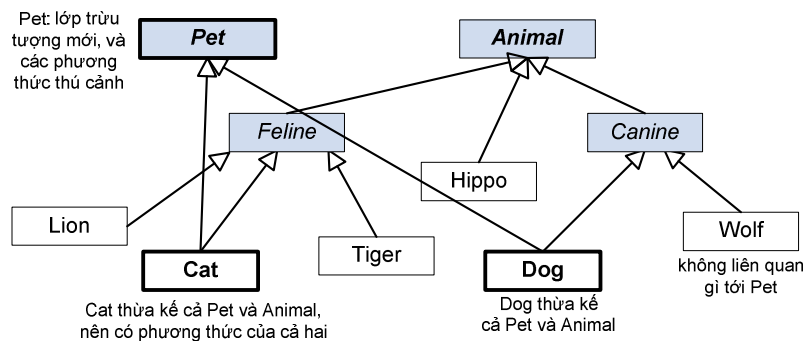
*Nhược điểm:* Hai vấn đề nghiêm trọng: Thứ nhất, phải có giao thức chung mà từ nay trở đi tất cả các lập trình viên cho các lớp Animal phải biết. Giao thức đó bao gồm các phương thức mà ta quyết định rằng tất cả các lớp thú cảnh phải có, tên là gì, trả về kiểu gì, đối số kiểu gì. Nói cách khác là hợp đồng của thú cảnh. Và ta hiện không có cách gì để đảm bảo sẽ không có ai nhầm.

Thứ hai, ta không có đa hình cho các phương thức thú cảnh đó. Không thể dùng tham chiếu Animal cho các phương thức thú cảnh.

Tóm lại, ta cần gì?

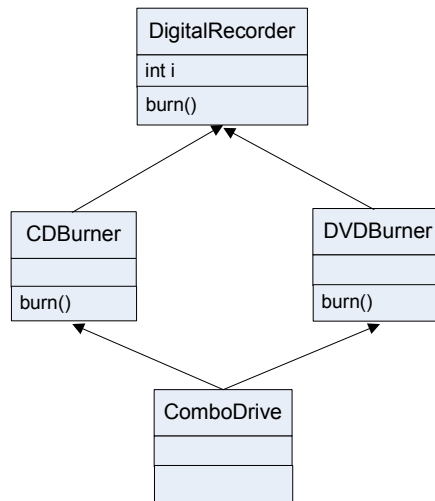
- Đặt hành vi thú cảnh tại các lớp thú cảnh và chỉ tại đó mà thôi.
- Đảm bảo rằng tất cả các lớp thú cảnh hiện có cũng như sẽ được viết sẽ phải có tất cả các phương thức đã được quy định (tên, đối số, kiểu trả về...) mà không phải ngời hy vọng rằng ai đó sẽ làm đúng.
- Tận dụng được lợi thế của đa hình, sao cho có thể gọi được phương thức của tất cả các loại thú cảnh mà không phải dùng riêng các kiểu đối số, kiểu trả về, dùng từng mảng riêng cho từng loại một.

Có vẻ như ta cần đến HAI lớp cha trong cây thừa kế.



Khi lớp con thừa kế từ nhiều hơn một lớp cha, ta có tình trạng được gọi là "đa thừa kế". Hình thức đa thừa kế này có tiềm năng gây ra một rắc rối nghiêm trọng được gọi là **Vấn đề Hình thoi** (*the Diamond problem*) như ví dụ trong Hình 8.5. Trong ví dụ đó, hai lớp DVDBurner (thiết bị ghi đĩa DVD) và CDBurner (thiết bị ghi đĩa CD) cùng là lớp con của DigitalRecorder (đầu thu kỹ thuật số), cả hai cài đặt phương thức burn() và cùng thừa kế biến thành viên i. Giả sử biến i được dùng tại DVDBurner cũng như CDBurner, nhưng với các giá trị khác nhau. Chuyện gì xảy ra nếu ComboDrive – lớp con thừa kế cả hai lớp trên – cần dùng đến cả hai giá trị i đó? Còn nữa, khi gọi phương thức burn() cho một đối tượng ComboDrive, phiên bản burn() nào sẽ được chạy?





Hình 8.5: Ví dụ về vấn đề Hình thoi của đa thừa kế.

Ngôn ngữ lập trình nào cho phép đa thừa kế sẽ phải giải quyết những tình trạng rối rắm trên, sẽ phải có những quy tắc đặc biệt để xử lý những tình huống nhập nhằng ngữ nghĩa có thể xảy ra. C++ là một trong những ngôn ngữ như vậy. Java được thiết kế theo tiêu chí đơn giản, nên nó không cho phép một lớp được thừa kế từ nhiều hơn một lớp cha.

Vậy ta phải giải quyết bài toán thú cảnh như thế nào với Java?

## 8.8. INTERFACE

Giải pháp mà Java cung cấp là **interface**. Thuật ngữ *interface* của tiếng Anh thường được dùng với nghĩa 'giao diện', chẳng hạn như "giao diện người dùng", hay như trong câu "Các phương thức public của một lớp là giao diện của nó đối với bên ngoài". Tuy nhiên, trong mục này, ta nói đến khái niệm interface với ý nghĩa là một cấu trúc lập trình của Java được định nghĩa với từ khóa interface (tương tự như cấu trúc lớp được định nghĩa với từ khóa class).

Cấu trúc interface này cho phép ta giải quyết bài toán đa thừa kế, cho ta hưởng phần lớn các ích lợi mang tính đa hình mà đa thừa kế mang lại, nhưng tránh cho ta các rắc rối nhập nhằng ngữ nghĩa như đã giới thiệu trong mục trước.

Nguy cơ nhập nhằng ngữ nghĩa được tránh bằng cách rất đơn giản: **phương thức nào cũng phải trừu tượng!** Theo đó, lớp con buộc phải cài đặt các phương thức. Nhờ vậy, khi chương trình chạy, máy ảo Java không phải bối rối lựa chọn giữa hai phiên bản mà một đối tượng được thừa kế.

Một interface, do đó, giống như một lớp thuần túy trừu tượng bao gồm toàn các phương thức trừu tượng và không có biến thực thể. Nhưng về cú pháp thì interface có khác lớp trừu tượng một chút. Để định nghĩa một interface, ta dùng từ khóa interface thay vì class như đối với lớp:

```
public interface Pet {...}
```

Đối với một lớp trừu tượng, ta cần tạo lớp con cụ thể. Còn đối với một interface, ta tạo lớp cài đặt các phương thức trừu tượng mà interface đó đã quy định. Lớp đó được gọi là lớp **cài đặt** interface mà ta đang nói đến.

Để khai báo rằng một lớp cài đặt một interface, ta dùng từ khóa **implements** thay vì **extends**, theo sau là tên của interface.

Một lớp có thể cài đặt một vài interface và đồng thời là lớp con của một lớp khác. Chẳng hạn lớp Dog vừa là lớp con của Canine, vừa là lớp cài đặt interface Pet:

```
class Dog extends Canine implements Pet {...}
```

Ví dụ cụ thể về interface Pet và lớp Dog cài đặt Pet được cho trong Hình 1.1. Các phương thức của interface đều ngầm định là public và abstract, do đó ta không bắt buộc phải dùng hai từ khóa public abstract khi khai báo các phương thức. Do là các phương thức trừu tượng nên chúng không có thân mà chỉ có một dấu chấm phẩy ở cuối dòng khai báo. Trong lớp Dog có hai loại phương thức: các phương thức cài đặt interface Pet, và các phương thức cài đặt lớp cha Canine như thông thường.

---

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}

public class Dog extends Canine implements Pet {
    public void beFriendly() {...}
    public void play() {...}
    public void roam() {...}
    public void eat() {...}
}
```

*các phương thức cài đặt Pet*

---

Hình 8.6: Lớp Dog cài đặt interface Pet.

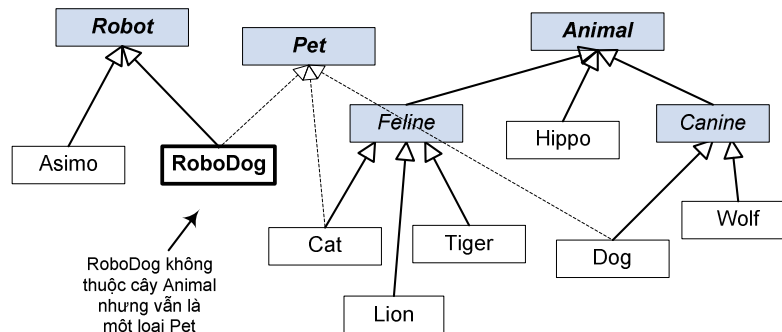
Như vậy ta có thể dùng cấu trúc interface để thực hiện một thứ gần giống đa thừa kế. Nó không hẳn là đa thừa kế ở chỗ: khác với lớp trừu tượng, ta không thể đặt mã cài đặt tại các interface.

Khi các phương thức tại interface đều trừu tượng, và do đó không thể tái sử dụng, ta được ích lợi gì ở đây? Câu trả lời là đa hình và đa hình. Khi ta dùng một interface thay cho các lớp riêng biệt làm tham số và giá trị trả về của phương thức, ta có thể truyền lớp bất kì nào cài đặt interface đó vào vị trí của tham số hay giá trị trả về đó. Không chỉ có vậy, các lớp nằm trên các cây thừa kế khác nhau có thể cùng cài đặt một interface.

Trong thực tế, đối với đa số thiết kế tốt, việc interface không thể chứa mã cài đặt không phải là vấn đề. Lí do là hầu hết các phương thức của interface có đặc điểm là

không thể được cài đặt một cách tổng quát, đằng nào cũng phải cài đặt các phương thức này ngay cả nếu chúng không bị buộc phải là phương thức trừu tượng.

Quay trở lại với ý rằng các lớp nằm trên các cây thừa kế khác nhau có thể cùng cài đặt một interface. Ta có ví dụ sau: Chó máy RoboDog là một loại robot và cũng là một loại thú cảnh. Lớp RoboDog thuộc cây thừa kế Robot chứ không thuộc cây Animal. Tuy nhiên, nó cũng có thể cài interface Pet như Cat và Dog.



Không chỉ có vậy, mỗi lớp còn có thể cài đặt nhiều hơn một interface. Sự linh hoạt của interface là đặc điểm vô cùng quan trọng đối với việc sử dụng Java API. Ví dụ, để một lớp đối tượng ở bất cứ đâu trên một cây thừa kế có thể được lưu ra file, ta có thể cho lớp đó cài interface Serializable.

Khi nào nên cho một lớp là lớp độc lập, lớp con, lớp trừu tượng, hay nên biến nó thành interface?

- Một lớp nên là lớp độc lập, nghĩa là nó không thừa kế lớp nào (ngoại trừ Object) nếu nó không thỏa mãn kiểm tra IS-A đối với bất cứ loại nào khác.
- Một lớp nên là lớp con nếu ta cần cho nó làm một phiên bản **chuyên biệt hơn** của một lớp khác và cần cài đặt hành vi có sẵn hoặc bổ sung hành vi mới.
- Một lớp nên là lớp cha nếu ta muốn định nghĩa một **khuôn mẫu** cho một nhóm các lớp con, và ta có một chút mã cài đặt mà tất cả các lớp con kia có thể sử dụng. Cho lớp đó làm lớp trừu tượng nếu ta muốn đảm bảo rằng không ai được tạo đối tượng thuộc lớp đó.
- Dùng một interface nếu ta muốn định nghĩa một **vai trò** mà các lớp khác có thể nhận, bất kể các lớp đó thuộc cây thừa kế nào.

#### Những điểm quan trọng:

- Khi muốn cấm tạo đối tượng từ một lớp, ta dùng từ khóa **abstract** tại định nghĩa lớp để tuyên bố lớp đó là lớp trừu tượng.
- Một lớp trừu tượng có thể có các phương thức trừu tượng cũng như không trừu tượng.
- Nếu một lớp có dù chỉ một phương thức trừu tượng, lớp đó buộc phải là lớp trừu tượng.

- Một phương thức trừu tượng không có thân, khai báo phương thức đó kết thúc bằng dấu chấm phẩy.
- Một lớp cụ thể phải cài đặt hoặc được thừa kế cài đặt của tất cả các phương thức trừu tượng.
- Mỗi lớp Java đều là lớp con trực tiếp hoặc gián tiếp của lớp Object.
- Nếu ta dùng một tham chiếu để gọi phương thức, tham chiếu đó được khai báo thuộc lớp gì hay interface gì thì ta chỉ được gọi các phương thức có trong lớp đó hoặc interface đó, bất kể đối tượng mà tham chiếu đó đang chiếu tới là đối tượng thuộc lớp nào.
- Một biến tham chiếu lớp cha có thể được gán giá trị là tham chiếu kiểu lớp con bất kì mà không cần đổi kiểu. Có thể dùng phép đổi kiểu để gán giá trị là tham chiếu kiểu lớp cha cho một biến tham chiếu kiểu lớp con, tuy nhiên khi chạy chương trình, phép đổi kiểu đó sẽ thất bại nếu đối tượng đang được chiếu tới không thuộc kiểu tương thích với phép đổi kiểu.
- Java không hỗ trợ đa thừa kế do vấn đề Hình thoi. Java chỉ cho phép mỗi lớp chỉ có duy nhất một lớp cha.
- Một interface tương tự với một lớp thuần túy trừu tượng. Nó chỉ định nghĩa các phương thức trừu tượng.
- Một lớp có thể cài đặt nhiều interface.
- Lớp nào cài đặt một interface thì phải cài tất cả các phương thức của interface đó, do tất cả các phương thức interface đều là các phương thức trừu tượng public.

## Đọc thêm

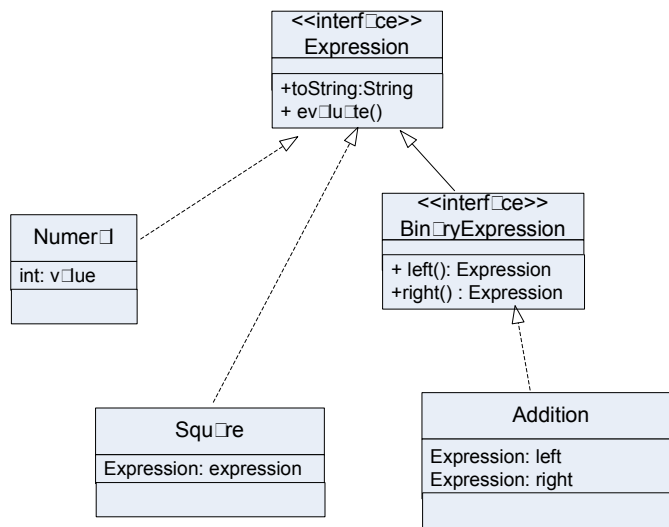
Bạn đọc có thể tìm hiểu sâu hơn về các mẫu thiết kế tại tài liệu sau:

1. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Đại học Công nghệ - HHOUGHN

## Bài tập

- Điền từ thích hợp vào các chỗ trống dưới đây
  - Nếu một lớp chứa ít nhất một phương thức trừu tượng thì nó phải là lớp \_\_\_\_\_
  - Các lớp mà từ đó có thể tạo đối tượng được gọi là các lớp \_\_\_\_\_
  - \_\_\_\_\_ cho phép sử dụng một tham chiếu kiểu lớp cha để gọi phương thức từ các đối tượng của lớp cha cũng như lớp con, cho phép ta lập trình cho trường hợp tổng quát.
  - Các phương thức không phải phương thức interface và không cung cấp cài đặt phương thức phải được khai báo với từ khóa \_\_\_\_\_
- Các phát biểu sau đây đúng hay sai:
  - Nếu một lớp cha khai báo một phương thức trừu tượng thì lớp con của nó buộc phải cài phương thức đó.
  - Một đối tượng thuộc một lớp cài đặt một interface có thể được coi là một đối tượng thuộc kiểu interface đó.
- Phương thức trừu tượng là gì? Hãy mô tả các tình huống mà ta nên khai báo một phương thức là phương thức trừu tượng.
- So sánh lớp trừu tượng và interface, khi nào ta nên dùng lớp trừu tượng, khi nào nên dùng interface?
- Đa hình hỗ trợ như thế nào cho khả năng mở rộng cây thừa kế?
- Liệt kê 4 kiểu gán tham chiếu lớp cha và lớp con cho các biến kiểu lớp cha và lớp con, mỗi kiểu có những thông tin quan trọng gì?
- Giải thích quan điểm rằng đa hình cho phép lập trình tổng quát thay vì lập trình cho từng trường hợp cụ thể. Dùng ví dụ minh họa. Lập trình tổng quát mang lại những ích lợi gì?
- Một lớp con có thể thừa kế giao diện hay cài đặt từ một lớp cha. Một cây thừa kế được thiết kế để cho thừa kế giao diện khác với cây thừa kế được dành cho thừa kế cài đặt như thế nào?
- Cài đặt 03 lớp và 02 interface trong sơ đồ sau. Trong đó các lớp Numeral (số) và Square (bình phương) cài đặt interface Expression (biểu thức, còn lớp Addition (phép cộng) cài đặt interface BinaryExpression (nhị thức – biểu thức có hai toán hạng), interface này lại thừa kế Expression.



Đại học Công nghệ - ĐHQGHN