ĐOÀN VĂN BAN

Lập trình hướng đối tượng với JAVA

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

ĐOÀN VĂN BAN

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VỚI JAVA

(Tái bản lần thứ nhất: có hiệu chỉnh và bổ sung)

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
HÀ NỘI - 2005

LỜI NÓI ĐẦU

Các phương pháp hướng đối tượng, đặc biệt là *lập trình hướng đối tượng* được xây dựng dựa trên nhiều khái niệm mới và được hỗ trợ bởi nhiều công cụ, ngôn ngữ lập trình ([2], [8]) rất mạnh giúp cho việc tạo ra những phần mềm ứng dụng có chất lượng cao, ngày càng đáp ứng tốt hơn yêu cầu của người sử dụng.

Ngôn ngữ Java do gãng Sun (http://java.sun.com) phát triển từ đầu những năm 90 đã trở thành ngôn ngữ lập trình hướng đối tượng rất được ưa chuộng trong những năm gần đây nhờ một số đặc điểm hết sức thích hợp với mạng Internet, hiện đã và đang được dùng phổ biến trên toàn thế giới nhằm đáp ứng các yêu cầu phát triển các ứng dụng trên mạng phục vụ cho nhiều người sử dụng với những môi trường thực hiện phần mềm khác nhau, Java là một ngôn ngữ lập trình hoàn chỉnh được thiết kế theo cách tiếp cận hướng đối tượng và kế thừa, sử dụng lại có nâng cấp của những ngôn ngữ lập trình trước nó.

Về mặt cú pháp, Java rất giống với C^{++}, một ngôn ngữ lập trình hướng đối tượng dùng phổ biến nhất hiện nay, nhưng loại đi một số tính khả dụng (*facilities*) quá mạnh nhưng khó và ít dùng, hoặc thừa về mặt ngôn ngữ [1] như: loại bỏ kế thừa nhiều lớp, vì chúng có thể tạo ra những lược đồ dữ liệu dạng đồ thị và là nguyên nhân chính có thể gây ra sự phức tạp và không đảm bảo tính nhất quán, tính đúng đắn trong quan hệ thông tin ([10], [11]); Java không cho phép thao tác số học trên kiểu con trỏ vì đây là nguồn gốc của những "con bọ" rất khó phát hiện ra khi biên dịch, v.v... Mục đích chính của Java là: *đơn giản, thân thiện, hướng đối tượng và cách tân nhằm tạo ra những phần mềm ứng dụng độc lập với môi trường sử dụng* ([7], [8]).

Cuốn sách này giới thiệu về lập trình hướng đối tượng sử dụng ngôn ngữ lập trình Java. Nội dung chính của cuốn sách được trình bày trong mười chương.

Chương I trình bày khái quát cách tiếp cận hướng chức năng và lập trình hướng đối tượng. Ngôn ngữ mô hình hóa hệ thống UML [2] được sử dụng để đặc tả các khái niệm cơ bản của lập trình hướng đối tượng. Chương II giới thiệu chu trình phát triển của các chương trình Java, quá trình dịch, thông dịch với JVM (Java Virtual Machine) và các thể loại chương trình ứng dụng, nhất là ứng dụng nhúng (*applet*) của Java. Chương III và chương IV trình bày những khái niệm cơ sở nhất của một ngôn ngữ lập trình và nêu cách xây dựng, tổ chức lớp các đối tượng trong các chương trình ứng dụng. Các lệnh điều khiển dòng thực hiện chương trình, đặc biệt là cơ chế xử lý ngoại lệ hỗ trợ để tạo ra những chương trình hoạt động tốt trong mọi tình huống, thích ứng được với mọi điều kiện trên cơ sở kiểm soát được các lỗi, các tình

huống có thể xảy ra, được giới thiệu chi tiết ở chương V. Chương VI đề cập đến một số lớp cơ sở nhất của Java và các kiểu cấu trúc dữ liệu phổ dụng như kiểu tuyển tập (*Collection*), kiểu tập hợp (Set), kiểu danh sách (List), v.v. Vấn đề phát triển những ứng dụng *applet* và sử dụng giao diện đồ họa của Windows (AWT) dưới dạng các trang Web với nhiều ví dụ minh họa được giới thiệu ở chương VII. Chương VIII giới thiệu các lớp xử lý các luồng dữ liệu vào/ra chuẩn và những vấn đề có tổ chức, đọc, ghi lên các loại tệp dữ liệu. Chương IX trình bày vấn đề kết nối các cơ sở dữ liệu với JDBC nhằm tạo ra những hệ thống phần mềm tích hợp từ nhiều loại hệ thống thông tin khác nhau trên mạng. Lần tái bản này được bổ sung thêm chương X giới thiệu các thành phần của Swing, cho phép tạo ra những phần mềm mà bạn "thấy và cảm nhận được". Trong các chương có nhiều ví dụ là những chương trình hoàn chỉnh, minh họa cho cách sử dụng những khái niệm đã nêu ở trên.

Cuốn sách được biên soạn dựa trên kinh nghiệm giảng dạy giáo trình phân tích, thiết kế và lập trình hướng đối tượng của tác giả trong nhiều năm tại các khóa cao học, đại học của ĐH Quốc Gia Hà Nội, ĐH Bách Khoa Hà Nội, ĐH Khoa Học Huế, v.v. Cuốn sách có thể dùng làm giáo trình học tập, tài liệu tham khảo cho sinh viên các hệ kỹ sư, cử nhân, học viên cao học CNTT và các bạn quan tâm đến vấn đề lập trình hướng đối tượng để phát triển những ứng dụng độc lập với môi trường, hay để xây dựng các Web Site trên mạng.

Tác giả bày tỏ lòng biết ơn chân thành tới các bạn đồng nghiệp trong Phòng các Hệ thống phần mềm tích hợp, đặc biệt cám ơn TS. Đặng Thành Phu, Viện Công nghệ Thông tin, TT KHTN & CNQG, PTS.TS. Đỗ Đức Giáo, Khoa Công nghệ, ĐH QG HN đã động viên, góp ý và giúp đỡ để hoàn chỉnh nội dung cuốn sách này. Xin cám ơn Nhà xuất bản Khoa học và Kỹ thuật đã hỗ trợ và tạo điều kiện để cuốn sách được tái bản.

Mặc dù rất cố gắng nhưng tài liệu này chắc chắn không tránh khỏi những sai sót. Chúng tôi rất mong nhận được các ý kiến đóng góp của bạn đọc để có chỉnh lý kịp thời.

Thư góp ý xin gửi về: Nhà xuất bản Khoa học và Kỹ thuật 70 Trần Hưng Đạo Hà Nội.

Hà Nội, tháng 5 năm 2005

Tác giả

CHƯƠNG I

GIỚI THIỆU VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

1.1. CÁC CÁCH TIẾP CẬN TRONG LẬP TRÌNH

Phương pháp lập trình truyền thống chúng ta vẫn áp dụng đó là *lập trình có cấu trúc*. Phương pháp lập trình này thực hiện theo cách tiếp cận *hướng chức năng dựa chủ yếu vào việc phân tách các chức năng chính* của bài toán thành những chức năng đơn giản hơn và thực hiện làm mịn dần từ trên xuống (*top-down*). Theo cách tiếp cận hướng chức năng, chương trình được xem như là một tập các hàm chức năng, trong đó dữ liệu và các hàm là tách rời nhau. Đối với những hệ thống lớn, phức hợp, độ phức tạp của chương trình tăng lên, sự phụ thuộc của nó vào các kiểu dữ mà nó xử lý cũng tăng theo. Các kiểu dữ liệu được xử lý trong nhiều chức năng bên trong chương trình có cấu trúc, và khi có sự thay đổi về kiểu dữ liệu thì cũng phải thực hiện thay đổi ở mọi nơi mà dữ liệu đó được sử dụng. Một nhược điểm nữa của lập trình hướng chức năng là khi có nhiều người tham gia xây dựng chương trình, mỗi người được giao viết một số chức năng (hàm) riêng biệt nhưng lại phải sử dụng chung dữ liệu. Khi có nhu cầu cần thay đổi dữ liệu sẽ ảnh hưởng rất lớn đến công việc của nhiều người, v.v.

Phương pháp lập trình mà ngày nay chúng ta nghe nói tới nhiều đó là *lập trình hướng đối tượng*. Lập trình hướng đối tượng dựa trên nền tảng là các *đối tượng*. Đối tượng (thực thể) được xây dựng trên cơ sở gắn dữ liệu với các phép toán sẽ thể hiện được đúng cách mà chúng ta suy nghĩ, bao quát về chúng trong thế giới thực [3], [5]. Chẳng hạn, ô tô có bánh xe, di chuyển được và hướng của nó thay đổi được bằng cách thay đổi tay lái. Tương tự, cây là loại thực vật có thân gỗ và lá. Cây không phải là ô tô và những gì thực hiện được với ô tô sẽ không làm được với cây.

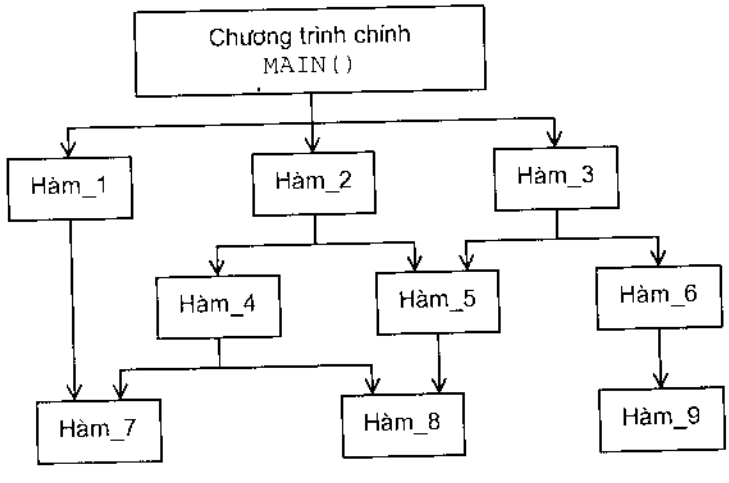
Lập trình hướng đối tượng cho phép chúng ta kết hợp những tri thức bao quát về các quá trình thực tế với những khái niệm trừu tượng được sử dụng trong máy tính. Chương trình hay hệ thống hướng đối tượng được xem như là tập các lớp đối tượng tương tác với nhau để thực hiện các yêu cầu của bài toán đặt ra.

Như vậy, hiện nay chúng ta có hai cách tiếp cận cơ bản để phát triển các hệ thống phần mềm: đó là cách tiếp cận hướng chức năng (*Function-Oriented*) và hướng đối tượng (*Object-Oriented*). Cả hai cách tiếp cận này đều áp dụng một nguyên lý chung để mô hình hóa hệ thống (để hiểu hệ thống) là thực hiện "*chia để trị*", phân chia bài toán thành những đơn vị tương đối đơn giản mà có thể dễ dàng quản lý được chúng một cách có hiệu quả.

Để hiểu rõ và áp dụng hiệu quả những phương pháp lập trình cần lựa chọn, chúng ta cần phân biệt những đặc trưng cơ bản nhất, đánh giá những mặt mạnh, mặt yếu của chúng.

1.1.1. LẬP TRÌNH HƯỚNG CHỨC NĂNG (THỦ TỤC)

Những ngôn ngữ lập trình bậc cao truyền thống như COBOL, FORTRAN, PASCAL, C v.v..., được gọi chung là ngôn ngữ lập trình *hướng chức năng*. Theo cách tiếp cận hướng chức năng thì một hệ thống phần mềm được xem như là dãy các công việc (chức năng) cần thực hiện như nhập dữ liệu, tính toán, xử lý, lập báo cáo và in ấn kết quả v.v... Mỗi công việc đó sẽ được thực hiện bởi một số hàm nhất định. Như vậy *trọng tâm của cách tiếp cận này là các hàm chức năng*. Cấu trúc của chương trình được xây dựng theo cách tiếp cận hướng chức năng có dạng như hình 1.1.

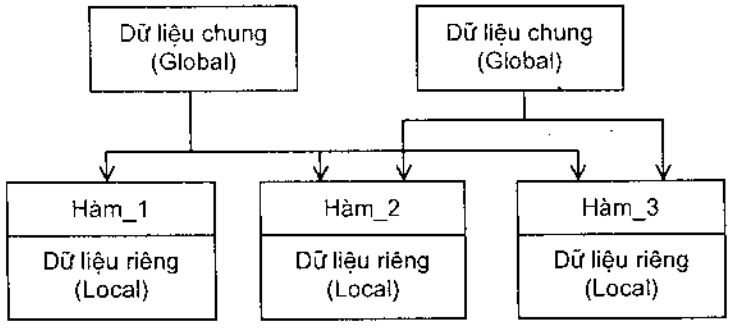


Hình 1.1. Cấu trúc của chương trình hướng chức năng.

Lập trình hướng chức năng (LTHCN) sử dụng *kỹ thuật phân rã các chức năng (hàm) theo cách tiếp cận top-down để tạo ra cấu trúc phân cấp. Chương trình* được xây dựng theo cách tiếp cận hướng chức năng thực chất là *tập các chương trình con* (có thể xem như là các hàm) mà theo đó máy tính cần thực hiện để hoàn thành những nhiệm vụ đặt ra của hệ thống.

Khi đặt trọng tâm vào các hàm thì dữ liệu, những cái mà các hàm sử dụng để thực hiện công việc của mình lại trở thành thứ yếu. Cái gì sẽ xảy ra đối với dữ liệu và gắn

dữ liệu với các hàm như thế nào? cùng nhiều vấn đề khác cần phải giải quyết khi chúng ta muốn xây dựng các phương pháp phù hợp để phát triển những hệ thống phần mềm giải quyết những yêu cầu đặt ra của thực tế. Hơn nữa, hệ thống luôn là một thể thống nhất, do vậy *các hàm trong một chương trình phải có liên hệ, trao đổi được với nhau*. Như chúng ta đã biết, các hàm (các bộ phận chức năng) chỉ có thể trao đổi được với nhau *thông qua các tham số, nghĩa là phải sử dụng biến chung (global)*. Mỗi hàm có thể có vùng dữ liệu riêng còn gọi là *dữ liệu cục bộ (local)*. Mối quan hệ giữa dữ liệu và hàm trong chương trình hướng chức năng được mô tả trong hình 1.2.



*Hình 1.2. **Quan hệ giữa dữ liệu và hàm trong LTHCN.***

Nhiều hàm có thể truy nhập, *sử dụng dữ liệu chung, làm thay đổi giá trị của chúng và vì vậy rất khó kiểm soát*. Nhất là đối với các chương trình lớn, phức tạp thì vấn đề càng trở nên khó khăn hơn. Khi chúng ta muốn thay đổi, bổ sung cấu trúc dữ liệu dùng chung cho một số hàm thì chúng ta phải thay đổi hầu như tất cả các hàm liên quan đến dữ liệu đó. Nhất là đối với những chương trình, dự án tin học lớn, phức tạp đòi hỏi nhiều người, nhiều nhóm tham gia thì những thay đổi của những biến dữ liệu chung sẽ *ảnh hưởng tới tất cả những ai có liên quan tới chúng*.

Một đặc tính nữa của cách tiếp cận hướng chức năng dễ nhận thấy là *tính mở (open) của hệ thống kém*. Thứ nhất, vì *dựa chính vào chức năng mà trong thực tế thì nhiệm vụ của hệ thống lại hay thay* đổi nên khi đó muốn cho hệ thống đáp ứng các yêu cầu thì phải thay đổi lại cấu trúc của hệ thống, nghĩa là phải thiết kế, lập trình lại hệ thống. Thứ hai, việc sử dụng các biến dữ liệu chung trong chương trình làm cho các nhóm chức năng phụ thuộc vào nhau về cấu trúc dữ liệu nên cũng hạn chế tính mở của hệ thống. Trong thực tế, cơ cấu tổ chức của mọi tổ chức thường ít thay đổi hơn là chức năng, nhiệm vụ phải thực hiện.

Mặt khác, cách tiếp cận hướng chức năng lại *tách dữ liệu khỏi chức năng xử lý* nên vấn đề che giấu, bảo vệ thông tin trong hệ thống là kém, nghĩa là vấn đề *an toàn, an ninh dữ liệu* là rất phức tạp.

Ngoài ra cách tiếp cận hướng chức năng cũng không hỗ trợ *việc sử dụng lại và kế*

thừa nên chất lượng và giá thành của các phần mềm rất khó được cải thiện. Những trở ngại mà chúng ta đã nêu ở trên sẽ làm cho mô hình được xây dựng theo cách tiếp cận hướng chức năng không mô tả được đầy đủ, trung thực hệ thống trong thực tế.

1.1.2. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Ngược lại, lập trình hướng đối tượng (LTHĐT) *đặt trọng tâm vào các đối tượng*, yếu tố quan trọng trong quá trình phát triển chương trình và nó không cho phép dữ liệu tách biệt, chuyển động tự do trong hệ thống. Dữ liệu được gắn chặt với các hàm thành phần và chúng được tổ chức, quản lý truy nhập theo nhiều mức khác nhau.

LTHĐT cho phép chúng ta phân tích bài toán thành *tập các thực thể* được gọi là các *lớp đối tượng*, sau đó xây dựng các dữ liệu thành phần cùng với các hàm thành phần thao tác trên các dữ liệu đó và trao đổi với những đối tượng khác để thực hiện những nhiệm vụ được giao.

Một chương trình, một hệ thống được xem như là một tập các lớp đối tượng và các đối tượng đó trao đổi với nhau thông qua việc gửi và nhận các thông điệp (message), do vậy một chương trình hướng đối tượng thực sự có thể hoàn toàn không cần sử dụng biến chung [1], [3].

Lập trình hướng đối tượng *dựa chủ yếu vào các đối tượng*, nên khi có nhu cầu thay đổi thì chỉ cần thay đổi ở một số lớp có liên quan, hoặc có thể bổ sung một số lớp mới trên cơ sở kế thừa và sử dụng lại nhiều nhất có thể. Mặt khác, như trên đã phân tích, một chương trình hướng đối tượng có thể không sử dụng hoặc hạn chế sử dụng các biến chung, do vậy dễ dàng tạo ra được những hệ thống có *tính mở* cao hơn.

Cơ chế bao bọc, che giấu thông tin của phương pháp hướng đối tượng giúp tạo ra được những *hệ thống an toàn, an ninh cao* hơn cách tiếp cận hướng chức năng.

Hơn nữa, phương pháp hướng đối tượng còn hỗ trợ rất mạnh nguyên lý *sử dụng lại nhiều nhất có thể và tạo ra mọi khả năng để kế thừa* những lớp đã được thiết kế, lập trình tốt để nhanh chóng tạo ra được những phần mềm có chất lượng, giá thành rẻ hơn và đáp ứng các yêu cầu của người sử dụng.

1.2. NHỮNG KHÁI NIỆM CƠ BẢN CỦA LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Để tiện lợi và phù hợp với xu thế phát triển hiện nay của công nghệ thông tin, chúng ta sẽ sử dụng UML (*Unified Modelling Language* [2], [7]) để đặc tả những khái niệm cơ bản của lập trình hướng đối tượng thông qua *Rational Rose* (http://www.rational.com) . UML là *ngôn ngữ mô hình hóa hình thức, thống nhất và*

trực quan. Phần lớn các thông tin trong mô hình được thể hiện bởi các ký hiệu đồ họa, biểu đồ thể hiện mối quan hệ giữa các thành phần của hệ thống một cách thống nhất và có logic chặt chẽ.

UML được sử dụng để *đặc tả, xây dựng và làm tài liệu cho các tác phẩm (Artifacts), các kết quả của của các pha phân tích, thiết kế và lập trình hướng đối tượng* dưới dạng các biểu đồ, bản mẫu hay các trang Web.

UML là ngôn ngữ chuẩn công nghiệp để lập kế hoạch chi tiết phát triển phần mềm. Hiện nay nhiều hãng sản xuất phần mềm lớn như: Microsoft, IBM, HP, Oracle, Digital Equipment Corp., Texas Instruments, Rational Software, v.v., sử dụng UML như là chuẩn cho ngôn ngữ mô hình hóa hệ thống phần mềm.

Phương pháp hướng đối tượng được xây dựng dựa trên một tập các khái niệm cơ sở:

1. Đối tượng (object),
2. Lớp đối tượng (class),
3. Trừu tượng hóa dữ liệu (Data Abstraction),
4. Bao bọc và che giấu thông tin (Encapsulation and Information Hiding),
5. Mở rộng, kế thừa giữa các lớp (Inheritance),
6. Đa xạ và nạp chồng (Polymorphism and Overloading),
7. Liên kết động (Dynamic Binding),
8. Truyền thông điệp (Message Passing).

1.2.1. ĐỐI TƯỢNG

Đối tượng là khái niệm cơ sở, quan trọng nhất của cách tiếp cận hướng đối tượng. Đối tượng là thực thể của hệ thống, của CSDL và được xác định thông qua định danh ID (*IDentifier*) của chúng. Mỗi *đối tượng có tập các đặc trưng bao gồm* cả các phần tài sản thường là *các dữ liệu thành phần* hay các thuộc tính mô tả các tính chất và *các phương thức, các thao tác trên các dữ liệu* để xác định hành vi của đối tượng đó.

Đối tượng là những thực thể được xác định trong thời gian hệ thống hướng đối tượng hoạt động. Như vậy đối tượng có thể biểu diễn cho người, vật, hay một bảng dữ liệu hoặc bất kỳ một hạng thức nào đó cần xử lý trong chương trình. Đối tượng cũng có thể là các dữ liệu được định nghĩa bởi người sử dụng (người lập trình) như *vector*, danh sách, các *record* v.v... Nhiệm vụ của LTHĐT là phân tích bài toán thành các đối tượng và xác định được bản chất của sự trao đổi thông tin giữa chúng. Đối tượng trong chương trình cần phải được chọn sao cho nó thể hiện được một cách gần nhất với những thực thể có trong hệ thống thực.

Như vậy, đối tượng được định nghĩa một cách trừu tượng như là một khái niệm, một

cấu trúc gộp chung cả phần dữ liệu (thuộc tính) với các hàm (phương thức) thao tác trên những dữ liệu đó và có thể trao đổi với những đối tượng khác. Theo quan điểm của người lập trình [3], *đối tượng được xem như là vùng bộ nhớ được phân chia trong máy tính để lưu trữ dữ liệu và tập các hàm tác động trên dữ liệu gắn với chúng*. Bởi vì các vùng phân hoạch bộ nhớ là độc lập với nhau nên các đối tượng có thể sử dụng bởi nhiều chương trình khác nhau mà không ảnh hưởng lẫn nhau.

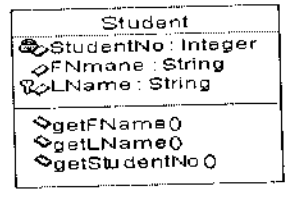
1.2.2. LỚP ĐỐI TƯỢNG

Lớp là bản mẫu hay một kiểu chung cho tất cả những đối tượng có những *đặc trưng giống nhau*, nghĩa là có các thuộc tính và hành vi giống nhau. Đối tượng chính là thể hiện (cá thể) của một lớp xác định. Trong lập trình hướng đối tượng, lớp được xem là đồng nhất với kiểu dữ liệu trừu tượng (ADT - Abstract Data Type được Barbara đề xuất vào những năm 70).

Phương pháp lập trình *hướng đối tượng* là cách phân chia chương trình thành các đơn thể (các lớp) bằng cách tạo ra các vùng bộ nhớ cho cả dữ liệu lẫn hàm và chúng sẽ được sử dụng như các mẫu để tạo ra bản sao từng đối tượng khi chúng được tạo ra trong hệ thống.

Nhiệm vụ của người lập trình là tìm cách xác định đầy đủ, chính xác danh sách các lớp đại diện cho tất cả các thực thể trong hệ thống và mối quan hệ giữa các lớp đối tượng đó [3], [5].

Như vậy, lớp chính là tập các đối tượng có cùng các thuộc tính và hành vi giống nhau. Các thành phần của lớp có thể chia thành ba vùng quản lý chính: công khai (*public*), được bảo vệ (*protected*) và vùng riêng (*private*). Trong ngôn ngữ mô hình hóa thống nhất UML, cấu trúc của lớp thường được đặc tả bởi: tên của lớp, tập các thuộc tính và tập các hàm.



Hình 1.3. Lớp **Student** *trong UML.*

Lớp có tên là *Student*. Lớp có ba thuộc tính: *StudentNo* có kiểu *Integer* là sở hữu riêng (*private*), bị khóa; thuộc tính *FName* có kiểu *String* là công khai (public) và thuộc tính *LName* là được bảo vệ (*protected*). Lớp *Student* có ba hàm : *getStudentNo(), getFName(), getLName()* đều là công khai.

1.2.3. TRỪU TƯỢNG HÓA DỮ LIỆU

Trừu tượng hóa là cách biểu diễn những đặc tính chính và bỏ qua những chi tiết. *Trừu tượng hóa* là sự mở rộng khái niệm kiểu dữ liệu và cho phép định nghĩa những phép toán trừu tượng trên các dữ liệu trừu tượng.

Để xây dựng các lớp, chúng ta phải sử dụng khái niệm trừu tượng hóa. Ví dụ, chúng ta có thể định nghĩa một lớp là danh sách các thuộc tính trừu tượng như là kích thước, hình dáng, mầu và các hàm xác định trên các thuộc tính này để mô tả các đối tượng trong không gian hình học. Trong lập trình, lớp được sử dụng như *kiểu dữ liệu trừu tượng*.

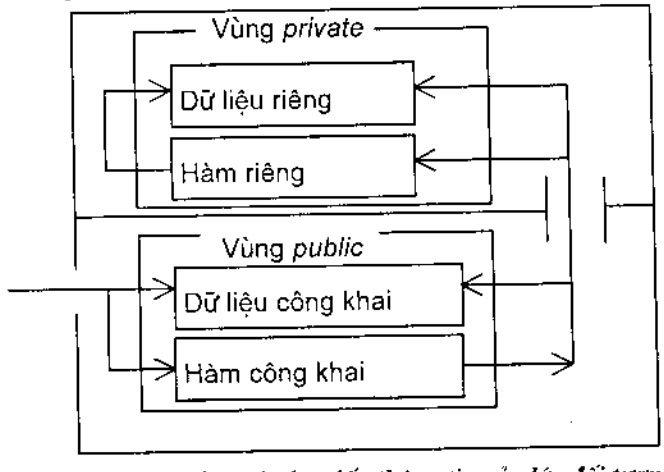
Khi xây dựng phần mềm thì nguyên lý trừu tượng hóa được thực hiện thông qua việc trừu tượng hóa các chức năng, hàm (thuật toán) và trừu tượng hóa các kiểu dữ liệu nguyên thủy.

1.2.4. BAO BỌC VÀ CHE GIẤU THÔNG TIN

Việc đóng *gói dữ liệu và các hàm vào một đơn vị cấu trúc* (gọi là lớp) được xem như một nguyên tắc *bao bọc thông tin*. Kỹ thuật này cho phép xác định các vùng đặc trưng *riêng, công khai hay được bảo vệ* bao gồm cả dữ liệu và các câu lệnh nhằm *điều khiển hoặc hạn chế những truy nhập tùy tiện* của những đối tượng khác. Dữ liệu được tổ chức sao cho thế giới bên ngoài (các đối tượng ở lớp khác) không truy nhập được vào những thuộc tính riêng và chỉ cho phép các hàm trong cùng lớp hoặc trong những lớp có quan hệ kế thừa với nhau được quyền truy nhập đến vùng được bảo vệ. Vùng công khai của lớp thì cho phép mọi đối tượng được phép truy nhập.

Chính các *hàm thành phần công khai của lớp sẽ đóng vai trò như là giao diện giữa các đối tượng* và với phần còn lại của hệ thống. Nguyên tắc bao bọc dữ liệu để ngăn cấm sự truy nhập trực tiếp trong lập trình được gọi là sự *che giấu thông tin*.

Nguyên lý bao bọc che giấu thông tin của lớp đối tượng được mô tả như trong hình 1.4.



*Hình 1.4. **Bao bọc và che giấu thông tin của lớp đối tượng.***

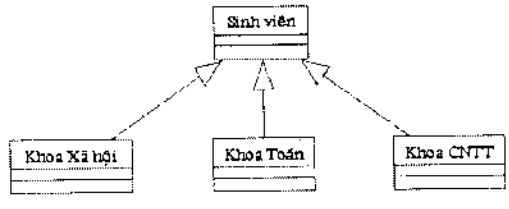
1.2.5. SỰ MỞ RỘNG, KẾ THỪA GIỮA CÁC LỚP

Nguyên lý kế thừa cho phép các đối tượng của lớp này được quyền sử dụng một số tính chất (cả dữ liệu và các hàm thành phần) của các lớp khác. Một lớp có thể là *lớp con* (*lớp dẫn* xuất) của một lớp khác, nghĩa là có thể bổ sung thêm một số tính chất để thu hẹp phạm vi xác định các đối tượng trong lớp mới cho phù hợp với ngữ cảnh trong thực tế.

Theo nguyên lý chung của kế thừa thì *chỉ những thuộc tính, hàm thành phần được bảo vệ và công khai là được quyền kế thừa*, còn những *thuộc tính, hàm thành phần riêng là không được phép kế thừa*. Những thuộc tính và hàm được kế thừa từ *lớp cha (lớp cơ sở)* được xem như là "tài sản" của chính các đối tượng con cháu, chúng có quyền sử dụng mà không cần phải khai báo hay định nghĩa lại. Như vậy, nguyên lý kế thừa trong lập trình hướng đối tượng hoàn toàn đồng nhất với nguyên lý kế thừa gia sản trong xã hội loài người.

Phương pháp hướng đối tượng nói chung hỗ trợ hai nguyên lý kế thừa: *kế thừa đơn và kế thừa bội*. Kế thừa đơn là một lớp có thể kế thừa từ một lớp cơ sở, còn kế thừa bội là một lớp có thể kế thừa từ nhiều hơn một lớp cơ sở. Ngôn ngữ C++ hỗ trợ cả hai nguyên lý kế thừa, nhưng *Java chỉ hỗ trợ thực hiện kế thừa đơn [8]*.

Nguyên lý kế thừa đơn hỗ trợ cho việc tạo ra cấu trúc cây phân cấp các lớp. Ví dụ, một trường đại học đào tạo sinh viên có ba khoa: Khoa Xã hội, Khoa Công nghệ Thông tin và Khoa Toán. Chúng ta có thể xây dựng lớp *Sinh viên* là lớp cơ sở để từ đó xây dựng tiếp ba lớp kế thừa là: *Khoa Xã hội, Khoa Toán* và *Khoa CNTT*. Hệ thống sẽ được thiết kế thành các lớp kế thừa và được mô tả trong UML như sau:



Hình 1.5. ***Cấu trúc phân cấp các lớp theo quan hệ kế thừa trong UML.***

Trong LTHĐT, khái niệm kế thừa kéo theo ý tưởng sử dụng lại. Nghĩa là từ một lớp đã được xây dựng chúng ta có thể bổ sung thêm một số tính chất tạo ra một lớp mới kế thừa lớp cũ mà không làm thay đổi những cái đã có.

Khái niệm kế thừa được hiểu như cơ chế sao chép ảo không đơn điệu. Trong thực tế, mọi việc xảy ra tựa như những lớp cơ sở đều được sao vào trong lớp con (lớp dẫn xuất) mặc dù điều này không được cài đặt tường minh (nên gọi là sao chép ảo) và việc sao chép chỉ thực hiện đối với những thông tin chưa được xác định trong các lớp

cơ sở (sao chép không đơn điệu). Do vậy, có thể diễn đạt cơ chế kế thừa như sau:

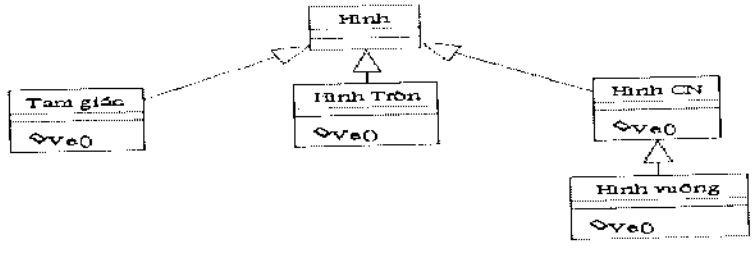
1. Lớp A kế thừa lớp B sẽ có (không tường minh) tất cả các thuộc tính, hàm đã được xác định trong B, ở những vùng được phép kế thừa (được bảo vệ, công khai),
2. Bổ sung thêm một số thuộc tính, hàm để mô tả được đúng các hành vi của những đối tượng mà lớp A quản lý.

1.2.6. ĐA XẠ (TƯƠNG ỨNG BỘI) VÀ NẠP CHỒNG

Một khái niệm quan trọng nữa trong LTHĐT là khái niệm *đa xạ* tương tự như trong toán học. Đa xạ là kỹ thuật được sử dụng để mô tả *khả năng gửi một thông điệp chung tới nhiều đối tượng mà mỗi đối tượng lại có cách xử lý riêng theo ngữ cảnh của mình.*

Đa xạ đóng một vai trò quan trọng trong việc tạo ra các đối tượng có cấu trúc với những nội dung thực hiện khác nhau mà lại có khả năng sử dụng chung một giao diện (cùng một tên gọi). Theo một nghĩa nào đó, đa xạ là sự mở rộng khái niệm sử dụng lại trong nguyên lý kế thừa. Hình 1.6 cho chúng ta thấy hàm có tên là *Ve()* có thể sử dụng để vẽ các hình khác nhau phụ thuộc vào đối tượng là các hình khi gọi để thực hiện.

Tam giác, Hình tròn và *Hình chữ nhật* là các lớp con của lớp *Hình*. Hàm *Ve()* là hàm đa xạ và nó được xác định tùy theo ngữ cảnh khi sử dụng. Khi gọi thực hiện đối tượng là *Tam giác* thì sẽ vẽ tam giác, nếu đối tượng là *hình tròn* thì vẽ hình tròn, v.v.



Hình 1.6. Tương ứng bội của hàm Ve().

Nạp chồng (*Overloading*) là một trường hợp của đa xạ. *Nạp chồng* là khả năng của một khái niệm (như các phép toán chẳng hạn) có thể được sử dụng với nhiều nội dung thực hiện khác nhau tùy theo ngữ cảnh, cụ thể là tuỳ thuộc vào kiểu và số các tham số của chúng. Ví dụ, hàm *Cong()* có thể được nạp chồng để cộng các số nguyên (*int*), số thực (*float*), số phức (*Complex*) hoặc ghép các xâu ký tự (*String*), v.v.

```
int Cong(int, int);                    // Cộng hai số nguyên
```

```
float Cong(float, float);          // Cộng hai số thực
Complex Cong(Complex, Complex);    // Cộng hai số phức
String Cong(String, String);       // Ghép hai xâu
String Cong(String, int);          // Ghép một xâu với một số nguyên
```

1.2.7. LIÊN KẾT ĐỘNG

Liên kết thông thường (liên kết tĩnh) là dạng liên kết được xác định ngay khi dịch chương trình. Hầu hết các chương trình được viết bằng Pascal, C đều là dạng liên kết tĩnh. *Liên kết động* là dạng liên kết các hàm, chức năng khi chương trình thực hiện các lời gọi các hàm, chức năng đó. Như vậy trong liên kết động, nội dung của đoạn chương trình ứng với chức năng, hàm sẽ không được xác định cho đến khi thực hiện lời gọi tới chức năng, hàm đó. Liên kết động liên quan chặt chẽ với những khái niệm đa xạ và kế thừa trong lập trình hướng đối tượng. Chính nhờ khái niệm liên kết động mà nhiều khái niệm của lập trình hướng đối tượng như khái niệm đa xạ thực hiện được.

Chúng ta hãy xét hàm *Ve()* trong hình 1.6. Theo nguyên lý kế thừa thì mọi đối tượng đều có thể sử dụng hàm này để vẽ các hình cụ thể khi hệ thống thực hiện. Hàm *Ve()* được định nghĩa lại ở trong các lớp dẫn xuất để vẽ tam giác, hình tròn hay hình chữ nhật theo các thuật toán tương ứng. Khi thực hiện, ví dụ: nếu đối tượng *Hình* là *Hình tròn* thì hệ thống sẽ liên kết với hàm *Ve()* được định nghĩa trong lớp *Hình tròn* để vẽ hình tròn.

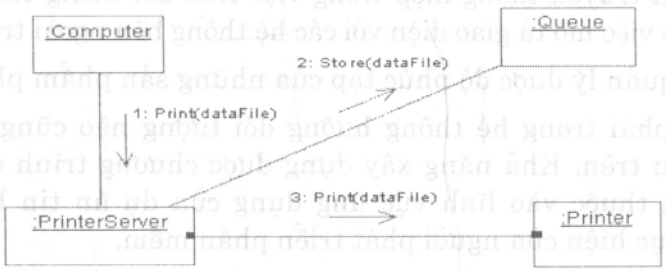
1.2.8. TRUYỀN THÔNG ĐIỆP

Chương trình hướng đối tượng (được thiết kế và lập trình theo phương pháp hướng đối tượng) bao gồm một tập các đối tượng và mối quan hệ giữa các đối tượng đó với nhau. Vì vậy, lập trình trong ngôn ngữ hướng đối tượng bao gồm các bước sau:

1. Tạo ra các lớp đối tượng và mô tả hành vi của chúng;
2. Tạo ra các đối tượng theo định nghĩa của các lớp;
3. Xác định sự trao đổi thông tin giữa các đối tượng trong hệ thống.

Các đối tượng gửi và nhận thông tin trong hệ thống hướng đối tượng cũng giống như con người trao đổi với nhau trong xã hội. Chính nguyên lý trao đổi thông tin bằng cách truyền thông điệp cho phép chúng ta dễ dàng xây dựng được hệ thống mô phỏng gần hơn với thực tế. Truyền thông điệp cho một đối tượng tức là báo cho nó phải thực hiện một việc, một yêu cầu (thỉnh cầu) nào đó. Cách ứng xử của đối tượng sẽ được mô tả ở trong lớp thông qua các hàm công khai (hay còn được gọi là lớp dịch vụ).

Trong chương trình, thông điệp gửi đến cho một đối tượng chính là để yêu cầu thực hiện một công việc cụ thể, nghĩa là sử dụng những hàm tương ứng để xử lý dữ liệu đã được khai báo trong lớp đối tượng đó. Vì vậy, trong thông điệp phải chỉ ra được hàm cần thực hiện của đối tượng nhận thông điệp. Hơn thế nữa, thông điệp truyền đi phải xác định tên đối tượng, tên hàm (thông điệp) và thông tin truyền đi. Ví dụ, khi hệ thống máy tính muốn in một tệp *dataFile* thì máy tính hiện thời (:*Computer*) gửi đến cho đối tượng *:PrinterServer* một yêu cầu *Print(dataFile)*. Bộ phận dịch vụ máy in sẽ kiểm tra xem nếu máy in đang bận thì lưu yêu cầu đó vào hàng đợi bằng cách gửi cho (*:Queue*) thông điệp là *Store(dataFile)*, ngược lại gửi yêu cầu in *Print(dataFile)* cho đối tượng *:Printer*. Hoạt động trao đổi thông điệp giữa các đối tượng trên có thể mô tả trong UML như hình 1.7.



Hình 1.7. Truyền thông điệp giữa các đối tượng.

Mỗi đối tượng chỉ tồn tại trong thời gian nhất định. Đối tượng được tạo ra khi nó được khai báo và sẽ bị hủy bỏ khi chương trình ra khỏi miền xác định của đối tượng đó. Sự trao đổi thông tin chỉ có thể thực hiện trong thời gian tồn tại của đối tượng.

1.2.9. CÁC ƯU ĐIỂM CỦA LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Như trên đã phân tích, lập trình hướng đối tượng đem lại một số lợi thế cho cả người thiết kế lẫn người lập trình. Cách tiếp cận hướng đối tượng giải quyết được nhiều vấn đề tồn tại trong quá trình phát triển phần mềm và tạo ra được những sản phẩm phần mềm có chất lượng cao. Những phương pháp này mở ra một triển vọng to lớn cho những người lập trình. Hy vọng sẽ có nhiều sản phẩm phần mềm tốt hơn, đáp ứng được những tính chất về sản phẩm chất lượng cao trong Công nghệ phần mềm và nhất là bảo trì hệ thống ít tốn kém hơn. Những ưu điểm chính của LTHĐT là:

1. Thông qua nguyên lý kế thừa, chúng ta có thể loại bỏ được những đoạn chương trình lặp lại, dư thừa trong quá trình mô tả các lớp và mở rộng khả năng sử dụng các lớp đã được xây dựng.

2. Chương trình được xây dựng từ những đơn thể (đối tượng) trao đổi với nhau nên việc thiết kế và lập trình sẽ được thực hiện theo quy trình nhất định chứ không phải dựa vào kinh nghiệm và kỹ thuật như trước. Điều

này đảm bảo rút ngắn được thời gian xây dựng hệ thống và tăng năng suất lao động.

3. Nguyên lý che giấu thông tin giúp người lập trình tạo ra được những chương trình an toàn không bị thay đổi bởi những đoạn chương trình khác một cách tuỳ tiện.
4. Có thể xây dựng được ánh xạ các đối tượng của bài toán vào đối tượng của chương trình.
5. Cách tiếp cận thiết kế đặt trọng tâm vào dữ liệu, giúp chúng ta xây dựng được mô hình chi tiết và phù hợp với thực tế.
6. Những hệ thống hướng đối tượng dễ mở rộng, nâng cấp thành những hệ lớn hơn.
7. Kỹ thuật truyền thông điệp trong việc trao đổi thông tin giữa các đối tượng giúp cho việc mô tả giao diện với các hệ thống bên ngoài trở nên đơn giản hơn.
8. Có thể quản lý được độ phức tạp của những sản phẩm phần mềm.
9. Không phải trong hệ thống hướng đối tượng nào cũng có tất cả các tính chất nêu trên. Khả năng xây dựng được chương trình có các tính chất đó còn phụ thuộc vào lĩnh vực ứng dụng của dự án tin học và vào phương pháp thực hiện của người phát triển phần mềm.

1.3. CÁC NGÔN NGỮ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Lập trình hướng đối tượng không là đặc quyền của một ngôn ngữ nào đặc biệt. Cũng giống như lập trình có cấu trúc, những khái niệm trong lập trình hướng đối tượng có thể cài đặt trong những ngôn ngữ lập trình như C hoặc Pascal. Tuy nhiên, đối với những chương trình lớn, phức hợp thì vấn đề lập trình sẽ trở nên phức tạp, nếu sử dụng những ngôn ngữ không phải là ngôn ngữ hướng đối tượng thì phải thực hiện nhiều thỏa hiệp. Những ngôn ngữ được thiết kế đặc biệt, hỗ trợ cho việc mô tả, cài đặt các khái niệm của phương pháp hướng đối tượng được gọi chung là ngôn ngữ hướng đối tượng.

Dựa vào khả năng đáp ứng các khái niệm về hướng đối tượng, chúng ta có thể chia ra làm hai loại:

1. Ngôn ngữ lập trình dựa trên đối tượng (object-based),
2. Ngôn ngữ lập trình hướng đối tượng (object-oriented).

Lập trình dựa trên đối tượng là kiểu lập trình hỗ trợ chính cho việc bao bọc, che giấu thông tin và định danh các đối tượng. Lập trình dựa trên đối tượng có những đặc tính sau:

- Bao bọc dữ liệu,
- Cơ chế che giấu và hạn chế truy nhập dữ liệu,

- Tự động tạo lập và hủy bỏ các đối tượng,
- Tính đa xạ.

Ngôn ngữ hỗ trợ cho kiểu lập trình trên được gọi là ngôn ngữ lập trình dựa trên đối tượng. Ngôn ngữ trong lớp này không hỗ trợ cho việc thực hiện kế thừa và liên kết động. *Ada* là ngôn ngữ lập trình dựa trên đối tượng.

Lập trình hướng đối tượng là kiểu lập trình dựa trên đối tượng và bổ sung thêm nhiều cấu trúc để cài đặt những quan hệ về kế thừa và liên kết động. Vì vậy đặc tính của LTHĐT có thể viết một cách ngắn gọn như sau:

Các đặc tính dựa trên đối tượng + kế thừa + liên kết động.

Ngôn ngữ hỗ trợ cho những đặc tính trên được gọi là ngôn ngữ LTHĐT, ví dụ như *Java, C++, Smalltalk, Object Pascal hay Eiffel*,v.v...

Việc chọn một ngôn ngữ để cài đặt phần mềm phụ thuộc nhiều vào các đặc tính và yêu cầu của bài toán ứng dụng, vào khả năng sử dụng lại của những chương trình đã có và vào tổ chức của nhóm tham gia xây dựng phần mềm. Một trong những ngôn ngữ lập trình hướng đối tượng thực sự được sử dụng phổ biến hiện nay là Java [8] sẽ được giới thiệu chi tiết ở các chương sau.

BÀI TẬP

1.1. Nêu các đặc trưng cơ bản của cách tiếp cận lập trình hướng chức năng và lập trình hướng đối tượng.

1.2. Tại sao lại gọi khái niệm lớp là kiểu dữ liệu trừu tượng trong lập trình hướng đối tượng.

1.3. Nêu nguyên lý hoạt động của khái niệm đa xạ, tương ứng bội trong lập trình hướng đối tượng, khái niệm này thực hiện được trong lập trình hướng chức năng hay không, tại sao?

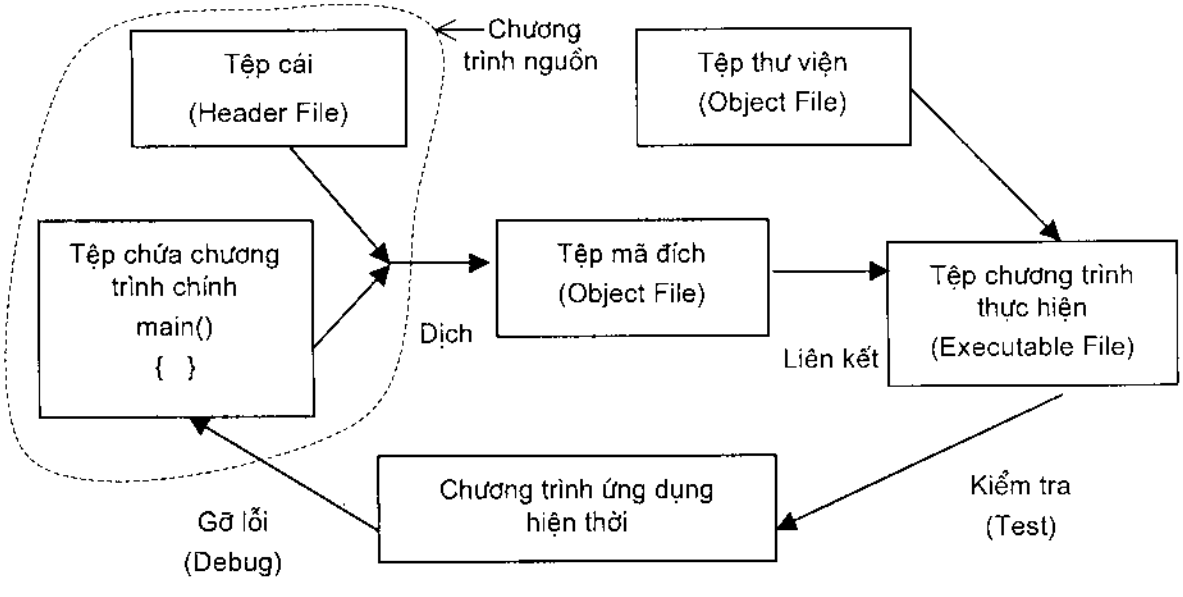
1.4. Khái niệm kế thừa và sử dụng lại trong lập trình hướng đối tượng là gì?, ngôn ngữ lập trình C++ và Java hỗ trợ quan hệ kế thừa như thế nào?.

CHƯƠNG II

GIỚI THIỆU VỀ LẬP TRÌNH VỚI JAVA

2.1. GIỚI THIỆU CHUNG

Các chương trình dịch của các ngôn ngữ lập trình truyền thống như C / C++, Pascal thường dịch các tệp chương trình nguồn sang các câu lệnh đặc biệt mà máy tính của bạn hiểu được. Những ngôn ngữ như C/C++ được xây dựng trên cơ sở tạo ra những chương trình dịch tối ưu với mã máy thực hiện hiệu quả, nhanh và linh hoạt. Chu trình phát triển và thực hiện của chương trình viết bằng những ngôn ngữ truyền thống như C/C++ có thể được mô tả như sau



Hình 2.1. ***Chu trình phát triển và thực hiện của chương trình C/C++.***

Tuy nhiên để tối ưu hóa được chương trình dịch thì nó phải được thực hiện dựa *trên một kiến trúc (tập các lệnh) xác định*, nghĩa là phụ thuộc vào cấu hình của máy đích. Để tạo ra được sự độc lập tương đối thì chương trình của bạn phải dịch lại để phù

hợp với kiến trúc máy mới mỗi khi bạn thay đổi môi trường thực hiện chương trình.

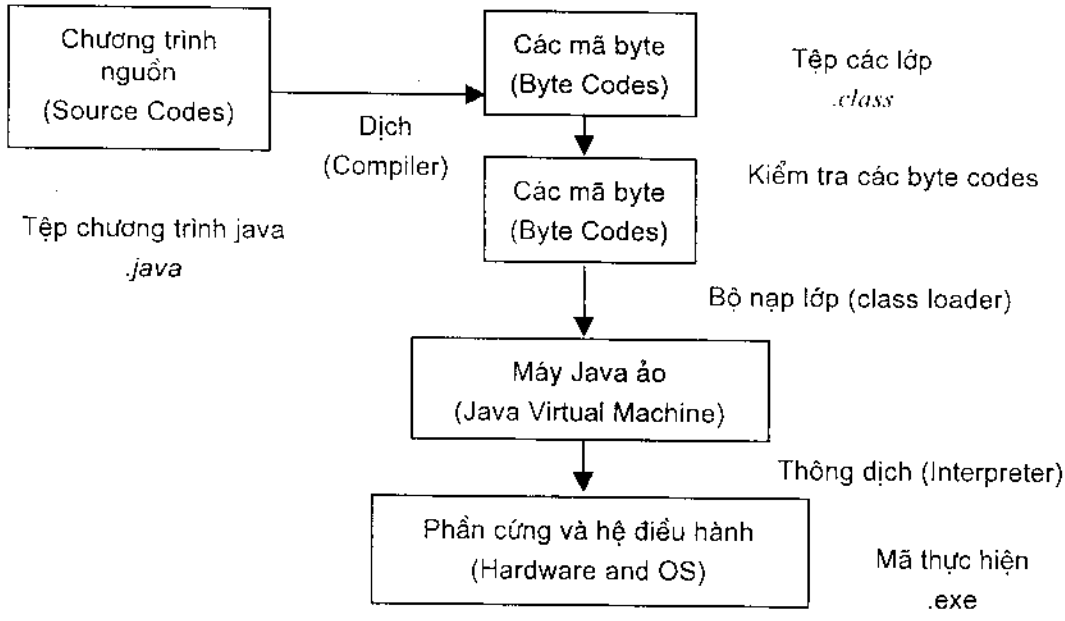
Hiện nay có nhiều loại máy tính với nhiều cấu hình khác nhau. Thông thường với mỗi loại ứng dụng chúng ta chọn một loại máy tính và cấu hình hợp lý để phát triển chương trình cho hiệu quả. Tuy nhiên, khi điều kiện thực hiện chương trình thay đổi thường gây rất nhiều trở ngại cho người sử dụng. Do đó những người phát triển chương trình ứng dụng (người lập trình) muốn phát kiến những phương thức làm việc mới có thể độc lập được với những thay đổi của môi trường và mong muốn có sự hỗ trợ của các ngôn ngữ lập trình.

Đặc biệt, *mạng Internet* được xem như là mạng của nhiều mạng khác nhau về nhiều lĩnh vực, cả phần cứng lẫn phần mềm. Mạng cung cấp nhiều dịch vụ tiện lợi cho nhiều ứng dụng khác nhau, nhất là các ứng dụng trên mạng của *công nghệ Web*. Để đưa được các yếu tố lập trình lên mạng và kết hợp với Web thì không nên sử dụng các chương trình viết bằng C/C++, bởi vì:

1. Vấn đề an toàn, an ninh dữ liệu trên mạng không đảm bảo,
2. Quan trọng hơn là các chương trình C/C++ được dịch sang một mã máy có cấu hình cố định, vì vậy khi được nạp từ trên mạng xuống một máy có cấu hình khác sẽ không thực hiện được. Ví dụ, một chương trình đã được dịch ở Machintosh thì sẽ không thực hiện được ở Windows và ngược lại.

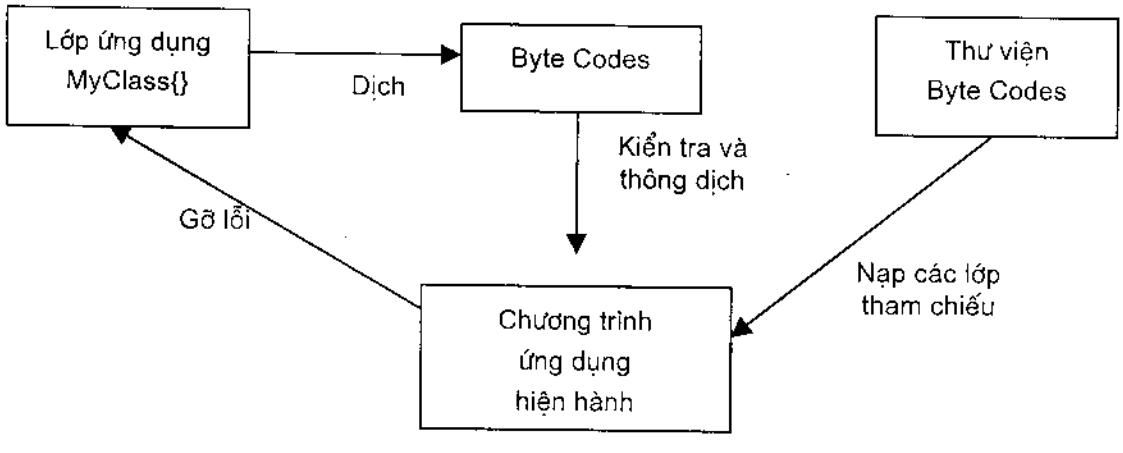
Điều mà chúng ta muốn nhất hiện nay là cần có một *ngôn ngữ thông dịch* thật mạnh để khắc phục những nhược điểm trên, đặc biệt để phát triển được dễ dàng các ứng dụng với Web trên mạng. Nhưng theo nguyên lý của chương trình thông dịch (*Interpreter*) thì nó thực hiện khá chậm. Nếu chương trình thông dịch phải thực hiện phân tích chương trình nguồn mỗi khi thực hiện thì các chương trình ứng dụng sẽ rất chậm.

Java vượt qua được các nhược điểm trên bằng cách dịch các chương trình nguồn sang ngôn ngữ máy ảo không phụ thuộc vào *chip* (*hệ lệnh cụ thể*) nào cả và sau đó khi cần thực hiện sẽ thông dịch sang hệ máy cụ thể. Kết quả của chương trình dịch không phải là mã đích (*Object Code*) như kết quả của các chương trình dịch truyền thống mà là chuỗi các bytes cơ sở bao gồm các mã lệnh thực hiện (*Opcode*) và các tham số của máy lý thuyết (máy ảo). Máy này được gọi là máy Java ảo (*JVM - Java Virtual Machine*). JVM có thể nhúng vào bất kỳ một biểu diễn, một môi trường cụ thể mà bạn có. Khi cài đặt trên một môi trường xác định (*máy có cấu hình cố định*) thì phần mềm được thông dịch và JVM trở thành máy cụ thể để có thể thực thi ứng dụng của bạn. Chương trình Java được thực hiện như sau:



Hình 2.2. ***Quá trình dịch và thông dịch chương trình Java.***

Công nghệ Java giải quyết vấn đề tốc độ bằng cách dịch chương trình nguồn sang các *mã byte* (*byte codes*). Khi JVM thực hiện, nó sẽ tìm các đối tượng cần tham chiếu của các lớp trong chương trình chính (*chương trình ứng dụng*) ở thời điểm thực hiện để nạp chúng xuống. Quá trình phát triển và thực hiện chương trình Java thực hiện như sau:



Hình 2.3. ***Quá trình phát triển chương trình Java.***

Như vậy, Java là *ngôn ngữ động* không yêu cầu mọi thứ phải sẵn sàng (xác định) ở lúc dịch mà JVM có thể nạp các lớp (*các đối tượng*) cần thiết (*tệp .class*) để làm việc khi thực hiện chương trình ứng dụng. Thông thường, mỗi lớp trong chương trình sẽ được dịch sang một tệp có đuôi (phần mở rộng) là *.class*. Theo cách tổ chức của Java

không có sự phụ thuộc giữa các tệp nguồn trong quá trình dịch. Nghĩa là khi các giao diện (*mối quan hệ giữa các lớp*) chưa bị thay đổi thì không cần phải dịch lại các tệp chứa mã nguồn của những lớp được tham chiếu hoặc các lớp tiện ích, trừ khi chính các lớp này bị thay đổi. Chúng ta chỉ phải dịch lại những tệp *.java* mới hoặc những tệp đã bị thay đổi.

Tóm lại, C/C++ và Java là những ngôn ngữ nhằm những mục đích khác nhau.

- Đối với C/C++ cái chính là tốc độ, mạnh mẽ và uyển chuyển trong lập trình. Khi thực hiện chương trình C/C++ thì mọi việc của hệ thống đều phải bắt đầu và kết thúc ở một chương trình con đặc biệt, hàm *main()*. Chúng ta có thể sử dụng thư viện động, những thứ yêu cầu được liên kết lại khi cần, nhưng về tổng thể là phải định nghĩa và xác định ngay từ trước khi dịch.
- Đối với Java mục đích chính là: đơn giản, thân thiện, hướng đối tượng và cách tân nhằm tạo ta những phần mềm độc lập với môi trường. Khi chạy một chương trình ứng dụng Java thì cần chỉ ra những lớp đối tượng mà bạn muốn chúng thực thi những công việc cần thực hiện và JVM sẽ nạp các lớp đó xuống khi có nhu cầu. Với cơ chế thông dịch thì phần lớn các yêu cầu chi tiết sẽ được xác định ở thời điểm thực hiện, chứ không phải thời điểm khi dịch. Do vậy bộ nhớ được cấp phát động lúc thực thi chương trình.

2.2. MÔI TRƯỜNG JAVA

Như trên chúng ta đã khẳng định, chương trình Java có thể dịch và thực hiện trong mọi môi trường điều hành, miễn là ở đó có chương trình thông dịch (*máy Java ảo - JVM*). Vấn đề đặt ra là làm thế nào để kết hợp được cơ chế thông dịch Java với các bộ trình duyệt Web (*Web Browsers*). Những vấn đề liên quan là:

- Cơ chế xử lý các *Web Site*,
- Tại sao lại sử dụng Java cho công nghệ *Web*,
- Tổ chức thực hiện các dự án ứng dụng với *Java*.

□ Xử lý các Web Site

Một đặc trưng của Java được đặc biệt chú ý là khả năng nạp xuống *(downloaded)* *Web* theo yêu cầu và thực hiện chương trình ứng dụng trên các máy khách. Trước khi *Java* xuất hiện thì *Web* được xem như là nguồn cung cấp các tư liệu *(document)* và chỉ thuần túy là các tư liệu.

Web sẽ được tăng thêm giá trị khi có thêm các dạng chứa thêm thông tin bổ sung. Trước đây các tư liệu chỉ chứa các dữ liệu thô, với *Java Web* được bổ sung thêm sự liên kết giữa các mẩu dữ liệu liên quan để biến đổi những dữ liệu thô thành những

thông tin.

Bản thân Java là biến đổi thông tin. Thay vì hiển thị sơ đồ dữ liệu, các bộ trình duyệt được Java hỗ trợ cho phép thao tác trên các dữ liệu, thông tin, thực hiện phân tích, thay đổi kích thước, phạm vi xử lý các dữ liệu để có những thông tin sinh động.

❑ Java và Web

Để trả lời được câu hỏi tại sao cần Java cho *Web*, chúng ta phải nắm bắt được các đặc điểm của *Web*.

(i) Độc lập với môi trường (Platform Independent)

+ Trước tiên các chương trình chạy trên *Web* đòi hỏi phải không phụ thuộc vào môi trường. *Web* phải mở đối với mọi người thông qua các giao thức *(protocol)* chuẩn để thực hiện trên mọi môi trường.
+ Ngôn ngữ thông dịch rất phù hợp với mục đích của *Web*: Trình duyệt thông dịch các lệnh của HTML sang các tư liệu có khuôn dạng đẹp sinh động, đáp ứng theo mọi yêu cầu.
+ Khi thực hiện chương trình, *JVM* nằm giữa phần cứng và chương trình ứng dụng, làm nhiệm vụ thông dịch các kết quả ở dạng mã *byte* vào môi trường của máy khách để thực hiện, do vậy đảm bảo được tính độc lập.

(ii) Đảm bảo an ninh (secure) thông tin

+ Bộ dịch Java đảm bảo không nạp các chương trình *virus* vào chương trình ứng dụng. Khi chương trình Java được nạp xuống theo yêu cầu của *Web*, nó kiểm soát tất cả các mã *byte* của chương trình được nạp xuống và đảm bảo rằng chúng phải tuân theo các qui định, các ràng buộc của ngôn ngữ.
+ Mặt khác, lời gọi các phương thức trong mã *byte* không phải là các địa chỉ như trong các ngôn ngữ truyền thống mà là các tên gọi (*thông qua định danh đối tượng*). Trong các chương trình C/C++, các lời gọi hàm là các địa chỉ và rất khó kiểm soát vì trong đó cho phép các phép nhảy tự do tới bất kỳ địa chỉ nào đó khi thực hiện chương trình. Trong Java, các đối tượng và các phương thức mô tả các hành vi của các đối tượng đều được xác định bằng định danh do vậy dễ dàng kiểm soát chúng và đảm bảo an ninh cao.

(iii) Đảm bảo an toàn (safe)

+ Nhiều khi sẽ rất là nguy hiểm nếu xảy ra các sự cố tràn bộ nhớ, vượt quá các chỉ số giới hạn. Java luôn kiểm soát được các tình huống đó. Ví dụ, cấu trúc dữ liệu mảng *(array)* và *String* trong Java là các lớp đối tượng và do vậy có cơ chế để tự kiểm soát các chỉ số và các giới hạn sử dụng của các phần tử trong mảng.

+ Hơn nữa, Java không sử dụng con trỏ *(pointer)* nên không cho phép nhảy tự do để thao tác tùy ý ở các vị trí bất kỳ trong bộ nhớ.

(iv) Thực hiện đa luồng (multithreads)

+ *Web* là môi trường mà ở đó có nhiều sự kiện có thể xảy ra đồng thời. Ngôn ngữ sử dụng để lập trình Web phải hỗ trợ để dễ dàng cho nhiều sự kiện có thể thực hiện cùng một lúc.
+ Java đáp ứng được tiêu chuẩn đa luồng vì: Thứ nhất nó cung cấp các lớp có thể thực hiện như là các luồng được điều khiển riêng biệt. Thứ hai là Java tự thực hiện được sự kết hợp giữa các phần dị bộ trong các luồng với nhau.

(v) Đảm bảo động (dynamic)

+ Các lớp không nhất thiết phải dịch và liên kết để tạo ra một tệp lớn có thể thực hiện ngay.
+ Thay vì đó, lúc thực hiện Java có thể xác định những lớp nào cần thì nạp xuống, còn những lớp khác có thể để trên mạng và sau đó nạp chúng vào môi trường Java khi cần thiết. Bằng phương pháp như trên, các ứng dụng Java có thể gồm nhiều lớp được tổ chức phân tán trên mạng.

(vi) Đảm bảo tương đối nhỏ và gọn nhẹ

+ Ý tưởng cơ sở của Java là ứng dụng có thể đặt ở trung tâm phục vụ (*máy chủ*) và sau đó được nạp về các máy trạm khi có yêu cầu để thực hiện công việc. Do vậy, các ứng dụng phải được tổ chức sao cho đủ nhỏ để việc trao đổi hiệu quả (nạp về nhanh) và dễ sử dụng với mọi người.
+ Chương trình Java có thể tổ chức nhỏ được vì chúng có thể tổ chức theo nguyên lý "đòn bẩy", nghĩa là các lớp có thể tổ chức thành các gói (*package*) và chúng sẽ được nạp về khi có nhu cầu.

(vii) Chuẩn hoá

+ Trong khi C/C++ là những ngôn ngữ chuẩn được xây dựng tốt, nhưng những bộ chương trình dịch khác nhau lại có thể cho những kết quả khác nhau. Ví dụ, một số bộ chương trình dịch qui định rằng hàm *main()* phải là hàm có giá trị trả lại thì một số khác lại cho phép gán mặc định kiểu *int*, hoặc các bộ chương trình dịch khác nhau cho phép dành số *byte* khác nhau cho các giá trị *int*.
+ Java là ngôn ngữ chuẩn và luôn cho cùng một dạng cài đặt.

□ Tổ chức các dự án ứng dụng với Java

Các tệp chương trình nguồn Java chỉ chứa đúng một loại là những mô tả về các lớp

mà chúng ta đã thiết lập, không có các biến toàn cục, không có hàm mẫu *(prototype)*.

Một số lớp có thể tổ chức thành các gói như là các hệ thống con. Khi một lớp muốn sử dụng những lớp được đặt trong một gói nào đó, ví dụ khi cần sử dụng các lớp để kết nối mạng, chúng ta phải nhập gói chứa chúng là gói *java.net* vào tệp chứa các lớp ứng dụng.

❑ Bộ JDK (Java Developer Kit)

Hiện nay có nhiều môi trường hỗ trợ để phát triển phần mềm với Java như: *Visual J++, Symatec's Cafe, Borland JBuilder, JDK, v.v.* Bộ *JDK* do *Sun* cung cấp thực hiện dựa chủ yếu trên các lệnh đơn được nhập vào từ các dòng lệnh và khá đơn giản, tiện lợi. JDK được sử dụng phổ biến còn bởi lẽ mọi người có thể dễ đọc được từ *Web Site:* http://java.sun.com [1]. Bộ JDK cung cấp các công cụ và các chương trình sau:

1. *javac* — Chương trình dịch chuyển mã nguồn sang mã *byte*.
2. *java* — Bộ thông dịch: Thực thi các ứng dụng độc lập, các tệp tin *.class* trực tiếp.
3. *appletviewer* — Bộ thông dịch: Thực thi các ứng dụng nhúng (*java applet*) từ tệp tin HTML mà không cần sử dụng trình duyệt như *Nestcape*, hay *Internet Explorer, v.v.*
4. *javadoc* — Bộ tạo tài liệu dạng HTML từ mã nguồn cùng các chú thích bên trong.
5. *jdb* — Bộ gỡ lỗi (*java debugger*) cho phép thực hiện từng dòng lệnh, đặt điểm dừng, xem giá trị của các biến, v.v.
6. *javah* — Bộ tạo lập *header* của C và cho phép chương trình C gọi các phương thức (*hàm*) của Java và ngược lại.
7. *javap* — Trình dịch ngược Assembler. Hiển thị các phương thức, dữ liệu truy nhập được bên trong của tệp tin *.class* đã được dịch và hiển thị nghĩa của *byte code*.

2.3. CÁC DẠNG CHƯƠNG TRÌNH ỨNG DỤNG CỦA JAVA

Có ba loại chương trình có thể phát triển với Java:

- Các chương trình ứng dụng độc lập,
- Các chương trình ứng dụng nhúng *(applet)*,
- Các chương trình kết hợp cả 2 loại trên.

Bởi vì Java là ngôn ngữ lập trình hướng đối tượng nên chương trình Java là tập của một hay nhiều lớp đối tượng. Mỗi tệp chương trình nguồn Java có thể chứa một hay nhiều định nghĩa lớp và tên của tệp phải trùng với tên của lớp công khai và có đuôi là "*.java*". Mỗi định nghĩa lớp trong tệp nguồn khi dịch sẽ được dịch và lưu trữ sang các tệp riêng trùng tên với tên lớp và có đuôi "*.class*". Tất cả các lớp phải được dịch trước khi thực hiện.

Để viết chương trình Java chúng ta có thể sử dụng những hệ soạn thảo phổ dụng như *NotePad, WordPad, TextPad, Jereator v.v.* Hệ soạn thảo đơn giản và thích hợp nhất có lẽ là TextPad. Bạn có thể nạp version mới nhất của TextPad từ địa chỉ của Web Site trên Internet: http://www.textpad.com, nạp *Jereator* từ http://www.jereator.com.

Sau đây chúng ta xét mẫu chung để phát triển các loại chương trình ứng dụng với Java.

2.3.1. CHƯƠNG TRÌNH ỨNG DỤNG ĐỘC LẬP

Chương trình ứng dụng độc lập là một chương trình theo nghĩa thông thường của chương trình: tệp chương trình nguồn mà sau khi dịch có thể thực hiện trực tiếp.

Để tạo lập một chương trình ứng dụng độc lập, chúng ta phải định nghĩa một lớp có chứa một phương thức đặc biệt tên là *main()*. Chương trình độc lập bắt đầu và kết thúc thực hiện ở *main()* giống như trong chương trình C/C++.

Khi xây dựng một ứng dụng độc lập cần lưu ý:

1. Tạo lập một lớp được định nghĩa bởi người sử dụng có phương thức *main()* gọi là lớp chính và đảm bảo nó được định nghĩa đúng theo mẫu qui định;
2. Kiểm tra xem liệu tệp chương trình có tên trùng với tên của lớp chính và đuôi "*.java*" hay không;
3. Dịch tệp chương trình để tạo ra các tệp mã *byte code* với các đuôi "*.class*" tương ứng;
4. Sử dụng chương trình thông dịch của Java để chạy chương trình đã dịch.

Ví dụ 2.1. Chương trình ứng dụng độc lập.

Bài toán: Xây dựng lớp CharStack là cấu trúc Stack và ứng dụng để đảo ngược các xâu ký tự.

```
// StandaloneApp.java
public class StandaloneApp{
  public static void main(String args[]){
```

```
  CharStack stack = new CharStack(80);// Tạo ra đối tượng stack
  String str = "maN teiV ,ioN aH";      // Tạo ra 1 xâu
  int leng = str.length();              // Số ký tự trong xâu
  System.out.println("Dao nguoc day ky tu");
  for(int k = 0; k < leng; k++)         // Đưa vào stack
     stack.push(str.charAt(k));
  while (!stack.isEmpty())  // Lấy ra từ stack theo thứ tự đảo ngược
     System.out.print(stack.pop());
  System.out.println();                 // Xuống dòng
  }
}
class CharStack {
   private char[] stackArray;           // Mảng các ký tự
   private int topOfStack;              // Đỉnh của stack
   private static int counter;          // (1)
   // Toán tử tạo lập đối tượng
   public CharStack(int capacity){      // (2)
     stackArray = new char[capacity];
     topOfStack = -1;
    counter++;
     }
    public void push(char e){stackArray[++topOfStack]= e;}
    public  char pop(){return stackArray[topOfStack--];}
    public  char peek(){return stackArray[topOfStack];}
    public boolean isEmpty() {return topOfStack < 0;}
    public boolean isFull() {
        return (topOfStack == stackArray.length - 1);}
  }
```

Tệp chương trình StandaloneApp có tên trùng với tên của lớp *StandaloneApp*, trong đó có hàm *main()*. Hàm này luôn có dạng:

```
  public static void main(String  args[]){
     // Nội dung thực hiện của chương trình
  }
```

Thuộc tính *public* của *main()* cho biết là hàm này được phép truy nhập đối với mọi lớp từ bên ngoài của lớp StandaloneApp và cho phép gọi để thực hiện bởi chương

trình thông dịch Java. Từ khóa *static* khai báo hàm này phụ thuộc vào cả lớp, không phụ thuộc vào đối tượng cụ thể, do vậy khi thực hiện không cần phải tạo lập đối tượng để gọi những hàm như thế. Từ *void* cho biết hàm *main()* không cần kết quả trả lại. Một điều cần lưu ý nữa là hàm *main()* có các đối số là mảng các xâu (đối tượng) args[] của *String*.

Trong chương trình trên còn định nghĩa lớp thứ hai là *CharStack*. Lớp này có các thuộc tính: mảng các ký tự *char[] stackArray*, biến kiểu *int topOfStack* và bộ đếm số ký tự *counter* trong xâu. Mỗi khi một đối tượng của lớp *CharStack* được tạo lập thì toán tử tạo lập (còn được gọi là cấu tử) tường minh (2) (được định nghĩa bởi người sử dụng) thực hiện để gán các giá trị cho các tham số theo yêu cầu của bài toán.

Dịch và thực hiện chương trình StandaloneApp.java

Chương trình nguồn Java có thể sử dụng chương trình dịch javac của bộ JDK:

```
javac StandaloneApp.java
```

Kết quả của lệnh dịch trên là hai tệp *StandaloneApp.class, CharStack.class* chứa mã *byte code* của hai lớp tương ứng. Những tệp lớp đã được dịch có thể thực hiện với chương trình thông dịch java như sau:

```
java StandaloneApp
```

Kết quả thực hiện của chương trình:

```
Ha Noi, Viet Nam
```

Lưu ý:

- ✓ Khi thực hiện với Java thì chỉ cần viết tên lớp chứa hàm *main()* và không cần đưa thêm đuôi *.class*,
- ✓ Khi soạn thảo chương trình nên tạo ra một thư mục riêng, ví dụ *c:\users\Lan* để ghi chương trình nguồn (StandaloneApp). Tất cả những tệp lớp (*.class*) đều được tạo ra ở thư mục chứa chương trình nguồn,
- ✓ Phải chỉ rõ thư mục chứa các chương trình dịch, thông dịch *javac.exe, java.exe*. Thường các chương trình này được cài đặt và được lưu ở thư mục, ví dụ *c:\jdk1.3\bin\java*.
- ✓ Mọi lớp trong Java đều mặc định xem là lớp con của lớp *Object* được xây dựng sẵn trong gói *java.lang* và gói này cũng được xem là mặc định sử dụng mà không cần nhập *(import)* vào như các gói khác khi sử dụng các lớp chứa trong đó.

2.3.2. CHƯƠNG TRÌNH ỨNG DỤNG NHÚNG APPLET

Applet là loại chương trình Java đặc biệt mà khi thực hiện phải được nhúng vào chương trình ứng dụng khác như các trình duyệt *Web Browser*, hoặc *appletviewer* của JDK. Cũng cần lưu ý là hầu hết các trình duyệt *Web Browser: Internet Explore* đều hỗ trợ để thực hiện *Java Applet*.

Sau đây chúng ta xây dựng một *applet* tên là AppletApp thực hiện Netscape đảo ngược dãy các ký tự như chương trình ứng dụng độc lập ở ví dụ 2.1.

Ví dụ 2.2. Java Applet

Bài toán: Nhiệm vụ tương tự như bài toán ở ví dụ 2.1 nhưng thực hiện theo applet

```
// AppletApp.java
import java.applet.Applet;        // Nhập thư viện chứa lớp Applet
import java.awt.Graphics;         // Nhập thư viện chứa lớp Graphics
// Mọi chương trình applet đều phải mở rộng (extends), kế thừa từ lớp Applet
public class AppletApp extends Applet{
    CharStack stack;
    String dayGoc, dayNguoc;
  // Nạp chồng lại hàm init() để thực hiện khi nạp và thực hiện applet
  public void init(){                    // (1)
      stack = new CharStack(80);
      dayGoc = new String("maN teiV ,ioN aH");
      int leng = dayGoc.length();        // Số ký tự trong xâu
  // Đặt các ký tự của dayGoc vào stack để sau đó lấy ra theo thứ tự ngược lại
      for(int k = 0; k < leng; k++)      // Đưa vào stack
          stack.push(dayGoc.charAt(k));
  // Đọc ra từ stack theo thứ tự đảo ngược và lưu vào dayNguoc
  dayNguoc = new String();               // Xâu mới rỗng
  while (!stack.isEmpty())
     dayNguoc += stack.pop();
     }

  // Nạp chồng hàm paint() để hiển thị (vẽ) các thông báo của applet
  public void paint(Graphics g){     // (2)
      g.drawString("Day ky tu dao nguoc:", 25, 25);
      g.drawString(dayNguoc, 25, 45);
      }
```

```
    }

class CharStack {
   private char[] stackArray;          // Mảng các ký tự
   private int topOfStack;             // Đỉnh của stack
   private static int counter;
   // Toán tử tạo lập đối tượng
   public CharStack(int capacity){
     stackArray = new char[capacity];
   topOfStack = -1;
   counter++;
    }

   public void push(char e){stackArray[++topOfStack]= e;}
   public  char pop(){return stackArray[topOfStack--];}
   public  char peek(){return stackArray[topOfStack];}
   public boolean isEmpty() {return topOfStack < 0;}
   public boolean isFull() {
      return (topOfStack == stackArray.length -1);}
}
```

Lớp *Applet* (tên đầy đủ *java.applet.Applet*) trong thư viện các lớp Java chuẩn (ở gói *java.applet*) cung cấp khuôn dạng và các chức năng chính để phát triển các *applet*. Lớp AppletApp mở rộng lớp Applet nên nó kế thừa tất cả các chức năng của Applet. Trong lớp *AppletApp* có hai hàm thành phần: *init(), paint()* được kế thừa từ lớp *Applet* nhưng được nạp chồng để thực hiện việc đảo ngược xâu và hiển thị thông tin của applet được tạo ra trên màn hình. Hai lệnh *import* ở đầu chương trình làm nhiệm vụ nhập hai gói thư viện Java chứa các lớp Applet và Graphics. Chú ý thấy ở đây hàm *main()* là không cần thiết đối với applet.

Trước khi đi tìm hiểu nguyên lý hoạt động của các *applet*, chúng ta hãy dịch và chạy thử ví dụ trên.

Dịch và chạy chương trình AppletApp

Dịch chương trình AppletApp thực hiện hoàn toàn giống như đối với chương trình ứng dụng độc lập ở ví dụ 2.1.

```
javac AppletApp.java
```

Kết quả chúng ta cũng có 2 tệp lớp AppletApp.class và CharStack.class.

Chương trình applet không thể thực hiện trực tiếp thông qua chương trình thông dịch java như đối với chương trình ứng dụng độc lập. Chương trình applet phải được đưa vào trang tư liệu định dạng HTML (HyperText Markup Language) để sau đó nạp được xuống thông qua Web Browser hoặc appletviewer của JDK.

Tệp HTML chứa các thông tin tương ứng về tên tệp lớp ứng dụng applet và những thông tin khác cần nạp applet xuống để thực hiện. Thông tin tối thiểu cần phải có đối với lớp applet là:

- Tên của tệp lớp applet,
- Kích cỡ của applet tính theo pixel

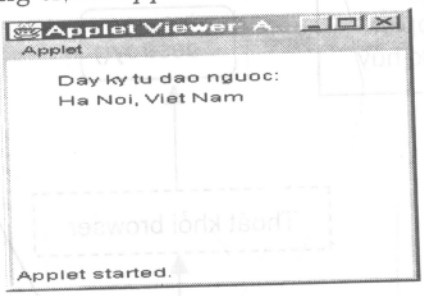
Ví dụ: <applet code = "AppletApp.class" width = 200 height = 300 >

</applet>

Tệp này được soạn bằng hệ soạn thảo bất kỳ và được ghi với tên gọi *AppletApp.html*. Tệp *AppletApp.html* có thể sử dụng Web Browser để chạy hoặc sử dụng *appletviewer* được JDK cung cấp như sau:

appletviewer AppletApp.html

Kết quả thực hiện của chương trình *applet*:



Hình 2.4. Thực hiện chương trình applet *với chương trình thông dịch* appletviewer.

Chu trình hoạt động của *applet*

Chương trình ứng dụng *applet* được thực hiện như sau:

- Khi một applet được nạp và chạy bởi Web Browser thì nó sẽ gửi thông điệp *init()* cùng với các dữ liệu, kích thước của Window để chương trình *applet* khởi động.
- Khi bắt đầu thực hiện, Web Browser thông báo cho *applet* bắt đầu bằng cách gọi phương thức *start()*.
- Khi rời khỏi trang Web có chứa *applet* thì chương trình *applet* này nhận được thông điệp *stop()* để dừng chương trình.

Hoạt động của chương trình *applet* được mô tả như trong hình 2.5, trong đó:

+ *init()*: Phương thức này được gọi khi *applet* được nạp lần đầu và được xem như là toán tử tạo lập cho applet,

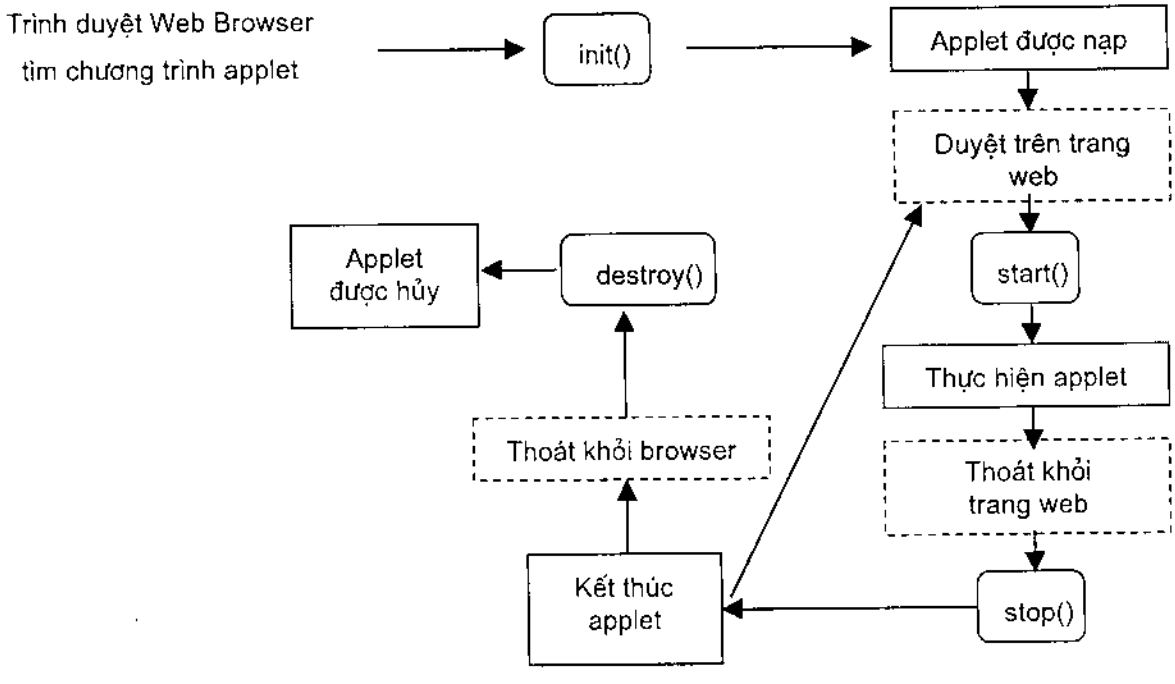
+ *start()*: Được gọi khi applet bắt đầu thực hiện, xuất hiện khi:

  - *applet* được nạp xuống,

  - *applet* được duyệt lại.

+ *stop()*: Được gọi khi *applet* dừng thực hiện, nhưng chưa bị xóa khỏi bộ nhớ,

+ *destroy()*: Được gọi ngay trước khi *applet* kết thúc, khi trình duyệt Web Browser tự đóng lại và *applet* bị xóa khỏi bộ nhớ.



*Hình 2.5. **Chu trình hoạt động của applet.***

2.3.3. CHƯƠNG TRÌNH ỨNG DỤNG Ở DẠNG APPLET LẪN DẠNG ĐỘC LẬP

Java cho phép xây dựng chương trình chạy được cả ở Web Browser lẫn như một ứng dụng độc lập. Một chương trình như thế phải:

- Định nghĩa lớp ứng dụng mở rộng, kế thừa từ lớp Applet,
- Trong lớp ứng dụng phải có hàm *main()*.

Ví dụ 2.3. Chương trình chạy được cả với Web Browser và chạy độc lập

```
import java.awt.Graphics;   // XinChao.java
import java.awt.Frame;
import java.applet.Applet;
public class XinChao extends Applet{
    public void init(){
      resize(200,160);        // Đặt lại kích thước
    }
    public void paint(Graphics g){
        g.drawString("Xin chao cac ban!", 60, 25);
    }
    // Hàm main() đảm bảo chương trình sẽ chạy được độc lập
    public static void main(String args[]){
        XinChao h = new XinChao();// Tạo ra một đối tượng của XinChao
        h.init();
        Frame f = new Frame("Chao Mung va Applet");
        f.resize(200, 160);
        f.add("Center", h);
        f.show();
    }
  }
```

Chương trình này sử dụng lớp Frame để tạo ra khung cỡ 200 × 160 khi nó chạy độc lập. Chi tiết hơn về đồ họa và các chức năng của *Frame* sẽ được đề cập chi tiết ở các chương VII.

Lưu ý: Tất nhiên để chương trình *XinChao* chạy được với Web Browser thì phải đưa vào tệp HTML (tệp *XinChao.html*) tương tự như ví dụ 2.2.

Tóm lại chương trình viết bằng Java có thể:

- Là các ứng dụng độc lập,
- Là các chương trình ứng dụng nhúng *applet*,
- Hoặc kết hợp cả hai loại trên.

Tuy nhiên giữa ứng dụng độc lập và *applet* của Java cũng có nhiều điểm khác nhau về cách khai báo và tổ chức thực hiện như trong bảng sau:

	Ứng dụng độc lập	Java Applet
Khai báo	Là lớp con của bất kỳ lớp nào trong các gói thư viện các lớp	Phải là lớp con của Applet
Giao diện đồ họa	Tùy chọn	Do trình duyệt Web quyết định
Yêu cầu bộ nhớ	Bộ nhớ tối thiểu	Bộ nhớ dành cho cả trình duyệt và applet đó
Cách nạp chương trình	Nạp bằng dòng lệnh	Thông qua trang Web
Dữ liệu vào	Thông qua các tham số trên dòng lệnh	Các tham số đặt trong tệp HTML gồm địa chỉ, kích thước của trình duyệt
Cách thức thực hiện	Mọi hoạt động được bắt đầu và kết thúc ở main() như trong C/C++	Gọi các hàm: init(), start(), stop(), destroy(), paint()
Kiểu ứng dụng	- Ứng dụng trên các máy chủ Server, - Công cụ phát triển phần mềm, - Ứng dụng trên các máy khách	Các ứng dụng trên Web

BÀI TẬP

2.1. Phân tích sự giống, khác nhau của quá trình dịch và chạy chương trình trong ngôn ngữ lập trình C/C++ và Java.

2.2. Nêu những đặc trưng cơ bản của ngôn ngữ Java, tại sao lại cần sử dụng Java để phát triển các chương trình ứng dụng.

2.3. Chương trình ứng dụng Java độc lập và các ứng dụng nhúng *applet* được thực hiện như thế nào?

2.4. Nêu cách sử dụng các chương trình của JDK để dịch và thực hiện các chương trình ứng dụng của Java.

2.5. Viết chương trình ứng dụng độc lập để hiển thị thông báo: "Xin moi cac ban lap trinh huong doi tuong voi Java!".

2.6. Viết chương trình ứng dụng *applet* để hiển thị thông báo: "Xin moi cac ban lap trinh huong doi tuong voi Java!".

2.7. Viết chương trình để thực hiện được cả ứng dụng độc lập lẫn ứng dụng *applet* để hiển thị thông báo: "Xin moi cac ban lap trinh huong doi tuong voi Java!".

2.8. So sánh sự giống, khác nhau của chương trình ứng dụng độc lập và ứng dụng nhúng *applet*.

CHƯƠNG III

CÁC THÀNH PHẦN CƠ SỞ CỦA JAVA

Chương III trình bày những thành phần cơ sở của ngôn ngữ lập trình Java, kiểu nguyên thủy, các phép toán và các biểu thức tính toán.

3.1. CÁC PHẦN TỬ CƠ SỞ CỦA JAVA

Giống như các ngôn ngữ lập trình khác, ngôn ngữ lập trình Java được định nghĩa bằng tập các qui tắc dẫn xuất của văn phạm cho phép kiểm tra xem các cấu trúc được xây dựng từ các phần tử của ngôn ngữ có hợp lệ (chính xác về mặt cú pháp) hay không và thông qua các định nghĩa về mặt ngữ nghĩa để kiểm chứng ý nghĩa của những cấu trúc đó.

Định danh (Tên gọi)

Tên gọi của các thành phần trong chương trình được gọi là *định danh* (*Identifier*). Định danh được sử dụng để xác định các phần tử như biến, kiểu, phương thức (*method*) hay còn gọi là hàm, đối tượng, lớp, v.v.

Trong Java định danh là một dãy các ký tự gồm các chữ cái, chữ số và một số các ký hiệu như: ký hiệu gạch dưới nối câu '_', các ký hiệu tiền tệ $, ¥, £, ¢, và không được bắt đầu bằng chữ số.

Lưu ý: Java phân biệt chữ thường và chữ hoa, ví dụ Hoa và hoa là hai định danh khác nhau. Độ dài (số ký tự) của định danh trong Java về lý thuyết là không bị giới hạn.

Ví dụ các định danh sau là hợp lệ: *so_nguyen, $My, danh_sach12, HocSinh*, còn những từ sau không phải là định danh *24gio, số_nguyên, Hoc-sinh*.

Xu thế chung hiện nay là nên sử dụng cách đặt tên theo một chuẩn nhất định để dễ phân biệt được các loại khác nhau của các thành phần sử dụng. Chúng ta qui ước cách đặt tên thống nhất như sau:

+ Định danh cho các lớp: *chữ cái đầu của mỗi từ trong định danh đều viết hoa*, ví

dụ MyClass, HocSinh, KhachHang là định danh của các lớp.

+ Định danh cho các biến, phương thức, đối tượng: *chữ cái đầu của mỗi từ trong định danh đều viết hoa trừ từ đầu tiên*, ví dụ myClass, hocSinh, khachHang là định danh của các đối tượng (biến) thuộc các lớp *MyClass, HocSinh, KhachHang* tương ứng.

Các từ khóa

Các từ khóa của một ngôn ngữ lập trình là những định danh định sẵn được định nghĩa trước của ngôn ngữ và không thể sử dụng cho những thực thể khác. Các từ khóa của Java có thể chia thành chín nhóm.

1. *Tổ chức các lớp*

package	Xác định một gói sẽ chứa một số lớp ở trong tệp nguồn.
import	Yêu cầu một hay một số lớp ở các gói chỉ định cần nhập vào để sử dụng trong ứng dụng hiện thời.

2. *Định nghĩa các lớp*

interface	Định nghĩa các biến, hằng, phương thức chung như là giao diện có thể chia sẻ chung giữa các lớp.
class	Định nghĩa tuyển tập các thuộc tính dữ liệu, phương thức mô tả các đặc tính và các hành vi của tập các đối tượng có quan hệ với nhau.
extends	Chỉ ra một lớp là mở rộng (kế thừa) của một lớp khác, hay còn gọi là lớp con của lớp cho trước.
implements	Xây dựng một lớp mới cài đặt những phương thức từ interface xác định trước.

3. *Các từ khóa cho các biến và các lớp*

abstract	Khai báo lớp trừu tượng không có thể hiện cụ thể.
public	Khai báo lớp, biến dữ liệu, phương thức công khai có thể truy nhập ở mọi nơi trong hệ thống.
private	Khai báo biến dữ liệu, phương thức riêng trong từng lớp và chỉ cho phép truy nhập trong lớp đó.
protected	Khai báo biến dữ liệu, phương thức được bảo vệ, cho phép truy nhập ở lớp chứa chúng và các lớp con của lớp đó.
static	Định nghĩa các biến, phương thức tĩnh của lớp, dùng chung cho tất cả các đối tượng trong một lớp.
synchronized	Chỉ ra là ở mỗi thời điểm chỉ có một đối tượng hoặc một lớp có

	thể truy nhập đến biến dữ liệu, hoặc phương thức loại đó, nghĩa là chúng được đồng bộ hóa.
volatile	Báo cho chương trình dịch biết là biến khai báo volatile có thể thay đổi dị bộ (tùy ý) trong các luồng (*thread*).
final	Chỉ ra các biến, phương thức không được thay đổi sau khi đã được định nghĩa.
native	Liên kết một phương thức với mã địa phương (*native code*), mã được viết trong những ngôn ngữ lập trình khác như C chẳng hạn.

4. *Các kiểu nguyên thủy (đơn giản)*

long	Kiểu số nguyên lớn với các giá trị chiếm 64 bit (8 byte).
int	Kiểu số nguyên với các giá trị chiếm 32 bit (4 byte).
short	Kiểu số nguyên ngắn với các giá trị chiếm 16 bit (2 byte).
byte	Kiểu byte với các giá trị nguyên chiếm 8 bit (1 byte).
char	Kiểu ký tự Unicode, mỗi ký tự chiếm 16 bit (2 byte).
float	Kiểu số thực với các giá trị biểu diễn theo dạng dấu phẩy động 32 bit.
double	Kiểu số thực chính xác gấp đôi với các giá trị biểu diễn theo dạng dấu phẩy động 64 bit (8 byte).
boolean	Kiểu logic với 2 trị: true, false.
void	Kiểu trống, sử dụng cho những hàm không trả lại giá trị.

5. *Những từ khóa cho các giá trị và các biến*

false	Giá trị kiểu boolean (sai).
true	Giá trị kiểu boolean (đúng).
this	Biến chỉ tới đối tượng hiện thời.
super	Biến chỉ tới đối tượng ở lớp cha.
null	Chỉ ra đối tượng không tồn tại.

6. *Xử lý ngoại lệ*

throw, throws	Bỏ qua một ngoại lệ để xử lý sau.
try	Thử thực hiện cho đến khi gặp một ngoại lệ.
catch	Đón nhận một ngoại lệ.
finally	Thực hiện một khối lệnh đến cùng bất chấp các ngoại lệ có thể xảy ra.

7. *Tạo lập và kiểm tra các đối tượng*

new	Tạo lập một đối tượng.
instanceof	Kiểm tra xem một đối tượng có nằm trong một lớp hay một *interface* hay không.

8. *Dòng điều khiển*

switch	Chuyển điều khiển chương trình theo các trường hợp ở *case*.
case	Trường hợp được tuyển chọn theo *switch*.
default	Trường hợp mặc định.
break	Thoát khỏi các chu trình.
if	Lệnh tuyển chọn theo điều kiện logic.
else	Rẽ nhánh theo điều kiện ngược lại của *if*.
continue	Tiếp tục thực hiện trong chu trình.
return	Trả lại giá trị cho các phương thức.
do	Thực hiện một chu trình.
while	Kết hợp với *do* để tạo ra một chu trình.
for	Chu trình *for* với các bước lặp thường là xác định.

9. *Những từ khóa chưa được sử dụng*

byvalue	*future*	*outer*
const	*genetic*	*rest*
goto	*inner*	*var*
cast	*operator*	

Chú thích (*Comment*)

Trong các chương trình thường cần phải có những đoạn tư liệu giải thích công việc hoặc cách thực hiện để người đọc hiểu và tiện theo dõi. Những phần giải thích đó chỉ nhằm làm tư liệu và được các bộ chương trình dịch bỏ qua. Java cung cấp 3 loại chú thích trong chương trình:

- Chú thích trên một dòng,
- Chú thích trên nhiều dòng,
- Chú thích trong tư liệu (javadoc).

Chú thích trên một dòng: Tất cả các ký tự sau // cho đến cuối dòng là chú thích.

```
// Từ đây đến cuối dòng là chú thích
```

Chú thích trên nhiều dòng: Giống như trong C, phần nằm giữa /* và */ là chú thích.

```
/*    Một chú thích
      ghi trên
      nhiều dòng
*/
```

Chú thích trong tư liệu: Đây là loại chú thích đặc biệt được đặt vào những chỗ thích hợp trong chương trình để javadoc có thể đọc và sử dụng để tạo ra tư liệu dạng HTML cho chương trình. Chúng được đặt vào trước phần định nghĩa các lớp, *interface*, phương thức và biến. Phần chú thích trong tư liệu được bắt đầu bằng /** và kết thúc bằng */.

```
/**
* Lớp này cài đặt giao diện Demo
* Tác giả: Trần An,
* Version 1.0
*/
```

Lưu ý: Không nên sử dụng các chú thích lồng nhau. Ví dụ

```
// ở đây /* có sự lồng nhau */ thì phần chú thích bên trong không xác định.
```

3.2. CÁC KIỂU DỮ LIỆU NGUYÊN THỦY

Mỗi ngôn ngữ lập trình đều định nghĩa sẵn một số kiểu dữ liệu cơ bản được gọi là kiểu nguyên thủy.

Các kiểu nguyên thủy của Java được chia thành 3 nhóm:

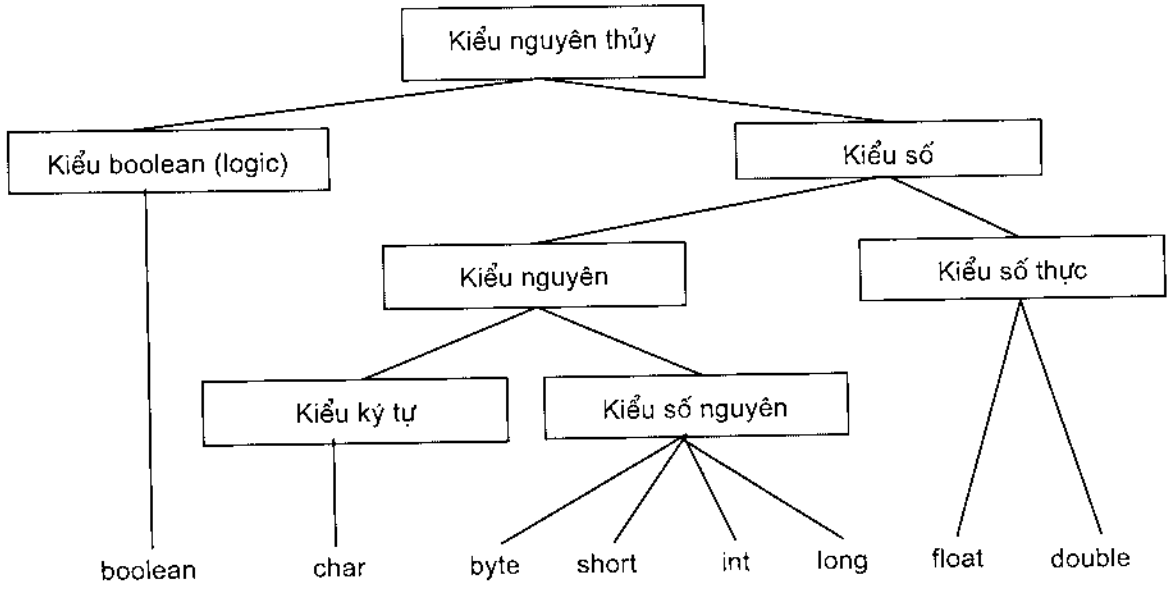
- Kiểu nguyên gồm các kiểu số nguyên và kiểu ký tự.

Các kiểu số nguyên gồm: *byte, short, int, long* biểu diễn cho các số nguyên.

Kiểu ký tự được thể hiện bằng kiểu *char*, biểu diễn cho các ký tự mã Unicode gồm các chữ cái, chữ số và các ký tự đặc biệt. Kiểu char có 65536 (2^{16}) ký tự trong tập mã Unicode 16 bit. Mã của 128 ký tự đầu của tập Unicode hoàn toàn trùng với mã của 128 ký tự trong tập mã ASCII 7-bit và mã của 256 ký tự đầu hoàn toàn tương ứng với 256 ký tự của tập mã ISO Latin-1 8 bit.

- *Kiểu dấu* phẩy động hay kiểu số thực: Loại này có hai kiểu float và double biểu diễn cho các số thập phân.
- Kiểu boolean: là kiểu *boolean* có hai giá trị *true* (đúng) và *false* (sai).

Hình 3.1 cho chúng ta bức tranh khái quát về các kiểu nguyên thủy của Java.



*Hình 3.1. **Các kiểu nguyên thủy trong Java.***

Lưu ý:

- ✓ Các giá trị dữ liệu kiểu nguyên thủy là nguyên tử (không phân chia được) và không phải là các đối tượng.
- ✓ Mỗi kiểu dữ liệu nguyên thủy có miền giá trị xác định và các phép toán được định nghĩa trên các giá trị đó như trong bảng 3.1.
- ✓ Mỗi kiểu nguyên thủy có một lớp bao bọc (Wrapper hay còn gọi lớp nguyên thủy) tương ứng để sử dụng các giá trị nguyên thủy như là các đối tượng. Ví dụ ứng với kiểu *int* có lớp *Integer*, ứng với *char* là *Char*, v.v.

3.3. KHAI BÁO CÁC BIẾN

Trong Java có bốn loại biến:

- *Các biến thành phần* là các thành phần của lớp và được khởi tạo giá trị mỗi khi một đối tượng của lớp được tạo ra. Nói cách khác, tất cả giá trị của các biến thành phần sẽ xác định trạng thái dữ liệu của một đối tượng.
- *Các biến tham chiếu đối tượng* (Object Reference) gọi tắt là biến tham chiếu và kiểu lớp còn được gọi là kiểu tham chiếu, là các biến được sử dụng để xử lý các đối tượng. Biến tham chiếu phải được khai báo và khởi tạo giá trị trước khi sử dụng.

```
HocSinh hs;    // Khai báo biến hs tham chiếu tới lớp HocSinh.
```

Lưu ý: khai báo như trên không hề tạo ra đối tượng của lớp *HocSinh*. Nó chỉ tạo ra biến *hs* để có thể sẵn sàng lưu trữ những thông tin về đối tượng của *HocSinh*. Muốn tạo ra đối tượng cho biến đó thường phải sử dụng toán tử *new*. Có thể kết hợp khai báo với việc khởi tạo giá trị cho các biến.

```
HocSinh hs = new HocSinh("Lan Anh"); //Khai báo kết hợp để khởi tạo giá trị
```

- *Các biến tĩnh (static)* cũng là các thành viên của lớp nhưng không phải đại diện cho từng đối tượng mà cho cả lớp.
- *Các biến cục bộ (local)* là những biến được khai báo trong các phương thức và trong các khối. Trong Java các biến local phải được khai báo trước khi sử dụng.

Bảng 3.1. Các giá trị của các kiểu nguyên thủy

Kiểu dữ liệu	Độ rộng (bits)	Giá trị cực tiểu	Giá trị cực đại
char	16	0x0	0xffff
byte	8	-128 (-2^7)	+127 (2^7-1)
short	16	-32768 (-2^{15})	32767 (2^{15}-1)
int	32	- 2^{31}, 0x80000000	+ 2^{31} - 1, 0x7fffffff
long	64	- 2^{63}	+ 2^{63} - 1
float	32	1.40129846432481707e-45	3.40282346638528860e+38
double	64	4.94065645841246544e-324	1.79769313486231570e+308

Biến lưu trữ các giá trị của các kiểu dữ liệu tương ứng. Mỗi biến có tên, kiểu dữ liệu và giá trị được gán. Việc khai báo biến là rất đơn giản cũng giống tương tự như trong C/C++: tên kiểu và sau đó khai báo các biến được cách nhau bởi dấu phẩy ','.

```
char b, c;
float giaBan, dienTich;
int  tuoi, soCongTrinh;
```

Khi khai báo cũng có thể kết hợp để khởi tạo các giá trị cho các biến.

```
int i = 10,
siSo  = 50;
float pi = 3.1415F;
```

3.4. KHỞI TẠO GIÁ TRỊ CHO CÁC BIẾN

Các giá trị mặc định cho các biến thành phần

Mỗi ngôn ngữ lập trình đều có qui định gán giá trị mặc định cho các biến khi đã được khai báo và định nghĩa. Giá trị mặc định cho các biến kiểu dữ liệu nguyên thủy trong Java được xác định như trong bảng 3.2.

Bảng 3.2. Các giá trị mặc định

Kiểu dữ liệu	Giá trị mặc định
boolean	false
char	'\u0000'
byte, short, int, long	0
float, double	+0.0F, +0.0D
Tham chiếu đối tượng	null

Lưu ý khi khai báo:

- ✓ Các biến tĩnh trong lớp luôn được khởi tạo với các giá trị mặc định nếu chúng không được gán giá trị tường minh .
- ✓ Các biến thành phần cũng được khởi tạo mặc định mỗi khi đối tượng của lớp có thành phần đó được khởi tạo nếu chúng không được giá trị tường minh.
- ✓ Biến tham chiếu được gán mặc định là null nếu không tạo lập tường minh theo toán tử *new* và toán tử tạo lập (*constructor*).

Ví dụ 3.1. // Light.java

```
class Light{
// Các biến tĩnh
static int boDem;      // Giá trị 0 được gán mặc định khi lớp được nạp
//Các biến thể hiện
int soWatts =100;  // Gán tường minh là 100
boolean bieuHien;  // Gán mặc định giá trị false
String viTri;      // Gán mặc định giá trị null
public static void main(String args[]){
  Light b1 = new Light();// Biến tham chiếu local
System.out.println("Bien tinh boDem: " +Light.counter);
    System.out.println("Bien thanh phan soWatts:" +b1.soWatts);
    System.out.println("Bien thanh phan bieuHien: " +b1.bieuHien);
```

```
        System.out.println(" Bien thanh phan viTri: " +bl.viTri);
    }
}
```

Lớp *Light* gồm biến thành phần tĩnh *boDem* kiểu *int* và các biến thành phần *soWatts* kiểu *int, bieuHien* kiểu *boolean* và biến tham chiếu *viTri* kiểu *String* (xâu ký tự). Lưu ý các biến tĩnh được khởi tạo giá trị khi lớp được nạp lần đầu tiên, còn các biến thành phần được khởi tạo giá trị mặc định, nếu không gán tường minh mỗi khi một đối tượng của lớp *Light* được tạo ra.

3.5. CẤU TRÚC TỆP CHƯƠNG TRÌNH JAVA

Tệp chương trình Java có thể có các phần được đặc tả như sau:

1. Định nghĩa một gói là tùy chọn thông qua định danh của gói (*package*). Tất cả các lớp, các *interface* được định nghĩa trong tệp chứa gói này đều thuộc gói đó. Nếu bỏ qua định nghĩa gói thì các định nghĩa ở tệp này sẽ thuộc vào gói mặc định.
2. Một số lệnh nhập *import* (0 hoặc nhiều).
3. Một số định nghĩa lớp và *interface* có thể định nghĩa theo thứ tự bất kỳ.

Cấu trúc chương trình Java vì vậy có thể khái quát như sau:

```
// Filename: NewApp.java
// Phần 1: Tùy chọn
// Định nghĩa gói
package GoiNhaTruong;

// Phần 2: (0 hoặc nhiều hơn)
// các gói cần sử dụng
import java.io.*;

// Phần 3: (0 hoặc nhiều hơn)
// Định nghĩa các lớp và các interface
public class NewApp{ ...}
class C1{...}
interface I1{...}
//
class Cn {...}
interface Im{...}
```

Lưu ý:

✓ Tệp chương trình Java luôn có tên trùng với tên của một lớp công khai (lớp chứa hàm *main()* nếu là ứng dụng độc lập) và có đuôi là .java.

✓ Tệp NewApp.java nếu là chương trình ứng dụng độc lập thì phải có một lớp có tên là NewApp và lớp này phải có phương thức *main()*. Phương thức này luôn có dạng:

```
public static void main(String args[]){
// Nội dung cần thực hiện của chương trình ứng dụng
}
```

Nội dung của phương thức *main()* xác định tất cả các công việc để hệ thống thực hiện.

✓ Khi dịch (javac) thì mỗi lớp trong tệp chương trình sẽ được dịch thành byte code và được ghi thành tệp riêng có tên trùng với tên của lớp và có đuôi .class. Những lớp này sẽ được nạp vào chương trình lúc thông dịch và thực hiện theo yêu cầu.

✓ Trong chương trình Java không có các khai báo biến, hàm tách biệt khỏi lớp và chỉ có các khai báo và định nghĩa các lớp, interface. *Như thế chương trình Java được xem như là tập các lớp, interface và các đối tượng của chúng trao đổi thông điệp với nhau (bằng các lời gọi hàm) để thực hiện các nhiệm vụ của ứng dụng.*

Ví dụ 3.2. Chương trình hiển thị các thông tin biết trước của sinh viên.

```
// Tệp Display.java
import java.io.*;          // Nhập vào gói java.io
class Display{             // Xây dựng lớp Display để hiển thị thông tin
  public static void main(String args[]){
      String hTen = "Lan Anh";   // Biến tham chiếu cục bộ được gán trị đầu
      int tuoi = 20;             // Biến cục bộ được gán trị đầu
      float diem = 8.5F;         // Biến cục bộ kiểu float được gán trị đầu số thực
      // Hiển thị các thông tin biết trước
      System.out.println("Ho va ten: " + hTen);
      System.out.println("Tuoi: " + tuoi);
      System.out.println("Diem thi: " + diem);
      }
  }
```

Tệp chương trình trên có hai phần: phần 2 và phần 3. Phần 3 có một lớp *Display* và lớp này có một hàm *main()*. Trong hàm *main()* có các biến cục bộ và sử dụng hàm *System.out.println("Ho va ten: " + hTen)* là phương thức thuộc lớp *System* được xây dựng sẵn ở gói java.io làm nhiệm vụ hiển thị xâu "Ho va ten: Lan Anh" là kết quả

của phép ghép ('+') hai xâu "Ho va ten: " với xâu trong biến tham chiếu *hTen* (đối tượng của lớp *String*) trên màn hình (thiết bị vào/ra chuẩn). Hàm *System.out.println()* hiển thị và xuống dòng.

Dịch chương trình *Display.java* với javac chúng ta được tệp *Display.class* và sau đó sử dụng *java* để thông dịch và hiển thị thông tin nêu trên.

Lưu ý: Các hằng số thực trong Java được xem là giá trị (mặc định) kiểu *double*.

```
float t = 3.14;     // Lỗi vì không tương thích kiểu
```

Giá trị 3.14 được mặc định là kiểu *double* và việc gán giá trị có độ chính xác gấp đôi cho biến kiểu *float* sẽ dẫn tới việc mất thông tin, do đó hệ thống sẽ thông báo lỗi. Vì thế, hoặc phải thông báo tường minh là số thực kiểu *float* (3.14F):

```
float t = 3.14F;
```

hoặc phải thực hiện ép sang kiểu (*float*)3.14,

```
float t = (float)3.14;
```

Ví dụ 3.3. Chương trình đọc vào 2 số và hiển thị giá trị trung bình của số đó.

```
// Tệp InputOutput.java
import java.io.*;
class InputOutput{
  public static void main(String args[]){
    float x, y;
    String str;
    // Đọc dữ liệu vào dưới dạng 1 xâu
    DataInputStream stream = new DataInputStream(System.in);
    // Hiển thị thông báo "x = " và không xuống dòng
    System.out.print("x = ");
    try{                                  // Thử thực hiện
        str = stream.readLine();  // Đọc vào 1 dãy các chữ số (xâu)
      }catch(IOException e){str="0.0";} // Đón nhận khi gặp ngoại lệ
    // Chuyển dãy các chữ số về số thực kiểu float
    try{
      // Chuyển dãy các chữ số sang giá trị kiểu float
      x = Float.valueOf(str).floatValue();
      }catch(NumberFormatException e){x = 0.0F;}
    System.out.print("y = ");
    try{
      str = stream.readLine();
      }catch(IOException e){str = "0.0F";}
    // Chuyển dãy các chữ số về số thực kiểu float
```

```
    try{
        y = Float.valueOf(str).floatValue();
        }catch(NumberFormatException e){y = 0.0F;}
    System.out.println("Gia tri trung binh cong cua " +
            + " va " + y + " la " + (x + y)/2);
    }
}
```

Vấn đề nhập dữ liệu vào từ bàn phím tương đối phức tạp trong chương trình Java vì mục đích chính của các ứng dụng Java là độc lập với các môi trường. Để đọc các giá trị số vào cho chương trình, chúng ta có nhiều cách khác nhau. Cách tương đối đơn giản là khai báo một biến tham chiếu thuộc lớp *String* và biến tham chiếu thuộc lớp *DataInputStream* có dạng như sau:

```
String str;
DataInputStream stream = new DataInputStream(System.in);
```

Vấn đề nhập dữ liệu vào từ bàn phím cũng thường gây ra nhiều ngoại lệ, như thay vì phải gõ các chữ số, chữ cái lại gõ vào ký hiệu điều khiển hay vượt phạm vi xác định chẳng hạn. Việc kiểm soát các ngoại lệ *vào/ra* trong Java có thể thực hiện theo lệnh:

```
try{
    str = stream.readLine();
}catch(IOException e){str = "0.0";}
```

Lệnh này thực hiện như sau: Đọc vào một dòng dữ liệu bằng phương thức *readLine()* của lớp *DataInputStream* và gán cho biến tham chiếu `str`. Lớp *IOException* làm nhiệm vụ xử lý các ngoại lệ *vào/ra*. Khi gặp ngoại lệ *vào/ra* thì gán mặc định cho `str` = "0.0". Câu lệnh try *{ ... }catch()* là lệnh xử lý ngoại lệ sẽ được trình bày chi tiết ở chương 5.

3.6. CÁC PHÉP TOÁN VÀ CÁC BIỂU THỨC

Nhiệm vụ của các chương trình phần lớn là thực hiện các phép tính toán để xử lý dữ liệu. Để lập trình đúng các yêu cầu tính toán, chúng ta phải nắm vững được các thứ tự ưu tiên cũng như các qui tắc kết hợp tính toán của các phép toán trong ngôn ngữ lập trình.

3.6.1. THỨ TỰ ƯU TIÊN VÀ QUI TẮC KẾT HỢP THỰC HIỆN CỦA CÁC PHÉP TOÁN

Mức ưu tiên và các qui tắc kết hợp của các phép toán cần phải được xác định để tính toán các biểu thức. Các mức ưu tiên và các qui tắc kết hợp của các phép toán trong

Java được xác định như trong bảng 3.3.

Khi sử dụng các phép toán chúng ta hãy lưu ý:

- ✓ Số ưu tiên là tỉ lệ nghịch với mức ưu tiên thực hiện trong các biểu thức. Trong biểu thức, những phép toán có số ưu tiên thấp hơn sẽ được thực hiện trước.
- ✓ Những phép toán có cùng số ưu tiên trong một biểu thức sẽ thực hiện theo qui tắc kết hợp từ trái qua phải hoặc từ phải qua trái như bảng 3.3.
- ✓ Để thay đổi thứ tự thực hiện theo ý muốn thì có thể sử dụng cặp ngoặc đơn (và) giống như trong toán học.

Bảng 3.3. Các phép toán trong Java

Số ưu tiên	Tên gọi	Các phép toán	Qui tắc kết hợp thực hiện
1	Phép toán 1 ngôi hậu tố (postfix)	[] . (tham_so) exp++ exp--	Thực hiện từ trái qua phải
2	Phép toán 1 ngôi tiền tố (prefix)	++exp --exp +exp -exp ~ !	Thực hiện từ phải qua trái
3	Tạo lập đối tượng và ép kiểu	new (type)	Thực hiện từ phải qua trái
4	Loại phép nhân	* / %	Thực hiện từ trái qua phải
5	Loại phép cộng	+ -	Thực hiện từ trái qua phải
6	Chuyển dịch	<< >> >>>	Thực hiện từ trái qua phải
7	Phép toán quan hệ	< <= > >= instanceof	Thực hiện từ trái qua phải
8	Phép so sánh đẳng thức	== !=	Thực hiện từ trái qua phải
9	Phép và (AND) trên bitwise/boolean	&	Thực hiện từ trái qua phải
10	Phép hoặc loại trừ (XOR) trên bitwise/boolean	^	Thực hiện từ trái qua phải
11	Phép hoặc (OR) trên bitwise/boolean	|	Thực hiện từ trái qua phải
12	Phép và (AND) logic	&&	Thực hiện từ trái qua phải
13	Phép hoặc (OR) logic	||	Thực hiện từ trái qua phải
14	Phép toán điều kiện	?:	Thực hiện từ trái qua phải
15	Các phép gán	= += -= *= /= %= <<= >>= >>>= &= ^= |=	Thực hiện từ phải qua trái

3.6.2. CÁC QUI TẮC CHUYỂN ĐỔI KIỂU

Ép kiểu

Java là ngôn ngữ rất chặt về kiểu, nó thường kiểm soát chặt chẽ sự tương thích giữa các kiểu trong các ngữ cảnh thực hiện ngay từ lúc dịch chương trình. Tuy nhiên, một số kiểm soát cũng chỉ thực hiện được lúc hệ thống thực hiện, ví dụ như kiểu *(lớp)* của các biến tham chiếu hay các trường hợp thực hiện các phép gán mà 2 toán hạng không tương thích với nhau về kiểu (ví dụ, gán giá trị kiểu *double* cho biến kiểu *int*). Trong các trường hợp như thế, người lập trình phải sử dụng qui tắc ép kiểu. Qui tắc ép kiểu có dạng:

(<type>) <exp>

Lúc thực hiện hệ thống sẽ chuyển kết quả tính toán của biểu thức <exp> sang kiểu được ép là <type>.

Ví dụ:

```
float f= (float) 100.15D; // Chuyển số 100.15 dạng kiểu double sang float
```

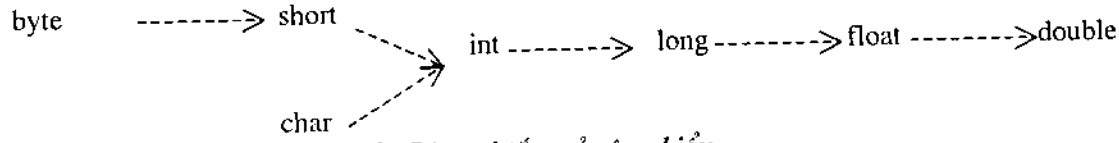
Lưu ý:

- ✓ Không cho phép chuyển đổi giữa các kiểu nguyên thủy với kiểu tham chiếu, ví dụ kiểu *double* không thể ép sang các kiểu lớp như *HocSinh* được.
- ✓ Kiểu giá trị *boolean* (logic) không thể chuyển sang các kiểu dữ liệu số và ngược lại.

Việc chuyển đổi kiểu theo qui tắc ép kiểu như trên được gọi là *chuyển đổi (ép) kiểu tường minh*. Java cũng giống như nhiều ngôn ngữ lập trình khác, ngoài cơ chế ép kiểu tường minh còn sử dụng cơ chế *ép kiểu không tường minh (mặc định)*, ví dụ có thể gán ký tự *char* cho biến kiểu *int*. Sau đây chúng ta xét hai cách chuyển đổi mở rộng và thu hẹp kiểu.

Mở rộng và thu hẹp kiểu

Đối với các kiểu dữ liệu nguyên thủy, giá trị của kiểu hẹp hơn (chiếm số byte ít hơn) có thể được chuyển sang những kiểu rộng hơn (chiếm số byte nhiều hơn) mà không tổn thất thông tin. Cách chuyển kiểu đó được gọi là *mở rộng kiểu*. Java cho phép thực hiện mở rộng kiểu theo các qui tắc được mô tả như trong hình 3.2.



*Hình 3.2. **Các qui tắc mở rộng kiểu.***

Ví dụ:

```
char c = 'A';
int k = c; // mở rộng kiểu char sang kiểu int (mặc định)
```

Chuyển đổi kiểu theo chiều ngược lại, từ kiểu rộng về kiểu hẹp hơn được gọi là *thu hẹp kiểu*. Lưu ý là *thu hẹp kiểu có thể dẫn tới mất thông tin*. Thực tế là mọi sự chuyển đổi kiểu được phép mà không phải là mở rộng kiểu như hình 3.2 thì đều là thu hẹp kiểu. Như vậy mọi chuyển đổi giữa *char* và hai kiểu *byte* và *short* cũng là thu hẹp kiểu. Ví dụ:

```
int  k = 10;
char c = (char) k; //Thu hẹp kiểu của k về kiểu của c
```

Thông thường việc thu hẹp kiểu phải sử dụng qui tắc ép kiểu tường minh như trên.

Cuối cùng chúng ta cũng cần lưu ý về ngữ cảnh phải thực hiện chuyển đổi kiểu:

- Thực hiện các phép gán đối với các biến kiểu nguyên thủy hoặc kiểu tham chiếu,
- Thực hiện các lời gọi hàm (phương thức) với các tham biến kiểu nguyên thủy hay kiểu tham chiếu,
- Thực hiện tính toán các biểu thức số học,
- Ghép các xâu kết hợp các đối tượng của lớp String và các kiểu dữ liệu khác.

Sau đây chúng ta xét một số nhóm các phép toán quan trọng trong các biểu thức.

3.6.3. CÁC PHÉP TOÁN SỐ HỌC

Các phép toán số học được chia thành hai loại:

1/ Các phép toán 1 ngôi *(đơn nguyên)*: + *(cộng)* và - *(trừ)*, các phép đổi dấu,

2/ Các phép toán 2 ngôi (nhị nguyên): * (nhân), / (chia), % (lấy modul - phép chia lấy số dư), + (cộng) và - (trừ).

Các phép toán *(toán tử)* số học được sử dụng để thiết lập các biểu thức toán học như trong đại số. Các toán hạng *(đối số)* của các phép toán là các biểu thức có giá trị kiểu số. Riêng phép + *(cộng)* còn được sử dụng nạp chồng để ghép các xâu nếu có một toán hạng là đối tượng của lớp *String*.

Lưu ý:

✓ Thứ tự kết hợp thực hiện của các phép toán số học đơn nguyên là từ phải qua trái:

```
int  val = -  -20;    // (- (-20)) cho 20
```

Chú ý giữa hai phép toán đơn nguyên là phải có dấu cách.

✓ Thứ tự kết hợp thực hiện của các phép toán số học nhị nguyên là từ trái qua phải:

```
int  newVal = 10 % 4 * 4;     // ((10 % 4) * 4) cho 8
int  iVal = newVal / 5;       // Thực hiện chia nguyên và cho kết quả là 1
```

✓ Lúc thực hiện, các toán hạng phải được tính toán từ trái qua phải trước khi áp dụng với phép toán. Khi 2 toán hạng khác nhau về kiểu thì thực hiện chuyển đổi kiểu như trên đã đề cập.

✓ Phép chia nguyên (2 toán hạng đều là kiểu nguyên) đòi hỏi số chia phải khác 0.

```
int iV = 10 / 0;      // Sinh lỗi ngoại lệ số học: ArithmeticException
```

✓ Phép chia số thực (ít nhất một toán hạng kiểu số thực) cho phép chia cho 0 và kết quả phép chia cho 0.0 là *INF* (số lớn vô cùng) hoặc *-INF* (số âm vô cùng), hai hằng đặc biệt trong Java.

```
float n, m = 4.5 / 0.0;      // Cho m là INF
      n = -4.5 /0.0;         // Cho n là -INF
```

✓ Trong Java, phép chia lấy số dư % thực hiện được cả đối với số thực.

```
float  m = 11.5 % 2.5;       // Cho m là 1.5
```

✓ Khi sử dụng các phép toán đơn nguyên đối với các đối số kiểu byte, short, hoặc char thì trước tiên toán hạng phải được tính rồi chuyển về kiểu int và kết quả là kiểu int. Ví dụ:

```
byte b1, b = 4;      // OK  số 4 kiểu int nhưng cho phép thu hẹp kiểu mặc định về byte
b = 1 - b;           // Lỗi vì 1 - b có kết quả kiểu int do vậy thu hẹp kiểu đòi
                     //  hỏi phải tường minh
b1 = (byte) (1 - b); //  Hoàn toàn đúng
short h = 30;        // OK: 30 kiểu int chuyển về short (mặc định đối với hằng nguyên
h = h + 4;           // Lỗi vì h + 4 cho kết quả kiểu int vì thế không thể gán trực tiếp
                 và  // cho h kiểu short. Nhưng,
h = (short) (h+4);   // lại đúng hoặc có thể viết h = h + (short)4;
```

3.6.4. CÁC PHÉP CHUYỂN DỊCH <<, >>, >>>

Các phép chuyển dịch <<, >>, >>> thực hiện dịch dạng biểu diễn nhị phân của toán hạng thứ nhất sang trái, sang phải số lần bằng giá trị số nguyên của toán hạng thứ hai. *Các đối số của chúng luôn là kiểu nguyên.*

Biểu diễn nhị phân của các số nguyên

Trước tiên chúng ta lưu ý rằng các giá trị nguyên biểu diễn cho cả giá trị dương và giá trị âm. Java sử dụng *phần bù 2 để lưu trữ các giá trị nguyên.*

Các bit trong dạng biểu diễn nhị phân của giá trị nguyên (kiểu *char, short, int, long*) được đánh số từ phải qua trái và bắt đầu bằng 0. Bit cao nhất là bit dấu, 0 cho số dương và 1 cho số âm.

Ví dụ: Bảng 3.4 cho một số biểu diễn phần bù 2 của các số chiếm 1 byte.

Lưu ý: Cách tính phần bù 2 thực hiện như sau:

Cho trước giá trị nguyên dương, ví dụ 41. Biểu diễn nhị phân của -41 được tính như sau:

	Biểu diễn nhị phân	Giá trị thập phân
Cho trước giá trị	00101001	41
Lấy phần bù 1	11010110	
Cộng thêm 1	00000001	
Kết quả là phần bù 2	11010111	-41

Tương tự như vậy đối với số âm, lấy phần bù 2 của số âm sẽ được số dương.

Bảng 3.4. Biểu diễn nhị phân của các số chiếm 1 byte

Giá trị thập phân	Biểu diễn nhị phân 8 bit	Giá trị Hexa
127	01111111	0x7f
126	01111110	0x7e
...	...	...
41	00101001	0x29
...	...	...
2	00000010	0x02
1	00000001	0x01
0	00000000	0x0
-1	11111111	0xff
-2	11111110	0xfe
...	...	...
-41	11010111	0xd7
...	...	...
-127	10000001	0x81
-128	10000000	0x80

Phép dịch trái: <<

a << n Dịch tất cả các bit của *a* sang trái *n* lần, điền số 0 vào bên phải.

Ví dụ: int i = 12;

int re = i << 4; //192

Java sử dụng 4 byte để lưu trữ số nguyên do vậy

12 << 4 = 00000000 00000000 00000000 00001100 << 4

= 00000000 00000000 00000000 11000000 = 192

Có thể nhận thấy mỗi lần dịch sang trái một vị trí thì tương đương với việc lấy giá trị trước đó nhân với 2 (với điều kiện bit cao nhất dịch ra ngoài phải là 0). Trong ví dụ trên re = 12 * 2^4 = 192.

Khi đối số bên trái có kiểu *byte* hoặc *short* thì sẽ được chuyển sang kiểu tương ứng của đối số thứ hai. Nếu đối số thứ hai là *int* thì bit dấu của giá trị *byte, short* sẽ được mở rộng để điền vào những bit cao hơn, ví dụ

byte b = -42; // 11010110

int re = b << 4; // -672

Khi giá trị byte là -42 được chuyển sang *int* thì bit dấu 1 sẽ được điền vào các bit cao hơn như sau:

b << 4 = 11111111 11111111 11111111 11010110 << 4

= 11111111 11111111 11111101 01100000 = 0xfffffd60 = -672

= -42 * 2^4

Phép dịch phải và điền bit dấu: >>

a >> n Dịch tất cả các bit của *a* sang phải *n* lần, điền bit dấu vào bên trái.

Ví dụ: int i = 12;

int re = i >> 2; // 3

Java sử dụng 4 byte để lưu trữ số nguyên do vậy

12 << 4 = 00000000 00000000 00000000 00001100 >> 2

= 00000000 00000000 00000000 00000011 = 0x00000003 = 3

Mặt khác có thể nhận thấy mỗi lần dịch sang phải 1 vị trí thì tương đương với việc lấy giá trị trước đó chia cho hai (chia đôi và làm tròn dưới). Trong ví dụ trên: re = 12 / 2^2 = 3.

Khi đối số bên trái có kiểu *byte* hoặc *short* thì sẽ được chuyển sang kiểu tương ứng của đối số thứ 2. Nếu đối số thứ hai là *int* thì bit dấu của giá trị *byte, short* sẽ được mở rộng để điền vào những bit cao hơn, ví dụ

```
byte b = -42;        // 11010110
int re = b >> 4;     // -3
```

Khi giá trị byte là -42 được chuyển sang *int* thì bit dấu 1 sẽ được điền vào các bit cao hơn như sau:

```
b >> 4   = 11111111 11111111 11111111 11010110 >> 4
         = 11111111 11111111 11111111 11111101 = 0xfffffffd  = -3
```

Phép dịch phải và điền bit 0: >>>

a >>> n Dịch tất cả các bit của *a* sang phải *n* lần, điền 0 vào bên trái.

Ví dụ:
```
byte b = -42;        // 11010110
int re = b >>> 4;    // 268435453 - khác kết quả trên rất nhiều
```

Khi giá trị byte là -42 được chuyển sang int thì bit dấu 1 sẽ được điền vào các bit cao hơn như sau:

```
b >>> 4 = 11111111 11111111 11111111 11010110 >> 4
        = 00001111 11111111 11111111 11111101 = 0x0fffffffd  = 268435453
```

Lưu ý:

- ✓ Giá trị của đối số bên phải (ở trên là n) luôn là số nguyên (dương) do vậy đối số bên trái (ở trên là a) nếu là *byte* hoặc *short* thì phải đổi sang kiểu *int*.
- ✓ Kết quả của các phép chuyển dịch vì vậy sẽ luôn là *int* (hoặc là kiểu *long* nếu đối số thứ nhất là *long*).

3.6.5. CÁC PHÉP GÁN MỞ RỘNG

Các phép toán gán mở rộng có cú pháp dạng:

<var> <op>= <exp>;

trong đó <var> là biến, <op> là một trong số các phép toán: +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, |= và <exp> là biểu thức tương ứng với các phép toán đã chọn.

Các phép gán số học mở rộng

Đối với các phép gán số học mở rộng thì qui tắc cú pháp trên tương đương ngữ nghĩa với lệnh sau:

<var> = (<type>) (<var> <op> (<exp>));

trong đó <type> là các kiểu số và <op> là phép toán số học +, -, *, /, %. Bảng 3.5 mô tả chi tiết hơn các phép gán số học mở rộng.

Bảng 3.5. Các phép gán số học mở rộng

Câu lệnh gán	Cho trước kiểu số T của biến x và biểu thức e
x += e;	x = (T) (x + (e));
x -= e;	x = (T) (x - (e));
x *= e;	x = (T) (x * (e));
x /= e;	x = (T) (x / (e));
x %= e;	x = (T) (x % (e));

Do qui định các phép gán số học mở rộng có dạng tương đương về mặt ngữ nghĩa như trên nên thực chất là đã có sự ép kiểu giữa các kết quả của biểu thức ở vế phải về kiểu T của biến x. Ví dụ:

```
int  i = 5;
i *= i + 2;      // Tính như là: i = (int) (i * (i+2));
byte b = 3;      // Các hằng nguyên cho phép thu hẹp kiểu như mặc định
b += 4;          // Tính như là : b = (byte) ((int) b + 4);
b = b + 4;       // Sai vì vế phải có kết quả kiểu int còn b kiểu byte đòi ép kiểu
```

Các phép gán logic mở rộng

Các phép gán logic mở rộng được định nghĩa chi tiết như trong bảng 3.6.

Bảng 3.6. Các phép gán logic mở rộng

Các lệnh gán	Cho b và biểu thức e kiểu boolean
b &= e;	b = (b & (e));
b ^= e;	b = (b ^ (e));
b |= e;	b = (b | (e));

Ví dụ:

```
boolean b1 = false, b2 = false, b3 = true;
b3 &= b3 & b1 | b2;        // false vì b3 = ((b3 & b1) | b2);
```

Các phép gán mở rộng trên bit (bitwise)

Các phép gán mở rộng trên bit có dạng ngữ nghĩa tương tự như các phép gán số học mở rộng, nghĩa là kết quả của biểu thức bên phải được ép về kiểu của biến nhận kết quả (vế trái). Những phép này được định nghĩa như trong bảng 3.7.

Bảng 3.7. Các phép gán mở rộng trên bit

Các lệnh gán	Cho T là kiểu nguyên của b và biểu thức e
b &= e;	b = (T) (b & (e));
b ^= e;	b = (T) (b ^ (e));
b |= e;	b = (T) (b | (e));

Ví dụ:

```
int v0 = - 42;
char v1 = ')';  // 41
byte v2 = 13;
v0 &= 15;  // 1...1101 0110 & 0...0000 1111 = 0...0000 0110 (=6)
```

Đối với những phép gán mở rộng trên bit được qui định như trong bảng 3.7, do vậy không cần ép kiểu khi thu hẹp kiểu.

Các phép gán chuyển dịch mở rộng

Các phép gán dịch chuyển mở rộng: <<=, >>=, >>>= được định nghĩa như trong bảng 3.8.

Bảng 3.8. Các phép gán dịch chuyển mở rộng

Các lệnh gán	Cho T là kiểu nguyên của b và biểu thức e
b <<= e;	b = (T) (b << (e));
b >>= e;	b = (T) (b >> (e));
b >>>= e;	b = (T) (b >>> (e));

Ví dụ:

```
int i = -42;  // Biểu diễn -42 ở dạng nhị phân phần bù 2: 1...11010110
```

```
i >>= 4;      // 1...11010110 >> 4 ⇒ 1...11111101 (= -3);
byte a = 12;
a <<= 5;      // Cho a = -128 vì a = (byte) ((int) a << 5)
a = a << 5;   // Sai bởi a << 5 là kiểu int do vậy đòi hỏi ép kiểu tường minh.
```

3.6.6. CÁC PHÉP TOÁN SO SÁNH ĐẲNG THỨC

Các phép toán so sánh <, <=, >, >= của Java được định nghĩa giống như trong các ngôn ngữ lập trình khác (như C chẳng hạn). Trong đó chỉ cần lưu ý là *2 đối số phải là các biểu thức số*. Riêng phép so sánh đẳng thức (bằng) thì phức tạp hơn. Sau đây chúng ta xét các dạng so sánh đẳng thức trong Java.

So sánh đẳng thức trên các giá trị kiểu nguyên thủy: ==, !=

Cho trước hai toán hạng a, b có kiểu dữ liệu nguyên thủy.

a == b *a* và *b* có bằng nhau không?, nghĩa là nếu chúng có các giá trị kiểu nguyên thủy bằng nhau thì cho kết quả đúng (true), ngược lại cho sai (false).

a != b *a* và *b* có khác nhau không?, nghĩa là nếu chúng có các giá trị kiểu nguyên thủy không bằng nhau thì cho kết quả đúng (true), ngược lại cho sai (false).

Lưu ý: Kiểu các đối số có thể là các kiểu số hoặc kiểu *boolean*, nhưng 2 đối số phải luôn có kiểu tương thích và sánh được với nhau. Ví dụ (<boolean value> == *c*) sẽ không hợp lệ nếu *c* có kiểu số. Tuy nhiên nếu biểu thức có nhiều phép == kết hợp thì phải thận trọng và căn cứ vào thứ tự thực hiện vì có thể tất cả các toán hạng đều có kiểu số nhưng biểu thức vẫn không hợp lệ, hoặc ngược lại khi các đối số có kiểu khác nhau nhưng vẫn hợp lệ. Ví dụ:

```
int a, b, c;
a = b = c = 10;
boolean hl = a == b == c; // Sai bởi phải thực hiện a == b cho giá trị true,
                          // sau đó so với c (giá trị số) thì 2 đối số khác kiểu.
boolean hl1 = a == b == true; // Lại hợp lệ.
```

So sánh đẳng thức trên các tham chiếu đối tượng: ==, !=

Cho trước *r* và *s* là hai biến tham chiếu. Các phép so sánh đẳng thức trên các biến tham chiếu được xác định như sau:

r == s — Cho giá trị true nếu *r, s* cùng tham chiếu tới cùng một trị (đối tượng), ngược lại sẽ cho giá trị *false*.

r != s — Cho giá trị true nếu *r, s* không cùng tham chiếu tới cùng một trị (đối tượng), ngược lại sẽ cho giá trị *false*.

Hai phép toán này được sử dụng để kiểm tra xem hai biến có chỉ tới cùng một đối tượng hay không.

Lưu ý: Các toán hạng phải có kiểu tương thích (chuyển đổi giữa chúng được mặc định), nếu không thì phải thực hiện ép kiểu.

Ví dụ 3.4. So sánh qua tham chiếu

```
    class SV{
String hTen;
int tuoi;
SV(String ht){      // Toán tử tạo lập đối tượng mới
hTen = ht;
}
public static void main(String[] args){
    SV sv1 = new SV("Lan Anh"); // Tạo ra 1 đối tượng mới
    SV sv2 = new SV("Lan Anh"); // Tạo ra 1 đối tượng mới
    SV sv3 = new SV("Tran Anh");      // Tạo ra 1 đối tượng mới
    boolean t1 = sv1 == sv2;     // false
    boolean t2 = sv1 == sv3;     // false
    SV sv4 = sv2;
    boolean t3 = sv4 == sv2;     // true
    System.out.println(" t1 = " + t1);
    System.out.println(" t2 = " + t2);
    System.out.println(" t3 = " + t3);
    }
}
```

3.7. TRUYỀN THAM SỐ VÀ CÁC LỜI GỌI HÀM

Như phần đầu chúng ta đã đề cập, các đối tượng trong chương trình trao đổi với nhau bằng cách trao đổi các thông điệp *(message). Một thông điệp được cài đặt như là lời gọi hàm* (phương thức) trong chương trình, gọi tới hàm thành phần của đối tượng đối tác. Những hàm tĩnh có thể gọi với tên của lớp. Các tham số trong các lời gọi hàm cung cấp cách thức trao đổi thông tin giữa đối tượng gửi và đối tượng nhận

thông điệp.

Cú pháp các lời gọi hàm có các dạng sau:

<Tham chiếu đối tượng >. <Tên hàm> (<danh sách tham biến hiện thời>)

<Tên lớp>. <Tên hàm tĩnh> (<danh sách tham biến hiện thời>)

<Tên hàm> (<danh sách tham biến hiện thời>)

Ví dụ:

```
hinhTron.ve();                  // hinhTron là đối tượng của lớp HinhTron
int i = java.lang.Math.abs(-4); // Gọi đầy đủ tên của lớp Math
int j = Math.abs(-4);           // Gọi theo tên của lớp Math
someMethod(ofValue);            // Đối tượng hoặc lớp không tường minh
```

Trong lập trình, chúng ta đã quen với khái niệm *danh sách tham biến hình thức* là danh sách các tham biến được định nghĩa trong định nghĩa các hàm. *Danh sách tham biến hiện thời* là danh sách tham biến được truyền vào cho các lời gọi hàm tương ứng. Các danh sách này có thể là rỗng. Hai danh sách hình thức và hiện thời phải tương thích với nhau:

- ✓ Số các tham biến của danh sách hình thức phải bằng số các tham biến của danh sách hiện thời.
- ✓ Kiểu của các tham biến hiện thời phải tương thích với kiểu của tham biến hình thức tương ứng. Bảng 3.9 tóm tắt cách truyền các giá trị phụ thuộc vào kiểu của các tham biến hình thức.

Bảng 3.9. Truyền tham số

Kiểu của tham biến hình thức	Giá trị được truyền
Các kiểu nguyên thủy	Giá trị kiểu nguyên thủy
Kiểu lớp (class)	Giá trị tham chiếu
Kiểu mảng (array)	Giá trị tham chiếu

Lưu ý: Trong Java, mọi tham biến đều được truyền theo tham trị (*passed by value*).

Truyền các giá trị kiểu nguyên thủy

Khi các tham biến hình thức có kiểu là nguyên thủy thì giá trị của các biến được sao sang biến hình thức trong các lời gọi hàm. Bởi vì *các biến hình thức là cục bộ trong định nghĩa của một hàm* nên mọi thay đổi của biến hình thức không ảnh hưởng đến

các tham biến hiện thời.

Các tham biến có thể là các biểu thức và chúng phải được tính trước khi truyền vào lời gọi hàm.

Qui ước chuyển đổi kiểu giữa tham biến hình thức và hiện thời cũng giống như đã được thảo luận ở 3.6.2.

Ví dụ 3.5. Truyền các giá trị nguyên thủy

```
class KhachHang1{ // Lớp khách hàng
public static void main(String[] arg){
    HangSX banh = new HangSX();          // Tạo ra một đối tượng
    int giaBan = 20;
    double tien = banh.tinh(10,giaBan);
    System.out.println("Gia ban: " + giaBan);// giaBan không đổi
    System.out.println("Tien ban duoc : " + tien);
    }
}
// Lớp Hãng sản xuất
class HangSX{
  double tinh(int num, double gia){
    gia = gia /2;
      return num * gia;     // Thay đổi gia nhưng không ảnh hưởng tới giaBan,
                            // nhưng số tiền vẫn bị thay đổi theo
  }
}
```

Khi thực hiện chương trình sẽ cho kết quả:

Gia ban: 20

Tien ban duoc: 100.0

Trong hàm *main()* của lớp *KhachHang* thực hiện:

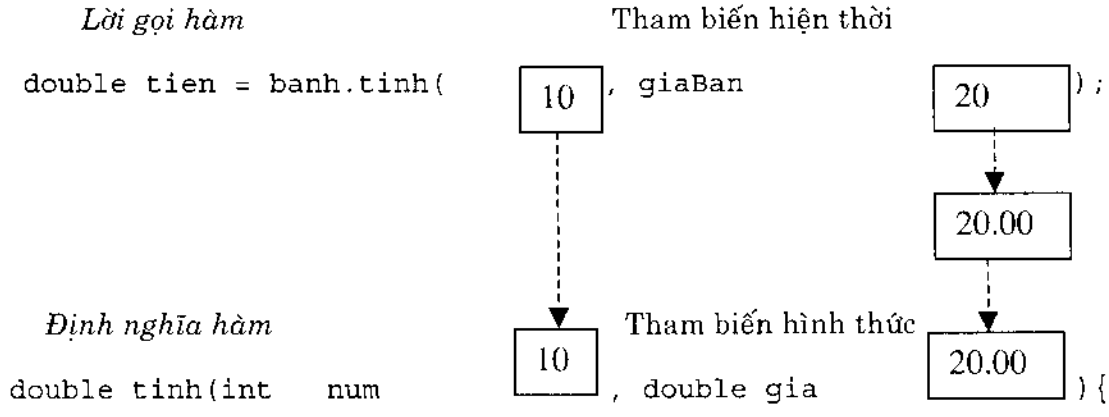
+ Tạo ra một biến đối tượng *banh* của lớp HangSX,

+ Định nghĩa biến thành phần *giaBan* = 20,

+ Gọi hàm thành phần *tinh()* phần của *banh* để tính số tiền bán được 10 chiếc bánh.

Mặc dù trong hàm *tinh()* của lớp *HangSX* đã thay đổi tham số hình thức *gia* nhưng khi gọi với tham số hiện thời *giaBan* thì việc thay đổi trên không ảnh hưởng tới giá trị của *giaBan*.

Chúng ta cũng cần lưu ý là kiểu của tham biến hiện thời thứ 2 của hàm *tinh()* là *int*, trong khi kiểu của biến hình thức tương ứng là *double*. Trước khi giá trị của biến hiện thời được sao cho biến hình thức thì giá trị (số 20) kiểu *int* phải được mở rộng sang *double* theo mặc định. Cơ chế truyền tham biến đối với các giá trị nguyên thủy có thể minh hoạ như sau:



Hình 3.3. mô tả cơ chế truyền tham biến đối với các trị nguyên thủy.

Truyền các giá trị tham chiếu đối tượng

Khi biến hiện thời tham chiếu tới đối tượng, thì giá trị tham chiếu của đối tượng sẽ được truyền cho biến hình thức. Nghĩa là cả biến hình thức và biến hiện thời là hai tên gọi khác nhau (bí danh) của đối tượng được tham chiếu tới trong lời gọi hàm. Do vậy, mọi thay đổi thực hiện đối với các thành phần của đối tượng thông qua tham biến hình thức cũng sẽ có hiệu quả cả sau lời gọi hàm và tác động lên biến hiện thời.

Ví dụ 3.6. Truyền theo giá trị tham chiếu

```
//KhachHang2.java
class KhachHang2{  // Lớp khách hàng
  public static void main(String[] arg){
  Banh banhMoi = new Banh();          // Tạo ra một đối tượng   (1)
  System.out.println("Nhoi thit vao banh truoc khi nuong:" + banhMoi.thit);
  nuong(banhMoi);                     //            (2)
  System.out.println("Thit cua banh sau khi nuong:"+banhMoi.thit);
  }
public static void nuong(Banh banhNuong){          // (3)
  banhNuong.thit = "Thit vit";        // Người nướng bánh đổi nhân thành thịt vit
```

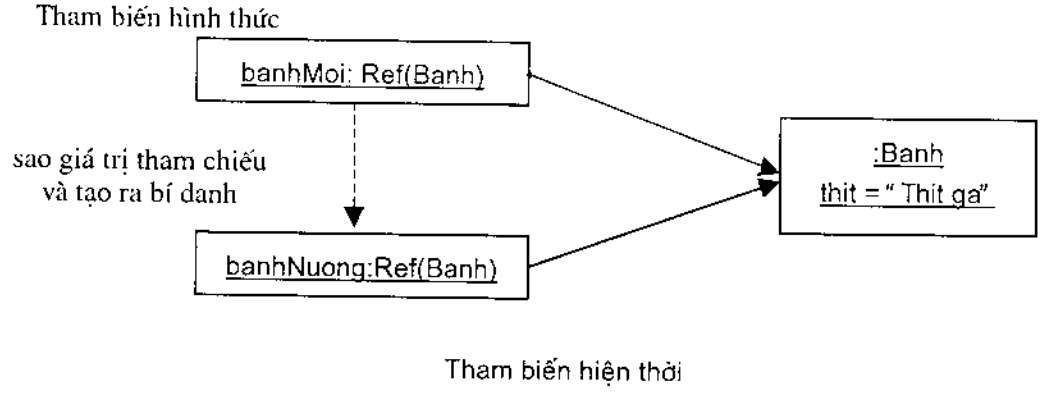
```
        banhNuong = null;                    // (4)
        }
    }
class Banh{                                  // Lớp Banh      (5)
            String thit = "Thit ga"; // Qui định của hãng làm nhân bánh bằng thịt gà
        }
```

Khi thực hiện chương trình sẽ cho kết quả:

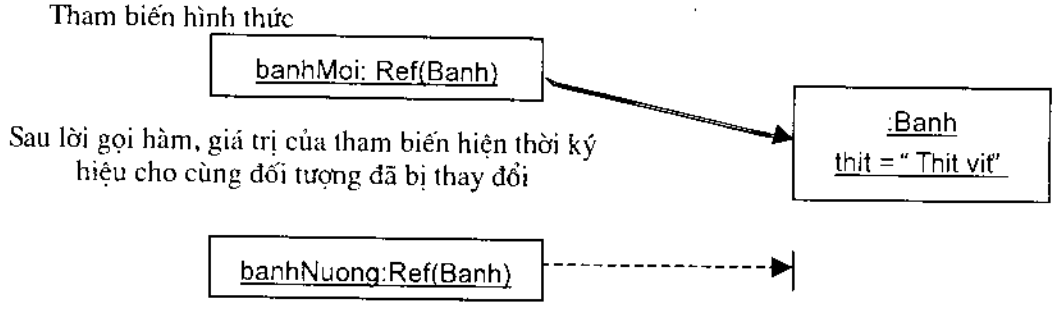
Nhoi thit vao banh truoc khi nuong: Thit ga

Thịt cua banh sau khi nuong: Thit vit

Một đối tượng *banhMoi* được tạo ra ở (1) với nhân nhồi "Thit ga" như qui định ở (5). Trong lời gọi hàm *nuong()* ở (2) giá trị tham chiếu đối tượng của biến hiện thời *banhMoi* được truyền cho biến hình thức *banhNuong* của hàm *nuong()* được định nghĩa ở (3). Trong hàm này biến *thit* của lớp *Banh* bị thay đổi thành "Thit vit". Những thay đổi này có ảnh hưởng tới trạng thái của đối tượng được tham chiếu. Nhưng việc thay đổi cả đối tượng như đặt *banhNuong* thành *null* (không có đối tượng) ở (4) lại không hề có tác động đối với đối tượng tham chiếu bởi biến hiện thời.



a/ Thực hiện lời gọi hàm



b/ Ngay sau khi thực hiện lời gọi hàm

Tóm lại, biến hình thức chỉ có thể làm thay đổi trạng thái (các thành phần) của đối tượng được tham chiếu tới trong lời gọi hàm.

Truyền các tham chiếu theo mảng

Mảng *(array)* trong Java được xem như là đối tượng. Các phần tử của mảng có thể có kiểu nguyên thủy hoặc kiểu tham chiếu *(kiểu lớp)*. Chi tiết về mảng chúng ta sẽ xét ở chương sau. Sau đây chúng ta tìm hiểu về cơ chế truyền tham chiếu kiểu mảng (các tham biến là mảng).

Ví dụ 3.7. Truyền các tham chiếu kiểu mảng

```
// Loc.java
class Loc{
  public static void main(String[] args){
    int[]  day = {8, 1, 4, 3, 2, 5};// Khởi tạo mảng day và gán trị đầu
    // Hiển thị các phần tử của dãy trước khi lọc
    for (int i = 0; i < day.length; i++)
        System.out.print("  " + day[i]);
    System.out.println();   // Xuống dòng mới
    int maxIndex = 0;
    // Lọc ra phần tử cực đại và đưa về cuối
    for (int index = 1; index < day.length; index++)
    {
        if (day[maxIndex]  > day[index])
          doiCho(day, maxIndex, index);      //(1)
          maxIndex = index;
    }
    // Hiển thị dãy sau khi lọc
    for (int i = 0; i < day.length; i++)
        System.out.print("  " + day[i]);
    System.out.println();
        }
  public static void doiCho(int[] bang,int i, int k){   // (2)
      int tg = bang[i]; bang[i] = bang[k]; bang[k] = tg;
      }
}
```

Kết quả của chương trình sau khi thực hiện:

```
8   1   4   3   2   5
1   4   3   2   5   8
```

Trong phần định nghĩa hàm *doiCho*() ở (2) biến hình thức *bang* là kiểu *array (mảng)*. Hàm này được gọi trong hàm *main*() ở (1), trong đó biến hiện thời *day* cũng là kiểu mảng. Vì trong Java các lời gọi hàm đều thực hiện theo tham trị (*call-by-value*) đối với kiểu nguyên thủy và tham chiếu đối với kiểu lớp nên hàm *doiCho()* như trên là thay đổi được các phần tử của *day* (biến hiện thời).

Các tham biến final

Tham biến hình thức có thể khai báo với từ khóa *final* đứng trước. Tham biến loại này được gọi là biến cuối "trắng", nghĩa là nó không được khởi tạo giá trị (là trắng) cho đến khi nó được gán một trị nào đó và khi đã được gán trị thì giá trị đó là cuối cùng, không thay đổi được.

Ví dụ 3.8. Sử dụng tham biến final

```
//KhachHang3.java
class KhachHang3{  // Lớp khách hàng
  public static void main(String[] arg){
    HangSX banh = new HangSX();          // Tạo ra 1 đối tượng
    int giaBan = 20;
    double tien = banh.tinh(10,giaBan);
    System.out.println("Gia ban: "+ giaBan); // giaBan không đổi
    System.out.println("Tien ban duoc : " + tien);
    }
}
// Lớp Hãng sản xuất
class HangSX{
  double tinh(int num,final double gia){//(1)
    gia = gia /2.0;                  //(2)
      return num * gia; // Thay đổi gia nhưng không ảnh hưởng tới giaBan,
                        // nhưng số tiền vẫn bị thay đổi theo
  }
}
```

Khi dịch hàm *tinh*(), chương trình dịch sẽ thông báo lỗi và không được phép thay đổi giá trị của biến final *gia* trong định nghĩa hàm như ở (2).

Các đối số của chương trình

Giống như chương trình C, chúng ta có thể truyền các tham số cho chương trình trên dòng lệnh, ví dụ:

```
java TinhTong 12 23 45
```

Chương trình java thông dịch lớp *TinhTong* để tính tổng các đối số 12, 23, 45. Chương trình *TinhTong* có thể viết như sau:

Ví dụ 3.9. Truyền tham số cho chương trình

```
// TinhTong.java
class TinhTong{
    public static void main(String args[]){
        float s = 0.0F;
        for (int i = 0; i < args.length; i++)
        s += Float.valueOf(args[i]).floatValue();
        // Chuyển dãy chữ số thành số
        System.out.print("Tong cua " + args[0]);
        for (int i = 1; i < args.length; i++)
            System.out.print(" + " + args[i]);
        System.out.println(" = " + s);
    }
}
```

Khi thực hiện chương trình cho kết quả

```
Tong cua 12 + 23 + 45 = 80.00
```

BÀI TẬP

3.1. Những từ sau có phải là định danh hay không?

a/ a4z, b/ ân2, c/ _class, d/ My$100

3.2. Dãy /* ... // ... */ có phải là chú thích hay không? tại sao?

Những lệnh nào trong số các lệnh sau là hợp lệ

(a) char a = '\u0061';

(b) char '\u0061' = 'a';

(c) long phone = 8235469L;

(d) float pi = 3.14159;

(e) double p = 314.159e-2;

3.3. Những khai báo nào của hàm main() sau đây là đúng

(a) static void main(String arg[]){ /* ... */ }

(b) public static int main(String arg[]){ /* ... */ }

(c) public static void main(String arg[]){ /* ... */ }

(d) public static void main(String[] arg){ /* ... */ }

(e) final static public void main(String[] arg){ /* ... */ }

3.4. Chương trình sau có lỗi hay không, nếu có lỗi thì hãy sửa cho đúng.

```
//NhietDo
PUBLIC CLASS  nhietDo{
      public VOID main(String  arg){
          double fahre = 62.5;
            double celsius = f2c(fahre);
          System.out.println(fahre + 'F = ' + celsius + 'c');
      }
      double f2c(float f){
          Return (f - 32) * 5 / 9;
      }
}
```

3.5. Chương trình sau khi dịch và chạy sẽ cho cái gì?

```
public class MyClass{
    public static void main(String[] arg){
        String a, b, c;
        c = new String("Chuột");
        a = new String("Mèo");
        b = a;
    a = "Chó";
    c = b;
    System.out.println(c);
  }
}
```

3.6. Điều gì sẽ xảy ra khi dịch và chạy chương trình sau:

```
public class Prog1{
    public static void main(String[] arg){
```

```
            int k = 1;
            int  i = ++k + k++ + +k;
          System.out.println(i);
    }
```

3.7. Điều gì sẽ xảy ra khi dịch và chạy chương trình sau:

```
public class Prog2{
    public static void main(String[] arg){
        int k = 0;
        int[]  a = {3, 6};
        a[k] = k = 9;   // Lưu ý mức ưu tiên của [] cao hơn phép gán =
      System.out.println(i + " " + a[0] + " " + a[1]);
    }
    }
```

3.8. Cho biết kết quả thực hiện của chương trình sau:

```
public class Prog3{
  public static void main(String[] arg){
      int k = 0, i = 0;
      boolean r, t = true;
         r = (t & 0 < (i += 1));
      r = (t && 0 < (i += 2));
        r = (t | 0 < (k += 1));
      r = (t || 0 < (k += 2));
      System.out.println(i + " " + k);
  }
  }
```

3.9. Kết quả dịch và chạy chương trình sau là gì?

```
public class Prog4{
    public static void main(String[] arg){
      int a = 0, b = 0;
      int[] bArr = new int[1]; bArr[0] = b;
      inC1(a);  inC2(bArr);
        System.out.println("a = "  + a + " b = " + b + "
          bArr = " + bArr[0]);
        }
        public static void inC1(int x){ x++;};
```

```
        public static void inC2(int[] x){ x[0]++;};
    }
```

3.10. Viết chương trình nhập vào một số nguyên và in ra dạng nhị phân của số đó.

3.11. Viết chương trình nhập vào ngày, tháng, năm và in ra ngày thứ bao nhiêu đó trong năm (365 hoặc 366 ngày đối với năm nhuận).

3.12. Viết chương trình nhập vào một số nguyên và in ra tất cả các số nguyên tố nhỏ hơn hoặc bằng số nguyên đó.

3.13. Viết chương trình nhập vào một số nguyên n và in ra tất cả các số hoàn thiện nhỏ hơn hoặc bằng số n đó.

CHƯƠNG IV

LỚP VÀ CÁC THÀNH PHẦN CỦA LỚP CÁC ĐỐI TƯỢNG

4.1. ĐỊNH NGHĨA LỚP

Định nghĩa một lớp là đặc tả một kiểu dữ liệu mới và mô tả cách cài đặt kiểu dữ liệu đó. Nó có cú pháp được qui định như sau:

[<Phạm vi hoặc kiểm soát truy nhập>] class <Tên lớp>

[extends <Tên lớp cha>] [implements <Tên giao diện>]

{

<Các thành phần của lớp>

}

trong đó `class`, `extends`, `implements` là các từ khoá. Những phần trong cặp [và] là tùy chọn. Những phần này sẽ được đề cập chi tiết ở các phần sau.

<Các thành phần của lớp> bao gồm các biến, các phương thức (hàm) thành phần và các *toán tử tạo lập* (*constructor*). Khai báo các biến thành phần như đã thảo luận ở mục 3.3. Các hàm và toán tử tạo lập sẽ được đề cập ở mục 4.2, 4.5 tiếp theo. Thân của một lớp lại có thể chứa các khai báo của các lớp, giao diện (*interface*) khác và xem chúng như là các thành phần của lớp.

4.2. ĐỊNH NGHĨA HÀM THÀNH PHẦN

Hành vi của các đối tượng của một lớp được xác định bởi các hàm thành phần của lớp đó. Cú pháp để định nghĩa các hàm có dạng tổng quát:

[<Phạm vi hoặc thuộc tính kiểm soát truy nhập>]<Kiểu trả lại> <Tên hàm >(

[<Danh sách tham biến hình thức>]) [<Mệnh đề throws>{

<Nội dung của hàm>

}

<Kiểu trả lại> có thể là kiểu nguyên thủy, kiểu lớp hoặc không có giá trị trả lại (kiểu void).

<Danh sách tham biến hình thức> bao gồm dãy các tham biến (kiểu và tên) phân cách với nhau bởi dấu phẩy.

Ví dụ 4.1. // Tệp Demo.java

```
public class Demo{
   public static void main(String args[]){
      if (args.length == 0) return; //Hàm kiểu void có thể sử dụng return;
      output(checkValue(args.length));
}
   static void output(int value) {// Hàm kiểu void không cần sử dụng return
      System.out.println(value);
      }
  static int checkValue(int i){ // Hàm khác kiểu void phải sử dụng return để trả
      if (i > 3) return 1;                                  // lại giá trị.
      else return 2;
      }
}
```

4.2.1. NẠP CHỒNG CÁC HÀM THÀNH PHẦN

Mỗi hàm đều có phần định danh hàm, bao gồm tên của hàm, kiểu trả lại giá trị và danh sách các tham biến. Trong lập trình hướng đối tượng cho phép sử dụng cùng một tên hàm nhưng định nghĩa nhiều nội dung thực hiện khác nhau. Những hàm như thế được gọi là hàm nạp chồng hay hàm tải bội (*overloading*).

Lưu ý: Cơ chế nạp chồng cho phép một hàm có cùng một tên gọi nhưng danh sách tham biến khác nhau, do vậy sẽ có các định danh khác nhau. Những hàm được nạp chồng với các định danh khác nhau có các phần cài đặt thực hiện những công việc khác nhau và có kiểu trả lại khác nhau.

JDK API đã xây dựng rất nhiều hàm được nạp chồng. Ví dụ, lớp *java.lang.Math* có hàm nạp chồng *min*() xác định giá trị cực tiểu của 2 số:

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

Cũng cần lưu ý là danh sách tham biến của các hàm nạp chồng phải khác nhau về số lượng hoặc về thứ tự các kiểu của các tham biến. Ví dụ:

```
public void methodA(int a, double b){/* ...*/}          //(1)
public int methodA(int a){return a;}                     //(2)
public int methodA(){return 1;}                          //(3)
public long methodA(double a,int b){return a*b;}         //(4)
public long methodA(int a, double b){return a;}   // No OK (5)
```

Các hàm trên có các định danh tương ứng là:

```
methodA(int, double)                                     //(1')
methodA(int)                                             //(2')
methodA()                                                //(3')
methodA(double, int)                                     //(4')
methodA(int, double)                  // (5') giống hệt như (1')
```

Bốn nội dung cài đặt đầu của hàm *methodA* là được nạp chồng chính xác vì danh sách các tham biến khác nhau. Định nghĩa cho hàm *methodA(int, double)* đã có ở (1) và sau đó lại bị lặp lại ở (5) do vậy phần cài đặt cuối cùng này không cho phép. Mỗi định danh hàm chỉ có một nội dung thực hiện được cài đặt. Lúc dịch, những nội dung cài đặt chính xác sẽ được gắn cho định danh hàm nạp chồng tương ứng để liên kết thực hiện các lời gọi hàm.

Ngoài cơ chế nạp chồng cho các hàm còn có cơ chế *viết đè (overriding)* cũng cần phải được phân biệt.

4.2.2. VIẾT ĐÈ CÁC HÀM THÀNH PHẦN VÀ VẤN ĐỀ CHE BÓNG CÁC BIẾN

Trong nhiều trường hợp, một lớp con có thể viết đè, thay đổi nội dung thực hiện của những hàm được thừa kế từ lớp cha. Khi những hàm này được gọi để thực hiện đối với những đối tượng của lớp con thì nội dung được định nghĩa mới ở lớp con sẽ được thực thi. Sau đây là một số chú ý khi sử dụng cơ chế viết đè.

- ✓ Định nghĩa mới của hàm viết đè phải có cùng định danh (tên gọi và danh sách tham biến) và cùng kiểu trả lại giá trị.
- ✓ Định nghĩa mới của hàm viết đè trong lớp con chỉ có thể xác định tất cả hoặc tập con các lớp ngoại lệ được kể ra trong mệnh đề cho qua ngoại lệ (*throws clause* - sẽ trình bày ở chương 5).
- ✓ Định nghĩa của những hàm sẽ viết đè không được khai báo *final* ở lớp cha.

Ví dụ 4.2. Viết đè và nạp chồng các hàm thành phần

```
// KhachHang.java
import java.io.*;
```

```
class Den {
  protected String loaiHoaDon = "Hoa don nho: ";          // (1)
  protected double docHoaDon(int giaDien)
            throws Exception {                            // (2)
    double soGio = 10.0,
    hoaDonNho = giaDien* soGio;
    System.out.println(loaiHoaDon + hoaDonNho);
    return hoaDonNho;
  }
}
class DenTuyp extends Den {
  public String loaiHoaDon ="Hoa don lon:";               // Bị che bóng (3)
  public double docHoaDon (int giaDien)
          throws Exception {                              // Viết đè hàm (4)
    double soGio = 100.0,
    hoaDonLon = giaDien* soGio;
    System.out.println(loaiHoaDon + hoaDonLon);
    return hoaDonLon;
  }
public double docHoaDon (){
    System.out.println("Khong co hoa don!");
    return 0.0;
  }
}

public class KhachHang {
  public static void main(String args[])
      throws Exception {                                  // (6)
      DenTuyp den1 = new DenTuyp();                       // (7)
      Den den2 = den1;                                    // (8)
      Den den3 = new Den();                               // (9)
      // Gọi các hàm đã viết đè
      den1.docHoaDon(1000);                               // (10)
      den2.docHoaDon(1000);                               // (11)
      den3.docHoaDon(1000);                               // (12)
      // Truy nhập tới các biến thành phần đã bị viết đè (bị che bóng)
      System.out.println(den1.loaiHoaDon);                // (13)
```

```
        System.out.println(den2.loaiHoaDon);          // (14)
        System.out.println(den3.loaiHoaDon);          // (15)
        // Gọi các hàm nạp chồng
        den1.docHoaDon();
    }
}
```

Kết quả thực hiện của chương trình KhachHang:

Hóa đơn lớn: 100000.00

Hóa đơn lớn: 100000.00

Hóa đơn nhỏ: 10000.00

Hóa đơn lớn:

Hóa đơn nhỏ:

Hóa đơn nhỏ:

Không có Hóa đơn!

Hàm *docHoaDon()* của lớp cha *Den* được định nghĩa ở (2) và sau đó được viết đè trong lớp con *DenTuyp* ở (4). Những hàm này sẽ bỏ qua ngoại lệ *Exception* (lớp thư viện ở gói java.io.*). Phạm vi lớp của *docHoaDon()* có thể thay đổi, ví dụ, chuyển từ *protected* sang *public*, và tham biến cũng có thể khai báo *final*, ví dụ *final int giaDien*. Lời gọi hàm *docHoaDon()* thực hiện đối với đối tượng của lớp con *DenTuyp* và lớp cha *Den* ở (10) và (11) tương ứng, kết quả thực hiện nội dung đã viết đè ở (4). Lời gọi hàm *docHoaDon()* đối với đối tượng của lớp cha (lớp *Den*) ở (12) tất nhiên sẽ thực hiện nội dung được định nghĩa ở (2).

Lưu ý: Các hàm *final, static* không được phép viết đè.

Cơ chế che bóng của các biến

Về nguyên tắc, một lớp con không được phép viết đè các biến thành phần của lớp cha, nhưng có thể bị che khuất chúng tương tự như biến cục bộ trong lớp con. Lớp con có thể định nghĩa lại (che) các biến đã được định nghĩa trong lớp cha. Khi đó các biến này của lớp cha sẽ không thể truy nhập trực tiếp theo tên ở lớp con. Trong hàm của lớp con có thể sử dụng toán tử super() (sẽ đề cập ở mục 4.5) để truy nhập tới các biến của lớp cha bị che khuất.

Trong ví dụ 4.2, biến *loaiHoaDon* được định nghĩa lại ở lớp con *DenTuyp* (3), do vậy với lệnh (13) thì *den1.loaiHoaDon* sẽ cho kết quả đã định nghĩa lại ở (3). Nhưng 2 đối tượng *den2, den3* lại thuộc lớp cha *Den* do đó vẫn giữ nguyên *loaiHoaDon* của lớp cha được định nghĩa ở (1).

Viết đè và nạp chồng là khác nhau

Trước tiên cần phân biệt rõ hai cơ chế viết đè và nạp chồng là khác nhau trong Java.

- ✓ Viết đè yêu cầu cùng định danh hàm (cùng tên gọi, cùng danh sách tham số) và cùng kiểu trả lại kết quả đã được định nghĩa tại lớp cha.
- ✓ Nạp chồng yêu cầu khác nhau về định danh, nhưng giống nhau về tên gọi của hàm, vì thế chúng sẽ khác nhau về số lượng, kiểu, hay thứ tự của các tham biến.
- ✓ Hàm có thể nạp chồng ở trong cùng lớp hoặc ở các lớp con cháu.
- ✓ Từ những lớp con khi muốn gọi tới các hàm ở lớp cha mà bị viết đè thì phải gọi qua toán tử đại diện cho lớp cha, đó là *super()*. Đối với hàm nạp chồng thì lại không cần như thế. Lời gọi hàm nạp chồng được xác định thông qua danh sách các đối số hiện thời sánh với đối số hình thức để xác định nội dung tương ứng.

4.3. PHẠM VI VÀ CÁC THUỘC TÍNH KIỂM SOÁT TRUY NHẬP CÁC THÀNH PHẦN CỦA LỚP

Trong lập trình hướng đối tượng, khái niệm lớp được sử dụng để bao gói và che giấu thông tin. Tuy nhiên giữa các đối tượng phải thường xuyên trao đổi thông điệp với nhau để thực thi những yêu cầu của hệ thống. Vậy vấn đề đặt ra là các thành phần của lớp được quản lý như thế nào? Sau đây chúng ta xét những luật quản lý truy nhập tới các thành phần (*biến, hàm*) của lớp trong Java.

4.3.1. PHẠM VI CỦA CÁC THÀNH PHẦN

Phạm vi của các thành phần có hai loại:

- Phạm vi lớp của các thành phần,
- Phạm vi khối của các biến cục bộ (*local*) .

1/ Phạm vi lớp

Phạm vi lớp xác định những thành phần được truy nhập bên trong của một lớp (kể cả lớp được kế thừa). Quyền truy nhập của chúng thường được xác định thông qua các *bổ ngữ (modifier): public, protected, private.*

Bên trong định nghĩa của lớp, các biến tham chiếu tới chính lớp đó và được quyền truy nhập tới tất cả các thành phần của nó mà không cần quan tâm tới các bổ ngữ của chúng.

Ví dụ 4.3. Phạm vi lớp của các thành phần

```
class BongDen{
```

```
//Các biến thành phần
private int soWatts;                    // Số watts của bóng đèn
private boolean batTat;                 // true - bóng sáng, false - tắt
private String viTri;                   // Nơi đặt bóng đèn
// Các hàm thành phần
public void batDen(){batTat = true;}    // Bật đèn
public void tatDen(){onOff = false;}    // Tắt đèn
public boolean tatHaySang(){return batTat;}
public BongDen nhanDoi(BongDen bongCu){ //(1)
  BongDen bongMoi = new BongDen();
  bongMoi.sOWatts = bongCu.sOWatts;     //(2)
  bongMoi.batTat = bongCu.batTat;       //(3)
  bongMoi.viTri = new String(bongCu.viTri);  //(4)
  }
}
```

Hàm *nhanDoi()* ở (1) của lớp *BongDen* có tham biến *bongCu* và biến cục bộ *bongMoi* đều tham chiếu tới lớp *BongDen*. Mặc dù các biến thành phần *sOWatts*, *batTat*, *viTri* là *private* (sở hữu riêng) nhưng chúng vẫn cho phép truy nhập đối với cả 2 tham chiếu trên ở (2), (3), và (4).

2/ Phạm vi khối

Trong chương trình, các lệnh khai báo và các lệnh thực hiện có thể gộp lại thành từng khối (*block*) bằng cách sử dụng {, }. Chú ý là thân của hàm, của lớp cũng là một khối. Các khối có thể lồng nhau và khi đó luật phạm vi được áp dụng cho các biến cục bộ. *Luật phạm vi* phát biểu tổng quát là *một biến được khai báo ở trong một khối có phạm vi xác định bên trong khối và không xác định ở bên ngoài khối đó.*

Lưu ý:

- ✓ Trong cùng một phạm vi không thể khai báo một biến hai lần. Các tham biến cũng không thể khai báo lại trong thân của hàm. Lưu ý là trong các khối thì lệnh khai báo là tự do, thứ tự không quan trọng, muốn khai báo ở chỗ nào cũng được miễn là phải khai báo trước khi sử dụng.
- ✓ Phạm vi khối của biến được xác định kể từ chỗ bắt đầu khai báo cho đến kết thúc khối (gặp dấu ngoặc '}' tương ứng).
- ✓ Có thể có nhiều khối lồng nhau nhưng không được cắt nhau và những biến khai báo ở khối ngoài đều có phạm vi xác định ở trong mọi khối bao bên trong nó.

Ví dụ: hình 4.1 minh họa các tính chất về phạm vi của các khối.

```
public static void main(String args[]){ // Khối 1
// String args = "";  không thể khai báo lại các tham biến  (1)
char chuSo;
    for (int k = 0; k < 10; ++k) { // Khối 2
  switch(chuSo) {  // Khối 3
     case '1': int i;  // (2)
     default:  int i;  // (3) Lỗi vì khai báo lại i trong cùng khối
  }// switch

  if (true){ // Khối 4
    int i;  // (4) OK
    int chuSo;// (5) Lỗi vì chuSo đã được khai báo ở khối ngoài, khối 1
    int k;        // (6) Lỗi vì k đã được khai báo ở khối ngoài, khối 2
     }
  }

int k;            // (6) OK
```

*Hình 4.1. **Minh họa phạm vi của các khối chương trình.***

4.3.2. CÁC THUỘC TÍNH KIỂM SOÁT TRUY NHẬP CÁC THÀNH PHẦN CỦA LỚP

Một trong những ưu điểm của phương pháp hướng đối tượng là có thể tổ chức dữ liệu theo nguyên lý bao gói và che giấu thông tin. Các thuộc tính và hàm thành phần của lớp có thể khai báo thêm một số bổ ngữ để kiểm soát quyền truy nhập đối với những thành phần đó. Khi thiết kế các thành phần của lớp đối tượng, chúng ta có thể sử dụng những bổ ngữ sau:

- public
- protected
- mặc định (không sử dụng thêm bổ ngữ khi định nghĩa lớp)
- private
- static
- final

- abstract
- synchronized
- native
- transient
- volatile

Trong phần thảo luận sau đây chúng ta sử dụng một số ký hiệu của UML (*Unified Modeling Language*) để minh họa. UML ký hiệu '+' cho thuộc tính *public,* '#' cho *protected* và '- ' cho *private.*

(i) Các thành phần `public`

Thuộc tính `public` xác định tính công khai của các thành phần. Thành phần khai báo công khai (*public*) cho phép truy nhập mọi nơi trong hệ thống, cả đối với các lớp cùng gói (*package*) lẫn những lớp ở các gói khác cũng nhìn thấy được.

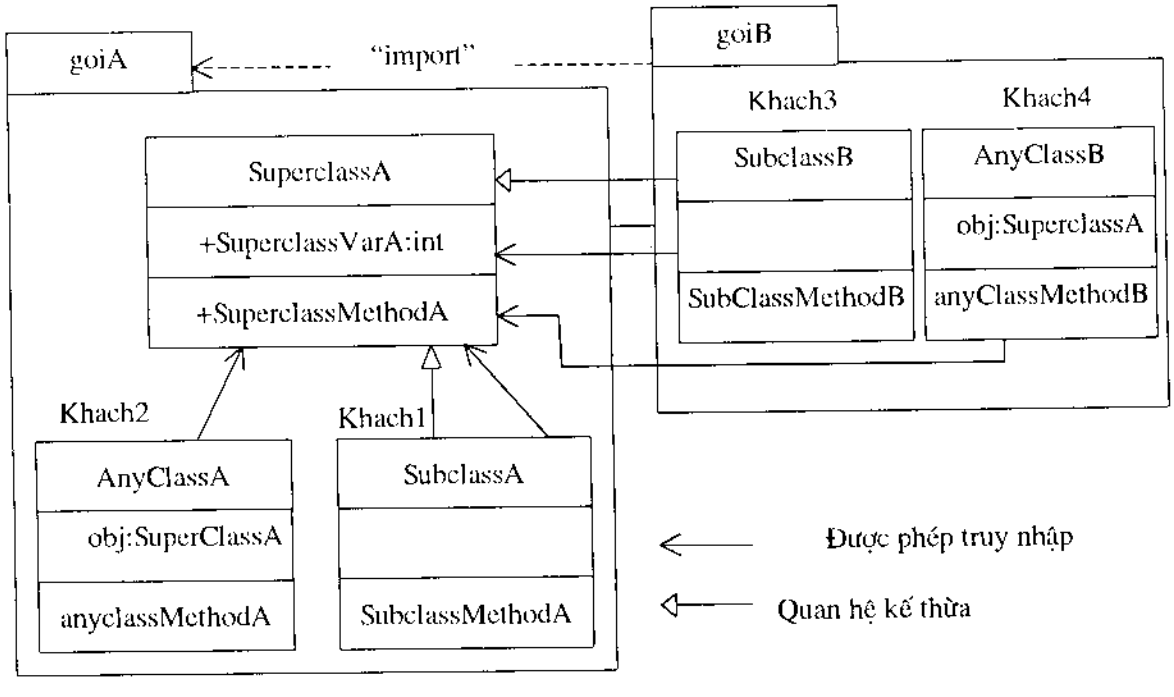
Ví dụ 4.4. Tính công khai (*public*) của các thành phần

```
// Tệp: SuperclassA.java                                        (1)
package goiA;                          // Định nghĩa gói có tên goiA
public class SuperclassA{
    public int superclassVarA;                              // (2)
    public void superclassMethodA(){ /* ...*/}              // (3)
}
class SubclassA extends SuperclassA{
    void subclassMethodA(){superclassVarA = 20; }   // (4)
}
class AnyClassA{
    SuperclassA obj = new SuperclassA();
    void anyClassMethodA(){
        obj.superclassMethodA();                            // (5)
    }
}
// Tệp: SubclassB.java                                          (6)
package goiB;
import goiA.*; // Nhập các lớp đã được định nghĩa ở gói có tên goiA.
public class SubclassB extends SuperclassA{
    void subclassMethodB(){superclassMethodA();}        // (7)
}
class AnyClassB{
```

```
        SuperclassA obj = new Sup rclassA();
        void anyClassMethodB(){
        obj.superclassVarA = 10;                    // (8)
    }
}
```

Ví dụ 4.4 có hai tệp nguồn gồm hai gói các lớp được định nghĩa ở (1) và (6). Các lớp trong gói *goiB* có thể sử dụng (truy nhập hay nhìn thấy được) các thành phần *public* của các lớp trong gói *goiA*. Lớp *SuperclassA* ở *goiA* có 2 lớp con (mở rộng hay kế thừa): một là *SubclassA* trong *goiA* và *SubclassB* trong *goiB*. Khả năng truy nhập tới các thành phần *public* của các lớp trong 2 gói trên được minh họa ở hình 4.2.

- *Khach1*: Từ lớp con trong cùng gói có thể truy nhập tới biến *superclassVarA* của lớp cha (4), vì các biến khai báo công khai của lớp cha được kế thừa sang cho các lớp con của nó;
- *Khach2*: Từ những lớp không kế thừa của lớp *SuperclassA*, nhưng trong cùng gói thì vẫn có thể truy nhập được tới thành phần công khai *superclassMethodA()* như ở (5);
- *Khach3*: Từ lớp con không trong cùng gói vẫn có thể truy nhập tới hàm *superclassMethodA()* (7), vì các hàm khai báo công khai của lớp cha được kế thừa sang cho các lớp con của nó;
- *Khach4*: Từ những lớp không kế thừa của lớp *SuperclassA* và ở khác gói thì vẫn có thể truy nhập được tới thành phần công khai *superclassVarA* như ở (8).



Hình 4.2. Khả năng truy nhập đối với các thành phần public.

Tóm lại các thành phần khai *public* có miền xác định rộng nhất, có thể nhìn thấy ở mọi nơi trong hệ thống.

(ii) Các thành phần protected

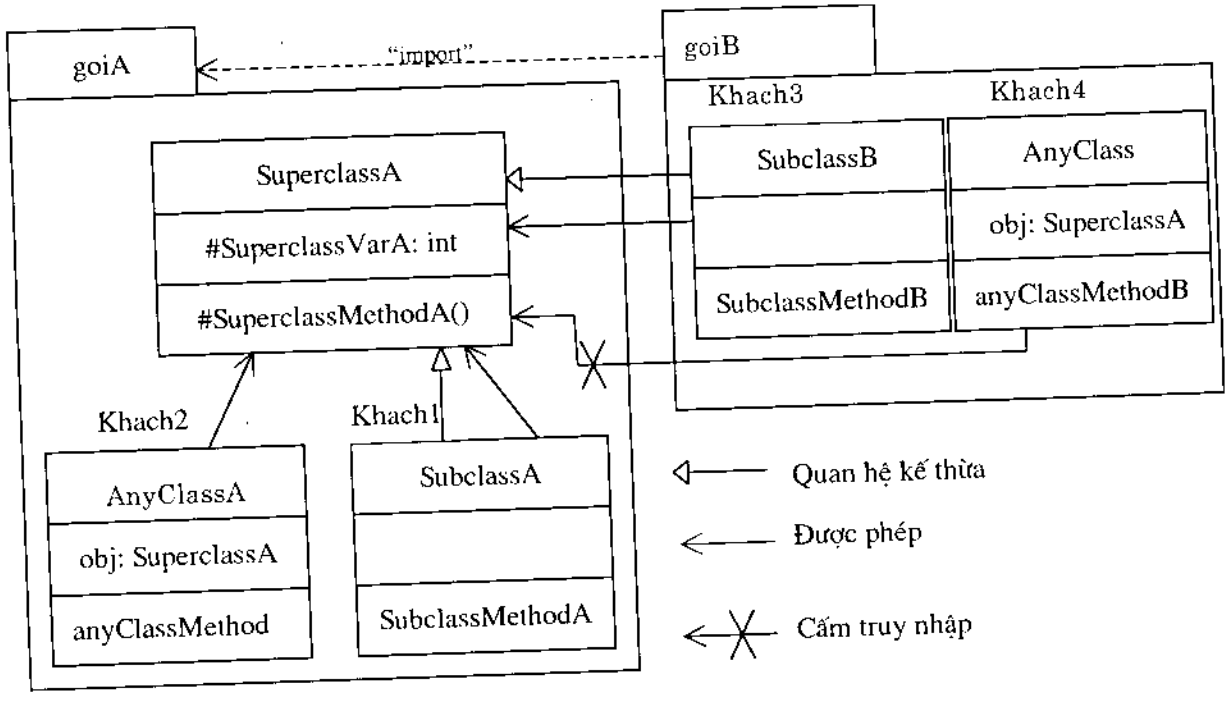
Những thành phần được bảo vệ *protected* cho phép truy nhập đối với tất cả các lớp trong gói chứa lớp đó và tất cả các lớp con (có thể ở những gói khác) của lớp chứa chúng. Nói cách khác, những lớp không phải là lớp con và ở những gói khác thì không được phép truy nhập tới các thành phần khai báo *protected*.

Ví dụ 4.5. Thuộc tính *protected* của các thành phần

```
// Tệp: SuperclassA.java                                          (1)
package goiA;                          // Định nghĩa gói có tên goiA
public class SuperclassA{
    protected int superclassVarA;                                  // (2)
    protected void superclassMethodA(){/*...*/}                    // (3)
}
class SubclassA extends SuperclassA{
    void subclassMethodA(){superclassVarA = 20; }                  // (4)
}
class AnyClassA{
    SuperclassA obj = new SuperclassA();
    void anyClassMethodA(){
        obj.superclassMethodA();                                   // (5)
    }
  }
  // Tệp: SubclassB.java (6)
package goiB;
import goiA.*;     // Nhập các lớp đã được định nghĩa ở gói có tên goiA.
public class SubclassB extends SuperclassA{
    SuperclassA objA = new SubclassB();                            // (7)
    SubclassB objB = new SubclassB();                              // (8)
    void subclassMethodB(){
        objB.superclassMethodA();                                  // Đúng (9)
        objA.superclassMethodA()                                   // Sai (10)
    }
}
```

```
class AnyClassB{
    SuperclassA obj = new SuperclassA();
    void anyClassMethodB(){
        obj.superclassVarA = 10;                //Không cho phép (11)
    }
}
```

Khả năng nhìn thấy được của các thành phần *protected* của các lớp ở hai gói trên được minh họa trong hình 4.3.



Hình 4.3. ***Khả năng truy nhập của các thành phần protected.***

Phân tích hoạt động của chương trình trên, chúng ta nhận thấy:

- *Khach1*: Từ lớp con trong cùng gói có thể truy nhập tới biến *superclassVarA* (4), vì các biến khai báo *protected* của lớp cha được kế thừa sang cho các lớp con của nó;
- *Khach2*: Từ lớp không kế thừa của lớp *SuperclassA*, nhưng trong cùng gói thì vẫn có thể truy nhập được tới thành phần *protected superclassMethodA()* như ở (5);
- *Khach3*: Từ lớp con không trong cùng gói vẫn có thể truy nhập tới hàm *superclassMethodA*() (9), vì các hàm khai báo *protected* của lớp cha được kế thừa sang cho các lớp con của nó; nhưng một đối tượng tham chiếu *objA* của *SubclassB* được tạo ra theo *SuperclassA* sẽ không thể nhìn thấy được thành phần được bảo

vệ của lớp cha ở gói khác (10). Điều hạn chế này đảm bảo rằng những lớp con nằm ở những gói khác với gói của lớp cha chỉ có thể truy nhập được tới những thành phần *protected* từ lớp cha thông qua sự tham gia của chúng trong cấu trúc phân cấp theo quan hệ kế thừa.

- *Khach4*: Từ lớp không kế thừa của lớp *SuperclassA* và ở khác gói thì không thể truy nhập được tới thành phần *protected superclassVarA* như ở (11).

(iii) Các thành phần private

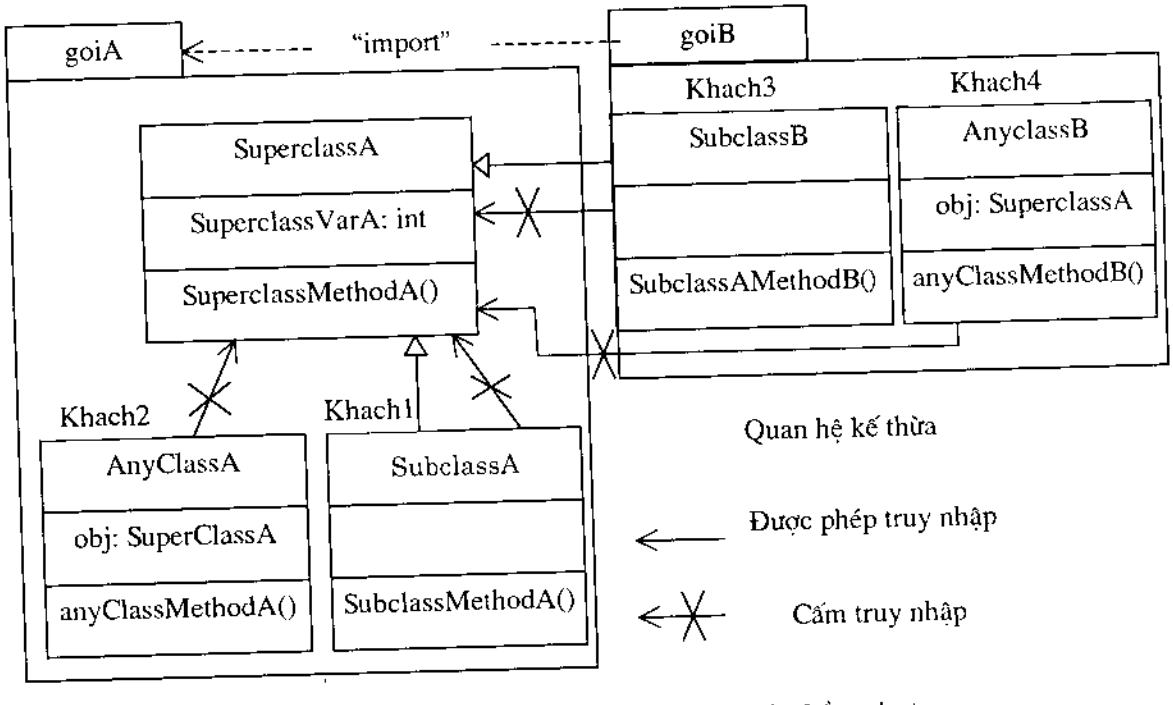
Những thành phần sở hữu riêng *private* được bảo vệ chặt chẽ nhất, không cho phép kế thừa và chỉ cho phép truy nhập đối với những đối tượng trong cùng lớp. Nói cách khác, những lớp khác dù là ở cùng một gói, hoặc là các lớp con cháu cũng không được phép truy nhập trực tiếp tới các thành phần riêng *private*.

Ví dụ 4.6. Thuộc tính *private* của các thành phần

```
// Tệp: SuperclassA.java                                        (1)
package goiA;   // Định nghĩa gói có tên goiA
public class SuperclassA{
    private int superclassVarA;                                  // (2)
    private void superclassMethodA(){
        superclassVarA = 10;}                                    // (3)
}
class SubclassA extends SuperclassA{
    void subclassMethodA(){
        superclassVarA = 20;}                    // Không được phép (4)
}
class AnyClassA{
    SuperclassA obj = new SuperclassA();
    void anyClassMethodA(){
        obj.superclassMethodA();                 // Không được phép (5)
    }
}
// Tệp: SubclassB.java                                          (6)
package goiB;
import goiA.*;   // Nhập các lớp đã được định nghĩa ở gói có tên goiA.
public class SubclassB extends SuperclassA{
  SuperclassA objA = new SubclassB();                            // (7)
  SubclassB objB = new SubclassB();                              // (8)
```

```
    void subclassMethodB(){
      objB.superclassMethodA();//   Không được phép          (9)
      objA.superclassMethodA();//   Không cho phép     (10)
  }
}
class AnyClassB{
  SuperclassA obj = new SuperclassA();
  void anyClassMethodB(){
      obj.superclassVarA = 10;                    // Không cho phép (11)
  }
}
```

Khả năng truy nhập tới các thành phần *private* của các lớp ở hai gói trên được minh họa trong hình 4.4.



Hình 4.4. ***Khả năng truy nhập của các thành phần private.***

Như nhận thấy ở trên, chỉ trong cùng lớp được phép truy nhập (nhìn thấy) các thành phần *private* của một lớp, còn các lớp khác không thể nhìn thấy được.

(iv) Các thành phần mặc định

Những thành phần mặc định (*default*, không khai báo thuộc tính *public, protected, private*) của một lớp chỉ cho phép truy nhập đối với những lớp trong cùng gói chứa

lớp đó, kể cả các lớp con của nó. Nói cách khác, những lớp ở những gói khác thì không được phép truy nhập tới các thành phần mặc định, nghĩa là các thành phần mặc định cũng không được phép kế thừa. Như vậy, các thành phần mặc định sẽ có phạm vi nhìn thấy được rộng hơn các thành phần *private*, nhưng hẹp hơn các thành phần *protected*.

Lưu ý: Khi không sử dụng lệnh định nghĩa gói

```
package tenGoi;
```

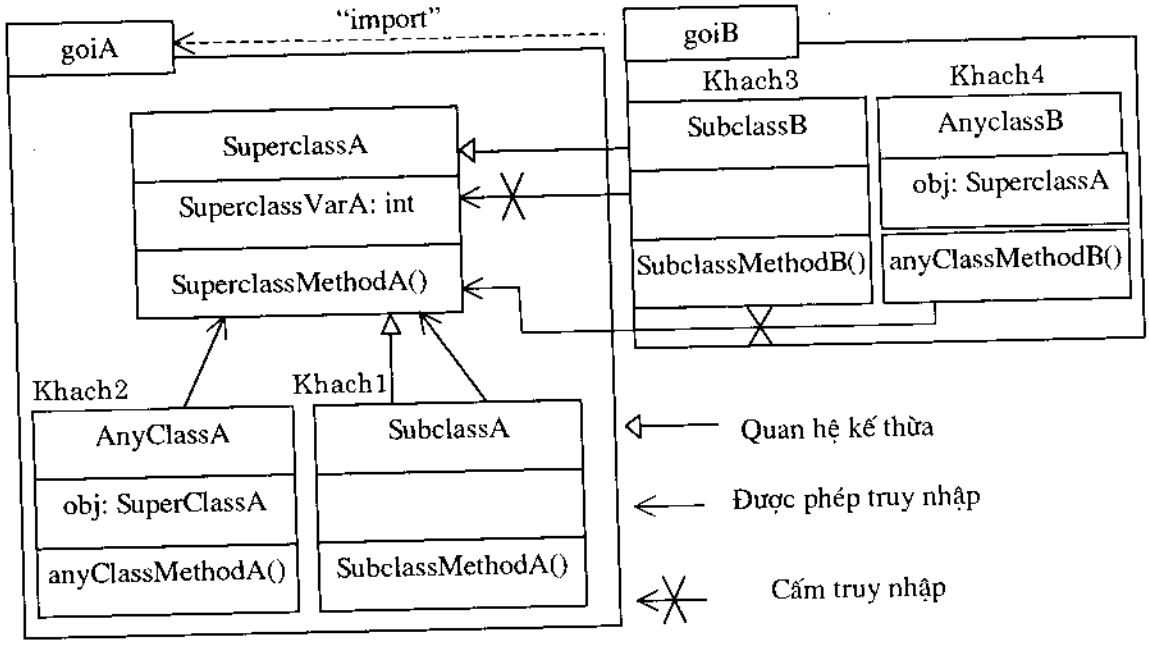
như trong các ví dụ 4.2, 4.3, 4.4, 4.5 thì tất cả các lớp được định nghĩa trong cùng một tệp chương trình sẽ được xem như là gói mặc định.

Ví dụ 4.7. Thuộc tính mặc định của các thành phần

```
// Tệp: SuperclassA.java                                          (1)
package goiA;      // Định nghĩa gói có tên goiA
public class SuperclassA{
  int superclassVarA;                                              // (2)
  void superclassMethodA(){/*...*/}                                // (3)
}
class SubclassA extends SuperclassA{
    void subclassMethodA(){superclassVarA = 20; }                  // (4)
}
class AnyClassA{
    SuperclassA obj = new SuperclassA();
    void anyClassMethodA(){
    obj.superclassMethodA();                                       // (5)
    }
}
// Tệp: SubclassB.java                                            (6)
package goiB;
import goiA.*;     // Nhập các lớp đã được định nghĩa ở gói có tên goiA.
public class SubclassB extends SuperclassA{
    void subclassMethodB(){
    objA.superclassMethodA();          // Không cho phép           (7)
    }
}
class AnyClassB{
    SuperclassA obj = new SuperclassA();
    void anyClassMethodB(){
```

```
        obj.superclassVarA = 10;                // Không cho phép      (8)
    }
}
```

Minh họa cho khả năng truy nhập tới các thành phần mặc định của các lớp ở hai gói trong ví dụ 4.5 là hình 4.5.



*Hình 4.5. **Khả năng truy nhập của các thành phần mặc định.***

Tóm lại bốn thuộc tính trên được sử dụng để xác định khả năng truy nhập đối với các thành phần của lớp đối tượng.

Thuộc tính	Các thành phần
public	Cho phép truy nhập ở mọi nơi, được phép kế thừa.
protected	Cho phép truy nhập đối với các lớp trong cùng gói và những lớp con (mở rộng) ở các gói khác. Được phép kế thừa.
Mặc định (không khai báo thêm bổ ngữ)	Chỉ cho phép truy nhập đối với các lớp trong cùng gói, kể cả các lớp con. Không cho phép kế thừa.
private	Chỉ cho phép truy nhập ở trong cùng lớp.Không cho phép kế thừa.

Ngoài các thuộc tính xác định phạm vi theo lớp như trên còn có một số thuộc tính (bổ ngữ) mô tả các đặc tính của các lớp, hàm, biến thành phần.

(v) Các thành phần static

Khi khai báo các thành phần của lớp có tính chất tĩnh, nghĩa là những thành phần đó là chung cho cả lớp chứ không phải là thành phần của cá thể nào cả thì sử dụng từ khóa *static* đứng trước phần khai báo. Như vậy các thành phần *static* sẽ là chung cho tất cả các đối tượng trong một lớp, còn những thành phần không *static* thì thường mỗi biến đều có bản sao các giá trị riêng của từng đối tượng.

Ví dụ 4.8. Minh họa về quản lý truy nhập các thành phần tĩnh *static*

```
// NhaKho.java
class BongDen{
// Các biến thành phần
    int soWatts;                                  // Số watts của bóng đèn
   boolean onOff;                                 // true - bóng sáng, false - tắt
  String viTri;                                   // Nơi đặt bóng đèn
// Thành phần tĩnh
   static int demBong;                            // Đếm số bóng của cả nhà kho (1)
// Định nghĩa toán tử tạo lập
BongDen(){
   soWatts = 45;                                  // Đặt mặc định bóng mới với công suất 45 watts
   onOff = true;                                  // Khi lắp bóng mới thì bật luôn
   viTri = new String("XYZ");
   ++demBong;                                     // Tăng demBong lên một khi lắp mới (2)
}
// Hàm thành phần tĩnh static
public static ghiBong(){
   System.out.println      ("So bong den cua nha kho: " + demBong);// (3)
   //System.out.println     ("Cong suat cua bong den: " + soWatts); // (4)
   }
}

public class NhaKho{
   public static void main(String args[]){
   BongDen.ghiBong();                             // Gọi hàm tĩnh theo tên lớp      (5)
   BongDen den1 = new BongDen(); // Tạo ra bóng mới
   System.out.println ("Gia tri cua bo dem: " + BongDen.demBong); //    (6)
   BongDen den2 = new BongDen(); // Tạo ra bóng mới
   den2.ghiBong();                                // Gọi hàm tĩnh theo tên đối tượng (7)
```

```
        System.out.println      ("Gia tri cua bo dem: " + den2.demBong); //   (8)
    }
}
```

Kết quả thực hiện của chương trình:

```
So bong den cua nha kho:  0
Gia tri cua bo dem:  1
So bong den cua nha kho:  2
Gia tri cua bo dem:  2
```

Lưu ý:

- ✓ Các biến dữ liệu tĩnh của lớp như biến *demBong* khai báo ở (1) sẽ được tự khởi tạo giá trị (*0* cho biến kiểu *int*) khi lớp được nạp vào chương trình để thực hiện và các giá trị của chúng là sử dụng chung cho cả lớp. Vì thế biến tĩnh *demBong* không được khởi tạo tường minh nhưng được mặc định vì vậy lệnh tăng giá trị lên *1* như ở (2) là thực hiện được.
- ✓ Truy nhập tới các thuộc tính dữ liệu tĩnh có thể thực hiện theo hai cách:
  - + Thông qua tên lớp như ở (5), (6),
  - + Thông qua tên đối tượng như ở (7), (8);
- ✓ Các hàm thành phần tĩnh (ví dụ hàm *ghiBong()* ở (3)) chỉ được phép truy nhập tới các thành phần tĩnh khác, ví dụ *demBong*; không được phép truy nhập tới những thành phần không khai báo *static*, ví dụ *soWatts* ở (4).

(vii) Các thành phần final

Biến thành phần *final* được xem như là hằng, giá trị của nó sẽ không thể thay đổi được ngay sau khi đã được khởi tạo.

Như ở chương 3 chúng ta đã biết, Java có hai loại kiểu: nguyên thủy và tham chiếu.

- Đối với các biến *final* kiểu nguyên thủy, khi chúng được khởi tạo giá trị thì sẽ không thay đổi được,
- Đối với các biến *final* kiểu tham chiếu (*reference*), giá trị tham chiếu (là đối tượng) sẽ không thay đổi được, nhưng các thành phần (trạng thái) của đối tượng có thể thay đổi được.

Ngoài ra cần lưu ý:

- ✓ Các biến *final* không cần khởi tạo giá trị khi khai báo và còn được gọi là biến "trắng". Song phải gán giá trị trước khi sử dụng. Biến *static* không cần khởi tạo giá trị trước khi sử dụng mà nó khởi tạo giá trị mặc định.
- ✓ Các hàm khai báo *final* ở trong lớp là đầy đủ (có đủ nội dung cần thực hiện) và

không thể viết đè ở các lớp con của lớp đó, nghĩa là không thay đổi được.

- ✓ Các biến *final static* được sử dụng chung trong chương trình như các hằng *(constant)* tương tự như từ khóa *const* của ngôn ngữ C. Thông thường tên của biến hằng được viết hoa và gán giá trị ngay khi khai báo như ở (1) trong ví dụ 4.9.
- ✓ Đối với các thành phần *final*, chương trình dịch có thể thực hiện được tối ưu hóa trong việc sinh mã.
- ✓ *Lớp cũng có thể khai báo final.* Lớp *final* là không mở rộng được. Nói cách khác, hành vi (các hàm thành phần) của lớp *final* là không thay đổi được ở các lớp con của nó. Như vậy có thể xem lớp *final* là loại lớp "đầy đủ" còn lớp *abstract* (sẽ đề cập ở phần sau) là không đầy đủ. Hiển nhiên một lớp không thể vừa là *final* vừa là *abstract.*
- ✓ Thư viện của Java API có nhiều lớp *final* đã được xây dựng, ví dụ java.lang.String là loại lớp không thể tạo ra những lớp đặc biệt là con cháu của nó được.

Ví dụ 4.9. Minh họa khả năng truy nhập của các thành phần *final* của lớp

```
// NhaKho.java
class BongDen{
// Biến thành phần final static                                        (1)
final static double KWH_GIA = 500.00;// Giá bán điện 500đ/KWH
int soWatts;
// Hàm thành phần final
final public void datWatt(int watt ){
  soWatts= watt; //          (2)
  }
public void datLaiGia(){
  KWH_GIA = 400.00; // Sai vì không cho phép thay đổi lại biến final static (3)
}
}
class DenTuyp extends BongDen{
// Không thể nạp chồng lại các hàm của BongDen
public void datWatt(int w) {// Thử viết đè hàm final do vậy chương trình
soWatts = 2 * w;          // dịch sẽ không thực hiện - Sai              (4)
}
}
public class NhaKho{
```

```
        public static void main(String args[]){
          final BongDen den = new BongDen();                    (5)
          den.soWatts =100;      // Được phép thay đổi trang thái đối tượng (6)
        den= new BongDen();      //Sai vì không cho phép thay đổi đối tượng (7)
    }
    }
```

public static double KWH_GIA được định nghĩa ở (1) như là *hằng* trong chương trình Java và do vậy sau đó không thể thay đổi (3). Hàm *final datWatt()* ở (2) không cho phép nạp chồng, viết đè như ở (4). Tương tự như vậy, đối tượng tham chiếu *final den* được định nghĩa ở (5) thì sau đó không thay đổi tham chiếu được (7), nhưng được phép thay đổi trạng thái của đối tượng đó (thay đổi thành phần).

(viii) Các hàm thành phần abstract

Hàm thành phần khai báo trừu tượng (*abstract*) có dạng:

abstract <Kiểu trả lại> <Tên hàm>([<Danh sách tham biến>])[<Mệnh đề throws>];

Hàm *abstract* là hàm *prototype*, chỉ khai báo phần định danh hàm mà không định nghĩa nội dung thực hiện, do vậy nó là hàm không đầy đủ. Hàm *abstract* thường chỉ tổ chức cho các lớp *abstract* và nó phải được cài đặt nội dung thực hiện ở trong các lớp con cháu của lớp chứa hàm đó.

Lưu ý:

- ✓ Hàm *final* không thể khai báo *abstract* và ngược lại.
- ✓ Các hàm trong *interface* đều là các hàm *abstract*.

(ix) Các lớp abstract

Lớp được khai báo *abstract* là lớp trừu tượng không có thể hiện. Cấu trúc này thường được sử dụng để nhằm thực hiện tổng quát hóa những khái niệm, những thực thể cụ thể trong thực tế. Ví dụ, chúng ta có thể sử dụng lớp *Vehicle* (phương tiện giao thông) như là lớp trừu tượng và làm lớp cơ sở để từ đó mở rộng thành những lớp cụ thể (không trừu tượng) *Car* (ô tô), *Bus* (ô tô bus), v.v.

Lớp *abstract* phải một hàm *abstract*. Những hàm này sau đó sẽ được cài đặt nội dung thực hiện trong các lớp con cháu của lớp chứa chúng.

Ví dụ 4.10. Minh họa khả năng truy nhập của lớp *abstract*

```
//NhaKho.java
abstract class BongDen{
    // Các biến thành phần                                        (1)
  int soWatts; 4
```

```
    boolean batTat;
    String viTri;
// Hàm thành phần
    public void batSang(){batTat = true; }
    public void tat(){batTat = false; }
    public boolean sangTat(){return batTat; }
    // Hàm abstract
    abstract public double xacDinhGia(); // Không cài đặt nội dung
}
class DenTuyp extends BongDen{
    // Biến thành phần
    int doDai;
    int mau;
// Cài đặt hàm abstract xacDinhGia()
    public double xacDinhGia(){
        return 500.00;
    }
}
class NhaKho{
    public static void main(String args[]){
    DenTuyp den1 = new DenTuyp();                          // (5)
    System.out.println("Gia dien: " + den1.xacDinhGia());
    BongDen den2;          // Được phép khai báo kiểu abstract      (6)
BongDen den3= new BongDen();      //Sai vì lớp abstract không có thể hiện
    }
}
```

(x) Các hàm thành phần đồng bộ synchronized

Java hỗ trợ chương trình thực hiện đa luồng (*multi threads*). Có thể có nhiều luồng muốn thực hiện đồng thời trên một đối tượng nào đó. Có những loại thiết bị, ví dụ như máy in, kênh truyền chẳng hạn, đòi hỏi phải có cơ chế để chỉ một luồng được thực hiện, nghĩa là phải thực hiện đồng bộ. Tại mỗi thời điểm, chỉ một luồng (tiến trình) được khai báo đồng bộ *synchronized* được thực hiện trên đối tượng chỉ định.

Ví dụ 4.11. Minh họa hoạt động của các hàm *synchronized*

```
class Stack{
  private Object[] stackArray;
  private int topfOfStack;
```

```
    synchronized public void push(Object elem){  // (1)
      stackArray[++topOfStack] = elem;
    }
    synchronized public Object pop(){          // (2)
      Object obj = stackArray[topOfStack];
      stackArray[topOfStack] = null;
      return obj;
    }
    // Những hàm khác
    public Object peek(){ return stackArray[topOfStack];
      }
  }
```

Hai hàm *push()* và *pop()* trong lớp *Stack* là đồng bộ *synchronized*. Do vậy khi có nhiều luồng muốn đẩy các phần tử vào *Stack* (hàm *push*()) hay muốn lấy ra các phần tử (hàm *pop*()) thì chỉ một luồng được phép thực hiện, còn những luồng khác sẽ phải chờ.

(xi) Các hàm thành phần native

Hàm khai báo *native* được gọi là hàm ngoại. Nội dung thực hiện của hàm *native* không được định nghĩa trong Java mà định nghĩa ở những ngôn ngữ lập trình khác như C/C++. Những hàm *native* chỉ cần khai báo *prototype* như là các thành phần của lớp.

JNI (*Java Native Interface*) là một loại API (*Abstract Programming Interface*) cho phép các hàm của Java gọi tới hàm ngoại được cài đặt trong C.

Ví dụ 4.12. Minh họa hoạt động của các hàm *native*

```
class Native{
/* Khối static này đảm bảo thư viện các hàm native được nạp xuống trước khi
   chúng được gọi.
*/
static {
  System.loadLibrary("NativeMethodLib");// Nạp thư viện
  }
native void nativeMethod();   // Hàm prototype native
// ...
}
class Khach{
```

```
    public static void main(String[] args){
      Native aNative = new Native();
      aNative.nativeMethod();
    }
  }
```

(xii) Các biến thành phần transient

Các đối tượng có thể lưu trữ tuần tự và việc tìm kiếm chúng sẽ trở nên đơn giản và nhanh khi các trạng thái của chúng là nhất quán, cố định.

Có những trường hợp, giá trị tham chiếu của đối tượng trong lớp là không nhất quán (hay thay đổi) khi đối tượng được lưu trữ. Những biến của các đối tượng như thế có thể khai báo với thuộc tính *transient* để chỉ ra rằng giá trị của nó sẽ không được lưu trữ khi đối tượng của lớp đó được tạo ra tuần tự trong bộ nhớ.

Trong ví dụ sau, biến *nhietDo* của lớp *ThiNghiem* được khai báo *transient* ở (1) bởi vì nó là yếu tố thay đổi nhiều nhất khi làm thí nghiệm ở những ngày tiếp theo. Biến *khoiLuong* khai báo ở (2) lại bình thường, ít thay đổi. Khi tổ chức lưu trữ các kết quả *ThiNghiem* một cách tuần tự trong bộ nhớ thì giá trị *nhietDo* sẽ không được ghi lại mà chỉ ghi lại các giá trị *khoiLuong*.

```
class ThiNghiem implements Serializable{
  // ...
  transient int nhietDo; // Giá trị hay thay đổi
  double khoiLuong;      // Giá trị cố định
}
```

Lưu ý: Thuộc tính *transient* không thể sử dụng cùng với biến *static*.

(xiii) Các thành phần volatile

Trong quá trình thực hiện, một số luồng có thể thực hiện che giấu những giá trị của các biến thành phần vì nhiều lý do khác nhau. Bởi vì một số luồng có thể cùng chia sẻ một biến và khi đó các thao tác đọc, ghi dữ liệu cho các biến chung đó có thể dẫn tới kết quả không nhất quán. Giá trị của những biến như thế có thể thay đổi mà không lường trước được. Java sử dụng bổ ngữ *volatile* để khai báo cho những biến có thể thay đổi bất thường và thông báo cho chương trình dịch không nên thực hiện tối ưu đối với những biến như thế.

Ví dụ trong lớp *DieuKhien* dưới đây, biến *docGio* của đồng hồ có thể khai báo *volatile* vì giá trị của nó có thể thay đổi trong quá trình điều khiển của một luồng

đang thực thi công việc mà các luồng khác không biết được và đảm bảo luôn đọc được giá trị hiện thời của đồng hồ.

```
class DieuKhien{
  // ...
  volatile long docGio;
  // 2 lần đọc liên tiếp sẽ cho kết quả khác nhau
}
```

Tóm lại, các từ khóa mô tả các đặc tính các thành phần được phép sử dụng cho lớp, hàm thành phần, biến thành phần và biến cục bộ để xác định phạm vi nhìn thấy của chúng trong hệ thống. Chúng ta tổng kết lại trong bảng sau.

Từ khoá	Lớp	Hàm thành phần	Biến thành phần	Biến cục bộ
abstract	✔	✔	-	-
static	-	✔	✔	-
public	✔	✔	✔	-
protected	✔	✔	✔	-
private	-	✔	✔	-
final	✔	✔	✔	✔
synchronized	-	✔	-	-
native	-	✔	-	-
transient	-	-	✔	-
volatile	-	-	✔	-

Trong đó dấu '✔' nghĩa là được phép sử dụng còn dấu '-' là không được phép sử dụng.

4.4. CÁC ĐỐI SỐ CỦA CHƯƠNG TRÌNH

Giống như chương trình của C/C++, chúng ta có thể truyền các tham số cho chương trình trên dòng lệnh, ví dụ:

java Colors red green blue

Chương trình java thông dịch lớp *Colors* để xác định số các chữ cái trong các đối số *red, green, blue* Chương trình *Colors* có thể viết như sau:

Ví dụ 4.13. Truyền tham số cho chương trình

```
// Colors.java
```

```
class Colors{
  public static void main(String args[]){
   for (int i = 0; i < args.length; i++)
    System.out.println("Đối số: " + (i+1) + "(" + args[i])
    + ")có " + args[i].length() + "ký tự");
  }
}
```

Khi thực hiện chương trình cho kết quả:

Đối số: 1 (red) có 3 ký tự
Đối số: 2 (green) có 5 ký tự
Đối số: 3 (blue) có 4 ký tự

4.5. TOÁN TỬ TẠO LẬP ĐỐI TƯỢNG

Mục đích chính của toán tử tạo lập (*constructor*) là đặt các giá trị khởi tạo cho các đối tượng khi một đối tượng được tạo ra bằng toán tử *new*.

Toán tử tạo lập có dạng khai báo như sau:

[<Thuộc tính>] <Tên lớp> ([<danh sách tham biến>]) {
  // Nội dung cần tạo lập
}

Toán tử tạo lập giống như các hàm thành phần và chỉ khác ở phần đầu của định nghĩa. Phần [<Thuộc tính>] <Tên lớp> gồm những thông tin sau:

- ✓ <Thuộc tính> chỉ có thể sử dụng: *public, protected, private* hoặc mặc định như đã trình bày ở phần trên.
- ✓ <Tên lớp> luôn trùng với tên của lớp chứa toán tử đó.

Một số lưu ý đối với các toán tử tạo lập:

- ✓ Không sử dụng các bổ ngữ khác với toán tử tạo lập,
- ✓ Toán tử tạo lập là loại hàm đặc biệt không có kiểu trả lại, ngay cả kiểu *void* cũng không được sử dụng,
- ✓ Toán tử tạo lập chỉ sử dụng được với toán tử *new*.

4.5.1. TOÁN TỬ TẠO LẬP MẶC ĐỊNH

Toán tử tạo lập mặc định (*default*) là toán tử tạo lập không có tham biến.

<Tên lớp> () {/* ... */}

Khi một lớp không định nghĩa một toán tử tạo lập nào cả thì hệ thống sẽ tự động cung cấp toán tử tạo lập mặc định không tường minh. Toán tử tạo lập mặc định không tường minh tương đương với dạng:

<Tên lớp> () { } // Không làm gì cả

Ví dụ dưới đây trong lớp *BongDen* không định nghĩa toán tử tạo lập và khi tạo lập đối tượng thì sử dụng toán tử tạo lập không tường minh.

```
class BongDen{
   // Biến thành phần                    (1)
   int soWatts;
   boolean batTat;
   String viTri;
        // Không định nghĩa toán tử tạo lập
        // ...
       }
 class NhaKho{
       public static void main(String args[]){
   BongDen den1 = new BongDen();// Gọi toán tử tạo lập mặc định
   // ...
   }
 }
```

Toán tử tạo lập mặc định không tường minh được cung cấp:

BongDen() {}

Như định nghĩa ở trên, nó chẳng làm gì cả. Tuy nhiên, khi sử dụng với toán tử *new* để tạo ra đối tượng thì các biến kiểu nguyên thủy như *soWatts, batTat, viTri* sẽ được khởi tạo các giá trị mặc định tương ứng là *0, false, null.*

Chúng ta cũng có thể định nghĩa toán tử tạo lập mặc định tường minh. Ví dụ

```
class BongDen{
   // Biến thành phần
   (1)
   int soWatts;
   boolean batTat;
   String viTri;
// Định nghĩa toán tử tạo lập mặc định
BongDen(){
```

```
      soWatts = 40;
      batTat = true;
      viTri = new String("XX");
      }
  // ...
      }
class NhaKho{
  public static void main(String args[]){
      BongDen den1= new BongDen();// Gọi toán tử tạo lập mặc định tường minh
      // ...
      }
}
```

Lưu ý: Khi trong một lớp đã định nghĩa một hay nhiều toán tử tạo lập thì hệ thống sẽ không cung cấp toán tử tạo lập mặc định không tường minh. Do đó khi muốn sử dụng toán tử tạo lập mặc định thì phải định nghĩa tường minh toán tử đó. Ví dụ:

```
class BongDen{
  // Biến thành phần          (1)
  int soWatts;
  boolean batTat;
  String viTri;
    // Định nghĩa toán tử tạo lập không mặc định
    BongDen(int w, boolean s, String v){
  soWatts = w;
    batTat = s;
  viTri = new String(v);
  }
   // ...
   }
  class NhaKho{
 public static void main(String args[]){
  BongDen d1=new BongDen(); // Sai vì toán tử tạo lập mặc định không được định nghĩa
    BongDen d2 = new BongDen(100, true, "Nha bep"); // OK
    // ...
    }
  }
```

4.5.2. NẠP CHỒNG TOÁN TỬ TẠO LẬP

Giống như các hàm thành phần, các toán tử tạo lập có thể nạp chồng (*tải bội - overloaded*) với nhiều nội dung thực hiện khác nhau. Ví dụ, trong lớp *BongDen* thì toán tử tạo lập *BongDen*() có định nghĩa mặc định tường minh (1), và nạp chồng để khởi tạo đối tượng với các giá trị xác định khác. Ví dụ,

```
class BongDen{
  // Biến thành phần                    (1)
  int soWatts;
  boolean batTat;
  String viTri;
  // Định nghĩa toán tử tạo lập mặc định

BongDen(){
  soWatts = 40;
  batTat = true;
  viTri = new String("XX");
  }
// Định nghĩa toán tử tạo lập không mặc định

 BongDen(int w, boolean s, String v){
    soWatts = w;
    batTat = s;
    viTri = new String(v);
    }
   //...
     }

 class NhaKho{
      public static void main(String args[]){
          BongDen d1=new BongDen();            // OK
          BongDen d2 = new BongDen(100, true, "Nha bep");    // OK
          //...
          }
 }
```

4.5.3. SỰ HOÀN THÀNH CỦA ĐỐI TƯỢNG

Một số đối tượng có thể gắn với những tài nguyên (như tệp, kết nối mạng) và chúng đòi hỏi phải giải phóng tường minh. Cơ chế hoàn thành của đối tượng cung cấp một kế sách để một đối tượng có thể thực hiện một hành động bất kỳ trước khi bộ nhớ bị giải phóng. Bộ dọn rác tự động (*Automatic Garbage Collector*) gọi hàm *finalize()* đã được xây dựng trong lớp *Object* có dạng:

protected void finalize() throws Throwable

để tập hợp rác lại (bỏ vào thùng rác) trước khi tiêu huỷ chúng.

Hàm này có thể được viết đè ở lớp con để thực hiện những công việc thích hợp trước khi hủy bỏ đối tượng.

Ví dụ 4.14. Sử dụng bộ finalizer

```
// Tệp: Finalizer.java
class Dien{                    // (1)
  static int dem;
  static int congChung;
  protected int lan;
  public Dien(){
  lan = dem++;
  ++congChung;
      }
     protected void finalize() throws Throwable {     // (2)
    super.finalize();
    --congChung;
      }
   }
   class TrinhDien extends Dien {        // (3)
     int[] fat;
     public TrinhDien(int n) {           // (4)
         fat = new int[n];
         System.out.println(lan + ": Hello");
    }
     protected void finalize() throws Throwable {  // (5)
         System.out.println(lan + ": Bye");
         super.finalize();
```

```
    }
  }
  public class Finalizer {
    public static void main(String args[]){   // (6)
      int soLan, thoiGian;
      try {
        soLan = Integer.parseInt(args[0]);// Đổi đối số 1 thành int
      thoiGian = Integer.parseInt(args[1]);
      }catch(IndexOutOfBoundsException e) {
       System.err.println("Usage: Finalizer <soLan> <thoiGian>");
       return;
      }
      for (int i = 0; i < soLan; i++){        // (7)
        new TrinhDien(thoiGian);
      }
      System.out.println(Dien.congChung+":tieptuc");   // (8)
    }
  }
```

Sau khi dịch và thực hiện lệnh

Java Finalizer 5 500000

Sẽ cho kết quả

```
0: Hello
1: Hello
1: Bye
2: Hello
2: Bye
3: Hello
3: Bye
4: Hello
1: tiep tuc
```

Ví dụ trên minh họa cách sử dụng *finalize()*. Chương trình tạo ra một số đối tượng (*soLan*) với kích thước (*thoiGian*) được đưa vào dưới dạng đối số của chương trình. Chu trình ở (7) trong hàm *main()* tạo ra đối tượng của lớp *TrinhDien*, nhưng không được lưu trữ để truy nhập tiếp. Việc tạo ra một đối tượng của *TrinhDien* được thực

hiện bởi toán tử tạo lập (4), xin cấp phát mảng các số nguyên cho *fat*. Mỗi lần tạo ra đối tượng của *TrinhDien* ở (4) lại in ra màn hình số *lan* và ": Hello". Hàm *finalize()* ở (5) được gọi trước khi đối tượng được đưa vào thùng rác. Mỗi lần thực hiện nó in ra số *lan*, ": Bye" và sau đó gọi hàm *finalize()* của lớp *Dien* ở (2). Kết quả của chương trình cho thấy một đối tượng chưa được dọn và nó còn tiếp tục hoạt động trong hệ thống. Lưu ý kết quả in ra màn hình (số đối tượng tạo ra và được dọn) phụ thuộc nhiều vào tham số thứ hai, *thoiGian* càng lớn thì hầu như chỉ có một đối tượng chưa được dọn, *thoiGian* nhỏ như khoảng nhỏ hơn 9000 thì cả 5 đối tượng sẽ vẫn tiếp tục hoạt động.

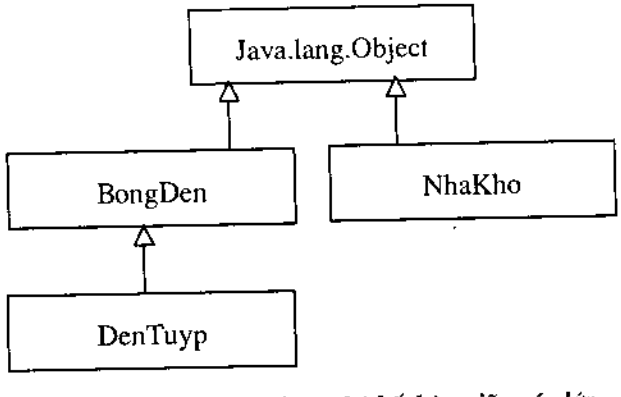
4.6. QUAN HỆ KẾ THỪA GIỮA CÁC LỚP

Một trong các cơ chế cơ cở để sử dụng lại được các đoạn chương trình trong lập trình hướng đối tượng là cơ chế kế thừa. Từ một lớp được xây dựng tốt, chúng ta có thể dẫn xuất ra nhiều lớp mới, nhiều kiểu mới (được gọi là lớp con, lớp dẫn xuất, hay lớp kế thừa). Lớp cơ sở còn được gọi là lớp cha, lớp cha mẹ. Lớp con có thể được bổ sung thêm một số thuộc tính và hàm để xác định thêm các tính chất, hành vi của những đối tượng cụ thể hơn hoặc có thể thay đổi các hàm thành phần được kế thừa (*public, protected*, mặc định như trình bày ở mục 4.3) từ lớp cha.

Java chỉ hỗ trợ kế thừa đơn (tuyến tính), nghĩa là một lớp chỉ kế thừa được từ một lớp cha. Do vậy giữa các lớp có quan hệ kế thừa với nhau có thể tổ chức thành cấu trúc phân cấp có thứ bậc. Trong Java lớp cơ sở nhất, làm cơ sở (gốc của cấu trúc phân cấp) cho tất cả các lớp kế thừa là lớp *Object*.

Chúng ta nhắc lại là *mọi lớp của Java đều là lớp con cháu mặc định của Object*.

Ví dụ, các lớp ở ví dụ 4.8, 4.9, 4.10 các lớp *BongDen, DenTuyp, NhaKho* có thể mô tả quan hệ kế thừa theo ký pháp của UML như hình 4.6.



*Hình 4.6. **Quan hệ kế thừa giữa các lớp.***

Cú pháp qui định quan hệ kế thừa trong Java là sự mở rộng của lớp cha, có dạng:

```
<Tên lớp con> extends <Tên lớp cha> {
        // Các thuộc tính dữ liệu bổ sung
        // Các hàm thành phần bổ sung hay viết đè
}
```

Lưu ý:

- ✓ Mọi đối tượng của lớp con cũng sẽ là đối tượng thuộc lớp cha. Do vậy việc gán một đối tượng của lớp con sang cho biến tham chiếu đối tượng của lớp cha là sự mở rộng kiểu và do đó không cần ép kiểu.
- ✓ Ngược lại gán một đối tượng của lớp cha cho biến tham chiếu đối tượng thuộc lớp con sẽ phải thực hiện ép kiểu. Lưu ý khi đó sẽ có thể bị tổn thất thông tin. Ví dụ,

```
class SuperClass  {/*  ...  */}
class SubClass extends SuperClass {/*  ...  */}
class UserClass {
    //
  publish static void main (String args []) {
    SuperClass super1 = new SuperClass ();  // Tạo ra đối tượng lớp cha
    SuperClass super2 = (SubClass) sub1;  // Thu hẹp kiểu nên phải ép kiểu
    SubClass sub1 = super1;   // Mở rộng kiểu
    }
```

4.6.1. TOÁN TỬ MÓC XÍCH GIỮA CÁC LỚP KẾ THỪA THIS() VÀ SUPER()

Các toán tử tạo lập không thể viết đè ở các lớp dẫn xuất (lớp con). Chúng có thể được nạp chồng nhưng phải trong cùng lớp.

Trong Java có 2 toán tử tạo lập đặc biệt có tên là *this()* và *super()* được sử dụng để móc xích giữa các lớp có quan hệ kế thừa với nhau.

Toán tử tạo lập this()

Toán tử tạo lập này được sử dụng để tạo ra đối tượng của lớp hiện thời.

Ví dụ 4.15. Nạp chồng toán tử tạo lập

```
// Tệp: NhaKho.java
class BongDen{
      // Biến thành phần                    (1)
     private int soWatts;
```

```
    private boolean batTat;
    private String viTri;
    // Định nghĩa toán tử tạo lập mặc định, số 1          (2)
    BongDen() {
      soWatts = 40;
      batTat = true;
      viTri = new String("XX");
      System.out.println("Toán tử số 1");
      }
    // Định nghĩa toán tử tạo lập không mặc định, số 2          (3)
      BongDen(int w, boolean s) {
      soWatts = w;
      batTat = s;
      viTri = new String("XX");
      System.out.println("Toán tử số 2");
      }
    // Định nghĩa toán tử tạo lập không mặc định, số 3 nạp chồng          (4)
    BongDen(int soWatts, boolean batTat, String viTri) {
      this.soWatts = soWatts;
      this.batTat = batTat;
      this.viTri = new String(viTri);
      System.out.println("Toán tử số 3");
      }
    // ...
    }
  public class NhaKho{
      public static void main(String args[]) {
          BongDen d1=new BongDen();          // OK
          BongDen d2 = new BongDen(100, true, "Nha bep");  // OK
          BongDen d3 = new BongDen(100, true);          // OK
          // ...
        }
    }
```

Trong lớp *BongDen* có 3 toán tử tạo lập được nạp chồng ở (1), (2) và (3). Tham chiếu *this* cũng được sử dụng trong hàm *main()* ở (4) để xác định các đối số tương ứng đối của toán tử tạo lập.

Ví dụ 4.16. Sử dụng toán tử *this()*

```
// Tệp: NhaKho.java
  class BongDen{
    // Biến thành phần                                                    (1)
    private int soWatts;
    private boolean batTat;
    private String viTri;
    // Định nghĩa toán tử tạo lập mặc định, số 1                          (2)
    BongDen() {
      this(40, true);
      System.out.println("Toán tử số 1");
      }
    // Định nghĩa toán tử tạo lập không mặc định, số 2                    (3)
    BongDen(int w, boolean s){
      this(w, s, "XX");
      System.out.println("Toán tử số 2");
      }
    // Định nghĩa toán tử tạo lập không mặc định, số 3 nạp chồng          (4)
    BongDen(int soWatts, boolean batTat, String viTri){
      this.soWatts = soWatts;
      this.batTat = batTat;
      this.viTri = new String(viTri);
      System.out.println("Toán tử số 3");
      }
    }
  public class NhaKho{
      public static void main(String args[]){
          BongDen d1 = new BongDen();                                    // OK
          BongDen d2 = new BongDen(100, true, "Nha bep");                // OK
          BongDen d3 = new BongDen(100, true);                           // OK
        // ...
      }
  }
```

Chương trình này hoạt động giống như chương trình ở ví dụ 4.15, nhưng khác là trong định nghĩa 2 toán tử tạo lập ở (1) và (2) có sử dụng cấu tử *this()* để gán các đối số tương ứng cho các đối tượng mỗi khi tạo lập ra chúng.

Toán tử tạo lập super()

Toán tử *super()* được sử dụng trong các toán tử tạo lập của lớp con (*subclass*) để gọi tới các toán tử tạo lập của lớp cha (*superclass*) trực tiếp.

Ví dụ 4.17. Sử dụng toán tử super()

```
// Tệp: NhaKho.java
    class BongDen{
        // Biến thành phần                                    (1)
        private int soWatts;
        private boolean batTat;
        private String viTri;
        // Định nghĩa toán tử tạo lập mặc định, số 1          (2)

    BongDen(){
        this(40, true);
        System.out.println("Toán tử số 1");
        }
    // Định nghĩa toán tử tạo lập không mặc định, số 2        (3)
    BongDen(int w, boolean s){
        this(w, s, "XX");
        System.out.println("Toán tử số 2");
        }
    // Định nghĩa toán tử tạo lập không mặc định, số 3 nạp chồng   (4)
    BongDen(int soWatts, boolean batTat, String viTri){
        this.soWatts = soWatts;
        this.batTat = batTat;
        this.viTri = new String(viTri);
        System.out.println("Toán tử số 3");
        }
    }
    class DenTuyp extends BongDen {
        private int doDai;
        private int mau;
        DenTuyp(int leng, int colo){                    // (5)
        this(leng, colo, 100, true, "Chua biet");
    }
```

```
        DenTuyp(int leng, int colo, int soWatt,
                boolean bt, String noi){    // (6)
        super(soWatt, bt, noi);
            this.doDai = leng;
            this.mau = colo;
    }
}
    public class NhaKho{
        public static void main(String args[]){
        System.out.println("Tao ra bong den tuyp");
        DenTuyp d = new DenTuyp(20, 5);
        }
    }
```

BongDen là lớp cha (trực tiếp) của lớp *DenTuyp*. Toán tử *super()* ở (6) là đại diện cho toán tử tạo lập của lớp cha *BongDen*.

Một số lưu ý khi sử dụng super() và this():

- ✓ Chúng chỉ sử dụng để xây dựng các toán tử tạo lập của các lớp và khi sử dụng thì chúng luôn phải là lệnh đầu tiên trong định nghĩa của toán tử tạo lập.
- ✓ *this()* được sử dụng để móc xích với cùng lớp chứa nó còn *super()* lại được sử dụng để móc xích với lớp cha của lớp đó.

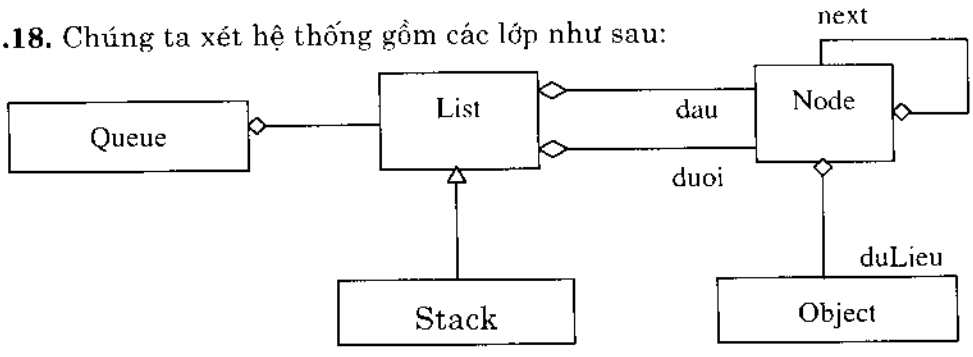
4.6.2. QUAN HỆ KẾ THỪA VÀ QUAN HỆ KẾT TẬP

Vấn đề rất quan trọng trong phát triển phần mềm hướng đối tượng là xây dựng được biểu đồ lớp mô tả đầy đủ mối quan hệ giữa các lớp của hệ thống ứng dụng [2]. Hai lớp có thể có các quan hệ sau:

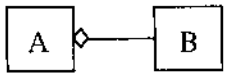
- ✓ Quan hệ liên kết (Association),
- ✓ Quan hệ kết tập (Aggregation),
- ✓ Quan hệ kế thừa (Inheritance),
- ✓ Quan hệ phụ thuộc (Dependency).

Hai quan hệ kế thừa và kết tập đều có đặc điểm chung là loại quan hệ giữa "một bộ phận" và phía "tổng thể". Quan hệ kế thừa là dạng "is a" (là một thành viên) còn quan hệ kết tập có dạng "has a" (có một thành viên). Nhưng chúng cũng có nhiều đặc trưng khác nhau. Vấn đề đặt ra cho người lập trình là chọn quan hệ nào thích hợp cho bài toán cần giải quyết. Chúng ta hãy xét biểu đồ lớp của UML được cho như trong hình 4.7.

Ví dụ 4.18. Chúng ta xét hệ thống gồm các lớp như sau:



Hình 4.7. Kế thừa và kết tập.

Trong đó 

ký hiệu cho quan hệ kết tập: Lớp A chứa các phần tử thuộc lớp B thể hiện mối quan hệ giữa tổng thể (bên chứa hình quả trám) và bên bộ phận.

Lớp *Queue* là một kết tập của *List*. *List* là lớp cơ sở để *Stack* kế thừa và *List* lại là kết tập của hai *Node* (dau, duoi). Tiếp theo *Node* lại là kết tập của *Object* (*duLieu*).

Chuyển đổi biểu đồ lớp trên sang Java chúng ta nhận được chương trình:

```
class Node {                                        // (1)
  private Object duLieu;                            // dulieu
  private Node next;                                // next node
  public Node(Object obj, Node link){               // Constructor
    duLieu = obj;
    next = link;
  }
// Accessor
  public void setDL(Object obj){duLieu = obj;}
  public Object getDL(){ return duLieu;}
  public void setNext(Node node){next = node;}
  public Node getNext() { return next;}
}
class List{                 // (2)
  protected Node dau = null;
  protected Node duoi = null;
  public void insertInFront(Object obj){
    if (isEmpty()) dau = duoi = new Node(obj,null);
    else dau = new Node(obj,dau);
  }
  public void insertAtBack(Object obj){
```

```
        if (isEmpty()) dau = duoi = new Node(obj,null);
        else { duoi.setNext(new Node(obj,dau));
          duoi = duoi.getNext();
          }
    }
    public Object deleteFront(){
      if(isEmpty()) return null;
      Node removed = dau;
      if(dau == duoi) dau = duoi = null;
      else dau = dau.getNext();
      return removed.getDL();
  }
  public boolean isEmpty(){ return dau == null;}
  }
  class Queue{
    private List qlist;                                    //(3)
    public Queue(){
      qlist = new List();
      }
    public void enqueue(Object item){
      qlist.insertAtBack(item);
      }
    public Object dequeue(){
      if (empty()) return null;
      else return qlist.deleteFront();
      }
    public Object peek(){
      Object obj = dequeue();
      if (obj != null) qlist.insertInFront(obj);
      return obj;
      }
    public boolean empty(){ return qlist.isEmpty();}
  }
  class Stack extends List{                                //(4)
    public void push(Object obj){insertInFront(obj);}
    public Object pop(){
      if(empty()) return null;
```

```
      else return deleteFront();
      }
    public Object peek(){
      return (isEmpty() ? null : dau.getDL());
      }
    public boolean empty() {return isEmpty();}
  }
  public class KhachHang{
    public static void main(String args[]){
      String xau1 = "Chung ta dang xep hang!";
      int leng = xau1.length();
      Queue q = new Queue();
      for(int i = 0; i < leng; i++)
        q.enqueue(new Character(xau1.charAt(i)));
      while(!q.empty())
        System.out.print((Character)q.dequeue());
      System.out.println();
      String xau2 = "pex nagn o gnad at gnuhC";
      int leng2 = xau2.length();
      Stack s = new Stack();
      for(int i = 0; i < leng2; i++)
        s.push(new Character(xau2.charAt(i)));
      while(!s.empty())
        System.out.print((Character)s.pop());
      System.out.println();
      }
  }
```

Kết quả chương trình in ra:

```
Chung ta dang xep hang!
Chung ta dang o ngan xep
```

4.7. GIAO DIỆN VÀ SỰ MỞ RỘNG QUAN HỆ KẾ THỪA TRONG JAVA

Như trên chúng ta đã thấy Java hỗ trợ kế thừa tuyên tính để tạo ra nhiều kiểu dữ liệu mới. Trong phương pháp hướng đối tượng quan hệ kế thừa còn rộng hơn. Ngoài kế thừa tuyến tính, còn có quan hệ kế thừa bội (*multiple*), nghĩa là một lớp có thể kế thừa từ nhiều hơn một lớp cha. Với quan hệ kế thừa bội thì biểu đồ lớp với quan hệ

kế thừa không còn là cấu trúc phân cấp như đã nêu ở trên mà sẽ là đồ thị. Nhiều vấn đề sẽ nảy sinh như vấn đề xung đột kiểu, vấn đề nhập nhằng dữ liệu, v.v., sẽ gây ra rất nhiều trở ngại cho người thiết kế hệ thống. Java không hỗ trợ trực tiếp loại kế thừa này, song để tránh những trở ngại trên Java cung cấp khái niệm *interface* và cho phép kế thừa bội đối với các *interface*.

Định nghĩa interface

interface định nghĩa mẫu hợp đồng thông qua việc phác thảo các hàm mẫu (*prototype*) và không cài đặt nội dung thực hiện.

```
interface <Tên interface > {
  <Nội dung của interface >
    }
```

<Nội dung của interface> thường chứa danh sách các hàm mẫu và các hằng.

Giao diện *interface* là loại kiểu lớp đặc biệt, trong đó tất cả các hàm thành phần đều là trừu tượng, do vậy không thể khởi tạo được giá trị, nghĩa là không thể tạo ra đối tượng của *interface*. Những hàm trong *interface* sẽ được cài đặt ở những lớp xây dựng mới theo dạng:

```
class <Tên lớp> implements <Tên interface> {
      // Bổ sung các thành phần;
      // Cài đặt các hàm mẫu đã cho trong <Tên interface>;
}
```

Sự mở rộng (kế thừa) của các interface

Giống như cấu trúc lớp, *interface* cũng có thể được mở rộng từ nhiều *interface* khác bằng mệnh đề *extends*. Tuy nhiên khác với sự mở rộng tuyến tính của các lớp, các *interface* được phép mở rộng không tuyến tính, nghĩa là có thể mở rộng nhiều hơn một *interface*. Ví dụ,

```
class MyClass implements Interface1, Interface2 {
      // Cài đặt một số hàm mẫu của Interface1, Interface2;
}
interface Interface1{
      // Khai báo các hằng và hàm mẫu;
}
interface Interface2{
      // Khai báo các hằng và hàm mẫu;
}
```

Ví dụ 4.19. Cách xây dựng và sử dụng các *interface*

```
interface IStack{          //      (1)
  void push(Object item);
  Object pop();
  }
class StackImpl implements IStack { //      (2)
  protected Object[] stackArray;        // Bổ sung mảng đối tượng
  protected int tos;                    // Bổ sung biến tos
  public StackImpl(int capacity) {
    stackArray = new Object[capacity];
    tos = -1;
  }
  public void push(Object item) { // Cài đặt hàm push() (3)
    stackArray[++tos] = item;
  }
  public Object pop() {      // Cài đặt hàm pop()(4)
    Object obj = stackArray[tos];
    stackArray[tos] = null;
    tos--;
    return obj;
  }
    public Object peek() {       // Bổ sung hàm peek() (5)
    return stackArray[tos];
    }
  }
  // Xây dựng thêm interface IsafeStack kế thừa Istack để kiểm tra các trạng thái của
  // Stack đảm bảo an toàn khi thao tác.
  interface ISafeStack extends IStack {       // (6)
    boolean isEmpty();
    boolean isFull();
  }
  // Xây dựng lớp để cài đặt các hàm của IsafeStack          // (7)
  class SafeStackImpl extends StackImpl implements ISafeStack {
    public SafeStackImpl(int capacity) {super(capacity);}
    public boolean isEmpty() {return tos < 0;}
    public boolean isFull() {return tos == stackArray.length-1;}
  }
```

```
// Xây dựng lớp sử dụng các lớp và giao diện đã cài đặt ở trên
public class StackUser {
  public static void main(String args[]){
    SafeStackImpl safe1 = new SafeStackImpl(80); // (8)
    StackImpl s1 = safe1;                        // (9)
    ISafeStack iSafe1 = safe1;                   // (10)
    IStack iS1 = safe1;
    Object obj = safe1;
    safe1.push("Khoa CNTT");
    s1.push("ĐHQG");
    System.out.println(iSafe1.pop());            // (11)
    System.out.println(iS1.pop());               // (12)
    System.out.println(obj.getClass());          // (13)
  }
}
```

Kết quả thực hiện chương trình:

ĐH QG

Khoa CNTT

class SafeStackImpl

Lớp *StackImpl* cài đặt *interface IStack*, trong đó định nghĩa nội dung thực hiện của các hàm mẫu *push()* và *pop()* ở (3), (4) tương ứng. Giao diện *interface ISafeStack* mở rộng *IStack* ở (6). Lớp *SafeStackImpl* thực hiện đồng thời mở rộng *StackImpl* và cài đặt giao diện *ISafeStack* ở (7). Một đối tượng của lớp *SafeStackImpl* được tạo ra ở (8) và sau đó có thể gán cho tất cả các biến tham chiếu các lớp cha (9), giao diện cha (10) (được gọi chung là kiểu cha). Lệnh *obj.getClass()* ở (13) cho lại tên của lớp mà *obj* là đại diện.

Lưu ý:

- ✓ Các hàm mẫu của *interface* được mặc định là *abstract* và không được khai báo *static*;
- ✓ Một lớp có thể chọn một số hàm mẫu để cài đặt, nghĩa là có thể chỉ cài đặt một phần của giao diện *interface*;
- ✓ Một giao diện có thể kế thừa từ nhiều hơn một giao diện;
- ✓ Một giao diện có thể kế thừa từ nhiều hơn một giao diện và lớp;

Ngoài ra cũng cần chú ý thêm rằng, các giao diện *interface* có thể định nghĩa các hằng (*constant*). Tất cả các thuộc tính dữ liệu trong *interface* luôn được xem mặc

định là *public static* và *final*, do vậy không cần quan tâm đến các bổ ngữ của chúng mà luôn xem chúng là hằng.

Ví dụ 4.20. Các biến hằng trong interface

```
//Tệp: Tinh.java
interface Constants {
    double PI = 3.14;
    String DON_VI_DT = "cm²";
    String DON_VI_CV = "cm";
}
public class Tinh implements Constants {
  public static void main(String args[]){
    double banKinh = 15.00;
    System.out.println("Dien tich hinh tron ban kinh" + banKinh +
        "là" + (PI * banKinh * banKinh) + DON_VI_DT); //(1)
    System.out.println("Chu vi hinh tron ban kinh " + banKinh + "là"
        + (2 * PI * banKinh) + Constants.DON_VI_CV); //(2)
  }
}
```

Các biến trong *interface* mặc định là *public static final* do vậy khi truy nhập tới chúng có thể truy nhập trực tiếp (1) hoặc thông qua tên *interface* (2).

BÀI TẬP

4.1. Tìm lỗi của chương trình sau và giải thích tại sao sai

```
public MyClass {
  long var;
  public void MyClass(long param) {      // (1)
    var = param;
    }
public static void main(String args){    // (2)
  MyClass a, b;
  a = new MyClass();                     // (3)
  b = new MyClass(15);                   // (4)
  }
}
```

4.2. Chương trình sau có chỗ nào sai ?, tại sao

```
    abstract class MyClass {
    abstract void f();
    final void g(){};                          // (1)
    final void h(){};
    protected static int i;
    private int j;
}
final class YourClass extends MyClass {
    YourClass(int n) {m = n; }                 // (2)
    public static void main(String arg[]){
        MyClass mc = new YourClass();
    }
    void f(){}
    void h(){}
    void k(){i++;}                             // (3)
    void l() {j++;}                            // (4)
int m;
}
```

4.3. Cho biết kết quả in ra màn hình của chương trình sau

```
class Base {
    int i;
    Base(){
        aad(1);
    }
    void add(int n){
        i += n;
    }
    void print(){
        System.out.println(i);
    }
  }
class Ext extends Base {
    Ext(){ add(2);  }
    void add(int n) { i += n *2; }
  }
```

```
public class Q3 {
      public static void main(String arg[]){
          bibo(new Ext));
      }
      static void bibo(Base b) {
          b.add(8);
          b.print();
      }
    }
```

4.4. Cho trước đoạn chương trình sau

```
class A {}
class B extends A {}
class C extends A {}
public class Q4{
  public static void main(String args[]){
    A x = new A();
    B y = new B();
    C z = new C();
    // Nơi cần chèn các lệnh thích hợp      (1)
  }
}
```

Hãy chọn những lệnh thích hợp sau:

(a) x = y;

(b) z = x;

(c) y = (B) x;

(d) z = (C) y;

(e) y = (A) y;

4.5. Cho biết kết quả thực hiện của chương trình sau

```
public class Q5{
    int a, b;
    public void f(){
        a = b = 0;
        int[] c = {0};
        g(b, c);
```

```
        System.out.println(a + " " + b + " " + c[0]);
          }
          public void g(int b, int[] c){
              a = b = 1;
              c[0] = 1;
          }
          public static void main(String args[]){
              Q5 e = new Q5();
              e.f();
          }
      }
```

4.6. Chương trình sau dịch và chạy cho những kết quả là gì?

```
public class Q6{
    public static void main(String args[]){
        int i = 4;
        float f = 4.3F;
        double d = 1.3;
        int c = 0;
        if(i == f) c++;
        if(((int)(f+d)) == ((int) f + (int) d))
          c += 2;
        System.out.println(c) ;
    }
}
```

4.7. Chương trình sau có một số lỗi, hãy sửa các lỗi đó!

```
import java.util.*;
package com.acme'
public class Exerc1 {
    int counter;
    void main(String args){
        Exerc1 ins = new Exerc1();
        ins.go();
  }
    public void go(){
        int sum, i = 0;
```

```
        while (i < 100) {
            if (i == 0) sum = 100;
            sum += i; i++;
        }
        System.out.println(sum);
    }
}
```

4.8. Viết chương trình nhập vào số nguyên và cho kết quả là dãy các chữ số 0, 1 biểu diễn nhị phân của số đó.

4.9. Hãy xây dựng lớp *SinhVien* có các thuộc tính riêng (*private*):

- Số báo danh,
- Họ và tên sinh viên,
- Địa chỉ,
- Môn học,
- Điểm thi học kì I, II.

a/ Viết các hàm truy nhập tới các thành phần dữ liệu và các toán tử tạo lập cho lớp *SinhVien*.

b/ Viết chương trình chính để tạo ra danh sách *SinhVien* và hiển thị thực đơn:

1. Nhập vào thông tin về sinh viên,
2. Xem thông tin về sinh viên,
3. Tìm sinh viên theo điểm,
4. Kết thúc chương trình.

c/ Viết các hàm thành phần của 1 lớp để thực hiện các nhiệm vụ trên

4.10. Một công ty được giao nhiệm vụ quản lý các phương tiện giao thông gồm các loại ôtô, xe máy, xe tải.

+ Mỗi phương tiện giao thông cần quản lý: mô hình, năm sản xuất, giá bán và màu

+ Các ôtô cần quản lý: số chỗ ngồi, kiểu động cơ

+ Xe máy: công suất ,

+ Xe tải: Trọng tải

a/ Hãy xây dựng các lớp *XeTai*, *XeMay* và *OTo* kế thừa từ lớp *PhuongTienGT*

b/ Xây dựng các hàm để truy nhập, hiển thị và kiểm tra các thuộc tính của các lớp,

c/ Phát triển lớp *KiemKe* để quản lý tất cả các phương tiên (xe máy, ôtô, xe tải), trong đó có hàm chính thực hiện theo thực đơn:

1. Nhập đăng ký phương tiện,
2. Tìm phương tiện theo mô hình,
3. Tìm phương tiện theo màu,
4. Kết thúc.

d/ Viết các hàm để thực hiện nhiệm vụ trên.

CHƯƠNG V

CÁC LỆNH ĐIỀU KHIỂN DÒNG THỰC HIỆN VÀ XỬ LÝ NGOẠI LỆ

5.1. CÁC CÂU LỆNH ĐIỀU KHIỂN RẼ NHÁNH CHƯƠNG TRÌNH

Java cung cấp các lệnh rẽ nhánh (tuyển chọn) để có thể chọn các phương án khác nhau trong quá trình thực hiện của chương trình. Có ba loại lệnh rẽ nhánh để thực hiện: lệnh *if* đơn giản, *if - else* và *switch*.

Câu lệnh if đơn giản

Câu lệnh *if* đơn giản (lệnh *if*) có dạng cú pháp qui định:

```
if <Biểu thức điều kiện>
    <Câu lệnh>                    // (1)
```

Ngữ nghĩa thực hiện rất đơn giản: Trước tiên tính giá trị của <Biểu thức điều kiện>, nếu nó có giá trị *true* (đúng) thì thực hiện <Câu lệnh>, sau đó tiếp tục thực hiện các lệnh còn lại của chương trình. Trường hợp nhận giá trị *false* (sai) thì <Câu lệnh> bị bỏ qua và tiếp tục thực hiện các lệnh còn lại của chương trình.

Lưu ý: <Câu lệnh> ở (1) có thể là câu lệnh phức (khối chương trình) bao gồm một dãy các lệnh chứa trong cặp '{' và '}'.

Câu lệnh if-else

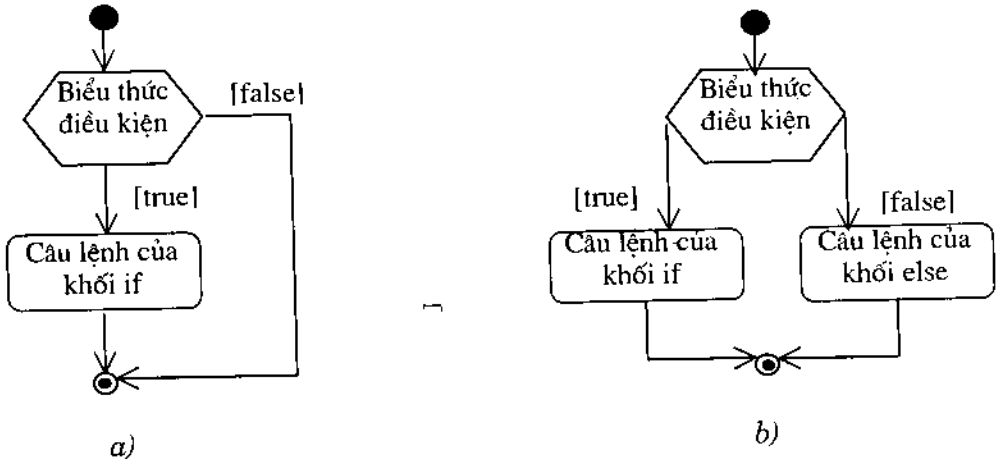
Cú pháp của lệnh *if-else* có dạng:

```
if <Biểu thức điều kiện>
    <Câu lệnh 1>                  // (2)
else
    <Câu lệnh 2>                  // (3)
```

Tương tự như trên, nếu <Biểu thức điều kiện> nhận giá trị *true* thì thực hiện <Câu lệnh 1>, còn gọi là câu lệnh *if*, ngược lại thực hiện <Câu lệnh 2>, còn gọi là câu lệnh *else*, sau đó tiếp tục thực hiện các lệnh còn lại theo dòng điều khiển của chương trình.

Lưu ý: <Câu lệnh 1> ở (2) và <Câu lệnh 2> ở (3) có thể là các câu lệnh phức (khối chương trình).

Hình 5.1 minh hoạ hoạt động của hai câu lệnh trên.



Hình 5.1. ***a) Câu lệnh if đơn giản,***
b/ Câu lệnh if- else .

Lưu ý: Các câu lệnh *if- else* có thể lồng nhau, nhưng khi đó chúng phải lồng nhau thực sự và không được cắt nhau. Chúng phải tuân theo qui tắc sau:

Qui tắc: Câu lệnh *else* luôn sánh với câu lệnh *if* gần nhất nếu câu lệnh đó chưa có lệnh *else* tương ứng. Ví dụ:

```
if (nhietDo >= canTren) {        // (1)
    if (nguyHiem) baoDong;       // (2)      Lệnh if đơn giản
    if (coNguyCo)                // (3)
        tinhToan();
    else                         // ứng với lệnh if ở (3)
        tatLoNhiet();
} else                           // ứng với lệnh if  (1)
    choLoNhietChay();
```

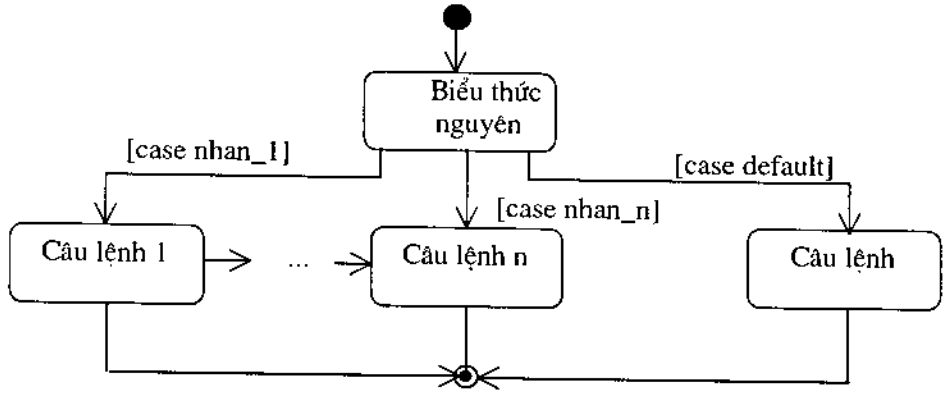
Câu lệnh switch

Câu lệnh này cho phép rẽ nhánh theo nhiều nhánh tuyển chọn *dựa trên các giá trị kiểu nguyên của biểu thức*. Nó có dạng:

```
switch (<Biểu thức nguyên>) {
        case nhan_1:   <Câu lệnh 1>
        case nhan_2:   <Câu lệnh 2>
        . . .
        case nhan_n:   <Câu lệnh n>
        default:   <Câu lệnh >
    }
```

Cách thực hiện như sau:

- ✓ Trước tiên tính giá trị của <Biểu thức nguyên>
- ✓ So sánh giá trị tính được với các *nhan_i, i = 1, 2, ... n*. Nếu giá trị đó bằng *nhan_i* thì thực hiện nhánh câu lệnh <*Câu lệnh i*> tương ứng. Sau khi thực hiện xong thì chuyển tiếp tới các lệnh sau đó theo điều khiển chương trình (nếu có *break* thì chuyển tới sau lệnh *switch*).



Hình 5.2. Sơ đồ mô tả hoạt động của câu lệnh **switch.**

Các câu lệnh *switch* có thể lồng nhau. Bởi vì mỗi lệnh *switch* định nghĩa các khối cục bộ riêng nên các nhãn của các *case* (trường hợp) bên trong không được xung đột với với các nhãn ở khối ngoài. Các nhãn có thể định nghĩa lại bên trong nhưng biến thì không cho phép khai báo lại.

Ví dụ 5.1. Các lệnh *switch* lồng nhau.

```
public class Mua {
public static void main(String args[]){
    int thang = 9;
    switch (thang) {            // Khối ngoài (1)
        case 11:
        case 12:
        case 1:
```

```
            System.out.println("Mua dong");
            break;
        case 2: case 3: case 4:
            System.out.println("Mua xuan");
            break;
        case 5: case 6: case 7:
            System.out.println("Mua he");
            break;
        case 8: case 9: case 10:
        switch(thang){              // Khối bên trong (2)
            case 8:
                System.out.println("Dau mua thu");
                break;
            case 9:
                System.out.println("Hoc sinh den truong");
                break;
            }
            System.out.println("Mua thu la vang roi!");
            break;
        default:
            System.out.println(thang + " khong phai la thang");
        }
    }
}
```

Kết quả thực hiện của chương trình Mua:

```
Hoc sinh den truong
Mua thu la vang roi!
```

5.2. CÁC CÂU LỆNH LẶP

Câu lệnh lặp (chu trình) cho phép một khối các câu lệnh thực hiện một số lần lặp lại. *Biểu thức boolean* được sử dụng để kiểm tra xem khi nào chu trình lặp kết thúc, được gọi là điều kiện kết thúc chu trình. Trong Java có ba cấu trúc điều khiển lặp:

- ✓ Câu lệnh *while*,
- ✓ Câu lệnh *do-while*,
- ✓ Câu lệnh *for*.

Câu lệnh while (chu trình while)

Câu lệnh *while* có dạng:

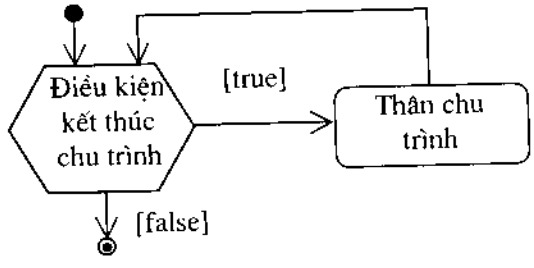
```
while (<Điều kiện kết thúc chu trình>)
    <Thân chu trình>
```

<Điều kiện kết thúc chu trình> là biểu thức *boolean* được tính trước khi thực hiện <Thân chu trình>. <Thân chu trình> sẽ được lặp lại cho đến khi <Điều kiện kết thúc chu trình> vẫn còn có giá trị *true*. Khi <Điều kiện kết thúc chu trình> nhận trị *false* thì chu trình kết thúc và thực hiện lệnh đứng sau trực tiếp câu lệnh chu trình.

Lưu ý:

- ✓ Có thể <Thân chu trình> không được thực hiện lần nào, nếu ngay từ đầu <Điều kiện kết thúc chu trình> có giá trị *false,*
- ✓ <Điều kiện kết thúc chu trình> phải là biểu thức *boolean.*

Dòng điều khiển của chu trình while được mô tả như trong hình 5.3.



Hình 5.3. ***Chu trình*** **while.**

Câu lệnh do-while (chu trình do-while)

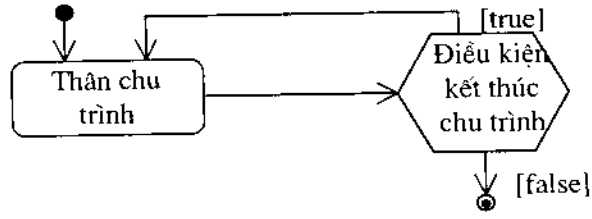
Câu lệnh *do-while* có dạng:

```
do
<Thân chu trình>
while (<Điều kiện kết thúc chu trình>)
```

<Điều kiện kết thúc chu trình> là biểu thức *boolean* được tính sau khi thực hiện <Thân chu trình>. <Thân chu trình> sẽ được lặp lại cho đến khi <Điều kiện kết thúc chu trình> vẫn còn có giá trị *true*. Khi <Điều kiện kết thúc chu trình> nhận trị *false* thì chu trình kết thúc và thực hiện lệnh đứng ngay sau câu lệnh chu trình đó.

Lưu ý: (Thân chu trình) phải thực hiện ít nhất một lần.

Hoạt động của chu trình do - while được mô tả như trong hình 5.4.



*Hình 5.4. **Chu trình do-while.***

Câu lệnh for (chu trình for)

Chu trình *for* là dạng tổng quát nhất của các câu lệnh lặp. Nó thường được sử dụng để điều khiển quá trình lặp khi số các lần lặp được biết trước.

Chu trình *for* có dạng:

for (<Biểu thức bắt đầu>; <Điều kiện lặp>; <Biểu thức gia tăng>)

<Thân chu trình>

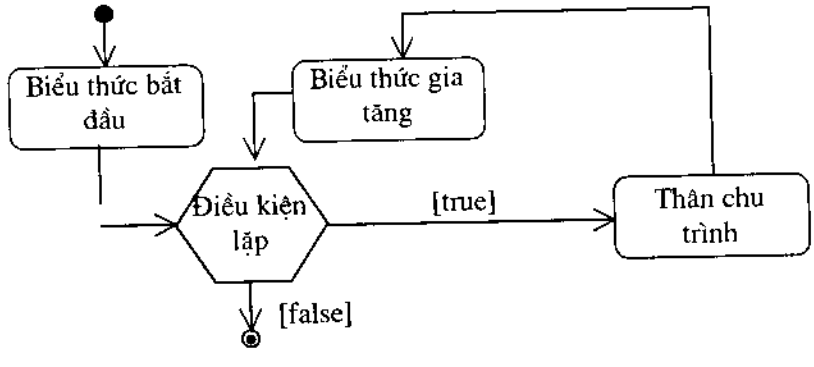
<Biểu thức bắt đầu> khai báo và gán các giá trị khởi đầu cho các biến điều khiển quá trình lặp của chu trình.

<Điều kiện lặp> là biểu thức *boolean*. Nếu <Điều kiện lặp> còn nhận trị *true* thì chu trình còn thực hiện, ngược lại, chu trình sẽ kết thúc và lệnh đứng sau nó sẽ được thực hiện.

Sau mỗi lần thực hiện <Thân chu trình> thì <Biểu thức gia tăng> lại được thực hiện. Biểu thức này thường phải làm thay đổi giá trị của các biến điều khiển số lần lặp của chu trình.

Lưu ý: <Biểu thức bắt đầu> chỉ thực hiện đúng một lần ngay khi câu lệnh *for* bắt đầu thực hiện.

Hoạt động của chu trình *for* được mô tả như sau:



*Hình 5.5. **Chu trình for.***

<Biểu thức gia tăng> có thể là danh sách các biểu thức (các biểu thức được phân cách nhau bằng dấu ','), còn <Biểu thức bắt đầu> lại có thể là danh sách các câu lệnh khai báo hoặc các biểu thức. Ví dụ, trong chu trình *for* sau có ba lệnh khai báo và một biểu thức trong <Biểu thức bắt đầu> và hai biểu thức trong <Biểu thức gia tăng> .

```
int[][] sqMatrix = {{3, 4, 6}, {3, 7, 4}, {5, 8, 9}};
for (int i =0, j = sqMatrix[0].length-1, diago = 0;
        j < sqMatrix.length;          // Biểu thức kết thúc
        i++, j++)                     // Biểu thức gia tăng
    diago += sqMatrix[i][j];
```

Lưu ý:

- ✓ Tất cả các biến được khai báo trong <Biểu thức bắt đầu> đều là cục bộ trong khối thân của chu trình *for*.
- ✓ Các thành phần của chu trình *for* là tùy chọn. Một trong <Biểu thức bắt đầu>, <Biểu thức gia tăng>, <Điều kiện kết thúc> có thể trống. Trường hợp <Điều kiện kết thúc> là trống thì điều kiện lặp của chu trình được xem là *true*.
- ✓ *for* (; ;) được sử dụng để xây dựng chu trình lặp vô điều kiện.

5.3. CÁC CÂU LỆNH CHUYỂN VỊ

Java cung cấp bốn cách để chuyển điều khiển thực hiện của chương trình:

- *break*
- `continue`
- `return`
- `try-catch-finally`

Câu lệnh break

Câu lệnh *break* được sử dụng trong các khối được gắn nhãn, trong các chu trình lặp (*for, while, do-while*) và câu lệnh *switch* để chuyển điều khiển thực hiện chương trình ra khỏi khối trong cùng chứa nó.

- ➢ Trong trường hợp khối được gắn nhãn, khi gặp lệnh *break* thì phần còn lại của khối bị cho qua và chương trình tiếp tục thực hiện lệnh đứng sau khối đó,
- ➢ Đối với các chu trình lặp, khi gặp lệnh *break* thì phần còn lại của thân chu trình được bỏ qua và kết thúc chu trình đó,
- ➢ Đối với lệnh *switch*, khi gặp lệnh *break* thì phần còn lại của lệnh *switch* bị bỏ qua và tiếp tục thực hiện lệnh đứng sau lệnh *switch* đó.

Ví dụ 5.2. Chương trình minh họa cách sử dụng khối được gắn nhãn và lệnh break

```
// Tệp Label.java
class Label {
    public static void main(String args[]){
        int[][] sqMatrix = {{4,3,5},{2,1,6},{9,7,8}};
        int sum = 0;
        outer:          // Gắn nhãn outer cho khối
        for (int i = 0; i < sqMatrix.length; i++){        // (1)
            for(int j = 0; j < sqMatrix[i].length; i++){// (2)
                if (i == j) break;// Kết thúc chu trình (2) và tiếp ở (5)
                System.out.println("Bang["+i+","+j+"] = " +
                        sqMatrix[i][j]);
                sum += sqMatrix[i][j];
                if (sum >10) break outer;     // Kết thúc khối outer (cả
                                              // hai chu trình for) và chuyển tới (6)
                }           // Kết thúc for bên trong
                // (5)
            }     // Kết thúc for bên ngoài
            // (6)
            System.out.println("Tong sum = " + sum);
        }
    }
```

Kết quả thực hiện của chương trình:

```
Bang[1, 0] = 2
Bang[2, 0] = 9
Tong sum   = 11
```

Câu lệnh continue

Câu lệnh *continue* được sử dụng trong các chu trình lặp *for, while, do-while* để dừng sự thực hiện của lần lặp hiện thời và bắt đầu lặp lại lần tiếp theo nếu điều kiện lặp còn thỏa mãn (còn *true*).

Đối với chu trình *while, do-while,* khi gặp *continue* thì phần còn lại của thân chu trình bị bỏ qua và tiếp tục kiểm tra điều kiện lặp để thực hiện quá trình lặp tiếp theo.

Đối với lệnh *for*, khi gặp *continue* thì phần còn lại của thân chu trình bị bỏ qua và tiếp tục thực hiện <Biểu thức gia tăng> sau đó thực hiện lặp lại nếu <Điều kiện kết thúc> còn *true*.

Ví dụ 5.3. Chương trình minh họa cách sử dụng lệnh *continue* để ngắt lần lặp hiện thời. Chương trình in ra màn hình các số 1, 2, 3 và 5 cùng với căn bậc 2 của chúng, còn với i = 4 thì không thực hiện.

```
class TiepTuc {
    public static void main(String args[]){
        for (int i = 1; i <= 5; ++i){
            if (i == 4) continue;
            // Phần còn lại sẽ bị bỏ qua
            System.out.println(i + "\t" + Math.sqrt(i));
            // Thực hiện ++i và kiểm tra để lặp lại
            }
        }
    }
```

Câu lệnh return

Câu lệnh *return* được sử dụng để kết thúc thực hiện của hàm hiện thời và chuyển điều khiển chương trình về lại sau lời gọi hàm đó. Lệnh này có hai dạng sử dụng tương ứng với hai loại hàm được xác định như trong bảng sau:

Dạng lệnh return	Hàm khai báo void	Hàm có kiểu trả lại khác void
return	Tùy chọn	Không cho phép
return <Biểu thức>	Không cho phép	Bắt buộc

Ví dụ 5.4. Chương trình minh họa các lệnh *return:* Kiểm tra xem số các đối số của chương trình có bằng 0 và in ra số đó nếu lớn hơn 3, ngược lại in ra số 2 khi có đối số.

```
class Label {
    public static void main(String args[]){
        if (args.length == 0) return;// Tùy chọn
        output(check(args.length));
    }
    static void output(int val){//
        System.out.println(val);
        // return 1;   Không được phép vì output() kiểu void
```

```
        }
    static int check(int i) {
        if (i > 3) return i;
        else return 2;// Nếu là return 2.0 sẽ sai vì không tương thích kiểu
        }
}
```

Câu lệnh *try-catch-finally* được sử dụng chủ yếu để xử lý các ngoại lệ. Câu lệnh này được đề cập chi tiết ở mục sau.

5.4. XỬ LÝ NGOẠI LỆ

Mục đích của công nghệ phần mềm là: Xây dựng các *module* phần mềm tốt, đảm bảo chúng làm việc được trong mọi tình huống và dễ dàng thích ứng với mọi điều kiện. Muốn đảm bảo hệ thống có được những tính chất trên thì phải có cơ chế để xử lý được tất cả các *ngoại lệ, các tình huống liên quan tới phạm vi xác định của các phần tử trong các cấu trúc dữ liệu.*

Một ngoại lệ (*exception*) trong chương trình Java là dấu hiệu chỉ ra rằng có sự xuất hiện một điều kiện không bình thường nào đó. Ví dụ, chương trình yêu cầu dữ liệu từ một tệp mà tệp đó lại không có trên đĩa, hay chỉ số của phần tử mảng vượt ra ngoài phạm vi xác định, v.v. Vấn đề kiểm soát tất cả các ngoại lệ như thế là rất phức tạp, nhàm chán và mất khá nhiều thời gian. Java cung cấp một cơ chế để kiểm tra một cách có hệ thống các điều kiện ngoại lệ. Để xử lý ngoại lệ trong Java, chúng ta sử dụng khối lệnh *try-catch-finally*.

5.4.1. CÁC KHỐI LỆNH TRY, CATCH VÀ FINALLY

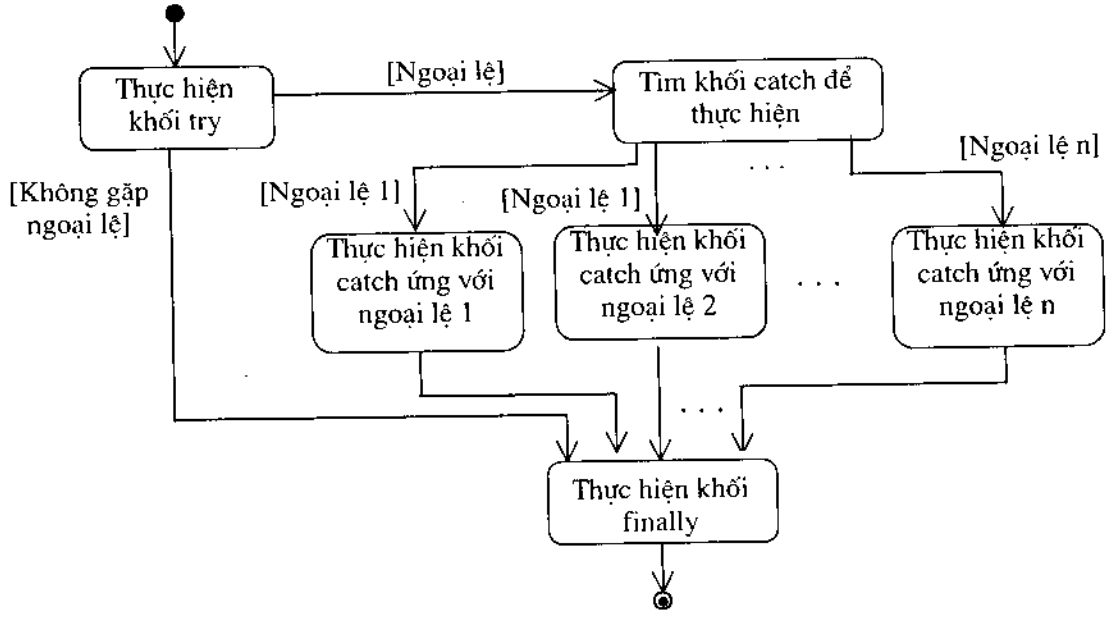
Cấu trúc *try-catch-finally* cho phép sử dụng để xử lý các ngoại lệ có dạng:

```
try {                                   // Khối try
    <Các câu lệnh>
} catch (<Kiểu ngoại lệ 1> <Tham biến 1>) {        // Khối catch
<Các câu lệnh xử lý khi xuất hiện kiểu ngoại lệ 1>
}
. . .
catch   (<Kiểu ngoại lệ n> <Tham biến n>) {        // Khối catch
<Các câu lệnh xử lý khi xuất hiện kiểu ngoại lệ n>
} finally {
```

```
        < Các câu lệnh phải thực hiện đến cùng>        // Khối finally
    }
```

Các ngoại lệ sẽ được cho qua trong quá trình thực hiện *khối try* và sẽ bị tóm lại để xử lý ở các *khối catch* tương ứng. *Khối finally* phải thực hiện đến cùng, bất luận có gặp phải ngoại lệ hay không.

Hoạt động của các khối trên được minh họa như sau:



*Hình 5.6. **Khối try-catch-finally.***

Khối try

Khối *try* xác định ngữ cảnh cần xử lý sự kết thúc thực hiện của một khối lệnh. Sự kết thúc thực hiện của khối lệnh xuất hiện khi:

- Gặp phải một ngoại lệ,
- Hoặc thực hiện thành công khối *try* (không gặp ngoại lệ).

Thực hiện xong khối *try* và xử lý xong các ngoại lệ khi chúng xuất hiện thì phải thực hiện khối *finally nếu* nó được chỉ ra trong cấu trúc đó.

Khối catch

Lối ra của khối *try* khi gặp phải ngoại lệ có thể chuyển điều khiển chương trình đến khối *catch*. Khối này chỉ được sử dụng để xử lý ngoại lệ. Khi một khối *catch* được

thực hiện thì các khối *catch* còn lại sẽ bị bỏ qua.

Trong trường hợp khối *finally* không xuất hiện thì chúng ta có cấu trúc *try-catch*.

Ví dụ 5.5. Cấu trúc *try-catch*

```
public class ChiaChoKhong {
    public void chia(){
        int n1 = 20;
        int n2 = 0;
        try {
            System.out.println(n1+" / "+n2+" = "+(n1/n2));
        } catch (ArithmeticException e) {           // (1)
            System.out.println("Gap phai loi: "+e);  // (2)
            }
        System.out.println("Ket thuc ham chia()");
        }                                   // (3)
    public static void main(String args[]){
        new ChiaChoKhong().chia();          // (4)
        System.out.println("Quay lai tu ham main!");  // (5)
        }
    }
```

Khi gọi hàm *chia()* ở (4) thì nó sẽ thực hiện phép chia *n1/n2 (n2 =0)*, do đó gặp ngoại lệ chia cho 0 trong phép chia của kiểu *int*. Ngoại lệ này được cho qua và nó được xử lý bởi khối *catch* ở (1), in ra lỗi gặp phải ở (2). Sau đó lại tiếp tục thực hiện phần còn lại của hàm *chia()*, in ra "Ket thuc ham chia()" mặc dù đã gặp phải ngoại lệ chia cho 0. Thực hiện xong hàm *chia()* của đối tượng được tạo ra ở (4), chương trình tiếp tục thực hiện phần còn lại ở (5). Kết quả thực hiện của chương trình *ChiaChoKhong*:

```
Gap phai loi: java.lang.ArithmeticException: /by zero
Ket thuc ham chia()
Quay lai tu ham main!
```

Khối finally

Khi khối *finally* có trong đặc tả khối *try-catch-finally* thì nó được đảm bảo phải thực hiện đến cùng bất luận trong khối *try* thực hiện như thế nào.

Ví dụ 5.6. Cấu trúc try-catch-finally

```
public class ChiaChoKhong1 {
    public void chia(){
```

```
        int n1 = 20;
        int n2 = 0;
        try {
          System.out.println(n1+" / "+n2+" = "+(n1/n2));   // (1)
        } catch (ArithmeticException e) {                  // (2)
          System.out.println("Gap phai loi: "+e);
        }finally {                                         // (3)
                System.out.println("Nhung viec can thuc hien");
          }
          System.out.println("Ket thuc ham chia()");       // (4)
        }
    public static void main(String args[]){
        new ChiaChoKhong1().chia();                        // (5)
        System.out.println("Quay lai tu ham main!");       // (6)
        }
    }
```

Kết quả thực hiện của chương trình ChiaChoKhong1:

```
Gap phai loi: java.lang.ArithmeticException: /by zero
Nhung viec can thuc hien
Ket thuc ham chia()
Quay lai tu ham main!
```

Ví dụ trên chỉ ra cách xử lý ngoại lệ xuất hiện ở (1) thông qua khối *catch* ở (2). Sau đó khối *finally* phải được thực hiện (3) rồi mới đến phần còn lại của hàm *chia()* ở (4).

Lưu ý: Khi không sử dụng khối *catch* để xử lý các ngoại lệ thì khối *finally* vẫn phải được thực hiện nhưng sau đó các phần còn lại của chương trình sẽ không được thực hiện nếu xuất hiện những ngoại lệ tệ hại như chia cho 0.

Ví dụ 5.7. Cấu trúc try-finally

```
public class ChiaChoKhong2 {
    public void chia(){
        int n1 = 20;
        int n2 = 0;
        try {
          System.out.println(n1+" / "+n2+" = "+(n1/n2));   // (1)
        }finally {                                         // (2)
                System.out.println("Nhung viec can thuc hien");
          }
```

```
        System.out.println("Ket thuc ham chia()");    // (3)
    }
public static void main(String args[]){
    new ChiaChoKhong2().chia();          // (4)
    System.out.println("Quay lai tu ham main!");   // (5)
    }
}
```

Kết quả thực hiện của chương trình ChiaChoKhong2:

```
Nhung viec can thuc hien
Gap phai loi: java.lang.ArithmeticException: /by zero
    at ChiaChoKhong2.chia(ChiaChoKhong2.java)
    ChiaChoKhong2.main(ChiaChoKhong2.java)
```

Sau khi thực hiện xong khối *finally*, chương trình *ChiaChoKhong2* thông báo lỗi gặp phải và những vị trí (tên lớp, hàm), tên tệp mà ở đó lỗi xuất hiện. Lưu ý khi không sử dụng khối *catch* để sử lý ngoại lệ nên chương trình kết thúc ngay khi gặp lỗi tệ hại (chia cho 0) và sau khi đã thực hiện khối *finally* nếu có.

Câu lệnh throw

Để tạm thời bỏ qua ngoại lệ chúng ta có thể sử dụng câu lệnh throw. Câu lệnh này có dạng như sau:

`throw` <Biểu thức tham chiếu đối tượng ngoại lệ>

<Biểu thức tham chiếu đối tượng ngoại lệ> là biểu thức xác định một đối tượng của lớp *Throwable* hoặc của một trong các lớp con của nó. Thông thường một đối tượng ngoại lệ sẽ được tạo ra trong câu lệnh *throw* để sau đó chúng sẽ được tóm lại và xử lý ở khối *catch*. Ví dụ câu lệnh

`throw` new ChiaCho0Exception(" / by zero");

Trong đó lớp *ChiaCho0Exception* là lớp con của *Exception* có thể định nghĩa đơn giản như sau:

```
class ChiaCho0Exception extends Exception {
    ChiaCho0Exception(String msg) {super(msg);}
}
```

Khi một ngoại lệ xuất hiện và được cho qua thì sự thực hiện bình thường của chương trình sẽ bị treo lại để đi tìm khối catch tương ứng và xử lý ngoại lệ đó. Sau đó khối finally được thực hiện nếu có. Nếu không tìm thấy bộ xử lý ngoại lệ tương ứng trong chương trình thì ngoại lệ đó được xử lý theo cơ chế xử lý ngoại lệ mặc định của hệ thống.

Ví dụ 5.8. Câu lệnh tạm thời cho qua ngoại lệ

```
class ChiaCho0Exception extends Exception {
        ChiaCho0Exception(String msg) {super(msg);}
    }
public class ChiaChoKhong3 {
    public void chia(){
        int n1 = 20;
        int n2 = 0;
        try {                                                    //(1)
          if(n2 == 0)throw new ChiaCho0Exception("/ by 0");      //(2)
          System.out.println(n1+" / "+n2+" = "+(n1/n2));         //(3)
    }catch (ChiaCho0Exception er){                               //(4)
          System.out.println("Gap loi: " +er);
                }finally {
          System.out.println("Nhung viec can thuc hien");        //(5)
            }
          System.out.println("Ket thuc ham chia()");             //(6)
        }
    public static void main(String args[]){
        new ChiaChoKhong3().chia();
        System.out.println("Quay lai tu ham main!");             //(7)
        }
    }
```

Kết quả thực hiện của chương trình ChiaChoKhong3:

```
Gap phai loi: ChiaChoException: /by zero
Nhung viec can thuc hien
Ket thuc ham chia()
Quay lai tu ham main!
```

Trong ví dụ 5.8, khối *try* (1) trong hàm *chia()* tạm thời cho qua ngoại lệ xuất hiện ở (2). Lưu ý phần còn lại (3) của khối *try* không được thực hiện. Khối *finally* (5) được thực hiện sau đó tiếp tục thực hiện bình thường các lệnh (6), (7).

Mệnh đề throws

Khi thiết kế các hàm thành phần, chúng ta có thể sử dụng mệnh đề *throws* để tạm thời cho qua ngoại lệ mà thực hiện một số công việc cần thiết khác. Những hàm này lại được sử dụng theo cấu trúc *try-catch-finally* sau đó để xử lý các ngoại lệ khi

chúng xuất hiện như trên. Chương trình dịch có thể cho qua những lỗi ngoại lệ đã phát hiện nếu trong định nghĩa các hàm có sử dụng kết hợp với mệnh đề *throws*. Định nghĩa hàm với mệnh đề *throws* có dạng:

<Thuộc tính của hàm> <tên hàm>(<Danh sách các tham biến>)

throws <Danh sách các kiểu ngoại lệ> { /* ... */}

Trong đó <Danh sách các kiểu ngoại lệ> là các lớp xử lý ngoại lệ được kế thừa từ lớp *Exception* và được phân tách bởi dấu ','. Ví dụ,

```
class A{
    protected void classMethA() throws FirstException, SecondException{
    /* . . . */
    }
```

Ví dụ 5.9. Mô tả cách sử dụng mệnh đề throws trong định nghĩa hàm

```
class ChiaCho0Exception extends Exception {      //(1)
        ChiaCho0Exception(String msg) {super(msg);}
    }
public class ChiaChoKhong4 {
    public void chia() throws ChiaCho0Exception {    //(2)
        int n1 = 20;
        int n2 = 0;
        if(n2 == 0)throw new ChiaCho0Exception("/ by 0"); //(3)
        System.out.println(n1+" / "+n2+" = "+(n1/n2));
        System.out.println("Ket thuc ham chia()");
    }
    public static void main(String args[]){
        try {
          new ChiaChoKhong3().chia();          //(4)
        }catch (ChiaCho0Exception er){
            System.out.println("Trong main() gap loi: " +er);
        }finally {
            System.out.println("Nhung viec can thuc hien"); //(5)
        }
        System.out.println("Quay lai tu ham main!");    //(7)
        }
    }
```

Chương trình thực hiện và cho kết quả:

```
Trong main() gap loi: ChiaCho0Exception: / by 0
Nhung viec can thuc hien
Quay lai tu ham main!
```

Trong ví dụ 5.8, hàm *main()* gọi hàm *chia()* được định nghĩa ở (2) cho qua ngoại lệ đã được kiểm tra thông qua mệnh đề *throws* . Hàm này không xử lý ngoại lệ ngay khi chúng xuất hiện mà để lại xử lý ở khối *try-catch-finally* trong hàm *main()* ở (4). Tất nhiên hàm *chia()* được định nghĩa như trên sẽ kết thúc thực hiện khi gặp ngoại lệ.

Lưu ý: Khi xây dựng hệ thống thường có rất nhiều loại lỗi (ngoại lệ) khác nhau có thể xảy ra. Để tiện lợi cho việc kiểm soát chúng, người lập trình nên đặt tên cho từng loại ngoại lệ đó cho phù hợp với ngữ cảnh mà chúng xuất hiện và sao cho dễ nhớ.

Ví dụ 5.10. Xây dựng *Stack* trong đó có các hàm được định nghĩa với cơ chế kiểm soát ngoại lệ để thực hiện:

- Đặt thêm các phần tử vào *Stack* nếu còn chỗ,
- Lấy ra một phần tử nếu *Stack* không rỗng.

Để tiện lợi cho việc kiểm soát các ngoại lệ khi thực hiện hai nhiệm vụ trên, chúng ta nên định nghĩa hai lớp ngoại lệ tương ứng.

```
class EmptyStack extends Exception { }
class FullStack extends Exception { }
```

Thực chất hai định nghĩa trên là cách đặt lại tên của lớp *Exception* cho phù hợp với ngữ cảnh xuất hiện của ngoại lệ.

Lớp *Stack* được định nghĩa như sau:

```
class Stack {
  int doCao;
  Object danhSach[];
  Stack(){ init(200);}
  Stack(int n) { init(n);}
  void init(int n){
    doCao = 0;
    danhSach = new Object[n];
  }
  void push(Object x) throws FullStack {
    if (danhSach.length == doCao) throw new FullStack();
    danhSach[doCao++] = x;
    }
```

```
    Object pop() throws EmptyStack {
      if (doCao == 0) throw new EmptyStack();
      return (danhSach[--doCao]);
      }
  }
```

Sau đó có thể sử dụng *Stack* để phân tích cú pháp cho các biểu thức bằng cách xây dựng lớp *PTCuPhap*.

```
class PTCuPhap {
  . . .
  void BieuThuc(){
    Stack s = new Stack();
    try {
      ...
      s.push(x); // x là hạng thức
      ...
    } catch (FullStack e) {
        System.err.println("Stack rong");
        abort();
    } catch (AnyOtherException er){
    // Xử lý các ngoại lệ khác
    }
    ...
}
```

5.4.2. CẤU TRÚC PHÂN CẤP CỦA CÁC LỚP XỬ LÝ NGOẠI LỆ

Ngoại lệ trong Java là các đối tượng. Tất cả các ngoại lệ đều được dẫn xuất từ lớp *Throwable*. Hình 5.7 chỉ rõ mối quan hệ giữa các lớp xử lý ngoại lệ trong cấu trúc phân cấp của chúng.

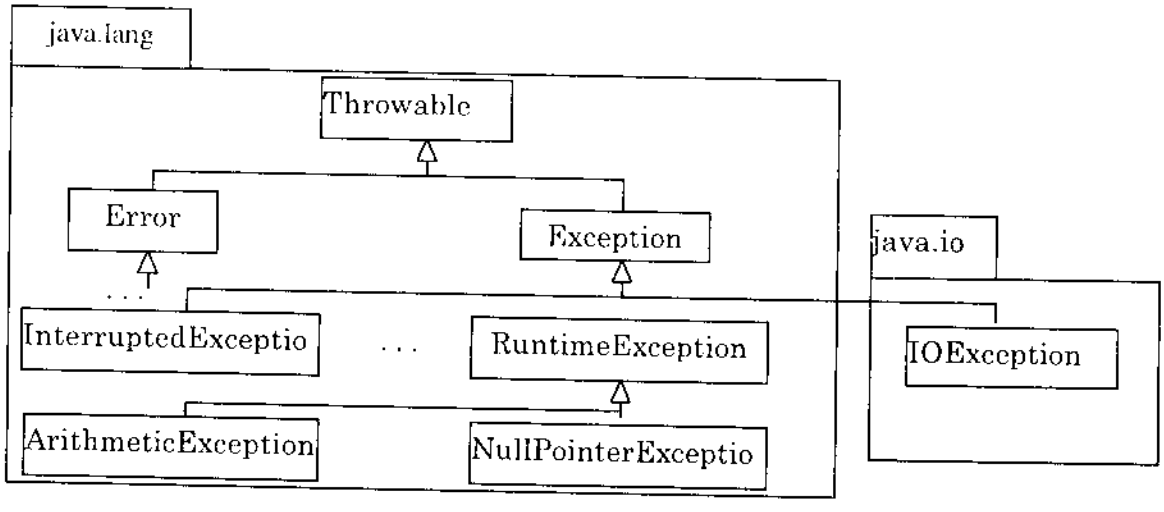
Trong gói *java.lang* lớp *Error* có các lớp con:

- LinkedError xử lý các tình huống ngoại lệ khi thực hiện liên kết,
- ThreadDeath xử lý tình huống liên quan đến sự sống/chết của các luồng (thread),
- VirtualMachineError xử lý các tình huống liên quan đến máy ảo.

Trong gói *java.lang* lớp *Exception* có các lớp con:

- ArrayIndexOutOfBoundsException xử lý các ngoại lệ khi chỉ số vượt khỏi phạm vi xác định của mảng,

- *NullPointerException* xử lý các tham chiếu đến đối tượng chưa được khởi tạo,
- *ClassCastException* xử lý các lỗi xuất hiện khi thực hiện ép kiểu giữa các lớp,
- *IlegalArgumentException* xử lý các lỗi liên quan đến các tham biến,
- *ArithmeticException* xử lý các lỗi số học, như chia cho 0,
- *NumberFormatException* xử lý các lỗi liên quan đến các format của các số.



*Hình 5.7. **Cấu trúc phân cấp của các lớp xử lý ngoại lệ.***

Trong gói java.io lớp *IOException* có các lớp con:

- *IOException* xử lý các lỗi vào/ra,
- *FileNotFoundException* xử lý các lỗi liên quan đến tệp tồn tại hay không tồn tại,
- *EOFException* xử lý các tình huống khi đã ở cuối tệp,
- *AWTException* xử lý các ngoại lệ liên quan đến GUI (giao diện đồ hoạ với người sử dụng).

Ngoài ra còn khá nhiều lớp ngoại lệ khác được định nghĩa trong Java, đặc biệt trong gói API.

BÀI TẬP

5.1. Hãy cho biết kết quả khi dịch và chạy chương trình sau:

```
public class KiemTraIF{
    static public void main(String args[]){
        if (true)
        if (false)
            System.out.println("AA");
        else
            System.out.println("BB");
        }
    }
```

5.2. Hãy cho biết kết quả khi dịch và chạy chương trình sau:

```
public class BreakDemo1{
     static public void main(String args[]){
    int i = 0;
    for( ; i < 10; i++)
    for(i = 0; ; i++) break;
    for(i =0; i < 10; ) i++;
      System.out.println(i);
    }
}
```

5.3. Hãy cho biết kết quả khi dịch và chạy chương trình sau:

```
public class BreakDemo2{
   static public void main(String args[]){
    int i = 0, k = 1;
    for(int n = 0; n <= 10; n++) {
        k++;
        if(n == 4) break;
        i++;
    }
      System.out.println(k + "," + i);
    }
}
```

5.4. Hãy cho biết kết quả khi dịch và chạy chương trình sau:

```
public class TryCatchDemo{
  static public void main(String args[]){
    int  k = 0;
    try {
        int i = 5/k;
    }catch(ArithmeticException e) {
        System.err.println(" Loi chia cho 0!");
    }catch(RuntimeException e){
        System.err.println(" Loi khi thuc hien!");
    }catch(Exception e){
        System.err.println(" Loi vao/ra!");
    }finally{
        System.out.println("Cong viec can thuc hien!");
    }
    System.out.println("Cuoi cung");
    }
}
```

5.5. Viết chương trình tìm các số nguyên tố nhỏ hơn một số *n* cho trước,

a/ Chỉ sử dụng chu trình *for*,

b/ Chỉ sử dụng chu trình *while*.

c/ Chỉ sử dụng chu trình *do-while*.

5.6. Chứng minh rằng chu trình *while* là tương dương ngữ nghĩa với chu trình *do-while*, nghĩa là có thể biến đổi từ chu trình *while* về *do-while* mà vẫn đảm bảo tương đương về kết quả tính toán và ngược lại.

5.7. Xây dựng cấu trúc *Queue (hàng đợi)* làm việc theo chế độ vào trước / ra trước *(FIFO - First Input / First Output)* có sử dụng mệnh đề *throws* để định nghĩa các các hàm thành phần thực hiện đưa phần tử vào hàng đợi hoặc lấy phần tử ra khỏi hàng đợi.

CHƯƠNG VI

CÁC LỚP CƠ SỞ VÀ CÁC CẤU TRÚC DỮ LIỆU

6.1. CẤU TRÚC MẢNG TRONG JAVA

Trong các ngôn ngữ lập trình truyền thống như C / C++, cấu trúc mảng (*array*) là một cấu trúc dữ liệu có kích thước (số phần tử) cố định các phần tử cùng kiểu. Bởi vì bộ nhớ giành cho cấu trúc mảng phải được xác định ngay từ lúc dịch nên số phần tử của nó phải được cố định trước (khởi tạo bởi *hằng - constant*).

Cấu trúc mảng như thế có các hạn chế:

- Rất khó kiểm soát khi chỉ số các phần tử vượt ra khỏi phạm vi xác định. Ví dụ, trong chương trình C/C++ không thể cấm việc truy nhập đến phần tử ở ngoài phạm vi xác định như phần tử A[100] của mảng khai báo:

```
int A[100];
```

Lưu ý: Chỉ số các phần tử của mảng trong C/C++ và Java được bắt đầu từ 0. Lỗi vượt ra ngoài phạm vi của mảng là rất khó kiểm soát.

- Việc quản lý bộ nhớ của các cấu trúc mảng cũng rất phức tạp, nếu không tổ chức tốt thì chương trình lớn có thể dễ bị dừng vì thiếu bộ nhớ.

Trong Java, *cấu trúc mảng* là một lớp có biến dữ liệu thành phần công khai *(public) length,* xác định kích thước của mảng và có các phần tử với kiểu bất kỳ (nguyên thủy hoặc lớp đối tượng).

Khai báo các biến mảng

Các biến mảng có thể khai báo theo một trong hai dạng sau:

<Kiểu các phần tử>[] <Tên mảng>;

<Kiểu các phần tử> <Tên mảng>[];

Trong đó <Kiểu các phần tử> có thể là kiểu nguyên thủy hoặc là kiểu lớp (tên lớp).

Lưu ý:

- ✓ Kích thước của mảng chưa xác định khi khai báo,
- ✓ Điều quan trọng cần phân biệt trong cấu trúc mảng của Java là việc khai báo biến mảng thì chưa hoàn toàn tạo ra cấu trúc đó mà mới khai báo cấu trúc để tham chiếu.

Ví dụ:

```
int mangInt[], bienInt;                    //(1)
BongDen[] dayBong, hangBong;               //(2)
```

Biến mảng *mangInt* được khai báo theo kiểu các phần tử là *int* (kiểu nguyên thủy), hai biến mảng *dayBong*, *hangBong* được khai báo với các phần tử có kiểu tham chiếu lớp *BongDen* (ở chương V).

Lưu ý: Khi ký hiệu [] đứng ngay sau kiểu khai báo thì tất cả các biến đứng sau đó đều là biến mảng, ví dụ ở (2), *dayBong* và *hangBong* đứng sau *BongDen*[] đều là biến mảng.

Tạo lập đối tượng mảng

Để tạo lập đối tượng mảng thì phải xác định số phần tử của mảng đó và phải sử dụng toán tử *new*.

<Tên mảng> = new <Kiểu các phần tử >[<Số phần tử>];

Trong đó <Kiểu các phần tử> là kiểu tương thích với kiểu mà mảng đã khai báo. Giá trị cực tiểu của <Số phần tử> là 0, nghĩa là trong Java có thể tạo ra mảng 0 phần tử.

Các biến mảng khai báo ở (1), (2) có thể được tạo lập như sau:

```
mangInt = new int[100];          // Mảng 100 phần tử kiểu int (3)
dayBong = new BongDen[10];       // Mảng 10 đối tượng của BongDen(4)
hangBong = new BongDen[15];      // Mảng 15 đối tượng của BongDen
```

Nhiều khi chúng ta có thể kết hợp cả khai báo với tạo lập mảng như sau:

<Kiểu các phần tử 1> <Tên mảng>[] = new <Kiểu các phần tử 2>[<Số phần tử>];

Khai báo và tạo lập mảng <Tên mảng> kiểu <Kiểu các phần tử 1> để chứa <Số phần tử> các phần tử có <Kiểu các phần tử 2>.

Lưu ý:

- ✓ <Kiểu các phần tử 1> và <Kiểu các phần tử 2> là hai kiểu tương thích với nhau. Nếu mảng đó có kiểu nguyên thủy thì hai kiểu đó phải trùng nhau. Ngược lại, đối với kiểu lớp thì <Kiểu các phần tử 2> phải là lớp con của <Kiểu các phần tử 1>.

Tất nhiên một lớp cũng được xem là lớp con của chính nó.

✓ Khi một mảng được tạo lập thì tất cả các phần tử của nó được khởi tạo giá trị mặc định (0 hoặc 0.0 đối với kiểu số, *false* đối với kiểu *boolean*, *null* cho kiểu lớp).

Ví dụ:

```
float mangFloat[] = new float[20];
                                    // Các phần tử được gán trị mặc định 0.0
Object[] dayDen = new BongDen[5];
                                    // Các phần tử được gán mặc định null
```

Khởi tạo các mảng

Java cung cấp cơ chế cho phép khai báo, tạo lập và gán ngay các giá trị ban đầu cho các phần tử của mảng:

<Kiểu các phần tử >[] <Tên mảng> = {<Các giá trị ban đầu>};

Ví dụ:

```
int[] mangInt = {1, 3, 5, 7, 9};
```

Tạo ra *mangInt* có 5 phần tử với phần tử đầu có giá trị là 1 (*mangInt[0] = 1*), phần tử thứ hai là 3 (*mangInt[1] = 3*) , v.v.

Object[] dayDT = {new BongDen(), new BongDen(), null};

Mảng *dayDT* được khai báo là mảng các đối tượng của lớp `Object`, được tạo lập để lưu giữ 3 đối tượng. Hai đối tượng đầu được khởi tạo bởi toán tử tạo lập đối tượng của lớp *BongDen*, còn phần tử thứ ba là `null`.

char[] charArr = {'a', 'b', 'a'}; // mảng 3 ký tự và hoàn toàn khác với "aba"

Sử dụng mảng

Toàn bộ cấu trúc mảng được tham chiếu thông qua tên của mảng, còn các phần tử của nó có thể truy nhập được bởi toán tử `[]` . Chỉ số các phần tử của mảng nhỏ nhất là 0 và nhỏ hơn kích thước của nó là *<Tên mảng>.length*. Phần tử thứ *i* của mảng là phần tử có chỉ số i - 1 (*<Tên mảng>[i-1]*).

Khi truy nhập đến các phần tử của mảng, Java sẽ tự động kiểm tra chỉ số của các phần tử xem có nằm trong phạm vi xác định hay không, do vậy người lập trình không phải bận tâm đến vấn đề vượt khỏi phạm vi của mảng như đối với các ngôn ngữ khác, như C/C++. Nếu chỉ số vượt ra ngoài phạm vi xác định (nhỏ hơn 0, hay lớn hơn hoặc bằng kích thước của mảng *<Tên mảng>.length)* thì hệ thống sẽ gọi tới hàm xử lý tình huống ngoại lệ *ArrayIndexOutOfBoundsException* để có thể tóm lại và xử

lý theo cấu trúc *try-catch-finally* (Chương V).

Ví dụ 6.1. Mô tả cách tạo ra năm mảng 15 các phần tử ngẫu nhiên và lưu lại năm giá trị cực tiểu của chúng để in ra màn hình. Lưu ý, ở đây chúng ta có thể sử dụng các hàm của lớp *Math* để tính giá trị cực tiểu, sinh số ngẫu nhiên.

```
class Mang {
  public static void main(String args[]){
    // Khai báo và tạo lập mảng
    double[] mang1 = new double[5];            // (1)
    double[] mang2 = new double[15];           // (2)
    for(int j = 0; j < mang1.length; ++j){     // (3)
      randomize(mang2);        // Tạo lập mảng ngẫu nhiên
      mang1[j] = cucTieu(mang2);  // Tìm giá trị cực tiểu
      }
    // In các giá trị cực tiểu ra màn hình          // (4)
    for(int j = 0; j < mang1.length; ++j){
      System.out.println(mang1[j]);
      }
    }
    public static void randomize(double[] mang){     // (5)
        for(int j = 0; j < mang.length; ++j)
            mang[j] = Math.random() * 100.0;
            }
    public static double cucTieu(double[] mang){     // (6)
        double giaTriMin = mang[0];
        for(int j = 0; j < mang1.length; ++j)
            giaTriMin = Math.min(giaTriMin, mang[j]);
        return giaTriMin;
        }
    }
```

Mảng nhiều chiều

Các phần tử của mảng có thể tham chiếu tới các mảng khác. Trong Java, mảng của mảng (mảng nhiều chiều) được khai báo như sau:

<Kiểu các phần tử>[][]...[] <Tên mảng>;

hoặc

<Kiểu các phần tử> <Tên mảng>[][]...[];

Toán tử [] có thể sử dụng ở cả hai vị trí, ví dụ

```
int[][] mang1;        // Mảng hai chiều
```

tương đương với

```
int mang1[][];
```

và cũng tương đương với

```
int[] mang1[];
```

Thông thường chúng ta có thể kết hợp cả khai báo với thiết lập mảng nhiều chiều tương tự như đối với mảng đơn.

```
int[][] mangA = new int[4][5];    // Ma trận có 4 hàng, 5 cột
```

Lệnh trên thiết lập mảng *mangA* có 4 phần tử, mỗi phần tử lại là mảng *mangA[i]* có 5 phần tử, với i = 1, 2, 3, 4. Như thế mảng hai chiều còn được gọi là mảng của mảng. Phần tử thứ j của *mangA[i]*, j = 1, 2, ..., 5 được truy nhập bởi *mangA[i][j]*. Do vậy, kích thước của mangA là *mangA.length* = 4 và mỗi phần tử của nó lại là mảng có kích thước là *mangA[i].length* = 5, i = 1, 2, ..., 4.

Hơn thế nữa, mảng nhiều chiều có thể được thiết lập và khởi tạo tường minh các giá trị ban đầu như đối với mảng đơn.

```
double[][] maTran = {
    {1, 2, 3, 4}, // hàng 1
    {0, 2, 0, 0}, // hàng 2
    {0, 0, 3, 0}, // hàng 3
    {0, 0, 0, 4}, // hàng 4
    };
```

Lưu ý: Các mảng trong mảng nhiều chiều không nhất thiết phải có số phần tử giống nhau. Ví dụ, mảng dayDen sau có 5 phần tử là các mảng có kích thước khác nhau.

```
BongDen[][] dayDen = {
    {new BongDen(), null, new BongDen()},   // Hàng có 3 phần tử (1)
    {null, new BongDen()},                  // Hàng có 2 phần tử (2)
    {},                                     // Hàng có 0 phần tử (3)
    {new BongDen()},                        // Hàng có 3 phần tử (4)
    {null}                                  // Hàng chưa được tạo lập (5)
    };
```

Không nên truy nhập đến những thành phần của mảng nếu nó chưa được xác định. Ví dụ, nếu chưa gán gì cho *dayDen[4]* thì sẽ nhận được *NullPointerException*, nếu bạn định truy nhập đến *dayDen[4].length*. Để tránh lỗi thì thường chúng ta nên kiểm tra trạng thái của các phần tử.

```
If (dayDen[4] != null) {
      // Truy nhập đến các phần tử của dayDen[4]
}
```

Lưu ý: Như chúng ta đã khẳng định từ đầu, tất cả các lớp đều là lớp con của *Object*, trong đó mảng cũng là lớp con của *Object*. Do vậy, giữa mảng và *Object* có thể thực hiện chuyển đổi kiểu:

+ Chuyển từ kiểu mảng về *Object* (Mở rộng kiểu),

+ Chuyển từ *Object* sang kiểu mảng (Thu hẹp kiểu).

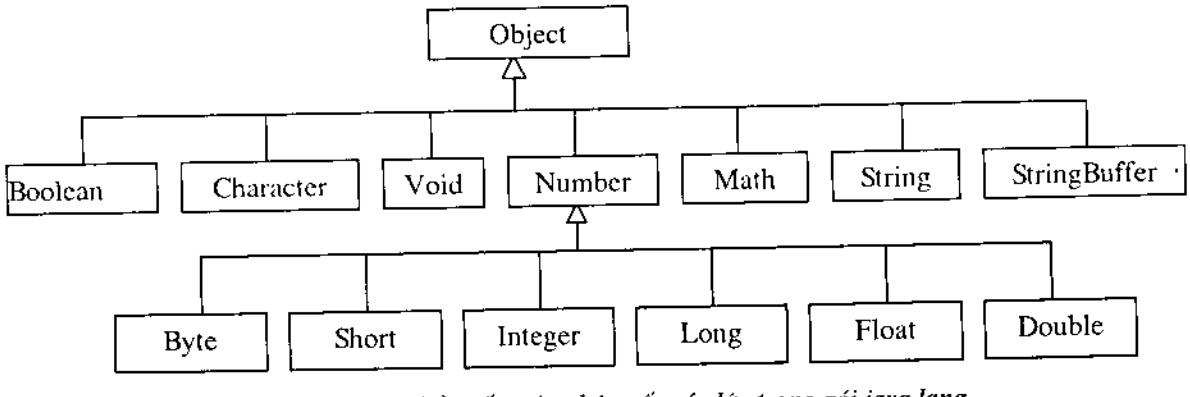
Ví dụ:

```
class EpKieu{
  static public void main(String[] args){
    int[] a = new int[5];
    Object o;
    o = a;          // int[] là con của Object
    int[] b = new int[5];
    b = (int[])o;   // Xem đối tượng của Object như là mảng int[]
    }
}
```

Ngoài ra cũng cần lưu ý thêm là cấu trúc mảng bảo toàn cấu trúc phân cấp giữa các lớp, nghĩa là nếu *B* là lớp con của lớp *A* thì *B[]* cũng là lớp con của lớp *A[]*.

6.2 CÁC LỚP CƠ BẢN CỦA JAVA

Phần này đề cập đến những lớp cơ bản, hay sử dụng của Java như: *Object*, các lớp *Wrapper* (lớp bao gói hay lớp nguyên thủy), *Math, String* và lớp *StringBuffer*. Những lớp này được xây dựng trong gói *java.lang* luôn mặc định được nạp vào chương trình. Tất cả đều là lớp con của *Object* và được xây dựng theo cấu trúc như trong hình 6.1.



*Hình 6.1. **Một phần cấu trúc phân cấp các lớp trong gói java.lang.***

6.2.1. LỚP OBJECT

Tất cả các lớp được xây dựng trong các chương trình Java là trực tiếp hoặc gián tiếp là mở rộng của *Object*. Đây là lớp cơ sở nhất, định nghĩa hầu như tất cả những hàm thành phần cơ sở để các lớp con cháu của nó sử dụng trực tiếp hoặc viết đè như đã đề cập ở chương 3.

Object cung cấp các hàm sau:

int hashCode()

Khi các đối tượng được lưu vào các bảng băm (*hash table*), hàm này có thể sử dụng để xác định duy nhất giá trị cho mỗi đối tượng. Điều này đảm bảo tính nhất quán của hệ thống khi thực hiện chương trình.

Class getClass()

Trả lại tên lớp của đối tượng hiện thời.

boolean equals(Object obj)

Cho lại kết quả *true* khi đối tượng hiện thời và *obj* là cùng một đối tượng. Hàm này thường được viết đè ở các lớp con cho phù hợp với ngữ cảnh so sánh bằng nhau trong các lớp mở rộng đó.

protected Object clone() throws CloneNotSupportedException

Đối tượng mới được tạo ra có cùng các trạng thái như đối tượng hiện thời khi sử dụng *clone()*, nghĩa là tạo ra bản copy mới của đối tượng hiện thời.

String toString()

Nếu các lớp con không viết đè hàm này thì nó sẽ trả lại dạng biểu diễn văn bản (*textual*) của đối tượng. Hàm *println()* ở lớp *PrintStream* sẽ chuyển các đối số của nó sang dạng văn bản khi sử dụng hàm *toString()*.

protected void finalize() throws Throwable

Hàm này được gọi ngay trước khi đối tượng bị dọn vào "thùng rác", nghĩa là trước khi đối tượng đó bị huỷ bỏ.

6.2.2. CÁC LỚP NGUYÊN THỦY (WRAPPER)

Các giá trị nguyên thủy không phải là đối tượng trong Java. Để có thể thao tác được trên các giá trị nguyên thủy (giá trị số và giá trị logic) thì gói *java.lang* cung cấp các lớp bao gói (Wrapper) cho từng kiểu dữ liệu nguyên thủy (gọi tắt là lớp nguyên

thủy). Các lớp nguyên thủy có những đặc tính chung sau:

1. Các toán tử tạo lập chung. Các lớp nguyên thủy (trừ lớp *Character* chỉ có một toán tử tạo lập) đều có hai toán tử tạo lập:

   - Toán tử tạo lập sử dụng giá trị nguyên thủy để tạo ra đối tượng tương ứng

```
Character charObj = new Character('a');
Boolean boolObj   = new Boolean(true);
Integer intObj    = new Integer(2002);
Float floatObj    = new Float(3.14F);
Double doubleObj  = new Double(3.14);
```

   - Toán tử thứ hai: chuyển các đối tượng lớp *String* biểu diễn cho các giá trị nguyên thủy về các lớp tương ứng. Các toán tử này sẽ cho qua *NumberFormatException* khi tham biến kiểu String không hợp lệ.

```
Boolean boolObj = new Boolean("true");
Integer intObj  = new Integer("2002");
Float floatObj  = new Float("3.14F");
Double doubleObj= new Double("3.14");
```

2. Có các hàm tiện ích chung: *valueOf(String s), toString(), typeValue(), equals().*

   - Mỗi lớp (trừ lớp *Character*) đều định nghĩa hàm *static valueOf(String s)* trả lại đối tượng tương ứng. Các hàm này đều cho qua NumberFormatException khi tham biến kiểu *String* không hợp lệ.

```
Boolean boolObj = Boolean.valueOf("true");
Integer intObj  = Integer.valueOf("2002");
Float floatObj  = Float.valueOf("3.14F");
Double doubleObj= Double.valueOf("3.14");
```

   - Các lớp viết đè hàm toString*()* trả lại là các đối tượng *String* biểu diễn cho các giá trị nguyên thủy ở dạng xâu.

```
String charStr   = charObj.toString();//  "a"
String boolStr   = boolObj.toString();//  "true"
String intStr    = intObj.toString(); // "2002"
String doubleStr = doubleObj.toString();// "3.14"
```

   - Các lớp định nghĩa hàm *typeValue()* trả lại các giá trị nguyên thủy tương ứng với các đối tượng nguyên thủy.

```
boolean b = boolObj.charValue();        // true
int i  = intObj.intValue();             // 2002
float f  = floatObj.floatValueOf();     // 3.14F
double d = doubleObj.doubleValueOf();   // 3.14
```

```
char c = charObj.charValue();      // 'a'
```

- Các lớp viết đè hàm *equals()* để thực hiện so sánh bằng nhau của các đối tượng nguyên thủy.

```
Character charObj = new Character('a');
boolean charTest  = charObj.equals('b');    // false
Integer intObj1   = Integer.valueOf("2010");
boolean intTest   = intObj.equals(intObj1); // false
```

Lớp Boolean

Lớp này định nghĩa hai đối tượng *Boolean*.TRUE, *Boolean*.FALSE biểu diễn cho hai giá trị nguyên thủy *true* và *false* tương ứng.

Lớp Character

Lớp *Character* định nghĩa hai giá trị cực tiểu, cực đại *Character.MIN_VALUE, Character.MAX_VALUE* và các giá trị kiểu ký tự Unicode. Ngoài ra lớp này còn định nghĩa một số hàm *static* để xử lý trên các ký tự:

```
static boolean isLowerCase(char ch)  // true nếu ch là ký tự thường
static boolean isUpperCase(char ch)  // true nếu ch là ký tự viết hoa
static boolean isDigit(char ch)      // true nếuch là chữ số
static boolean isLetter(char ch)     // true nếu ch là chữ cái
static boolean isLetterOrDigit(char ch)
                                     // true nếu ch là chữ hoặc là số
static char  toUpperCase(char ch)    // Chuyển ch về chữ viết hoa
static char  toLowerCase(char ch)    // Chuyển ch về chữ viết thường
static char  toTitleCase(char ch)    // Chuyển ch về dạng tiêu đề.
```

Các lớp nguyên thủy

Các lớp *Byte, Short, Integer, Long, Float, Double* là các lớp con của lớp *Number*. Trong các lớp này đều xác định hai giá trị:

<Lớp nguyên thủy>.MIN_VALUE

<Lớp nguyên thủy>.MAX_VALUE

là các giới hạn của các số trong kiểu đó. Ví dụ,

```
byte minByte = Byte.MIN_VALUE;      // -128
int maxInt   = Integer.MAX_VALUE;   // 2147483647
```

Trong mỗi lớp nguyên thủy có hàm *typeValue()* để chuyển các giá trị của các đối

tượng nguyên thủy về giá trị số:

```
byte byteValue()
short  shortValue()
int  intValue()
long  longValue()
float  floatValue()
double  doubleValue()
```

Trong mỗi lớp nguyên thủy còn có hàm *static parseType(String s)* để chuyển các giá trị được biểu diễn dưới dạng xâu về các giá trị số:

```
byte value1  = Byte.parseByte("16");
int  value2    = Integer.parseInt("2002");
double  value3 = Double.parseDouble("3.14");
```

Ví dụ 6.2. Viết chương trình để nhập vào một dãy số tùy ý và sắp xếp theo thứ tự tăng dần.

```
import java.io.*;
class SapXep{
    static int[] day;
    static void nhap(){
        String str;
        int n = day.length;
        DataInputStream stream = new DataInputStream(System.in);
        System.out.println("Nhap vao " + n + " so nguyen");
        for (int k = 0; k < n; k++){
            try{
              System.out.print(k + ": ");
              str = stream.readLine();
              day[k] = Integer.valueOf(str).intValue();
            }catch(IOException e){
                System.err.println("I/O Error!");
              }
            }
        }
    static void hienThi(){
        int n = day.length;
        for (int k = 0; k < n; k++)
            System.out.print(day[k] + " ");
```

```
        System.out.println();
        }
    static void sapXep(){
        int x, max, k;
        for(int i =day.length-1; i > 0; i--){
            max = day[i];k = i;
            for (int j = 0; j < i; j++)
                if (max < day[j]){
                    max = day[j];
                    k = j;
                }
            day[k] = day[i];
            day[i] = max;
            }
        }
    public static void main(String[] args){
        String str;
        int n;
        DataInputStream stream = new DataInputStream(System.in);
        System.out.print("\nCho biet bao nhieu so nhap vao: ");
        try{
          str = stream.readLine();
        }catch(IOException e){
            System.err.println("I/O Error!");
            str = "0";
            }
        n = Integer.valueOf(str).intValue();
        SapXep.day = new int[n];
        nhap();
        sapXep();
        System.out.println("Day so duoc sap xep: ");
        hienThi();
        }
    }
```

Lớp Void

Lớp này ký hiệu cho đối tượng của lớp *Class* biểu diễn cho giá trị *void*.

6.2.3. LỚP MATH

Lớp *final class Math* định nghĩa một tập các hàm tĩnh để thực hiện các chức năng chung của toán học như các phép làm tròn số, sinh số ngẫu nhiên, tìm số cực đại, cực tiểu, v.v.

Lớp *final class Math* còn cung cấp những hằng số như số *e* (cơ số của logarithm), số *pi* thông qua *Math.E* và *Math.PI*.

Các hàm làm tròn và xử lý các giá trị giới hạn

static int abs(int i)

static long abs(long l)

static float abs(float f)

static double abs(double d)

Hàm *abs()* được nạp chồng để trả lại giá trị tuyệt đối của đối số.

static double ceil(double d)

Hàm *ceil()* trả lại giá trị nhỏ nhất kiểu *double* mà không nhỏ hơn đối số và lại bằng số nguyên. Ví dụ *ceil(3.14)* cho giá trị 4.0 là số trần trên của đối số.

static double floor(double d)

Hàm *floor()* trả lại giá trị lớn nhất kiểu *double* mà không lớn hơn đối số và lại bằng số nguyên. Ví dụ *floor(3.14)* cho giá trị 3.0 là số sàn dưới của đối số.

static int round(float f d)

static long round(double d)

Hàm *round() được nạp chồng* để trả lại số nguyên gần nhất của đối số.

static int max(int a, int b)

static long max(long a, long b)

static float max(float a, float b)

static double max(double a, double b)

Hàm *max() được nạp chồng* để trả lại giá trị cực đại của hai đối số.

static int min(int a, int b)

static long min(long a, long b)

static float *min(float a, float b)*
static double *min(double a, double b)*

Hàm *min()* *được nạp chồng* để trả lại giá trị cực tiểu của hai đối số.

Các hàm lũy thừa

static double pow(double d1, double d2)

Hàm *pow()* trả lại giá trị là lũy thừa của *d1* và *d2* ($d1^{d2}$).

static double exp(double d)

Hàm *exp()* trả lại giá trị là luỹ thừa cơ số *e* và số mũ *d* (e^d).

static double log(double d)

Hàm *log()* trả lại giá trị là lô-ga-rit tự nhiên (cơ số e) của *d*.

static double sqrt(double d)

Hàm *sqrt()* trả lại giá trị là căn bậc hai của *d* , hoặc giá trị *NaN* nếu đối số âm.

Các hàm lượng giác

static double sin(double d)

Hàm *sin()* trả lại giá trị là *sine* của góc *d* được cho dưới dạng radian.

static double cos(double d)

Hàm *cos()* trả lại giá trị là *cose* của góc *d* được cho dưới dạng radian.

static double tan(double d)

Hàm *tan()* trả lại giá trị là tangent của góc *d* được cho dưới dạng radian.

Hàm sinh số ngẫu nhiên

static double random()

Hàm *random()* cho lại giá trị là số ngẫu nhiên trong khoảng từ 0.0 đến 1.0.

6.3. LỚP STRING

Việc xử lý các xâu ký tự trong Java được hỗ trợ bởi hai lớp *String* và *StringBuffer*. Lớp *String* dùng cho những xâu ký tự bất biến, nghĩa là những xâu *chỉ đọc* và chúng

được tạo lập hay khởi tạo giá trị đầu tiên sau đó không thay đổi. Lớp *StringBuffer* được sử dụng đối với những xâu ký tự *động, có thể thay đổi tùy ý.*

Tạo lập và khởi tạo các xâu

Lớp *final class String* có một số toán tử tạo lập và khởi tạo giá trị của các đối tượng của lớp *String* dựa trên các kiểu của các đối số.

Riêng đối với lớp *String* trong *Java* còn sử dụng cách tạo lập và khởi tạo xâu theo cách truyền thống (không sử dụng toán tử tạo lập) dựa vào dãy các ký tự như:

```
String str = "Dung dong den toi";
```

Các cách tạo lập khác đều sử dụng các toán tử tạo lập *String()* được nạp chồng theo các đối số.

```
String(String s)
String(<Kieu>[] mang)
String(StringBuffer buf)
```

Ví dụ:

```
byte[] bytes = {97, 98,98,97};
char[] characters = {'a', 'b', 'b','a'};
StringBuffer buff = new StringBuffer("abba");
String byteStr = new String(bytes); // Chuyển mảng bytes về xâu
String charStr = new String(charactes);
String buffStr = new String(buff);
String str     = new String("abba");
```

Đọc từng ký tự trong xâu

int length()

Hàm *length()* cho kết quả là số các ký tự trong xâu còn được gọi là kích thước của xâu.

char charAt(int index)

Hàm *charAt(int index)* cho lại ký tự thứ *index* (bắt đầu từ 0) của xâu hiện thời *(0≤ index ≤ length())*. Nếu *index* vượt ra khỏi phạm vi xác định thì hệ thống sẽ cho qua ngoại lệ *StringIndexOutOfBoundsException.*

Ví dụ 6.3. Sử dụng các hàm *length()* và *charAt()* để đếm số lần lặp của các ký tự trong xâu, kể cả dấu cách.

```
public class Dem{
```

```
public static void main(String[] args){
    int[] mangDem = new int[Character.MAX_VALUE];
    String str = "Khong duoc dung den toi!";
    for (int i = 0; i < str.length(); i++)   // (1)
    try{
        mangDem[str.charAt(i)]++;        // (2)
    }catch(StringIndexOutOfBoundsException e){
        System.out.println("Chi so vuot khoi gioi han");
    }
    for (int i = 0; i < mangDem.length; i++)
        if (mangDem[i] != 0)
            System.out.println((char)i + ":" + mangDem[i]);
    }
}
```

So sánh các xâu

Các ký tự được so sánh dựa trên các giá trị mã Unicode. Hai xâu được so sánh theo thứ tự từ điển bằng cách so sánh hai ký tự tương ứng của hai xâu đó. Xâu "abba" đứng trước (nhỏ hơn) xâu "aha" vì ký tự thứ hai của xâu "abba" là 'b' đứng trước 'h' của "aha".

Trong lớp *String* có những hàm sau được sử dụng để so sánh các xâu.

boolean equals(Object obj)

boolean equalsIgnoreCase(String str2)

Hàm *equals()* được viết đè ở *String* để kiểm tra xem đối tượng lớp *String* và đối tượng ở tham số có cùng tập các ký hiệu hay không. Tương tự hàm *equalsIgnoreCase()* thực hiện so sánh, nhưng không phân biệt các trường hợp chữ thường, chữ hoa.

boolean startWith(String str)

boolean endWith(String str)

Hai hàm này cho kết quả *true* nếu xâu bắt đầu (kết thúc) bằng đối số *str*.

int compareTo(String str2)

`int compareTo(Object obj)`

Hàm *compareTo()* đầu thực hiện só sánh hai xâu và trả lại kết quả dựa vào kết quả đối sánh:

+ 0 nếu xâu hiện thời bằng xâu đối số,

+ Giá trị nhỏ hơn 0, nếu xâu đó nhỏ hơn xâu đối số theo thứ tự từ điển,

+ Giá trị lớn hơn 0, nếu xâu đó lớn hơn xâu đối số theo thứ tự từ điển,

Hàm *compareTo()* thứ hai thực hiện tương tự như hàm đầu nếu đối số là chuyển về được đối tượng xâu, ngược lại sẽ cho qua để xử lý ngoại lệ *ClassCastException*.

Ví dụ:

```
String strA = new String("Ha Noi, Viet Nam");
String strB = new String("Ha noi, Viet nam");
boolean b1 = strA equals(strB);          // false
boolean b2 = strA equalsIgnoreCase(strB); // true
String str1 = new String("abba");
String str2 = new String("aha");
int compVal = str1.compareTo(str2); // Giá trị âm vì str1<str2
```

Chuyển đổi các trường hợp của ký tự

Để chuyển các ký tự viết thường về chữ viết hoa hoặc ngược lại, chúng ta sử dụng những hàm sau:

```
String toUpperCase()
String toLowerCase()
```

Lưu ý: Nếu không có ký tự nào phải chuyển đổi thì hàm cho lại xâu gốc, ngược lại khi có ký tự phải chuyển thì đối tượng mới của *String* được tạo ra và cho lại kết quả đó.

Ví dụ:

```
String strA = new String("Ha Noi, Viet Nam");
String strB = new String("ha noi, viet nam");
String str1 = strA.toUpperCase();
String str2 = strB.toLowerCase();
boolean b1 = str1 == strA;       // false
String str2 = str2 == strB;     // true
```

Ghép các xâu

Ghép hai xâu cho kết quả là một xâu, trong đó các ký tự của xâu đầu đứng trước các ký tự của xâu thứ hai. Phép ghép không có tính giao hoán. Như ở các phần trước chúng ta đã thấy, phép + được nạp chồng để thực hiện phép ghép của các xâu. Ngoài

ra chúng ta còn có thể sử dụng hàm thành phần của lớp *String*:

```
String concat(String str)
```

Ví dụ:

```
String tin = "Hoc lap trinh ";
tin = tin + "huong doi tuong ";
tin = tin.concat("voi Java!");
```

Kết quả cho xâu mới: "Hoc lap trinh huong doi tuong voi Java!".

Tìm các ký tự và các xâu con

Những hàm sau được nạp chồng để tìm chỉ số của ký tự hay chỉ số bắt đầu, chỉ số cuối của một xâu con. Nếu không tìm được các hàm này sẽ cho kết quả là -1.

int indexOf(int ch)

Tìm chỉ số của lần xuất hiện đầu tiên của *ch* trong xâu.

int indexOf(int ch, int fromIndex)

Tìm chỉ số của lần xuất hiện đầu tiên của *ch* trong xâu bắt đầu từ *fromIndex*.

int indexOf(String str)

Tìm chỉ số của lần xuất hiện đầu tiên của xâu con *str* trong xâu hiện thời.

int indexOf(String str, int fromIndex)

Tìm chỉ số của lần xuất hiện đầu tiên của xâu con *str* bắt đầu từ *fromIndex* trong xâu hiện thời.

int lastindexOf(int ch)

Tìm chỉ số của lần xuất hiện cuối cùng của ký tự *ch* trong xâu.

int lastindexOf(int ch, int fromIndex)

Tìm chỉ số của lần xuất hiện cuối cùng của *ch* bắt đầu từ *fromIndex* trong xâu.

int lastindexOf(String str)

Tìm chỉ số của lần xuất hiện cuối cùng của xâu con *str* trong xâu hiện thời.

int lastindexOf(String str, int fromIndex)

Tìm chỉ số của lần xuất hiện cuối cùng của xâu con *str* bắt đầu từ *fromIndex* trong xâu hiện thời.

String replace(char cu, char moi)

Thay tất cả các lần xuất hiện của ký tự *cu* bằng ký tự *moi* trong xâu.

Trích ra các xâu con

String trim()

Hàm này được sử dụng để tạo ra một xâu mới, trong đó các ký tự "trắng" (những ký tự có giá trị nhỏ hơn giá trị dấu cách ' ') ở trong xâu đều bị loại bỏ.

String substring(int startIndex)

Cho kết quả là một xâu con được triết ra từ vị trí *startIndex đến cuối của* xâu.

String substring(int startIndex, int endIndex)

Cho kết quả là một xâu con được triết ra từ vị trí *startIndex* đến vị trí *endIndex* của xâu.

Nếu chỉ số không tương thích (vượt ra ngoài phạm vi xác định) thì cho qua ngoại lệ *StringIndexOutOfBoundsException*.

Ví dụ:

```
String cau = "\t\n Java Applet \n\t";
cau = cau.trim();              // "Java Applet"
cau = cau.substring(5);        // "Applet"
cau = cau.substring(2,4);      // "ple"
```

Chuyển các đối tượng của Object về String

Lớp *String* viết đè hàm *toString()* của lớp *Object* để tự trả lại đối tượng của lớp *String*. Đồng thời nó còn có các hàm nạp chồng *valueOf()* để chuyển các giá trị số về dạng xâu ký tự.

static String valueOf(Object obj)

static String valueOf(char[] characters)

Hai hàm này được nạp chồng để chuyển các đối tượng *obj* và mảng các ký tự *characters* về xâu ký tự.

static String valueOf(boolean b)

static String valueOf(char c)

Hàm đầu được nạp chồng để chuyển hai giá trị boolean: *true, false* về xâu ký tự

"true", "false" tương ứng.

static String valueOf(int i)

static String valueOf(long l)

static String valueOf(float f)

static String valueOf(double d)

Các hàm này được nạp chồng để chuyển các giá trị số về dạng xâu ký tự.

6.4. LỚP STRINGBUFFER

Ngược lại với lớp *String*, lớp các xâu chỉ cho đọc, lớp *StringBuffer* lại cài đặt các xâu cho cả đọc lẫn ghi. Các thành phần trong xâu của lớp *StringBuffer* không những được phép thay đổi mà kích thước của nó cũng có thể thay đổi tùy ý.

Hai lớp *String* và *StringBuffer* là tương đối gần nhau, song chúng lại là hai lớp độc lập.

Tạo lập đối tượng StringBuffer

Lớp *final class StringBuffer* có ba toán tử tạo lập để tạo lập, khởi tạo và đặt độ dài (số ký tự) cho đối tượng của lớp *StringBuffer*.

StringBuffer(String str)

Tạo ra đối tượng mới có nội dung giống như xâu đối số *str*. Lưu ý kích thước của xâu *buffer* được đặt bằng kích thước của xâu *str* cộng với chỗ để cho 16 ký tự nhiều hơn.

StringBuffer(int length)

Tạo ra đối tượng mới chưa có nội dung và đặt kích thước của xâu *buffer* bằng đối số *length*, nếu đối số lớn hơn 0.

StringBuffer()

Tạo ra đối tượng mới chưa có nội dung và đặt kích thước của xâu *buffer* là 16.

```
StringBuffer strBuf1  = new StringBuffer("Java"); // Kích thước = 20
StringBuffer strBuf2 = new StringBuffer(10); // Kích thước = 10
StringBuffer strBuf3 = new StringBuffer();       // Kích thước = 16
```

Đọc và thay đổi các ký tự trong StringBuffer

int length()

Cho số ký tự trong xâu *buffer*.

char charAt(int index)

void setCharAt(int index, char ch)

Hàm đầu đọc một ký tự và hàm thứ hai thay đổi ký tự ở vị trí *index* trong xâu *buffer* thành *ch*. Hệ thống sẽ cho qua ngoại lệ *StringIndexOutOfBoundsException* nếu các chỉ số không tương thích.

Chuyển StringBuffer sang String

Lớp *StringBuffer* nạp chồng hàm *toString()* để chuyển xâu *buffer* về xâu của lớp cố định *String*.

Bổ sung, chèn và xóa các ký tự trong StringBuffer

Hàm *append()* được nạp chồng để bổ sung các ký tự vào cuối của xâu.

StringBuffer append(Object obj)

Đối tượng obj được chuyển về xâu (bằng hàm *String.valueOf()*) và sau đó bổ sung vào xâu *buffer*.

StringBuffer append(String str)

StringBuffer append(char[] str)

StringBuffer append(char[] str, int offset, int len)

StringBuffer append(char c)

Các mảng ký tự *str* và ký tự *c* được chuyển về xâu và sau đó bổ sung vào xâu *buffer*.

StringBuffer append(boolean b)

StringBuffer append(int i)

StringBuffer append(long l)

StringBuffer append(float f)

StringBuffer append(double d)

Các giá trị nguyên thủy được chuyển về xâu và sau đó bổ sung vào xâu *buffer*.

StringBuffer insert(int offset, Object obj)

StringBuffer insert(int offset, String str)

StringBuffer insert(int offset, char[] str)

StringBuffer insert(int offset, char c)

StringBuffer insert(int offset, boolean b)

StringBuffer insert(int offset, int i)

StringBuffer insert(int offset, long l)

StringBuffer insert(int offset, float f)

StringBuffer insert(int offset, double d)

Hàm *insert()* được nạp chồng để chèn đối số thứ hai (sau khi được chuyển về dạng xâu bằng cách sử dụng hàm *String.valueOf()*) vào từ vị trí *offset* trong xâu *buffer*.

StringBuffer delete(int index)

StringBuffer delete(int start, int end)

Hàm đầu xóa đi ký tự ở vị trí *index*, hàm sau xóa đi một xâu con kể từ vị trí *start* đến *end*.

StringBuffer reverse()

Nội dung của xâu *buffer* được đảo ngược lại (soi gương).

Ví dụ:

```
// Sử dụng phép bổ sung, chèn và xóa trong xâu,
StringBuffer buf = new StringBuffer("Lan dem");
buff = buff.delete(3,6);        // "Lan"
buff = buff.append(12);         // "Lan12"
buff = buff.insert(3, " co ");  // "Lan co 12"
buff = buff.setCharAt(0,'T');   // "Tan co 12"
buff = buff.reverse();          // "21 oc naT"
```

Kiểm soát cỡ của StringBuffer

int capacity()

Cho lại kích thước (khả năng chứa) của xâu *buffer*.

void ensureCapacity(int minCap)

Đảm bảo đủ chỗ cho ít nhất *minCap* ký tự của xâu *buffer*.

void setLength(int newLen)

Đặt độ dài của xâu *buffer* là *newLen*.

6.5. CẤU TRÚC TUYỂN TẬP ĐỐI TƯỢNG

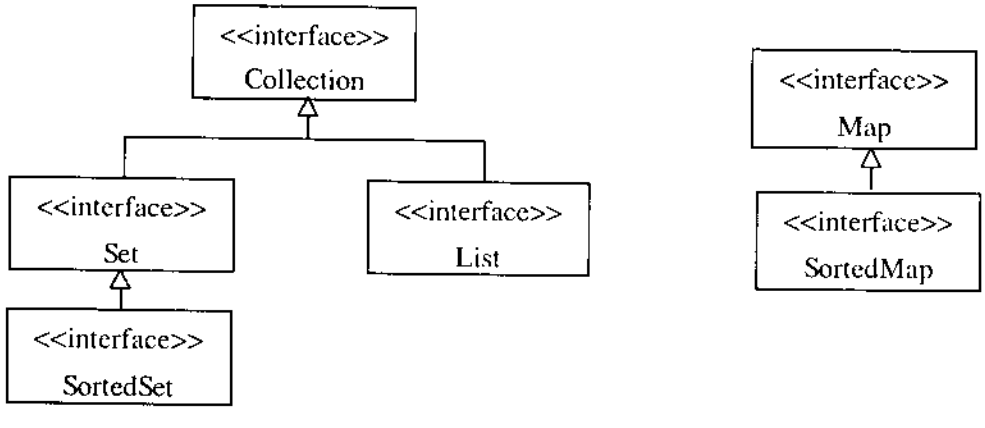
Cấu trúc *tuyển tập* (Collection) cho phép tập hợp các đối tượng lại để xử lý như một đơn vị. Các đối tượng bất kỳ có thể được lưu trữ, tìm kiếm và được thao tác như là các phần tử của *tuyển tập và các phần tử của nó là không thuần nhất*.

Thiết kế một hệ thống thường đòi hỏi phải xử lý một nhóm các đối tượng có liên quan với nhau. Cấu trúc tuyển tập giới thiệu một tập các lớp tiện dụng chuẩn phục vụ cho việc xử lý trên các tuyển tập. Những cấu trúc này được cung cấp ở gói *java.util* và được chia thành ba phần.

- Giao diện lõi (*Core interface*) cho phép các tuyển tập được xử lý độc lập với cài đặt (hình 6.2).
- Tập các cài đặt của các giao diện cung cấp cấu trúc dữ liệu cho các chương trình có thể sử dụng (hình 6.3).
- Tập các thuật toán được sử dụng để thực hiện các phép toán trên các tuyển tập.

Phần giao diện lõi

Giao diện (*interface*) *Collection* là cơ sở để phát triển, mở rộng thành các giao diện khác như *Set, List, SortedSet* và *Map* là giao diện cơ sở để mở rộng thành *SortedMap*. Hình 6.2 mô tả cấu trúc phân cấp theo quan hệ kế thừa của các giao diện lõi.



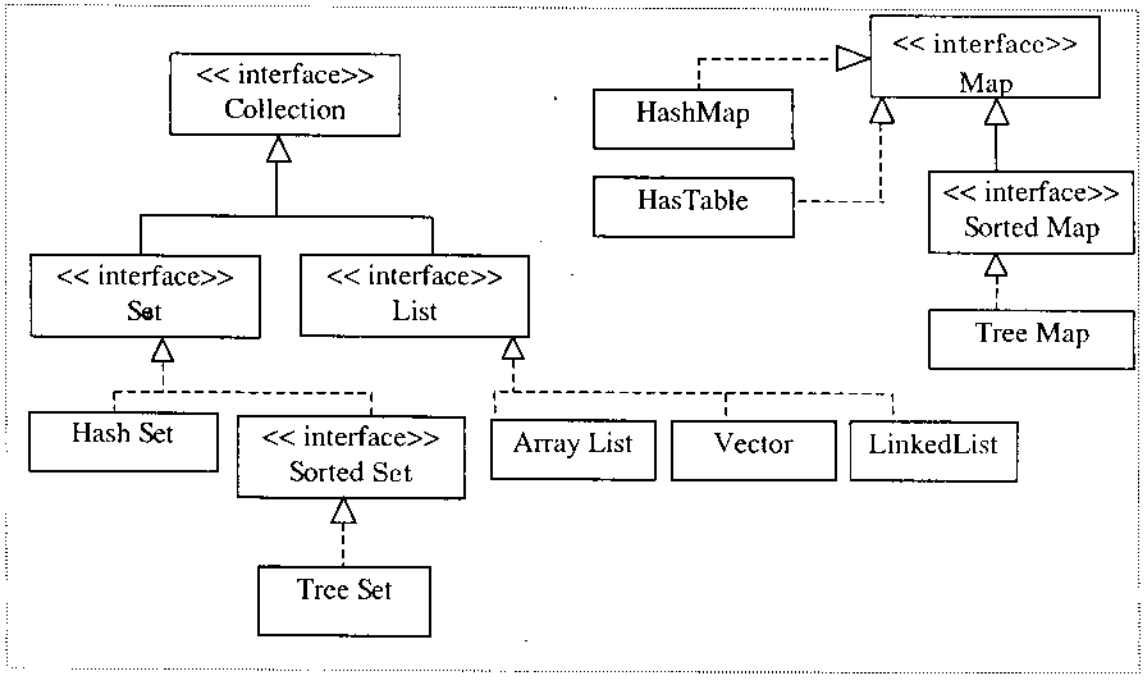
*Hình 6.2. **Các giao diện lõi của Collection.***

Các giao diện lõi của cấu trúc *Collection* được mô tả trong bảng sau:

interface	Mô tả
Collection	*interface* cơ sở định nghĩa tất cả các phép toán cơ bản cho các lớp cần duy trì thực hiện và cài đặt chúng
Set	Mở rộng *Collection* để cài đặt cấu trúc tập hợp, trong đó không có phần tử được lặp và chúng không được sắp xếp
SortedSet	Mở rộng *Set* để cài đặt cấu trúc tập hợp được sắp, trong đó không có phần tử được lặp và chúng được sắp xếp theo thứ tự
List	Mở rộng *Collection* để cài đặt cấu trúc danh sách, trong đó các phần tử được sắp xếp theo thứ tự, và có lặp
Map	*interface* cơ sở định nghĩa các phép toán để các lớp sử dụng và cài đặt các ánh xạ từ khóa sang các giá trị
SortedMap	Mở rộng của Map để cài đặt các ánh xạ khóa theo thứ tự

Phần cài đặt

Gói *java.util* cung cấp tập các lớp cài đặt các giao diện lõi để tạo ra những cấu trúc dữ liệu thường sử dụng như: *Vector, HashTable, HashSet, LinkedList, TreeSet*, v.v. Những lớp này và giao diện lõi được xây dựng theo cấu trúc phân cấp như trong hình 6.3.



Hình 6.3. ***Các giao diện lõi và các lớp cài đặt chúng.***

Trong hình 6.3, ký hiệu —▷ biểu diễn cho quan hệ kế thừa và --▷ biểu diễn cho cơ chế cài đặt các giao diện của Java trong UML.

Phần thuật toán

Lớp *java.util.Collection* (cần phân biệt với giao diện *Collection*) cung cấp một số hàm *static* thực hiện những thuật toán đa xạ cho những phép toán khác nhau trên tuyển tập, kể cả sắp xếp, tìm kiếm và dịch chuyển các phần tử. Một số hàm trong số đó là:

static int binarySearch(List list, Object key)

Sử dụng thuật toán tìm kiếm nhị phân để xác định chỉ số của phần tử *key* trong danh sách *list*.

static void fill(List list, Object obj)

Thay thế tất cả các phần tử trong danh sách *list* bằng *obj*.

static void shuffle(List list)

Hoán vị các phần tử của danh sách *list* một cách ngẫu nhiên.

static void sort(List list)

Sắp xếp các phần tử trong danh sách *list* theo thứ tự tăng dần.

6.5.1. COLLECTION

Giao diện *Collection* được xây dựng như là mẫu hợp đồng cho tất cả các cấu trúc tuyển tập có thể dựa vào đó mà thực thi và cài đặt. Gói *java.util* cung cấp các lớp tuyển tập (hình 6.3) cài đặt hầu hết các hàm của *Collection*.

Các phép toán cơ sở

int size();

Xác định kích thước của tuyển tập.

boolean isEmpty();

Trả lại *true* nếu tuyển tập rỗng, ngược lại *false*.

boolean contains(Object obj);

Trả lại *true* nếu tuyển tập chứa obj, ngược lại *false*.

boolean add(Object obj); // *Tùy chọn*

boolean remove(Object obj); // *Tùy chọn*

Trả lại *true* nếu tuyển tập thực hiện thành công việc bổ sung (loại bỏ) *obj*, ngược lại *false*.

Một số phép toán khác

Những phép toán sau thực hiện trên tuyển tập như một đơn vị cấu trúc dữ liệu.

boolean cotainsAll(Collection c); // *Tùy chọn*

Kiểm tra xem tuyển tập hiện thời có chứa cả tuyển tập *c* hay không.

boolean addAll(Collection c); // *Tùy chọn*

Thực hiện phép hợp hai tuyển tập

boolean removeAll(Collection c); // *Tùy chọn*

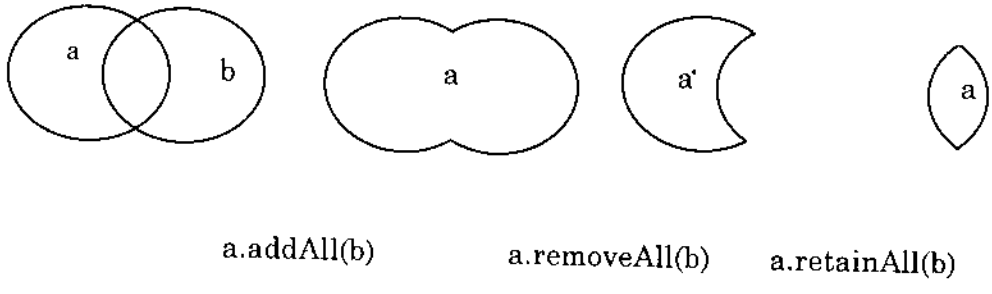
Thực hiện phép trừ hai tuyển tập

boolean retainAll(Collection c); // *Tùy chọn*

Thực hiện phép giao hai tuyển tập

void clear() // *Tùy chọn*

Hủy bỏ đối tượng trong tuyển tập các phép trên trả lại *true* nếu thực hiện thành công. Kết quả thực hiện các phép toán trên được minh họa như trong hình 6.4, trong đó a, b là hai tuyển tập bất kỳ.



*Hình 6.4. **Các phép toán trên các tuyển tập.***

6.5.2. SET (TẬP HỢP)

Tập hợp *Set* là cấu trúc dữ liệu, trong đó không có sự lặp lại và không có sự sắp xếp của các phần tử. Giao diện *Set* không định nghĩa thêm các hàm mới mà chỉ giới hạn lại các hàm của *Collection* để không cho phép các phần tử của nó được lặp lại. Thực chất các hàm này là các phép toán của tập hợp trong toán học.

Giả sử *a, b* là hai tập hợp (hai đối tượng của các lớp cài đặt *Set*). Kết quả thực hiện trên *a, b* có thể mô tả như trong bảng sau:

Các hàm trong Set	Các phép hợp tương ứng
a.containsAll(b)	$b \subseteq a$? (Tập con)
a.addAll(b)	$a = a \cup b$ (Hợp tập hợp)
a.removeAll(b)	a = a - b (Hiệu tập hợp)
a.retainAll(b)	$a = a \cap b$ (Giao tập hợp)
a.clear()	$a = \varnothing$ (Tập rỗng)

Sau đây chúng ta xét một số lớp thực thi cài đặt giao diện *Set*.

HashSet

Một dạng cài đặt nguyên thủy của *Set* là lớp *HashSet,* trong đó các phần tử của nó là không được sắp. Lớp này có các toán tử tạo lập:

HashSet()

Tạo ra một tập mới không có phần tử nào cả (tập rỗng).

HashSet(Collection c)

Tạo ra một tập mới chứa các phần tử của tuyển tập *c* nhưng không cho phép lặp.

HashSet(int initCapacity)

Tạo ra một tập mới rỗng có kích thước (khả năng chứa) là initCapacity.

HashSet(int initCapacity, float loadFactor)

Tạo ra một tập mới rỗng có kích thước (khả năng chứa) là *initCapacity* và yếu tố được nạp vào là *loadFactor*.

Ví dụ 6.4. Mô tả cách tạo ra các tập hợp và kiểm tra mối quan hệ giữa chúng. Khi thực hiện các đối số được đưa vào sau tên chương trình theo dòng lệnh. Chương trình bắt đầu với *tap1* là rỗng và lấy các ký tự của đối số đầu tiên để tạo ra *tap2*. So sánh hai tập đó, thông báo kết quả ra màn hình, sau đó cộng dồn *tap2* vào *tap1* và lại tiếp tục như thế đối với đối số tiếp theo cho đến hết.

```
import java.util.*;
public class TapKT {
    public static void main(String args[]){
```

```
        int nArgs = args.length;      // Số đối số của chương trình
        Set tap1 = new HashSet(); // Tạo ra tập thứ nhất là rỗng
        for (int i = 0; i < nArgs; i++){
            String arg = args[i]; // Lấy từng đối số của chương trình
            Set tap2 = new HashSet(); // Tạo ra tập thứ 2
            int size = arg.length(); // Số ký tự trong mỗi đối số
            for (int j = 0; j < size; j++) // Tập thứ 2 chứa các ký tự của arg
              tap2.add(new Character(arg.charAt(j)));
            // Tạo ra tập tapChung chính bằng tap1
            Set tapChung = new HashSet(tap1);
            tapChung.retainAll(tap2); // tapChung = tap1 ∩ tap2
            boolean b = tapChung.size() == 0;
            if (b)
              System.out.println(tap2+" va "+tap1+" la roi nhau");
            else {
             // tap2 có phải là tập con của tap1?
              boolean isSubset = tap1.containsAll(tap2);
             // tap1 có phải là tập con của tap2?
              boolean isSuperset = tap2.containsAll(tap1);
             // tap1 có bằng tap2?
              if (isSuperset && isSubset)
                System.out.println(tap2 + " bang tap " + tap1);
              else if (isSubset)
                System.out.println(tap2+" la tap con cua "+tap1);
              else if (isSuperset)
                System.out.println(tap2+" la tap cha cua "+tap1);
              else
                System.out.println(tap2 + " va " + tap1 + " co "
                     + tapChung + " la phan chung");
              }
              tap1.addAll(tap2);// hợp tap2 vào tap1
            }
        }
    }
```

Dịch và thực hiện chương trình với các đối số như sau:

```
java TapKT em voi em anh
```

sẽ cho kết quả:

[m, e] va [ ] la roi nhau

[v, i, o] va [m, e] la roi nhau

[m, e] la tap con cua [m, v, i, e, o]

[a, h, n] va [m, v, i, e, o] la roi nhau

6.5.3. LIST (DANH SÁCH)

Cấu trúc *List* là dạng tuyển tập các phần tử được sắp theo thứ tự (còn được gọi là dãy tuần tự) và trong đó cho phép lặp (hai phần tử giống nhau). Ngoài những hàm mà nó được kế thừa từ *Collection*, *List* còn bổ sung thêm những hàm như:

Object get(int index)

Cho lại phần tử được xác định bởi *index*.

Object set(int index, Object elem) // *Tùy chọn*

Thay thế phần tử được xác định bởi *index* bằng *elem*.

void add(int index, Object elem) // *Tùy chọn*

Chèn *elem* vào sau phần tử được xác định bởi *index*.

Object remove(int index) // *Tùy chọn*

Bỏ đi phần tử được xác định bởi *index*.

boolean addAll(int index, Collection c) // *Tùy chọn*

Chèn các phần tử của tuyển tập *c* vào vị trí được xác định bởi *index*.

int indexOf(Object elem)

Cho biết vị trí lần xuất hiện đầu tiên của *elem* trong danh sách.

int lastIndexOf(Object elem)

Cho biết vị trí lần xuất hiện cuối cùng của *elem* trong danh sách.

List subList(int fromIndex, int toIndex)

Lấy ra một danh sách con từ vị trí *fromIndex đến toIndex* .

ListIterator listIterator()

Cho lại các phần tử liên tiếp bắt đầu từ phần tử đầu tiên.

ListIterator listIterator(int index)

Cho lại các phần tử liên tiếp bắt đầu từ phần tử được xác định bởi *index*.

Trong đó *ListIterator* là giao diện mở rộng giao diện *Iterator* đã có trong *java.lang*.

Các lớp ArrayList, Vector và LinkedList

Ba lớp này có những toán tử tạo lập để tạo ra những danh sách mới rỗng hoặc có các phần tử lấy theo các tuyển tập khác.

Vector và *ArrayList* là hai lớp kiểu mảng động (kích thước thay đổi được). Hiệu suất sử dụng hai lớp này là tương đương nhau, tuy nhiên nếu xét theo nhiều khía cạnh khác thì *ArrayList* là cấu trúc hiệu quả nhất để cài đặt cấu trúc danh sách *List*.

Ví dụ 6.5. Hệ thống có một dãy N_DIGIT (5) chữ số bí mật. Hãy viết chương trình nhập vào N_DIGIT chữ số để đoán xem có bao nhiêu chữ số trùng và có bao nhiêu vị trí các chữ số trùng với dãy số cho trước.

```
import java.util.*;
public class NhapDoanSo {
    final static int N_DIGIT = 5;
    public static void main(String args[]){
        if(args.length != N_DIGIT) {
         System.err.println("Hay doan " + N_DIGIT + " chu so!");
         return;
            }
        List biMat = new ArrayList(); // Tạo danh sách biMat là rỗng
        biMat.add("5");          // Bổ sung các số vào dãy biMat
        biMat.add("3");
        biMat.add("2");
        biMat.add("7");
        biMat.add("2");
        List doan = new ArrayList(); // Tạo danh sách doan là rỗng
        for(int i = 0; i < N_DIGIT; i++)
          doan.add(args[i]); // Đưa các số từ đối số chương trình vào doan
        List lap = new ArrayList(biMat); // Lưu biMat sang lap
        int nChua = 0;
        // Đếm số các chữ số trùng nhau, nghĩa là thực hiện được phép bỏ đi remove()
        for(int i = 0; i < N_DIGIT; i++)
          if (lap.remove(doan.get(i))) ++nChua;
        int nViTri = 0;
```

```
        ListIterator kiemTra = biMat.listIterator();
        ListIterator thu = doan.listIterator();
        // Tìm những vị trí đoán trúng trong hai dãy có lặp
        while (kiemTra.hasNext()) // Khi còn phần tử tiếp theo
        // Kiểm tra xem lần lượt các vị trí của hai dãy có trùng nhau hay không
          if (kiemTra.next().equals(thu.next())) nViTri++;
        // Thông báo kết quả ra màn hình
        System.out.println(nChua + " chu so doan trung.");
        System.out.println(nViTri + " vi tri doan trung.");
        }
    }
```

Dịch và thực hiện chương trình:

```
java NhapDoanSo 3 2 2 2 7
```

sẽ cho kết quả:

```
4 chu so doan trung
1 vi tri doan trung
```

6.5.4. MAP (ÁNH XẠ)

Map định nghĩa các ánh xạ từ các khóa (*keys*) vào các giá trị. Lưu ý: các khóa phải là duy nhất (không cho phép lặp). Mỗi khóa được ánh xạ sang nhiều nhất một giá trị, được gọi là ánh xạ đơn.

Các ánh xạ không phải là các tuyển tập, giao diện *Map* cũng không phải là mở rộng của các *Collection*. Song, phép ánh xạ có thể xem như là một loại tuyển tập theo nghĩa: các khóa (*key*) tạo thành tập hợp và tuyển tập các giá trị (*value*) hoặc tập các cặp <*key, value*>.

Giao diện *Map* khai báo những hàm sau:

Object put(Object key, Object value); *// Tùy chọn*

Chèn vào một cặp <key, value>

Object get(Object key);

Đọc giá trị được xác định bởi *key* nếu có ánh xạ, ngược lại cho *null* nếu không có ánh xạ ứng với *key*.

Object remove(Object key); *// Tùy chọn*

Loại bỏ ánh xạ được xác định bởi *key*.

boolean containsKey(Object key);

Cho giá trị *true* nếu *key* được ánh xạ sang một giá trị nào đó, ngược lại là *false*.

boolean containsValue(Object value);

Cho giá trị *true* nếu *value* được ánh xạ bởi một *key* nào đó, ngược lại là *false*.

int size();

Cho số các cặp ánh xạ *<key, value>*.

boolean isEmpty();

Cho giá trị *true* nếu ánh xạ rỗng, ngược lại là *false*.

Một số phép toán thường dùng

void putAll(Map t); // *Tùy chọn*

Sao lại các ánh xạ từ *t*.

void clear(); // *Tùy chọn*

Xóa đi tất cả các ánh xạ.

Set keySet();

Xác định tập các khóa.

Collection values();

Xác định tuyển tập các giá trị.

Set entrySet();

Xác định tập các ánh xạ *<key, value>*.

Các lớp HashMap và HashTable

Hai lớp này cài đặt giao diện *Map* và được xây dựng trong *java.lang*. Chúng cho phép tạo ra các ánh xạ mới có thể rỗng hoặc có những kích thước tùy ý.

Ví dụ 6.6. Viết chương trình nhập vào các trọng lượng và in ra tần suất của các trọng lượng đó trong các nhóm cách nhau 5 đơn vị (kg).

```
import java.util.*;
public class NhomTrongluong {
    public static void main(String args[]){
```

```
        // Tạo ra một ánh xạ để lưu tần suất của mỗi nhóm
        Map demNhom = new HashMap();
        int nArgs = args.length;
        // Đọc các trọng lượng được nhập vào từ đối số và chia nhóm cách nhau 5 đơn vị.
        for(int i = 0; i < nArgs; i++){
         double trongL = Double.parseDouble(args[i]);
         Integer nhomTL=new Integer((int)Math.round(trongL/5)*5);
         Integer demCu = (Integer)demNhom.get(nhomTL);
        // Tăng số lần trọng lượng trong cùng nhóm, nếu là lần đầu (demCu = null) thì đặt là 1.
         Integer demMoi = (demCu == null)?
            new Integer(1): new Integer(demCu.intValue()+1);
         demNhom.put(nhomTL, demMoi);
          }
        // Lấy ra tập các giá trị từ ánh xạ demNhom
        List keys = new ArrayList(demNhom.keySet());
        // Sắp xếp lại theo các nhóm trọng lượng
        Collections.sort(keys);
        ListIterator keyIterator = keys.listIterator();
        // Tìm tần suất của các trọng lượng được nhập vào trong các nhóm
        while(keyIterator.hasNext()) {
              Integer nhom = (Integer) keyIterator.next();
              Integer dem = (Integer) demNhom.get(nhom);
              int demInt = dem.intValue();
              // Sử dụng hàm fill() của lớp Array để tạo ra xâu gồm demInt các dấu '*'
              char[] bar = new char[demInt];
              Arrays.fill(bar, '*');
              System.out.println(nhom+"\t" + new String(bar));
            }
        }
    }
```

Dịch và chạy chương trình *NhomTrongLuong* với các tham số:

```
java NhomTrongLuong 75 72 93 12 34
```

sẽ cho kết quả:

```
10      *
35      *
75      **
```

95 *

Như vậy, nhóm 10 kg có 1, 35 kg có 1, 75 kg có 2 và 95 kg có 1.

6.5.5. SORTEDSET (TẬP ĐƯỢC SẮP) VÀ SORTEDMAP (ÁNH XẠ ĐƯỢC SẮP)

Các cấu trúc tập hợp *(set)* và ánh xạ *(map)* có giao diện đặc biệt là *SortedSet* và *SortedMap* như trong hình 6.5 để cài đặt những cấu trúc có các phần tử được sắp theo thứ tự chỉ định.

Giao diện SortedSet

SortedSet là giao diện mở rộng của Set cung cấp các hàm để xử lý các tập được sắp.

SortedSet headSet(Object toElem);

Cho lại tập được sắp gồm những phần tử đứng trước *toElem*.

SortedSet tailSet(Object fromElem);

Cho lại tập được sắp gồm những phần tử cuối đứng sau *fromElem*.

SortedSet subSet(Object fromElem, Object toElem);

Cho lại tập được sắp gồm những phần tử kể từ *fromElem* đến *toElem*.

Object first();

Cho lại phần tử đầu tiên (cực tiểu) của tập được sắp.

Object last();

Cho lại phần tử cuối cùng (cực đại) của tập được sắp.

Comparator comparator()

Cho lại thứ tự so sánh của cấu trúc được sắp, cho *null* nếu các phần tử được sắp theo thứ tự tự nhiên (tăng dần)

Giao diện SortedMap

SortedMap là giao diện mở rộng của Map cung cấp các hàm để xử lý các ánh xạ được sắp theo thứ tự của khóa (key).

SortedMap headMap(Object toKey);

Cho lại ánh xạ được sắp gồm những phần tử đứng trước *toKey*.

SortedMap tailMap(Object fromKey);

Cho lại ánh xạ được sắp gồm những phần tử cuối đứng sau *fromKey*.

SortedMap subMap(Object fromKey, Object toKey);

Cho lại ánh xạ được sắp gồm những phần tử kể từ *fromKey* đến *toKey*.

Object firstKey();

Cho lại phần tử đầu tiên (cực tiểu) của ánh xạ được sắp.

Object lastKey();

Cho lại phần tử cuối cùng (cực đại) của ánh xạ được sắp.

TreeSet và TreeMap

Hai lớp này cài đặt hai giao diện *SortedSet* và *SortedMap* tương ứng. Chúng có bốn loại toán tử tạo lập như sau:

TreeSet()

TreeMap()

Tạo ra những tập hoặc ánh xạ mới và rỗng, được sắp theo thứ tự tăng dần của các phần tử hoặc của khóa.

TreeSet(Comparator c)

TreeMap(Comparator c)

Tạo ra những tập hoặc ánh xạ mới được sắp và xác định thứ tự so sánh theo *c*.

TreeSet(Collection c)

TreeMap(Map m)

Tạo ra những tập hoặc ánh xạ mới được sắp và có các phần tử lấy từ *c* hoặc từ *m* tương ứng.

TreeSet(SortedSet s)

TreeMap(SortedMap m)

Tạo ra những tập hoặc ánh xạ mới được sắp và có các phần tử lấy từ *s* hoặc từ *m* tương ứng.

BÀI TẬP

6.1. Xây dựng lớp các ma trận cấp $n \times m$ có các hàm:

+ Nhập, hiển thị các ma trận,
+ Thực hiện các phép cộng, trừ, nhân hai ma trận,
+ Tính định thức của ma trận nếu là ma trận vuông.

6.2. Xây dựng lớp các phân số (số hữu tỉ) có các hàm:

+ Các toán tử tạo lập phân số có nghĩa (mẫu số khác 0, không có dấu âm ở mẫu số, tối giản, v.v
+ Thực hiện các phép +, -, *, / trên các phân số.

6.3. Những lệnh nào trong số các lệnh sau là không hợp lệ?

```
a/  int[] i[] = {{1, 2},{1},{},{1, 2, 3}};
b/ int i[] = new int[2] {1, 2};
c/ int i[][] = new int[][] {{1, 2, 3}, {4, 5, 6}};
d/ int i[][] = {{1, 2},new int[2]};
e/ int i[] = {1, 2, 3, 4};
```

6.4. Viết chương trình tính số *pi*, biết rằng

pi = 4 * (1 - 1/3 + 1/5 - 1/7 + ...)

6.5. Xây dựng lớp *KhachHang* có các thuộc tính mô tả: họ và tên, địa chỉ liên hệ, các số điện thoại, địa chỉ E-mail và các hàm thành phần để:

+ Nhập, hiển thị các thông tin về khách hàng,
+ Tìm kiếm theo họ hoặc theo tên gọi của khách hàng,
+ Tìm theo địa chỉ và không phân biệt chữ viết thường hay chữ viết hoa.

6.6. Cho biết kết quả thực hiện của chương trình sau:

```
import java.util.*;
public class Sets {
  public static void main(String[] args){
    List list = new ArrayList();
    list.add("1");
    list.add("2");
    list.add(1,"3");
    List list2 = new ArrayList(list);
```

```
        list.addAll(list2);
        list2 = list.subList(2,5);
        System.out.printl(list2);
        list2.clear();
        System.out.printl(list);
    }
}
```

6.7. Xây dựng lớp *VanBan* để xử lý các văn bản và lớp này có các hàm:

+ Nhận một xâu ký tự và cho lại xâu không có ký tự nào bị lặp lại,
+ Nhận một xâu bất kỳ và cho lại kết quả là số các ký tự chỉ xuất hiện một lần,
+ Nhận ba xâu, thực hiện tìm và thay thế tất cả các lần xuất hiện của xâu thứ hai trong xâu thứ nhất bằng xâu thứ ba.

CHƯƠNG VII

LẬP TRÌNH ỨNG DỤNG APPLET VÀ AWT

7.1. LẬP TRÌNH *APPLET*

Trong chương II chúng ta đã giới thiệu khái quát về chương trình ứng dụng nhúng (*applet*). *Applet* là loại chương trình Java đặc biệt mà khi thực hiện phải được nhúng vào chương trình ứng dụng khác như các trình duyệt *Web Browser*, hoặc chương trình thông dịch *appletviewer* của JDK.

Lớp *java.applet.Applet* cung cấp mọi chức năng đã được cài đặt trước phục vụ cho việc thực thi của các chương trình ứng dụng nhúng *applet*. Trong cấu trúc phân cấp ở hình 7.2, lớp *Applet* là một thành phần của AWT (*Abstract Windowing Toolkit*) nhưng lớp này không nằm trong gói *java.awt* mà nằm ở gói *java.applet*. Lớp *Applet* cung cấp các chức năng sau:

- Các thành phần GUI (*Graphics User Interface*) và các bộ chứa (*Container*) được nhúng vào một *applet* để tạo ra các giao diện đồ họa cho ứng dụng đó.
- Các thao tác thực hiện của *applet* được viết đè trong hàm *paint()*.

Để chạy được các chương trình *applet* thì các thông tin về các tệp chứa các lớp đối tượng đã được dịch của *applet* đó phải được đưa vào tệp HTML. Trong các Version trước của HTML 4.0, phần tử APPLET (hoặc *applet* vì trong HTML không phân biệt chữ thường với chữ hoa) được sử dụng để đặc tả các thông tin về *applet*. Trong các Version sau của HTML (sau 4.0) thì có thể sử dụng OBJECT để thay thế cho APPLET. Bởi vì không phải tất cả các Web Browser đều hỗ trợ OBJECT, vì vậy ở đây chúng ta cần tìm hiểu cả hai phần tử APPLET và OBJECT của HTML.

Phần tử APPLET của HTML

Phần tử APPLET trong HTML cung cấp các thông tin cho Web Browser để xác định khung cơ sở và thực hiện ứng dụng trên trang Web. Trang HTML có thể chứa các

thông tin về *applet:*

```
<HTML>
  <HEAD>
    <TITLE> Tên gọi (tiêu đề) của chương trình ứng dụng  </TITLE>
  </HEAD>
  <BODY>
    <P> Mô tả về chức năng, nhiệm vụ của chương trình applet  </P>
    <APPLET code = "AppletName.class"
          width = w height = h > </APPLET>
  </BODY>
</HTML>
```

Dạng cú pháp tổng quát của APPLET qui định:

```
<APPLET
    CODE  = ClassFile
    WIDTH = w
    HEIGHT= h
    [CODEBASE = URLDirectory]
    [ARCHIVE = JarFileName]
    [OBJECT = SerializedAppletFile]
    [NAME = AnotherAppletName]
    [ALIGN = AligmentOnPage]
    [HSPACE= HorizonSpace]
    [VSPACE= VerticalSpace]
    [ALT= TextMessage] >
  [<PARAM NAME = param_1 [VALUE = paramValue_1] >]
  [<PARAM NAME = param_2 [VALUE = paramValue_2] >]
      . . .
  [<PARAM NAME = param_n [VALUE = paramValue_n] >]
  </APPLET>
```

Trong đó, các phần trong cặp [và] là tùy chọn. Các giá trị của các thuộc tính được cho dưới dạng các xâu (*string*). Phần lớn các trường hợp có phân biệt chữ thường với chữ hoa như: tên tệp, tên tham biến, các giá trị của các tham biến, v.v.

Lưu ý : các phần tử PARAM chỉ được xuất hiện trong cặp thẻ <APPLET> - </APPLET>, còn các thuộc tính khác được sử dụng bên trong thẻ <APPLET>.

Phần tử OBJECT của HTML

Phần tử OBJECT trong HTML rất giống với cú pháp của APPLET nêu trên. Trang HTML có thể chứa các thông tin về *applet:*

```
<HTML>
  <HEAD>
    <TITLE> Tên gọi (tiêu đề) của chương trình ứng dụng  </TITLE>
  </HEAD>
  <BODY>
    <P> Mô tả về chức năng, nhiệm vụ của chương trình applet  </P>
    <OBJECT classid = "AppletName.class"
          width = w height = h > </OBJECT>
  </BODY>
</HTML>
```

Điểm khác cơ bản là OBJECT sử dụng thuộc tính CLASSID để xác định phần mã cần thực hiện, còn APPLET sử dụng thuộc tính CODE.

Dạng cú pháp tổng quát của OBJECT qui định:

```
<OBJECT
    CLASSID  = ClassFile
    WIDTH = w
    HEIGHT= h
    [CODEBASE = URLDirectory]
    [ARCHIVE = JarFileName]
    [OBJECT = SerializedAppletFile]
    [NAME = AnotherAppletName]
    [ALIGN = AligmentOnPage]
    [HSPACE= HorizonSpace]
    [VSPACE= VeriticalSpace]
    [ALT= TextMessage]>
  [<PARAM NAME = param_1 [VALUE = paramValue_1]>]
  [<PARAM NAME = param_2 [VALUE = paramValue_2]>]
      . . .
  [<PARAM NAME = param_n [VALUE = paramValue_n]>]
  </OBJECT>
```

Các thuộc tính của *applet* được chia thành hai nhóm:

- Các thuộc tính đề cập đến cách bài trí của *applet* (trang Web),

- Các thuộc tính đề cập đến mã của *applet*.

Các thuộc tính đặc tả cách bài trí chương trình *applet*

WIDTH và HEIGHT

Hai thuộc tính này xác định độ rộng và chiều cao của khung cho *applet* và được tính theo *pixel*. Trong Web Browser, khung của *applet* không thay đổi được, chỉ có Window của trang HTML là được thay đổi chứ không phải là kích thước của *applet*. Tuy nhiên, chương trình thông dịch *appletviewer* lại cho phép thay đổi kích thước của *applet*. Trong chương trình có thể lấy lại được kích thước của *applet* bằng cách gọi hàm *getSize()* được kế thừa từ lớp *Component*.

ALIGN

Thuộc tính này là tùy chọn, được sử dụng để căn lề cho trang HTML. Nó có các giá trị sau:

+ *LEFT, RIGHT:* Các văn bản hiển thị được căn tương ứng theo lề trái, hoặc phải của một *applet*.
+ *BOTTOM:* Các văn bản hiển thị được căn tương ứng theo mép dưới (đáy) của *applet*. Đây là giá trị mặc định.
+ *TOP:* Các văn bản hiển thị được căn tương ứng theo mép trên (đỉnh) của *applet*.
+ *MIDDLE:* Các văn bản hiển thị ở giữa (trung tâm) của *applet*.

Ví dụ:

```
<APPLET code = "MyApplet.class" width = 200 height = 300
    align = TOP>
</APPLET>
```

HSPACE và VSPACE

HSPACE xác định khoảng trống tính theo *pixel* của chiều ngang kể từ mép trái và mép phải, VSPACE xác định khoảng trống của chiều dọc kể từ mép trên và mép dưới của *applet*.

Ví dụ:

```
<APPLET code = "MyApplet.class" width = 200 height = 300
    hspace = 10 vspace = 15>
</APPLET>
```

Các thuộc tính đặc tả mã chương trình

CODE (APPLET) và CLASSID (OBJECT)

Thuộc tính này xác định tên của tệp chứa lớp thực hiện *applet* đã được dịch. Tên tệp được thông dịch theo thư mục của trang HTML hiện thời, nếu không chỉ định cụ thể bằng thuộc tính CODEBASE. Khi thuộc tính CODEBASE được chỉ định, thì tên tệp *ClassName* được xác định theo thư mục của CODEBASE để thông dịch.

CODEBASE

Thuộc tính này là tùy chọn và có thể sử dụng để xác định vị trí (thư mục) từ xa URL chứa *ClassName* để thông dịch.

Ví dụ:

```
<APPLET codebase = "user/project"
      code = "MyApplet.class" width = 200 height = 300
     hspace = 10 vspace = 15>
</APPLET>
```

Tệp *MyApplet.class* ở thư mục *./users/project* được thông dịch và thực hiện. Muốn đọc lại thư mục gốc đó thì gọi hàm *getCodeBase()*.

ARCHIVE

Tất cả các lớp và các tài nguyên cần cho một *applet* thực hiện có thể đặt vào tệp JAR thông qua thuộc tính *ARCHIVE*.

Ví dụ:

```
<APPLET archive = "MyProject.jar"
      code = "MyApplet.class" width = 200 height = 300
     hspace = 10 vspace = 15>
</APPLET>
```

OBJECT (chỉ sử dụng với APPLET)

Các đối tượng trong Java có thể thực hiện tuần tự ghi và đọc thông tin thông qua thuộc tính OBJECT. Lưu ý, khi đã sử dụng thuộc tính CODE thì không được sử dụng OBJECT và ngược lại.

Ví dụ:

```
<APPLET object= "MyApplet.class" width = 200 height = 300
     hspace = 10 vspace = 15>
</APPLET>
```

NAME

Thuộc tính này được sử dụng để xác định tên của *applet* bên trong một trang HTML.

Ví dụ:

```
<APPLET code = "MyApplet.class" width = 200 height = 300
      name = "My" >
</APPLET>
```

ALT

Thuộc tính này được sử dụng để hiển thị văn bản TextMessage khi vì một lý do nào đó mà *applet* không hoạt động được và ngược lại nó sẽ bị bỏ qua.

Ví dụ:

```
<APPLET code = "MyApplet.class" width = 200 height = 300
      alt = "JVM co le khong hoat dong!" >
</APPLET>
```

Các tham biến

Các thông tin có thể cung cấp cho *applet* thông qua các tham biến PARAM.

```
[<PARAM NAME = param_i [VALUE = paramValue_i]>]
```

Trong đó NAME là phải chỉ định còn VALUE thì là tùy chọn. Lưu ý: các NAME và VALUE luôn là các xâu ký tự. Khi thực hiện thì *applet* phải kiểm tra tính hợp lệ của các tên gọi và giá trị tương ứng.

Ví dụ:

```
<APPLET code = "MyApplet.class" width = 200 height = 300
      <PARAM NAME = "background" VALUE = "DDFFBB">
      <PARAM NAME = "foreground" VALUE = "880000">
</APPLET>
```

Ví dụ 7.1. Mô tả cách sử dụng các tham số của *applet*.

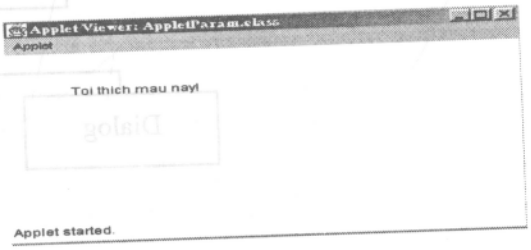
```
// Tệp AppletParam.java
import java.applet.*;
import java.awt.*;
public class AppletParam extends Applet{
    public void init(){ // Đọc hai tham biến xác định màu
        Color foreground = getColorParam("foreground");
        Color background = getColorParam("background");
        if (foreground != null) setForeground(foreground);
        if (background != null) setBackground(backgound);
```

```
    }
  private Color getColorParam(String paramName){
      // Đọc giá trị màu dưới dạng xâu ký tự
      String paramVal = getParameter(paramName);
      try{  // Chuyển tham biến màu thành giá trị nguyên hệ hexa
          return new Color(Integer.parseInt(paramVal, 16));
          } catch (NumberFormatException e) { return null;}
      }
  public void paint(Graphics gf){  // Nạp chồng hàm  paint()
      gf.translate(getInsets().left,getInsets().top);
      gf.drawString("Toi thich mau nay!", 50,50);
      }
  // Cung cấp các thông tin để hiển thị trong hộp thoại
  public String getAppletInfor(){
      return "AppletParam Version 1.0 writtten by My Lan";
      }
  // Cung cấp các thông tin về tham biến của applet
  public String[][] getParameterInfo() {return paramInfo;}
  private String[][] paramInfo = {
      // Tên, kiểu và mô tả tham biến
      {"foreground","Hex RGB color value", "Foreground color"},
      {"background","Hex RGB color value","Background color"},
      };
  }
```

Tệp HTML có thể viết đơn giản:

```
// Tệp AppletParam.html
<applet code = "AppletParam.class" width = 300 height = 300>
</applet>
```

Kết quả thực hiện *appletviewer AppletApp.html* của JDK là:



Hình 7.1. Applet với các tham biến.

7.2. CÁC THÀNH PHẦN CỦA AWT

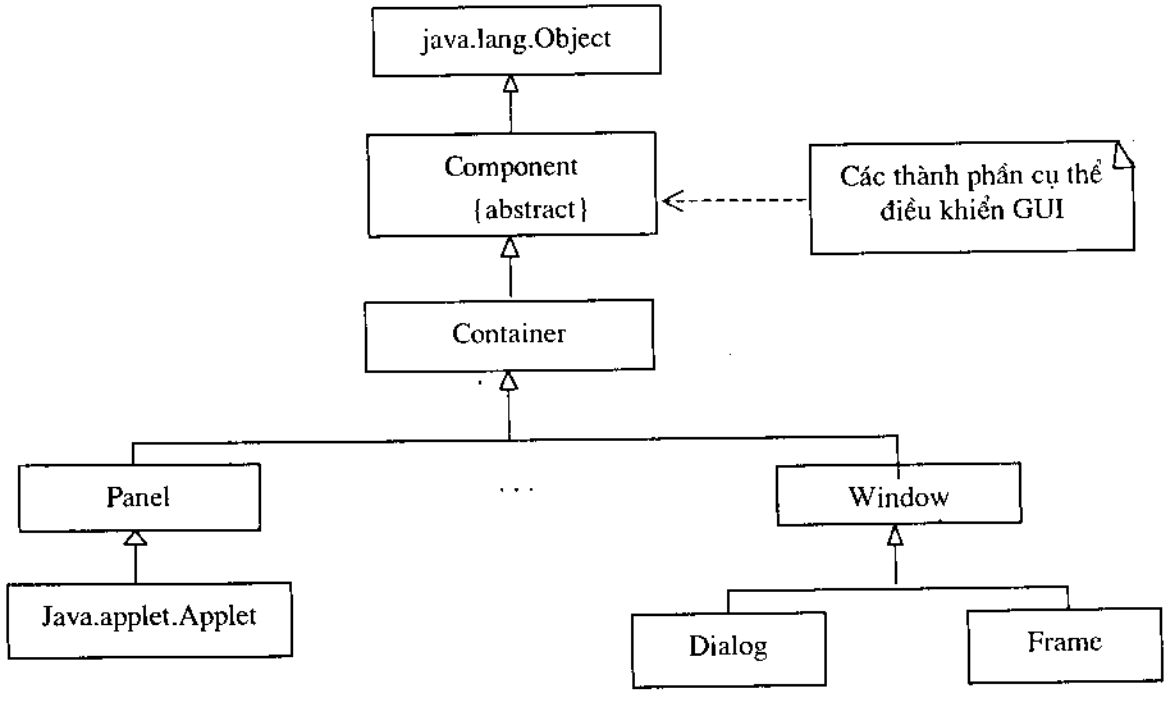
Các lớp cơ sở của Java (*Java Foundation Classes - JFC*) cung cấp bộ công cụ lập trình trên *Window* trừu tượng AWT (Abstract Windowing Toolkit) để xây dựng những ứng dụng dựa trên giao diện đồ họa với người sử dụng (*GUI- Graphics User Interface*).

Gói *java.awt* cung cấp các chức năng nguyên thủy của AWT để:

- Quản lý cách bài trí các thành phần bên trong các đối tượng chứa,
- Hỗ trợ để xử lý các sự kiện cần thiết cho sự tương tác của người sử dụng với hệ thống thông qua GUI,
- Xử lý đồ họa trong giao diện GUI sử dụng màu, *font*, hình ảnh âm thanh, v.v.

7.2.1. LỚP COMPONENT VÀ CONTAINER

Hình 7.2 mô tả một phần cấu trúc phân cấp của các lớp theo quan hệ kế thừa, trong đó các lớp con của lớp chứa *Container* sẽ cung cấp những chức năng chính để phát triển các ứng dụng dựa trên GUI.



*Hình 7.2. **Các thành phần của AWT và các lớp con của Container.***

Component

Lớp trừu tượng cơ sở của tất cả các thành phần cung cấp các chức năng để xử lý các sự kiện, thay đổi kích thước của thành phần, điều khiển *font,* màu và vẽ các thành phần.

Container

Một lớp chứa là một thành phần. Nó có thể bao gồm một số các thành phần khác và cũng có thể là các lớp chứa khác. Các lớp chứa (các lớp con của *Container*) hỗ trợ để xây dựng những giao diện đồ họa phức tạp với người sử dụng.

Panel

Bảng *panel* là một thành phần chứa lý tưởng để nhóm các thành phần khác và các *panel* khác để tạo ra cấu trúc phân cấp các thành phần.

Applet

Một *applet* là một loại *panel* đặc biệt có thể sử dụng để phát triển những chương trình thực hiện trong *Web Browser.*

Window

Lớp *Window* đại diện cho *Window* ở mức đỉnh, trong đó không chứa tiêu đề, thực đơn hoặc các đường biên.

Frame

Một khung *(frame)* là tùy chọn được sử dụng để thay đổi kích thước, dịch chuyển *Window* và trong đó có thanh tiêu đề, thực đơn, các đường biên.

Dialog

Lớp *Dialog* định nghĩa một *window* độc lập, tùy chọn, có thể thay đổi lại được kích thước và trong đó chỉ có thanh tiêu đề và các đường biên.

Lớp Component

Lớp *Component* cung cấp các hàm tiện ích chung cho các lớp con của nó.

Dimension getSize()

void setSize(int width, int height)

void setSize(Dimension d)

Hàm *getSize()* cho lại đối tượng thuộc lớp *Dimension* gồm *width* (chiều rộng),

height (chiều cao) xác định kích thước của một thành phần tính theo *pixel*. Hàm *setSize()* đặt lại kích thước của thành phần.

Point getLocation()

void setLocation(int x, int y)

void setLocation(Point p)

Hàm *getLocation()* cho lại tọa độ *(kiểu Point)* trên cùng bên trái (tọa độ gốc) của thành phần đang xét. Hàm *setLocation()* đặt lại các tọa độ được chỉ định cho một thành phần.

Rectangle getBounds()

void setBounds(int x, int y)

void setBounds(Rectangle r)

Hàm *getBounds()* cho lại đường biên là hình chữ nhật *Rectangle* bao gồm tọa độ gốc và chiều dài, chiều rộng của hình chữ nhật. Hàm *setBounds()* đặt lại đường biên cho một thành phần.

void setForeground(Color c)

void setBackground(Color c)

Hàm *setForeground()* được sử dụng để đặt màu vẽ cho thành phần đồ họa, còn *setBackground()* đặt màu nền cho thành phần đồ họa. Các tham số của hai hàm này là đối tượng của lớp *Color* sẽ được giới thiệu ở phần sau.

Font getFont()

void setFont(Font f)

Hàm *getFont()* được sử dụng để biết được *font* của các chữ đang xử lý trong thành phần đồ họa. Hàm *setFont()* đặt lại *font* chữ cho một thành phần.

void setEnabled(boolean b)

Nếu đối số *b* của hàm *getEnabled()* là *true* thì thành phần đang xét hoạt động bình thường, nghĩa là có khả năng kích hoạt (*enable*), có thể trả lời các yêu cầu của người sử dụng và sinh ra các sự kiện như mong muốn. Ngược lại, nếu là *false* thì thành phần tương ứng sẽ không kích hoạt được, nghĩa là không thể trả lời được các yêu cầu của người sử dụng. *Lưu ý:* Tất cả các thành phần giao diện khi khởi tạo đều được kích hoạt.

void setVisible(boolean b)

Một thành phần đồ họa có thể được hiển thị lên màn hình (nhìn thấy được) hoặc bị che giấu tùy thuộc vào đối số của hàm *setVisible()* là *true* hay *false*.

Lớp Container

Lớp *Container* là lớp con của lớp trừu tượng *Component*. Các lớp chứa (lớp con của *Container*) cung cấp tất cả các chức năng để xây dựng các giao diện đồ họa ứng dụng, trong đó có hàm *add()* được nạp chồng để bổ sung một thành phần vào lớp chứa cho trước.

Lớp *Frame*

Lớp *Frame* là lớp con của lớp *Window* được sử dụng để tạo ra những *window* cho các giao diện ứng dụng GUI. Đối tượng của *Frame* thường là điểm khởi đầu của một ứng dụng với GUI và có thể chứa một số bảng *Panel*, trong đó có thể lại chứa các thành phần giao diện khác.

Kịch bản chung để xây dựng giao diện ứng dụng là:

1. Tạo ra một *frame* có tên, ví dụ "*My Frame*" :

```
Frame guiFrame = new Frame("My Frame");
```

2. Xây dựng một cấu trúc phân cấp các thành phần bằng cách sử dụng hàm *add()* để bổ sung thêm *panel* hoặc những thành phần điều khiển GUI vào cấu trúc đó:

```
guiFrame.add(new Button("OK"); // Đưa vào một nút (Button) có tên "OK"
```

3. Đặt lại kích thước cho *frame* sử dụng hàm *setSize()*:

```
guiFrame.setSize(200, 300); // Đặt lại khung frame là 200 × 300
```

4. Gói khung *frame* đó lại bằng hàm *pack()*:

```
guiFrame.pack();
```

5. Làm cho *frame* đó nhìn thấy được:

```
guiFrame.setVisible(true);
```

7.2.2. CÁC THÀNH PHẦN ĐIỀU KHIỂN CỦA GUI

Các thành phần điều khiển của GUI là các lớp giao diện đồ họa, các lớp con của lớp *Component* cho phép tương tác với người sử dụng. Bảng 7.1 mô tả tóm tắt các lớp giao diện đồ họa nguyên thủy đó.

Để sử dụng được các thành phần giao diện đồ họa, cần phải thực hiện theo ba bước như sau:

1. Tạo ra một thành phần giao diện bằng cách sử dụng toán tử tạo lập tương ứng với thành phần đó, ví dụ:

```
Button guiComponent = new Button("OK");
```

2. Bổ sung thành phần vừa được tạo ra vào thành phần chứa (*frame*), ví dụ:

```
guiFrame.add(guiComponent);
```

3. Nhận và xử lý các sự kiện khi chúng xuất hiện, đặc biệt là các yêu cầu của người sử dụng.

Bảng 7.1. Các thành phần giao diện đồ họa trong AWT

Button	Một nút (*button*) với tên gọi xác định, được thiết kế cho một hành động và khi muốn thực hiện nó thì nhấn nút tương ứng.
Canvas	Một thành phần tổng quát để vẽ và thiết kế các thành phần giao diện đồ họa mới.
Checkbox	Một hộp kiểm tra (*Checkbox*) có tên gọi và được sử dụng để đánh dấu chỉ ra là một mục nào đó được kiểm tra hay không, tương ứng với hai trạng thái *true* hoặc *false*.
Choice	Cài đặt thực đơn (*menu*) *pop-up* để chọn lựa khi thực hiện, trong đó chỉ một mục hiện thời là được kích hoạt.
Label	Nhãn (*label*) là một thành phần được hiển thị trên một dòng và chỉ cho đọc.
List	Danh sách các mục văn bản.
TextField	Trường chứa các dòng văn bản (text) soạn thảo được.
TextArea	Thành phần chứa các văn bản soạn thảo được.

Ví dụ 7.2. Tạo ra các nút và cách sử dụng chúng.

```
import java.awt.*;
import java.applet.*;

public class CheckboxApplet extends Applet{
    public void init(){
        CheckboxGroup amGroup = new CheckboxGroup();
        // Tạo ra nút có tên "Am cao" và không được chọn
        Checkbox amCao = new Checkbox("Am cao", amGroup, false);
        // Tạo ra nút có tên "Am trung binh" và kích hoạt nó (đặt true)
        Checkbox amTBinh=new Checkbox("Am trung binh",true,amGroup);
        // Tạo ra nút có tên "Am thap" và không được chọn
```

```
        Checkbox amThap = new Checkbox("Am thap", false);
        amThap.setCheckboxGroup(amGroup);
        // Bổ sung các nút trên vào nhóm các thành phần
        add(amCao);
        add(amTBinh);
        add(amThap);
        }
    }
```

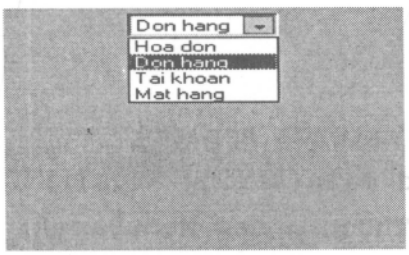
Ví dụ 7.3. Tạo ra một danh sách các mục và chọn lựa một trong số các mục đó.

```
import java.awt.*;
import java.applet.*;
public class ChoiceApplet extends Applet{
  public void init(){
    Choice muc = new Choice();          // Tạo ra một thực đơn pop-up
    muc.add("Hoa don");                 // Đưa mục "Hoa don" vào muc
    muc.add("Don hang");                // Đưa mục "Đon hang" vào muc
    muc.add("Tai khoan");               // Đưa mục "Tai khoan" vào muc
    muc.add("Mat hang");                // Đưa mục "Mat hang" vào muc
    add(muc);                           // Đưa muc vào khung ứng dụng
    muc.select("Don hang");             // Chọn mục "Đon hang"
  }
}
```

Dịch và tạo ra tệp *ChoiceApplet.html* có lệnh đơn giản:

```
<applet code = "ChoiceApplet.class" width = 200 height = 200>
</applet>
```

Thực hiện *ChoiceApplet.html* trong Web Browser, ví dụ Window Explore sẽ nhận được kết quả dạng:



Hình 7.3. Tạo ra danh sách các mục để chọn.

7.2.3. THÀNH PHẦN *MENU*

Lớp *abstract class MenuComponent* là lớp cơ sở cho tất cả các lớp thực hiện những vấn đề liên quan đến thực đơn (*menu*).

Lớp *MenuBar* cài đặt thanh thực đơn và trong đó có thể chứa các thực đơn pull-down.

Lớp *MenuItem* định nghĩa từng mục của thực đơn.

Lớp *Menu* cài đặt các thực đơn *pull-down* để có thể đưa vào một thực đơn bất kỳ.

Lớp *PopUpMenu* biểu diễn cho thực đơn pop-up .

Lớp *CheckboxMenuItem* chứa các mục được chọn để kiểm tra trong các mục thực đơn.

Việc tạo lập một thanh thực đơn cho một *frame* được thực hiện như sau:

1. Tạo ra một thanh thực đơn,

   `MenuBar thanhThDon = new MenuBar();`

2. Tạo ra một thực đơn,

   `Menu thucDon = new Menu("Cac loai banh");`

3. Tạo ra các mục trong thực đơn và đưa vào thực đơn,

   `MenuItem muc = new MenuItem("Banh day");`
   `thucDon.add(muc);` // Đưa *muc* vào *thucDon*

4. Đưa các thực đơn vào thanh thực đơn,

   `thanhThDon.add(thucDon);` // Đưa thucDon vào thanhThDon

5. Tạo ra một *frame* và đưa thanh thực đơn vào *frame* đó.

   `Frame frame = new Frame("Cac mon an");`
   `frame.add(thanhThDon);` // Đưa thanhThDon vào frame

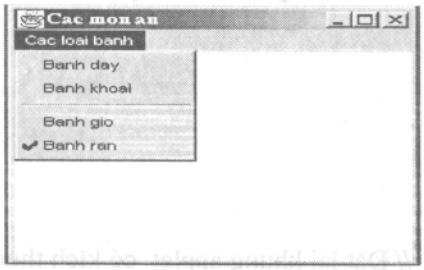
Ví dụ 7.4. Tạo lập và sử dụng các thực đơn.

```
import java.awt.*;
import java.applet.*;

public class MenuDemo extends Applet{
    public static void main(String args[]){
        MenuBar thanhThDon = new MenuBar();
        Menu thucDon = new Menu("Cac loai banh");
        MenuItem b1 = new MenuItem("Banh day");
        thucDon.add(b1);
```

```
        MenuItem b3 = new MenuItem("Banh khoai");
        thucDon.add(b3);
        thucDon.addSeparator(); // Tạo ra một thanh phân cách
        MenuItem b4 = new MenuItem("Banh gio");
        thucDon.add(b4);
        // Tạo ra mục "Banh ran" và đánh dấu để lựa chọn
        thucDon.add(new CheckboxMenuItem("Banh ran"));
        thanhThDon.add(thucDon);
        Frame frame = new Frame("Cac mon an");
        frame.setMenuBar(thanhThDon);
        frame.pack();
        frame.setVisible(true);
        }
    }
```

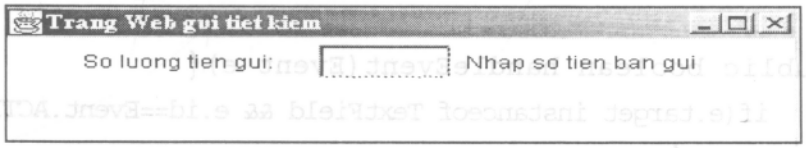
Dịch và thực hiện *java MenuDemo* sẽ cho kết quả dạng:



*Hình 7.4. **Tạo ra các thực đơn.***

Trên đây chúng ta đã xét các thành phần của GUI. Vấn đề quan trọng trong chương trình là phải xử lý được những sự kiện xảy ra khi người sử dụng chọn một trong các thành phần đó. Sau đây chúng ta xét hai ví dụ đơn giản minh họa cho việc sử dụng giao diện đồ họa để nhập và hiển thị thông tin thực hiện được cả hai chế độ, thực hiện độc lập và thực hiện trong Web Browser.

Ví dụ 7.5. Viết chương trình có giao diện đồ họa như sau:

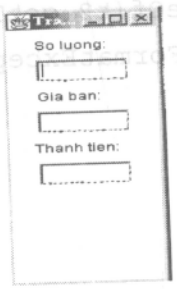


để nhập vào *số tiền gửi tiết kiệm* và hệ thống sẽ tính và cho ra số tiền lãi, với lãi xuất ví dụ 10%. Người sử dụng nhập xong *số lượng gửi,* ví dụ 100, hệ thống sẽ hiển thị *Tiền lãi: 10.0.*

```
// LaiXuat.java
import java.awt.Label;
import java.awt.TextField;
import java.awt.Event;
import java.awt.Frame;
import java.applet.Applet;
public class LaiSuat extends Applet{
    Label l1 = new Label("So luong tien gui:  ");
    Label l2;
    TextField t1;
    int num = 0;
    int getNum(){
        int n;
        try{
            n = Integer.valueOf(t1.getText()).intValue();
        }catch (NumberFormatException e){
            n = 0;
            }
        return n;
        }
    public void init(){
        resize(400,300); // Đặt lại khung applet có kích thước 400×300
        add(l1);        // Bổ sung nhãn l1 vào applet
        t1 = new TextField(6);
        add(t1);        // Bổ sung trường text t1 vào applet
      l2 = new Label ("Nhap so tien ban gui");
        add(l2);        // Bổ sung nhãn l2 vào applet
    }
    void showTotal(int k){
        l2.setText("Tien lai " + String.valueOf(k* 0.1F));
        }
    public boolean handleEvent(Event e){
       if(e.target instanceof TextField && e.id==Event.ACTION_EVENT)
            {
            num = getNum();
            showTotal(num);
            return true;
```

```
        }
    return false;
    }
public static void main(String args[]){
    LaiSuat gui = new LaiSuat();
    gui.init();
    Frame fr = new Frame("Trang Web gui tiet kiem");
    fr.resize(400,300);    // Đặt lại kích thước của frame
    fr.add("Center", gui);  // Bổ sung frame fr vào giữa trang applet
    fr.show();
    }
}
```

Ví dụ 7.6. Viết chương trình có giao diện đồ họa:



để nhập vào *số lượng hàng bán được* và *giá bán*, hệ thống sẽ tính và hiển thị *số tiền bán được* của một mặt hàng nào đó được tính bằng tích của *số lượng* và *giá bán*. Khi nhập xong *số lượng*, hay *giá bán*, người sử dụng phải nhấn phím ENTER báo cho hệ thống (*trường Text*) biết để nhận và xử lý theo sự kiện nhập dữ liệu vào trường *Text*.

```
// BanHang.java
import java.io.*;
import java.awt.Label;
import java.awt.TextField;
import java.awt.Event;
import java.awt.Frame;
import java.applet.Applet;
public class BanHang extends Applet{
    Label l1 = new Label("So luong:  ");
    Label l2 = new Label("Gia ban:   ");
    Label l3 = new Label("Thanh tien:");
    TextField t1, t2, t3;
```

```
int num = 0;
float price, total;
int getNum(){
    int n;
    try{
        n = Integer.valueOf(t1.getText()).intValue();
    }catch (NumberFormatException e){
        n = 0;    // Khi số nhập vào không đúng format thì đặt là 0
        }
    return n;
    }
float getPrice(){
        float p;
        try{
        p = Float.valueOf(t2.getText()).floatValue();
        }catch (NumberFormatException e){
            p = 0.0F;
            }
        return p;
    }
public void init(){
    resize(100,300);
    add(l1);
    t1 = new TextField(6);
    add(t1);
    add(l2);
    t2 = new TextField(6);
    add(t2);
    add(l3);
    t3 = new TextField(6);
    add(t3);
    }
void showTotal(int k, float p){
    t3.setText(String.valueOf(k*p));
    }
public boolean handleEvent(Event e){  // Xử lý các sự kiện nhập dữ liệu
    if (e.target instanceof TextField
```

```
            && e.id == Event.ACTION_EVENT && num == 0 )
            num = getNum();          // Chuyển dãy các chữ số thành số
        else if (e.target instanceof TextField
            && e.id == Event.ACTION_EVENT && num != 0){
            price = getPrice();      // Chuyển dãy các chữ số thành số
        showTotal(num, price);       // Hiển thị kết quả ở trường Text t3
        num = 0;
        return true;
         }
        return false;
        }
    public static void main(String args[]){
        BanHang sale = new BanHang();
        sale.init();
        Frame fr = new Frame("Trang Web ban hang");
        fr.resize(100,300);
        fr.add("Center", sale);
        fr.show();
        }
    }
```

Lưu ý: Hai chương trình trên chạy được cả dạng ứng dụng độc lập lẫn ứng dụng nhúng. Chi tiết hơn về xử lý các sự kiện sẽ được đề cập ở phần 7.5.

7.3. CÁC LỚP XỬ LÝ ĐỒ HỌA

Lớp *abstract class java.awt.Graphics* cung cấp giao diện hỗ trợ để xử lý đồ họa độc lập với các thiết bị (card) đồ họa. Các lớp con của lớp *Graphics* thực thi cài đặt cho những môi trường và các thiết bị đồ họa khác nhau. Để xử lý đồ họa chúng ta phải xác định và tạo ra ngữ cảnh của đồ họa máy tính.

Ngữ cảnh đồ họa

- Đích của việc sử dụng đồ họa,
- Màu để vẽ và màu nền,
- *Mode* vẽ,
- *Font* chữ được chọn để viết (vẽ) chữ,
- Vùng gốc của đối tượng đồ họa,

- Các yếu tố xác định chiều dài, chiều cao của đối tượng ảnh,

Ngữ cảnh đồ họa cung cấp nhiều hàm khác nhau để làm thay đổi các trạng thái thông tin và để vẽ đồ họa. Các chức năng đồ họa gồm có:

- Vẽ các đường, vẽ (và tô màu) các hình chữ nhật, *oval*, đa giác,
- Xử lý văn bản với các *font, kiểu* chữ khác nhau trong chế độ đồ họa,
- Hiển thị, cắt dán, quay và các thao tác khác đối với các ảnh.

7.3.1. LỚP GRAPHICS

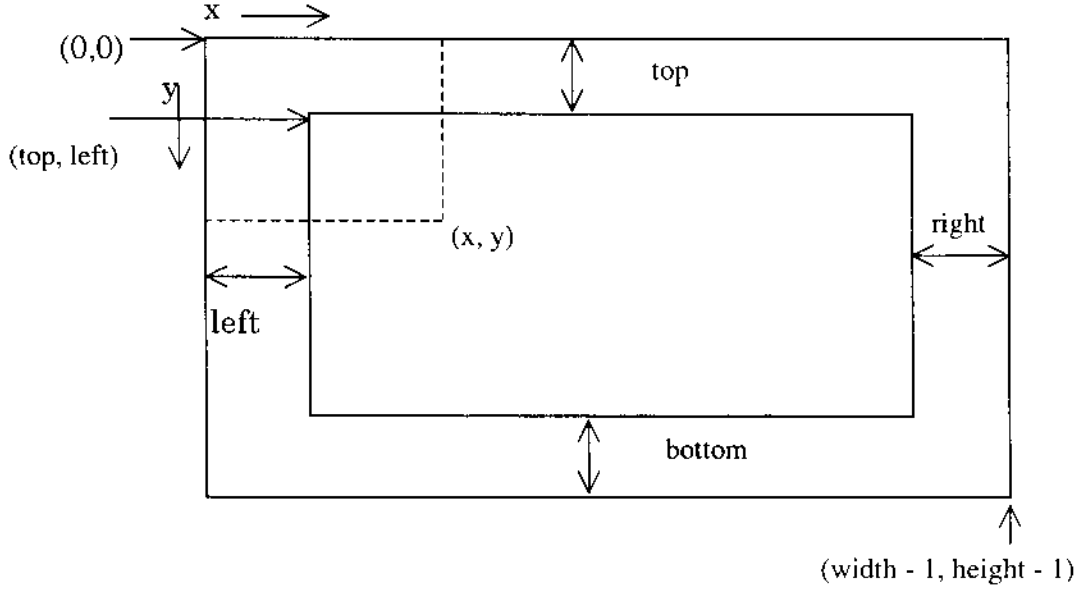
Ngữ cảnh đồ họa không thể tạo ra được trực tiếp bằng cách sử dụng toán tử tạo lập *Graphics()* vì lớp *Graphics* là trừu tượng. Để xác định được ngữ cảnh đồ họa có thể thực hiện theo hai cách:

1) Sử dụng một đối tượng đồ họa có sẵn để tạo ra đối tượng mới bằng cách sử dụng hàm *create()* của lớp *Graphics*.

2) Bởi vì mỗi thành phần đồ họa đều có ngữ cảnh xác định và ngữ cảnh của một đồ họa có thể nhận được bằng cách gọi hàm *getGraphics()* của lớp *Component*.

Lưu ý: Bộ tạo lập ngữ cảnh đồ họa phải đảm bảo là hàm *dispose()* sẽ phải được gọi để giải phóng các thiết bị khi ngữ cảnh đó không còn tiếp tục sử dụng.

Mặt khác khi vẽ một đồ họa chúng ta cũng cần phải xác định các tọa độ, kích thước của hình đồ họa cần vẽ. Hình 7.5 mô tả các tọa độ của các thành phần đồ họa trên máy tính.



*Hình 7.5. **Các tọa độ của các thành phần đồ họa.***

Vùng vẽ trong một thành phần đồ họa không nhất thiết phải có kích thước giống như kích thước của thành phần đó. Kích thước của thành phần đồ họa có thể xác định được bằng hàm:

```
Dimension getSize();      // (độ rộng: w, chiều cao: h)
```

Kích thước của các vùng biên có thể nhận được bởi hàm:

```
Insets getInsets();      // (top, bottom, left, right)
```

Từ đó có thể xác định được kích thước của vùng vẽ đồ họa:

```
Dimension vungVe = getSize();
Insets vungBien = getInsets();
int chieuCao = vungVe.height - vungBien.top - vungBien.bottom;
int chieuRong = vungVe.width - vungBien.left - vungBien.right;
```

Mọi nét vẽ, chữ trong vùng vẽ đều được sử dụng với màu, chế độ vẽ và *font* chữ hiện thời.

7.3.2. LỚP MÀU COLOR

Lớp Color định nghĩa 13 màu cơ bản với các hằng cho trước như sau:

```
Color.black        // Màu đen
Color.blue         // Màu xanh da trời
Color.cyan         // Màu lục lam
Color.darkGray     // Màu xám tối
Color.gray         // Màu xám
Color.green        // Màu xanh lá cây
Color.lightGray    // Màu xám sáng
Color.magenta      // Màu đỏ tươi
Color.orange       // Màu da cam
Color.pink         // Màu (hoa) hồng
Color.red          // Màu đỏ
Color.white        // Màu trắng
Color.yellow       // Màu vàng
```

Các toán tử tạo lập của lớp Color

Color(int r, int g, int b)

Tạo ra một màu theo *mode* RGB (*Red, Green, Blue)* với các giá trị màu *r, g, b* trong khoảng (0-255).

Color(int rgb)

Tạo ra một màu theo *mode* RGB với các giá trị màu *rgb* là tổ hợp của thành phần: màu đỏ được xác định trong các bit từ bit 16 đến 23, màu xanh từ bit 8 - 15 và màu xanh da trời từ bit 0 - 7.

Color(float r, float g, float b)

Tạo ra một màu theo *mode* RGB (*Red, Green, Blue)* với các giá trị màu *r, g, b* trong khoảng (0.0-1.0).

Ví dụ 7.7. Mô tả cách sử dụng màu vẽ và màu nền trong đồ họa.

```
import java.applet.*;
import java.awt.*;
public class ColorApplet extends Applet {
    public void init() {                  // Nạp chồng hàm init()
    setForeground(Color.blue);            // Đặt màu vẽ là blue
    setBackground(new Color(127, 255, 212)) // Đặt màu nền
    }
    public void paint(Graphics g){
      g.drawString("Ve chu mau xanh da troi (blue)!", 75, 50);
    }
  }
```

7.3.3. CHỮ VÀ CÁC FONT CHỮ TRONG ĐỒ HỌA

Tương tự như màu trong chế độ đồ họa, *font* chữ có thể được xác định theo ngữ cảnh hoặc theo các thành phần của đồ họa.

Viết chữ (văn bản - text) trong đồ họa có thể sử dụng các hàm của lớp *Graphics:*

void drawString(String str, int x, int y)

Xâu ký tự *str* được viết (vẽ) ở chế độ đồ họa với màu (màu nét chữ và màu nền) và *font* chữ hiện thời và ký tự đầu tiên được viết ở tọa độ *(x, y).*

void drawChars(Char[] day, int offset, int len, int x, int y)

Xâu ký tự có độ dài *len*, bắt đầu từ vị trí *offset* của *day* (mảng các ký tự) được viết ở chế độ đồ họa với màu (vẽ, nền) và *font* chữ hiện thời và ký tự đầu tiên được viết ở tọa độ *(x, y).*

void drawBytes(Byte[] day, int offset, int len, int x, int y)

Xâu ký tự có độ dài *len*, bắt đầu từ vị trí *offset* của *day* (mảng các byte) được

viết ở chế độ đồ họa với màu (vẽ, nền) và font chữ hiện thời và ký tự đầu tiên được viết ở tọa độ *(x, y)*.

Font getFont()

Cho lại *font* đang vẽ (*font* hiện thời).

void setFont(Font f)

Đặt *font* để vẽ (*font* hiện thời) là *f*.

Lớp *Font* định nghĩa toán tử tạo lập để tạo ra các *font* khác nhau đã có trong hệ thống:

Font(String ten, int kieu, int coChu)

Trong đó *ten* là tên gọi chuẩn của các *font* chữ cho mọi môi trường và được chuyển tương ứng sang các môi trường xác định:

+ "Serif" loại chữ có chân *Serif*,
+ "SansSerif" loại chữ không chân *SansSerif*,
+ "Monospaced" loại chữ đơn giản, *font* chữ cố định,
+ "Dialog" loại chữ cho hội thoại,
+ "DialogInput" loại chữ để nhập vào hội thoại,
+ "Symbol" loại chữ được ánh xạ sang *font Symbol*.

Biến *kieu* là một trong các hằng xác định kiểu *font* chữ:

+ *Font.BOLD* kiểu chữ đậm,
+ *Font.ITALIC* kiểu chữ nghiêng,
+ *Font.PLAIN* kiểu chữ thông thường,
+ *(Font.BOLD | FONT.ITALIC)* Kiểu chữ cả đậm và nghiêng.

Biến *coChu* xác định co chữ được tính theo số điểm (1 point = 1/ 72 inch).

Ví dụ 7.8. Mô tả cách sử dụng *font* chữ chuẩn trong đồ họa.

```
import java.applet.*;
import java.awt.*;
public class FontApplet extends Applet {
    public void init(){
        setBackground(Color.white);
        setForeground(Color.black);
    }
```

```
    public void paint(Graphics g) {
        g.setFont(new Font("Serif", Font.BOLD, 20));
        g.drawString("Serif", 50, 50);
        g.setFont(new Font("SansSerif", Font.ITALIC, 20));
        g.drawString("SansSerif", 50, 75);
        //g.setFont(new Font("Monospaced", Font.PLAIN, 20));
        g.setFont(new Font("DialogInput",Font.BOLD|Font.ITALIC, 20));
        g.drawString("Dialog", 50, 100);
        g.setFont(new Font("DialogInput", Font.BOLD, 20));
        g.drawString("DialogInput", 50, 125);
        g.setFont(new Font("Symbol", Font.PLAIN, 20));
        g.drawString("\u03A4\u03A6\u03C4", 50, 150);
    }
}
```

Sau khi dịch được kết quả là *FontApplet.class*, chúng ta tạo ra tệp HTML (*FontApplet.html*) có lệnh đơn giản:

```
<applet code = "FontApplet.class" width = 300 height = 300>    </applet>
```

Thực hiện *FontApplet.html* trong một trình duyệt, ví dụ Window Explore sẽ cho kết quả:

Hình 7.6. Các font chữ chuẩn.

7.3.4. VẼ CÁC HÌNH HỌC NGUYÊN THỦY

Lớp *Graphics* cung cấp các hàm để vẽ các đường thẳng, vẽ hoặc tô màu các hình chữ nhật, *oval*, cung và các đa giác. Hầu hết các hình học nguyên thủy, trừ đường thẳng đều tô màu được. Để vẽ các hình học nguyên thủy chúng ta thường sử dụng hàm thành phần của lớp *Graphics* có tên bắt đầu bằng *draw* và tiếp theo là tên gọi ngắn gọn của các hình, như *drawLine, drawRect,* v.v. Tương tự như vậy, để tô màu một hình chúng ta thường sử dụng hàm bắt đầu bằng *fill* sau đó là tên ngắn gọn của hình định tô màu, ví dụ *fillRect, fillRoundRect,* v.v.

Vẽ đường thẳng

void drawLine(

int x1, int y1, // Từ tọa độ (x1, y1)

int x2, int y2) // Đến tọa độ (x2, y2)

Vẽ đường thẳng từ tọa độ *(x1, y1)* đến tọa độ *(x2, y2)* sử dụng màu hiện thời.

Vẽ hình chữ nhật

void drawRect(

`int x, int y,` // Góc trên bên trái từ tọa độ (x, y)

`int width, int height)` // Chiều rộng width và chiều cao height

Vẽ hình chữ nhật có góc trên bên trái ở tọa độ *(x, y)*, có độ rộng *width*, chiều cao *height* và sử dụng màu hiện thời.

void fillRect(

int x, int y, // Góc trên bên trái từ tọa độ (x, y)

int width, int height) // Chiều rộng width và chiều cao height

Tô hình chữ nhật được xác định như trên với màu hiện thời.

void drawRoundRect(

int x, int y, // Góc trên bên trái từ tọa độ (x, y)

int width, int height, // Chiều rộng width và chiều cao height

int arcWidth, int arcHeight) // Độ rộng và chiều cao của cung xác định góc tròn

Vẽ hình chữ nhật có góc tròn bắt đầu từ đỉnh trên bên trái ở tọa độ *(x, y)*, có độ rộng *width*, chiều cao *height* với độ rộng *arcWidth*, chiều cao *arcHeight* của cung xác định độ tròn của góc và sử dụng màu hiện thời.

void fillRoundRect(

int x, int y, // Góc trên bên trái từ tọa độ (x, y)

int width, int height, // Chiều rộng width và chiều cao height

int arcWidth, int arcHeight) // Độ rộng và chiều cao của cung xác định góc tròn

Tô hình chữ nhật có góc tròn được xác định như trên với màu hiện thời.

void draw3DRect(

int x, int y, // Góc trên bên trái từ tọa độ (x, y)

int width, int height, // Chiều rộng width và chiều cao height

int arcWidth, int arcHeight, // Độ rộng và chiều cao của cung xác định góc tròn

boolean raised) // Độ nổi hay chìm của hình

Vẽ hình chữ nhật 3- chiều (3-D) được xác định như trên. Biến *raised* được dùng để chỉ ra là hình nổi hay chìm tùy thuộc giá trị là *true* hay *false*.

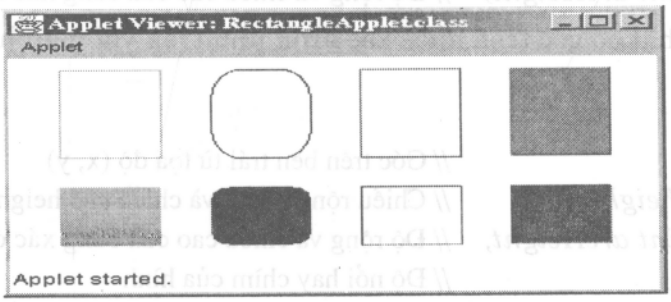
Ví dụ 7.9. Mô tả cách sử dụng các hàm của lớp *Graphics* để vẽ, tô màu các hình.

```
// Các hình chữ nhật
import java.applet.*;
import java.awt.*;
public class RectangleApplet extends Applet{
    public void paint(Graphics g) {
        // Vẽ hình chữ nhật và tô màu hình vuông
        g.setColor(Color.orange);
        g.drawRect(25, 10, 50, 75);
        g.fillRect(25, 110, 50, 50);
        // Vẽ hình chữ nhật và tô màu hình vuông có góc tròn
        g.setColor(Color.blue);
        g.drawRoundRect(100, 10, 50, 75, 30, 50);
        g.fillRoundRect(100, 110, 50, 50,20, 20);
        // Vẽ hình chữ nhật và hình vuông 3D
        g.setColor(Color.red);
        g.draw3DRect(175, 10, 50, 75, true);
        g.draw3DRect(175, 110, 50, 50, false);
        // Tô màu hình chữ nhật và hình vuông 3D
        g.setColor(Color.green);
        g.fill3DRect(250, 10, 50, 75, true);
        g.fill3DRect(250, 110, 50,50, false);
    }
}
```

Sau khi dịch được kết quả là *RectangleApplet.class*, chúng ta tạo ra tệp HTML *RectangleApplet.html* có lệnh đơn giản:

```
<applet code = "RectangleApplet.class" width = 400 height = 200>
</applet>
```

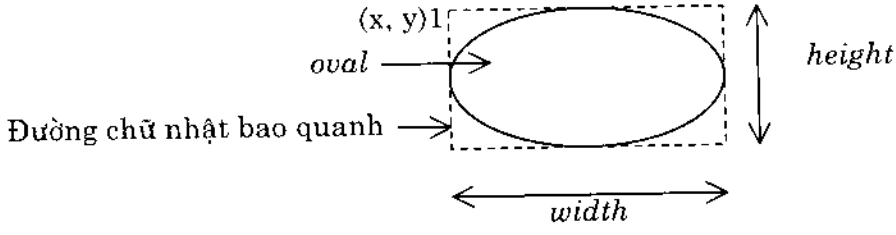
Thực hiện *appletviewer FontApplet.html* sẽ cho kết quả:



Hình 7.7. Vẽ các hình chữ nhật.

Vẽ hình *oval*

Lớp *Graphics* cung cấp hàm thành phần để vẽ, tô màu các hình *oval (ellipse)*. Hình *oval* được vẽ nội tiếp bên trong hình chữ nhật như hình 7.8.



Hình 7.8. ***Hình* oval *nội tiếp trong hình chữ nhật.***

void drawOval(

int x, int y, // Góc trên bên trái từ tọa độ (x, y)

`int width, int height)` // Chiều rộng width và chiều cao height của hình chữ nhật

Vẽ hình *oval* nội tiếp trong hình chữ nhật có góc trên bên trái ở tọa độ *(x, y)*, có độ rộng *width*, chiều cao *height* và sử dụng màu hiện thời.

void fillOval(

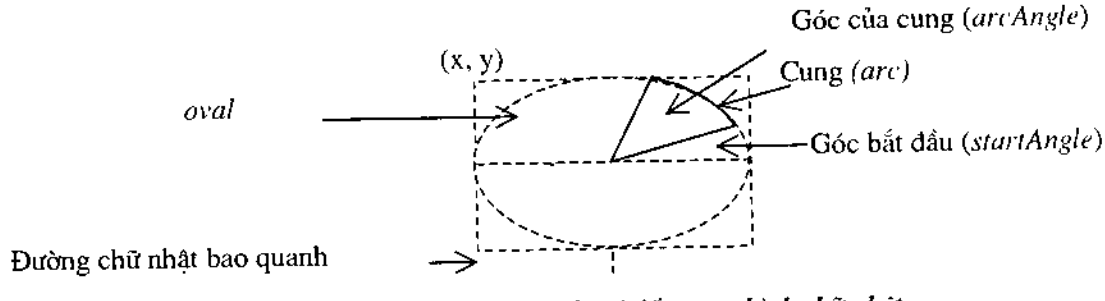
int x, int y, // Góc trên bên trái từ tọa độ (x, y)

int width, int height) // Chiều rộng width và chiều cao height của hình chữ nhật

Tô màu hình *oval* nội tiếp trong hình chữ nhật được xác định như trên theo màu hiện thời.

Vẽ các cung

Lớp *Graphics* cung cấp hàm thành phần để vẽ, tô màu các cung, các góc cánh quạt của hình *ellipse*. Các cung của hình *ellipse* nội tiếp bên trong hình chữ nhật được vẽ như trong hình 7.9.



Hình 7.9. ***Hình* oval *nội tiếp trong hình chữ nhật.***

void drawArc(

```
int x, int y,                // Góc trên bên trái từ tọa độ (x, y)
int width, int height,       // Chiều rộng width và chiều cao height của hình chữ nhật
int startAngle,              // Góc bắt đầu của cung
int arcAngle)                // Góc của cung (cánh quạt)
```

Vẽ một cung được xác định như hình 7.9 với màu hiện thời.

void fillArc(

```
int x, int y,                // Góc trên bên trái từ tọa độ (x, y)
int width, int height)       // Chiều rộng width và chiều cao height của hình chữ nhật
int startAngle,              // Góc bắt đầu của cung
int arcAngle)                // Góc của cung (cánh quạt)
```

Tô một cung được xác định như hình 7.9 theo màu hiện thời.

Ví dụ 7.10. Vẽ và tô màu các cung.

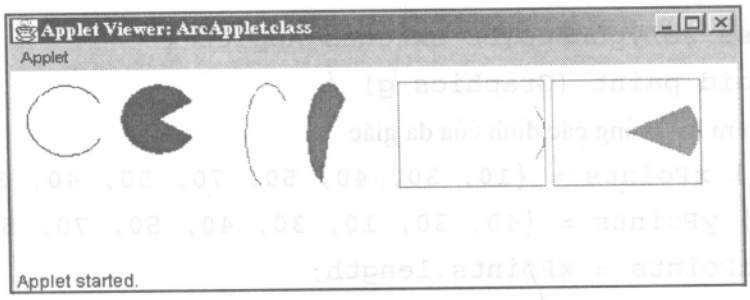
```
// ArcApplet.java
import java.applet.*;
import java.awt.*;
public class ArcApplet extends Applet {
    public void paint(Graphics g) {
        // Vẽ một cung và tô màu blue (xanh da trời)
        g.setColor(Color.blue);
        g.drawArc(10, 10, 50, 50, 30, 300);
        g.fillArc(70, 10, 50, 50, 30, 300);
        // Vẽ một cung ellipse và tô màu red (đỏ)
        //Elliptical arc
        g.setColor(Color.red);
        g.drawArc(150, 10, 30, 75, 40, 210);
        g.fillArc(190, 10, 30, 75, 40, 210);
        g.setColor(Color.magenta);
        g.drawRect(250, 10, 95, 75); // Vẽ hình chữ nhật màu magenta
        g.drawArc(250, 10, 95, 75, -30, 60); // Vẽ một cung
        g.drawRect(350, 10, 95, 75);
        g.fillArc(350, 10, 95, 75, -30, 60);
    }
}
```

Sau khi dịch được kết quả là *ArcApplet.class*, chúng ta tạo ra tệp HTML

ArcApplet.html có lệnh đơn giản:

```
<applet code = "ArcApplet.class" width = 400  height = 200>
</applet>
```

Thực hiện *appletviewer ArcApplet.html* của JDK sẽ cho kết quả:



Hình 7.10. Vẽ và tô màu các cung.

Vẽ các hình đa giác (Polygon)

Hình đa giác là dãy khép kín các đoạn thẳng. Cho trước một dãy các điểm (gọi là các đỉnh), các đoạn thẳng nối liên tiếp các đỉnh cho đến đỉnh cuối và từ đỉnh cuối lại nối tiếp với đỉnh đầu tiên.

Lớp *Polygon* có toán tử tạo lập sau:

Polygon(int[] xpoints, int[] ypoints, int npoints)

Tạo ra hình đa giác có các đỉnh được xác định bởi các tọa độ liên tiếp (*xpoints[i], ypoints[i]*) với i = 0, 1, ..., *npoints*.

Lớp *Graphics* có hai hàm để vẽ hình đa giác:

void drawPolygon(int[] xpoints, int[] ypoints, int npoints)

Vẽ hình đa giác có các đỉnh được xác định bởi các tọa độ liên tiếp là (*xpoints[i], ypoints[i]*) với i = 0, 1, ..., *npoints*.

void drawPolygon(Polygon p)

Vẽ hình đa giác *p*.

void fillPolygon(int[] xpoints, int[] ypoints, int npoints)

Tô màu hình đa giác có các đỉnh được xác định bởi các tọa độ liên tiếp (*xpoints[i], ypoints[i]*) với i = 0, 1, ..., *npoints* theo màu hiện thời.

void fillPolygon(Polygon p)

Tô màu hình đa giác *p* theo màu hiện thời.

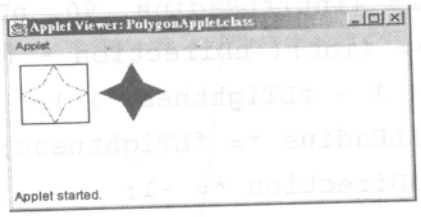
Ví dụ 7.11. Vẽ các hình đa giác.

```
// Vẽ và tô màu hai hình đa giác
import java.applet.*;
import java.awt.*;
  public class PolygonApplet extends Applet {
    public void paint (Graphics g) {
        // Tạo ra hai mảng các đỉnh của đa giác
        int[] xPoints = {10, 30, 40, 50, 70, 50, 40, 30};
        int[] yPoints = {40, 30, 10, 30, 40, 50, 70, 50};
        int nPoints = xPoints.length;
        // Tạo ra hình đa giác và vẽ hình đó
        Polygon hinhDaGiac= new Polygon(xPoints, yPoints, nPoints);
        g.setColor(Color.red);
        g.drawPolygon(hinhDaGiac);
        // Đặt hình đa giác vào trong hình chữ nhật
        g.setColor(Color.black);
        Rectangle bounds = hinhDaGiac.getBounds();
        g.drawRect(bounds.x,bounds.y, bounds.width, bounds.height);
        // Tô màu hình đa giác có tập các đỉnh tương tự như trên, chỉ khác là các tọa
        // độ gốc của x được chuyển qua phải 70 điểm
    g.setColor(Color.darkGray);
    g.fillPolygon(
          new int[] {80, 100, 110, 120, 140, 120, 110, 100},
      new int[] {40, 30, 10, 30, 40, 50, 70, 50},8);
        }
     }
```

Sau khi dịch được kết quả là *PolygonApplet.class*, chúng ta tạo ra tệp HTML *PolygonApplet.html* có lệnh đơn giản:

```
<applet code = "PolygonApplet.class" width = 400  height = 200>
</applet>
```

Thực hiện *appletviewer PolygonApplet.html* của JDK sẽ cho kết quả:



*Hình 7.11. **Vẽ và tô màu hai hình đa giác.***

Sử dụng các hình học nguyên thủy, kết hợp với các phương pháp vẽ và tô màu theo các bước khác nhau có thể tạo ra nhiều hình ảnh khá sinh động.

Ví dụ 7. 12. Vẽ hình xoắn ốc.

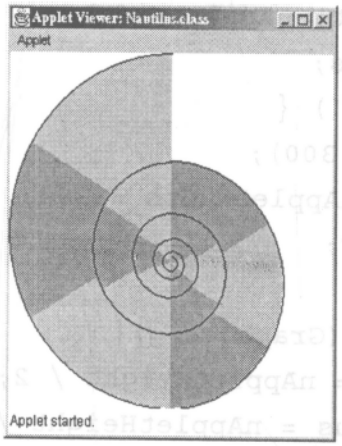
```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
public class HinhOc extends Applet {
     int nAppletHeight, nAppletWidth;
     float fLTightness;
     public void init() {
         resize( 300, 300);
    nAppletHeight = nAppletWidth = 300;
    fLTightness =0.7F;
    }
     public void paint(Graphics g){
          int nCenter = nAppletHeight / 2;
          float fLRadius = nAppletHeight / 2;
          int nDirection = 1;
          for( int nI = 0; nI < 10; nI++) {
               int x = (nAppletHeight / 2) - (int)fLRadius;
               int y = nCenter - (int)fLRadius;
               for( int nJ = 0; nJ < 3; nJ++) {
                    if( 0 != (nJ + ((1+nDirection) / 2)) % 2)
                         g.setColor( Color.orange);
               else g.setColor( Color.magenta);
               g.fillArc(x,y,2*(int)fLRadius,2* (int)fLRadius,
                  90 + (nDirection * nJ * 60),nDirection * 60);
               }
               g.setColor( Color.black );
               g.drawArc( x, y, 2 *(int)fLRadius,
```

```
                2* (int)fLRadius, 90, nDirection * 180 );
        nCenter += (int)( nDirection * (int)fLRadius *
                ( 1 - fLTightness ) );
                fLRadius *= fLTightness;
                nDirection *= -1;
            }
        }
}
```

Sau khi dịch được kết quả là *HinhOc.class*, chúng ta tạo ra tệp HTML *HinhOc.html* có lệnh đơn giản:

```
<applet code = "HinhOc.class" width = 400 height = 200>
</applet>
```

Thực hiện *appletviewer HinhOc.html* của JDK sẽ cho kết quả:



Hình 7.12. ***Hình xoán ốc.***

Các *mode* vẽ

Mode (*mốt*) vẽ của đối tượng đồ họa được mặc định là *viết đè*. Trong chế độ đó, thì các điểm vẽ sau sẽ vẽ đè lên bề mặt của ảnh. Để thay đổi *mode* vẽ chúng ta có thể sử dụng:

Graphics.setPaintMode()

Đặt lại *mode* vẽ là *Mode* vẽ đè.

Graphics.setXORMode(Color c1)

Đặt *mode* vẽ là XOR (*Mode* loại trừ), trong đó những bề mặt chung của các

hình là bị loại trừ giữa màu hiện thời và màu chỉ định *c1*. Đối tượng ảnh ở *mode* XOR để xử lý các hình học nguyên thủy, văn bản và các ảnh được vẽ trên các bề mặt theo *mode* loại trừ nhau. Ba màu cơ bản được tổ hợp lại để xác định màu vẽ như sau:

colorAfterRendering(x, y) = colorBeforeRendering(x, y) ⊗ graphics.foregroundColor ⊗ graphics.alternateColor

Trong đó, ⊗ là phép toán hoặc loại trừ trên bit, *alternateColor* được xác định chỉ khi ở *mode* vẽ *XORMode*. Ví dụ, hình 7.13 minh họa cách sử dụng cách pha màu và các mốt vẽ để tạo ra các hiệu ứng khác nhau.

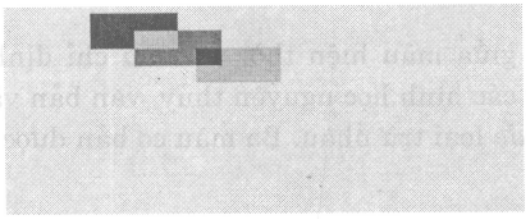
Ví dụ 7.13. Các *mode* vẽ trong chế độ đồ họa

```
import java.applet.*;
import java.awt.*;
public class XORModeApplet extends Applet{
  public void paint(Graphics g){
    // Đặt màu nền là yellow (màu vàng)
    setBackground(Color.yellow);
    // Đặt màu vẽ (màu hiện thời) là blue (xanh) và vẽ hình vuông ở đỉnh trên
    g.setColor(Color.blue);
    g.fillRect(50,10,50,50);
    // Đặt màu vẽ (màu hiện thời) là cyan (lục lam) và vẽ hình vuông ở đỉnh dưới
    g.setColor(Color.cyan);
    g.fillRect(110,60,50,50);
    // Đặt màu loại trừ là blue
    g.setXORMode(Color.blue);
    // Sử dụng màu hiện thời là cyan và vẽ hình vuông ở giữa
    g.fillRect(75,35,50,50);
    }
  }
```

Sau khi dịch được kết quả là `XORModeApplet.class`, chúng ta tạo ra tệp HTML `XORModeApplet.html` có lệnh đơn giản:

```
<applet code = "XORModeApplet.class" width = 400  height = 200>
</applet>
```

Thực hiện `XORModeApplet.html` trong Web Browser sẽ cho kết quả:



Hình 7.13. Các mode vẽ trong đồ họa.

Ví dụ sau minh họa cách sử dụng *XORMode* để vẽ các hình *oval* .

Ví dụ 7. 14. Vẽ các hình *oval* trong mốt *XORMode*.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;
public class Ovals extends Applet {
     int nAppletHeight, nAppletWidth;
     float fLTightness;
     public void init() {
         resize( 300, 300);
    nAppletHeight = nAppletWidth = 300;
    }
public void paint(Graphics g) {
    float fLLongAxisLength = nAppletWidth;
    float fLShortAxisLength = nAppletHeight / 2;
    boolean fLongAxisVertical = true;
    g.setColor( Color.white );
     for( int nI = 0 ; nI < 10 ; nI++) {
         int x, y, width, height;

         if( fLongAxisVertical ) {
             x = (nAppletWidth / 2) -
                 (int)(fLShortAxisLength / 2);
             y = (nAppletHeight / 2) -
                 (int)(fLLongAxisLength / 2);
             width = (int)fLShortAxisLength;
             height = (int)fLLongAxisLength;
         } else {
             x = (nAppletWidth / 2) -
                 (int)(fLLongAxisLength / 2);
```

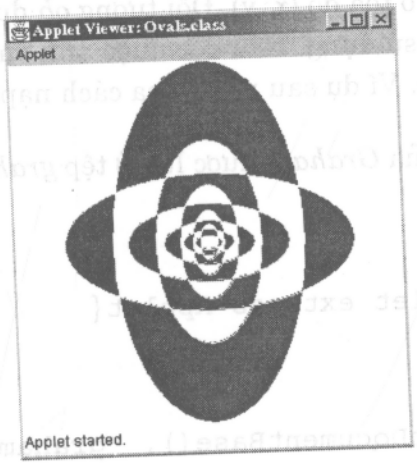
```
                y = (nAppletHeight / 2) -
                    (int)(fLShortAxisLength / 2);
                width = (int)fLLongAxisLength;
                height = (int)fLShortAxisLength;}
            g.setXORMode( Color.blue);
            g.fillOval( x, y, width, height);
            g.setPaintMode();
            g.drawOval( x, y, width, height);

            fLLongAxisLength *= 0.75;
            fLShortAxisLength *= 0.75;
            fLongAxisVertical = ! fLongAxisVertical;}
        return;
    }
}
```

Sau khi dịch được kết quả là *Ovals.class*, chúng ta tạo ra tệp HTML *Ovals.html* có lệnh đơn giản:

```
<applet code = "Ovals.class" width = 400  height = 200>
</applet>
```

Thực hiện chương trình trên với *appletviewer* chúng ta có kết quả:



Hình 7.14. Vẽ các hình oval trong các mode XOR.

7.3.5. XỬ LÝ HÌNH ẢNH VÀ ÂM THANH

Applet có thể xử lý các ảnh dạng *format GIF, JPEG* và các tệp âm thanh (audio) dạng AU, AIFF, WAV và MIDI.

Xử lý ảnh

Lớp *Image* cung cấp các giao diện để xử lý ảnh trên màn hình độc lập với môi trường. Một ảnh muốn hiển thị lên màn hình thì trước tiên phải nạp được xuống từ nơi chứa ảnh đó hoặc phải được tạo ra bằng một cách nào đó.

Lớp Toolkit cung cấp các hàm được nạp chồng để đọc và nạp các tệp ảnh được lưu theo *format* GIF hoặc JPEG. Ví dụ: đọc một tệp đồ họa có tên *Cover.gif* vào đối tượng *image1* của lớp *Image*:

```
Toolkit currnetTK = Toolkit.getDefaultToolkit();
Image image1 = currnetTK.getImage("Cover.gif");
```

Đối với những tệp ảnh ở trên mạng thì phải sử dụng lớp URL:

```
URL url1 = new URL("http://www.example.com/Cover.gif");
Image image2 = currnetTK.getImage(url1);
```

Lớp *Applet* có hàm *getImage()* được nạp chồng để đọc các tệp đồ họa và hàm *drawImage()* để vẽ ảnh.

Image getImage(URL url)

Image getImage(URL url, String name)

url là địa chỉ của URL, *name* là tên của tệp ảnh.

boolean drawImage(Image img, int x, int y, ImageObserver ob)

Vẽ ảnh có tên là *img* ở tọa độ (*x*, y). Đối tượng *ob* được sử dụng để thông báo có nhiều ảnh được phép sử dụng. Nếu đọc được ảnh và nạp thành công thì trả lại *true*, ngược lại là *false*. Ví dụ sau minh họa cách nạp và hiển thị ảnh.

Ví dụ 7.15. Hiển thị ảnh của *Graham* được lưu ở tệp *graham.gif*.

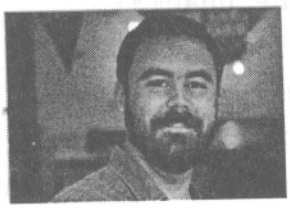
```
import java.applet.*;
import java.awt.*;
public class ImageApplet extends Applet{
  Image img;
  public void init(){
    img = getImage(getDocumentBase(), "graham.gif");
    }
  public void paint(Graphics g){
    g.drawImage(img, 10, 10, this);
    }
  }
```

Sau khi dịch được kết quả là `ImageApplet.class`, chúng ta tạo ra tệp HTML

`ImageApplet.html` có lệnh đơn giản:

```
<applet code = "ImageApplet.class" width = 550 height = 400>
</applet>
```

Thực hiện `ImageApplet.html` trong Web Browser sẽ cho kết quả:



*Hình 7.15. **Hiển thị hình ảnh.***

Lưu ý: Tệp ảnh *graham.gif* phải có ở thư mục hiện thời cùng với thư mục chứa các tệp `ImageApplet.html`, `ImageApplet.class`.

Xử lý âm thanh

Giao diện *AudioClip* khai báo các hàm mẫu và được cài đặt để xử lý âm thanh:

void loop()

void play()

void stop()

Hàm *loop()* được sử dụng chạy lặp lại một đoạn âm thanh (*audio clip*), *play()* chạy một đoạn âm thanh và nếu đoạn âm thanh đang chạy thì có thể sử dụng *stop()* để dừng lại.

Lớp *Applet* cung cấp hàm *getAudioClip()* để tìm các tệp âm được xác định bởi URL.

AudioClip getAudioClip(URL url)

AudioClip getAudioClip(URL url, String fileName)

url là địa chỉ tuyệt đối (địa chỉ hiện thời nơi chứa tệp các lớp chương trình) hoặc địa chỉ từ xa được xác định bởi URL và *fileName* là tên tệp chứa các đoạn âm thanh cần xử lý.

void play(URL url)

void play(URL url, String fileName)

Hàm *play()* được nạp chồng để tìm và chạy một đoạn âm thanh lưu ở tệp *fileName* và được xác định bởi URL.

Ví dụ 7.16. Chương trình dạo một bản nhạc

```
import java.applet.*;
import java.awt.*;
public class SoundApplet extends Applet{
    AudioClip audio;
    String msg = "Hay nghe nhac!";
    public void init(){
        String audioFile = getParameter("audiofile");
        if(audioFile != null){
          audio= getAudioClip(getDocumentBase(), audioFile);
          msg = "Hãy lắng nghe!";
            }
        else{
          showStatus("Audio file cannot be found!");
          msg = "Hãy im lặng!";
        }
    }
    public void start(){
        if(audio != null) audio.loop();
        }
    public void stop(){
        if(audio != null) audio.stop();
        }
    public void destroy(){
        audio = null;
        }
    public void paint(Graphics gfx){
    gfx.translate(getInsets().left,getInsets().top);
        gfx.drawString(msg, 50, 50);
        }
}
```

Dịch xong chương trình trên thì tạo ra tệp .html có dạng:

```
<title>SoundApplet </title>
<applet archive = "JavaClass.jar" code = "SoundApplet.class"
        width = 200 height = 200>
<param name ="audiofile" value = "music.au">
</applet>
```

Trong đó tệp âm thanh "*music.au*" được lưu trữ ở thư mục hiện thời, nơi chứa *SoundApplet.html.*

7.4. BỐ TRÍ VÀ SẮP XẾP CÁC THÀNH PHẦN GIAO DIỆN TRONG CÁC ỨNG DỤNG

Khi thiết kế giao diện đồ họa cho một ứng dụng, chúng ta phải quan tâm đến kích thước và cách bố trí (*layout*) *các thành phần giao diện* như: *Button, Checkbox, TextField, Menu, v.v.* sao cho tiện lợi nhất đối với người sử dụng. Java có các lớp đảm nhiệm những công việc trên và quản lý các thành phần giao diện GUI trong các phần tử chứa. Bảng 7.2 cung cấp bốn lớp quản lý *layout* (cách bố trí và sắp xếp) các thành phần GUI.

Bảng 7.2. Các lớp quản lý cách tổ chức các thành phần giao diện

Tên lớp	Mô tả
FlowLayout	Xếp các thành phần giao diện trước tiên theo hàng từ trái qua phải, sau đó theo cột từ trên xuống dưới. Cách sắp xếp này là mặc định đối với *Panel* và *Applet.*
GridLayout	Các thành phần giao diện được sắp xếp trong các ô lưới hình chữ nhật lần lượt theo hàng từ trái qua phải và theo cột từ trên xuống dưới trong một phần tử chứa. Mỗi thành phần giao diện chứa trong một ô.
BorderLayout	Các thành phần giao diện (ít hơn 5) được đặt vào các vị trí theo các hướng: *north (bắc), south (nam), west (tây), east (đông)* và *center (trung tâm)).* Cách sắp xếp này là mặc định đối với lớp Window, Frame và Dialog.
GridBagLayout	Cho phép đặt các thành phần giao diện vào lưới hình chữ nhật, nhưng một thành phần có thể chiếm nhiều ô.

Các phương pháp thiết kế *layout*

Để biết *layout* hay để đặt lại *layout* cho chương trình ứng dụng, chúng ta có thể sử dụng hai hàm của lớp *Container:*

LayoutManager getLayout();

void setLayout(LayoutManager mgr);

Các thành phần giao diện sau khi đã được tạo ra thì phải được đưa vào một phần tử chứa nào đó. Hàm *add()* của lớp *Container* được nạp chồng để thực hiện nhiệm vụ đưa các thành phần vào phần tử chứa.

Component add(Component comp)

Component add(Component comp, int index)

Component add(Component comp, Object constraints)

Component add(Component comp, Object constraints, int index)

Trong đó, đối số *index* được sử dụng để chỉ ra vị trí của ô cần đặt thành phần giao diện *comp* vào. Đối số *constraints* xác định các hướng để đưa *comp* vào phần tử chứa.

Ngược lại, khi cần loại ra khỏi phần tử chứa một thành phần giao diện thì sử dụng các hàm sau:

void remove(int index)

void remove(Component comp)

void removeAll()

Lưu ý: Nếu không sử dụng các lớp quản lý *layout* để bố trí sắp xếp các thành phần giao diện của ứng dụng theo một *layout* cố định thì khi thay đổi kích thước của khung chương trình (*frame*), những thành phần đó có thể bị thay đổi, bị xê dịch lung tung không theo thứ tự cần thiết.

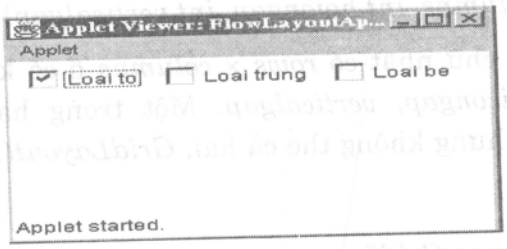
Ví dụ 7.17. Minh họa về các thành phần có thể bị thay đổi khi thay đổi kích thước *frame*.

```
import java.awt.*;
import java.applet.*;
public class FlowLayoutApplet extends Applet{
public void init(){
   CheckboxGroup nhom = new CheckboxGroup();
   Checkbox cb1 = new Checkbox("Loai to", true);
   Checkbox cb2 = new Checkbox("Loai trung", false);
   Checkbox cb3 = new Checkbox("Loai be", false);
   add(cb1);
   add(cb2);
   add(cb3);
  }
}
```

Sau khi dịch được kết quả là `FlowLayoutApplet.class`, chúng ta tạo ra tệp HTML `FlowLayoutApplet.html` có lệnh đơn giản:

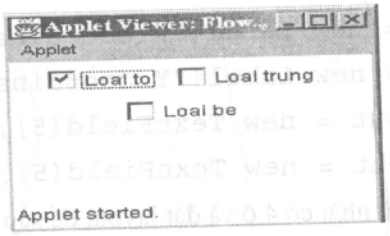
```
<applet code = "FlowLayoutApplet.class" width = 200 height = 200>
</applet>
```

Thực hiện *appletviewer* `FlowLayoutApplet.html` sẽ cho kết quả:



*Hình 7.16. **Các thành phần giao diện có thể thay đổi.***

Nhưng trong chương trình trên, các thành phần giao diện (*Button*) chưa được sắp xếp theo một *layout* cố định nên khi thay đổi kích thước của khung *applet*, như thu hẹp chiều rộng thì các nút của *Checkbox* sẽ bị di chuyển. Ví dụ, khi thu hẹp lại chiều rộng của hình 7.16 sẽ cho hình 7.17.



*Hình 7.17. **Thay đổi lại kích thước của khung chương trình.***

Lớp FlowLayout

Lớp *FlowLayout* cung cấp các hàm tạo lập để sắp hàng các thành phần giao diện:

FlowLayout()
FlowLayout(int aligment)
FlowLayout(int aligment, int horizongap, int verticalgap)
public static final int LEFT
public static final int CENTER
public static final int RIGHT

Đối số *aligment* xác định cách sắp theo hàng: từ trái, phải hay trung tâm, *horizongap* và *verticalgap* là khoảng cách tính theo *pixel* giữa các hàng các cột. Trường hợp mặc định thì khoảng cách giữa các hàng, cột là 5 *pixel*.

Lớp GridLayout

Lớp *GridLayout* cung cấp các hàm tạo lập để sắp hàng các thành phần giao diện:

GridLayout()
GridLayout(int rows, int columns)

GridLayout(int rows, int columns, int hoiongap, int verticalgap)

Tạo ra một lưới hình chữ nhật có *rows* × *columns* ô có khoảng cách giữa các hàng các cột là *horizongap, verticalgap*. Một trong hai đối số *rows* hoặc *columns* có thể là 0, nhưng không thể cả hai, *GridLayout(1,0)* là tạo ra lưới có một hàng.

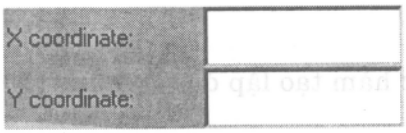
Ví dụ 7.18. Mô tả cách sử dụng *GridLayout*

```
import java.awt.*;
import java.applet.*;
public class GridLayoutApplet extends Applet {
    public void init() {
        //Cread a list of colors
        Label xLabel = new Label("X coordinate: ");
        Label yLabel = new Label("Y coordinate: ");
        TextField xInput = new TextField(5);
        TextField yInput = new TextField(5);
        // Tạo ra lưới hình chữ nhật có 4 ô và đặt layout để sắp xếp các thành phần
        // xLabel, xInput, yLabel, yInput
        setLayout(new GridLayout(2,2));
        add(xLabel); // Đặt xLabel vào ô thứ nhất
        add(xInput); // Đặt xInput vào ô thứ hai
        add(yLabel); // Đặt yLabel vào ô thứ ba
        add(yInput); // Đặt yInput vào ô thứ tư
    }
}
```

Sau khi dịch được kết quả là `GridLayoutApplet.class`, chúng ta tạo ra tệp HTML `GridLayoutApplet.html` có lệnh đơn giản:

```
<applet code ="GridLayoutApplet.class" width =200 height =60>
</applet>
```

Thực hiện `GridLayoutApplet.html` trong Web Browser sẽ cho kết quả:



*Hình 7.18. **Tạo ra lưới 4 ô để sắp xếp các thành phần giao diện.***

Lớp *BorderLayout*

Lớp BorderLayout cho phép đặt một thành phần giao diện vào một trong bốn hướng: bắc (NORTH), nam (SOUTH), đông (EAST), tây (WEST) và ở giữa (CENTER).

BorderLayout()

BorderLayout(int horizongap, int verticalgap)

Tạo ra một *layout* mặc định hoặc có khoảng cách giữa các thành phần (tính bằng *pixel*) là *horizongap* theo hàng và *verticalgap* theo cột.

Component add(Component comp)

void add(Component comp, Object constraint)

public static final String NORTH

public static final String SOUTH

public static final String EAST

public static final String WEST

public static final String CENTER

Trường hợp mặc định là *CENTER*, ngược lại, có thể chỉ định hướng để đặt các thành phần *comp* vào phần tử chứa theo *constraint* là một trong các hằng trên.

Ví dụ 7.19. Đặt một nút *Button* có tên "Vẽ" ở trên (NORTH), trường văn bản "Hiển thị thông báo" ở dưới (SOUTH) và hai thanh trượt (SCROLLBAR) ở hai bên cạnh (WEST và EAST).

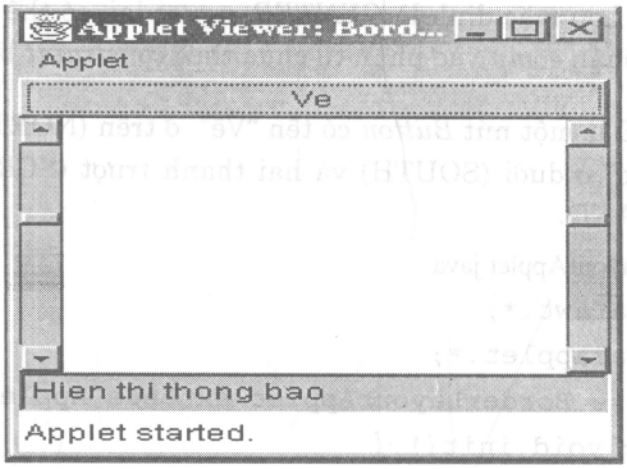
```
// Tệp BorderLayoutApplet.java
import java.awt.*;
import java.applet.*;
public class BorderLayoutApplet extends Applet {
    public void init() {
        // Tạo ra trường text
        TextField msg = new TextField("Hien thi thong bao");
        msg.setEditable(false);
        // Tạo ra nút Button: nutVe
        Button nutVe = new Button("Ve");
        // Tạo ra vungVe để vẽ tranh
        Canvas vungVe = new Canvas();
        vungVe.setSize(150, 150); // Đặt kích thước cho vungVe vẽ tranh
        vungVe.setBackground( Color.white); // Đặt màu nền là trắng
```

```
        Scrollbar sb1=new Scrollbar(Scrollbar.VERTICAL,0,10,-50,100);
        Scrollbar sb2=new Scrollbar(Scrollbar.VERTICAL,0,10,-50,100);
        // Đặt layout theo BorderLayout
        setLayout(new BorderLayout());
        add(nutVe, BorderLayout.NORTH); // Đặt nutVe ở trên (NORTH)
        add(msg, BorderLayout.SOUTH);   // Đặt msg ở dưới (SOUTH)
        add(vungVe,BorderLayout.CENTER); // Đặt vungVe ở giữa (CENTER)
        add(sb1, BorderLayout.WEST);    // Đặt sb1 ở bên trái (WEST)
        add(sb2, BorderLayout.EAST);    // Đặt sb2 ở bên phải (EAST)
    }
}
```

Sau khi dịch được kết quả là *BorderLayoutApplet.class*, chúng ta tạo ra tệp HTML *BorderLayoutApplet.html* có lệnh đơn giản:

```
<applet code ="BorderLayoutApplet.class" width =200 height =60>
</applet>
```

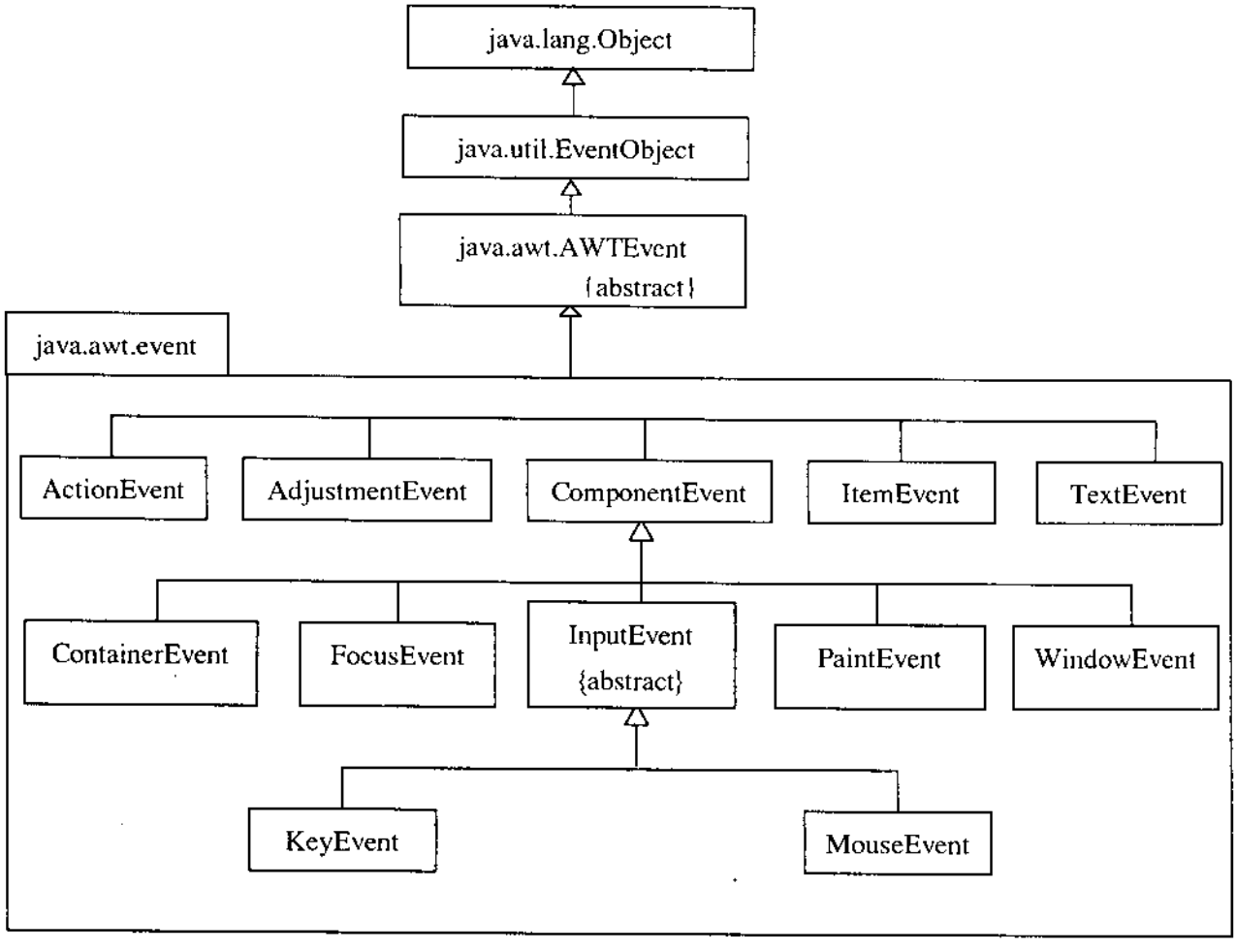
Thực hiện *appletviewer BorderLayoutApplet.html* sẽ cho kết quả:



Hình 7.19. Sắp xếp các thành phần giao diện theo các hướng.

7.5. XỬ LÝ CÁC SỰ KIỆN

Các ứng dụng với GUI thường được hướng dẫn bởi các sự kiện (*event*). Việc nhấn một nút, mở, đóng các Window hay gõ các ký tự từ bàn phím, v.v. đều tạo ra các sự kiện (*event*) và được gửi tới cho chương trình ứng dụng. Trong Java các sự kiện được thể hiện bằng các đối tượng. Lớp cơ sở nhất, lớp cha của tất cả các lớp con của các sự kiện là lớp *java.util.EventObject*. Tập các lớp xử lý các sự kiện tạo thành cấu trúc phân cấp như hình 7.15.



*Hình 7.20. **Các lớp xử lý các sự kiện.***

Các lớp con của *AWTEvent* được chia thành hai nhóm:

1. Các lớp mô tả về ngữ nghĩa của các sự kiện,
2. Các lớp sự kiện ở mức thấp.

7.5.1. CÁC LỚP NGỮ NGHĨA

a/ ActionEvent

Sự kiện này được phát sinh bởi những hoạt thực hiện trên các thành phần của GUI. Các thành phần gây ra các sự kiện hành động bao gồm:

- *Button* - khi một nút button được khích hoạt,
- *List* - khi một mục trong danh sách được kích hoạt đúp,
- *MenuItem* - khi một mục trong thực đơn được chọn,

- *TextField* - khi gõ phím ENTER trong trường văn bản (*text*).

Lớp ActionEvent có các hàm:

String getActionCommand()

Cho biết tên của lệnh tương ứng với sự kiện xảy ra, là tên của nút, mục hay text.

b/ AdjustmentEvent

Thành phần gây ra sự kiện căn chỉnh (adjustment):

- Scrollbar - khi thực hiện một lần căn chỉnh trong thanh trượt *Scrollbar*.

Lớp này có hàm:

int getValue()

Cho lại giá trị hiện thời được xác định bởi lần căn chỉnh sau cùng.

c/ ItemEvent

Các thành phần của GUI gây ra các sự kiện về các mục gồm có:

- *Checkbox* - khi trạng thái của hộp kiểm tra *Checkbox* thay đổi,
- *CheckboxMenuItem* - khi trạng thái của hộp kiểm tra *Checkbox* ứng với mục của thực đơn thay đổi,
- *Choice* - khi một mục trong danh sách các thành phần được chọn hoặc bị loại bỏ,
- *List* - khi một mục trong danh sách được chọn hoặc bị loại bỏ.

Lớp *ItemEvent* có hàm:

Object getItem()

Cho lại đối tượng được chọn hay vừa bị loại bỏ.

d/ TextEvent

Các thành phần của GUI gây ra các sự kiện về *text* gồm có:

- TextArea - khi kết thúc bằng nhấn nút ENTER,
- TextField - khi kết thúc bằng nhấn nút ENTER.

Bảng 7.3. Các hàm xử lý ngữ nghĩa các sự kiện

Kiểu sự kiện	Nguồn gây ra sự kiện	Các hàm đón nhận và di dời các sự kiện	Giao diện Listener tương ứng
AcitionEvent	Button	addComponentlistener	AcitionListener
	List	remove ActiontListener	
	TexField		
AdjustmentEvent	Scrollbar	addAdjustmentListener	AdjustmentListener
		removeAdjustmentListener	
ItemEvent	Choice	addItemListener	ItemListener
	Checkbox	removeItemListener	
	CheckboxMenuItem		
	List		
TextEvent	TextArea	addTexListener	TextListener
	TexField	removeTextListener	

7.5.2. CÁC SỰ KIỆN Ở MỨC THẤP

a/ ComponentEvent

Sự kiện này xuất hiện khi một thành phần bị che giấu, hiển thị hay thay đổi lại kích thước. Lớp *ComponentEvent* có hàm:

Component getComponent()

Cho lại đối tượng tham chiếu kiểu *Component.*

b/ ContainerEvent

Sự kiện này xuất hiện khi một thành phần được bổ sung hay bị loại bỏ khỏi phần tử chứa (*Container*).

c/ FocusEvent

Sự kiện loại này xuất hiện khi một thành phần nhận được một phím từ bàn phím.

d/ KeyEvent

Lớp *KeyEvent* là lớp con của lớp trừu tượng *InputEvent* được sử dụng để xử lý các

sự kiện liên quan đến các phím của bàn phím. Lớp này có các hàm:

int getKeyCode()

Đối với các sự kiện KEY_PRESSED hoặc KEY_RELEASED, hàm này được sử dụng để nhận lại giá trị nguyên tương ứng với mã của phím trên bàn phím.

char getKeyChar()

Đối với các sự kiện KEY_PRESSED, hàm này được sử dụng để nhận lại giá trị nguyên, mã *Unicode* tương ứng với ký tự của bàn phím.

e/ MouseEvent

Lớp *MouseEvent* là lớp con của lớp trừu tượng *InputEvent* được sử dụng để xử lý các tín hiệu của chuột. Lớp này có các hàm:

int getX()

int getY()

Point getPoint()

Các hàm này được sử dụng để nhận lại tọa độ *x, y* của vị trí liên quan đến sự kiện do chuột gây ra.

void translatePoint(int dx, int dy)

Hàm *translate()* được sử dụng để chuyển tọa độ của sự kiện do chuột gây ra đến (*dx, dy*).

int getClickCount()

Hàm *getClickCount()* đếm số lần kích chuột.

f/ PaintEvent

Sự kiện này xuất hiện khi một thành phần gọi hàm *paint()* / *update()* để vẽ.

g/ WindowEvent

Sự kiện loại này xuất hiện khi thao tác với các Window. Lớp này có hàm:

Window getWindow()

Hàm này cho lại đối tượng của lớp Window ứng với sự kiện liên quan đến Window đã xảy ra.

Bảng sau mô tả các hàm xử lý các sự kiện ở mức thấp.

Bảng 7.4. Xử lý các sự kiện ở mức thấp

Kiểu sự kiện	Nguồn gây ra sự kiện	Các hàm đón nhận và di dời các sự kiện	Giao diện Listener tương ứng
ComponentEvent	Component	addComponentListener removeComponentListener	ComponentListener
ContainerEvent	Container	addContainerListener removeContainerListener	ContainerListener
FocusEvent	Component	addFocusListener removeFocusListener	FocusListener
KeyEvent	Component	addkeyListener removeKeyListener	KeyListener
MouseEvent	Component	addMouseListener remoMouseListener addMouseMotionListener remoMouseMotionListener	MouseMotionListener WindowListener
WindowEvent	Window	addWindowListener removeWindowListener	

Ví dụ 7.20. Một ví dụ đơn giản minh họa cách sử dụng nút *Next Shape* và khi nhấn liên tục thì chương trình vẽ *đường thẳng, hình chữ nhật, hình chữ nhật góc tròn, hình oval, hình đa giác và tô màu hình đa giác* sau đó lặp lại.

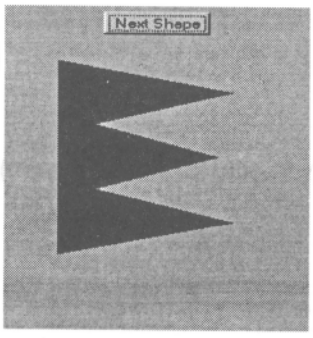
```
// Tệp ShapeApplet.java
import java.awt.*;
import java.applet.*;
public class ShapeApplet extends Applet{
    int shape;
    Button button;
    public void init(){
        shape = 0;
        button = new Button("Next Shape");
        add(button);
        }
    // Nạp chồng hàm paint() để vẽ các hình nguyên thủy
    public void paint(Graphics g){
      int x[] = {35, 150, 60, 140,60, 150, 35};
```

```
        int y[] = {50, 80, 110, 140, 170, 200, 230};
        int numPts = 7;
        switch(shape){
            case 0: g.drawLine(35,50,160,230); break;
            case 1: g.drawRect(35,50,125,180); break;
            case 2: g.drawRoundRect(35,50,125,180, 15, 15);
                    break;
            case 3: g.drawOval(35,50,125,180); break;
            case 4: g.drawArc(35,50,125,180, 90, 180); break;
            case 5: g.drawPolygon(x, y, numPts); break;
            case 6: g.fillPolygon(x, y, numPts); break;
            }
    }
    // Xử lý các sự kiện khi nhấn nút Next Shape
    public boolean action(Event event, Object arg){
        ++shape;
        if(shape == 7)
            shape = 0;
        repaint();
        return true;
    }
}
```

Sau khi dịch được kết quả là *ShapeApplet.class*, chúng ta tạo ra tệp HTML *ShapeApplet.html* có lệnh đơn giản:

```
<applet code ="ShapeApplet.class" width =200 height =60>
</applet>
```

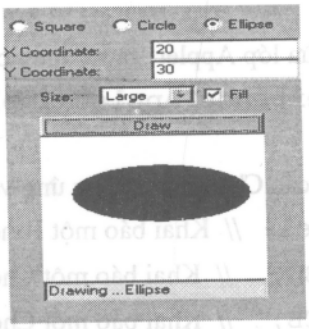
Thực hiện *ShapeApplet.html* trong Web Browser và sau đó nhấn nút *Next Shape* bốn lần chương trình sẽ cho kết quả dạng:



*Hình 7.21. **Nhấn nút Next Shape để vẽ các hình nguyên thủy.***

Tiếp theo chúng ta có thể xây dựng ứng dụng có giao diện đồ họa được bố trí theo thiết kế định trước và xử lý các sự kiện tương ứng với các giao diện đó.

Ví dụ 7.21. Xây dựng ứng dụng có giao diện đồ họa GUI:



Hình 7.22. Giao diện đồ họa của chương trình ứng dụng.

Chương trình được thực hiện như sau:

+ Người sử dụng có thể đánh dấu các nút (Button) có tên: Hình vuông (*Square*), Hình tròn (*Circle*), Hình *Ellipse* để vẽ các hình tương ứng ở vùng vẽ tranh.
+ Tại các dòng: *Tọa độ x (X Coordinate), Toạ độ y (Y Coordinate)* người sử dụng có thể nhập vào tọa độ gốc của hình.
+ Trong thực đơn *Kích thước (Size):* người sử dụng có thể chọn các mục: *nhỏ (Small), trung bình (Medium)* và *lớn (Large)* cho các hình được hiển thị. Kích thước của chúng được cho trước.
+ Sau khi đã xác định đầy đủ các thông tin thì nhấn nút *Hãy vẽ (Ve)* để có hình theo yêu cầu. Nếu nút *Tô màu (fill)* được đánh dấu thì hình vẽ sẽ được tô màu.

Sau đây là chương trình được xây dựng trên cơ sở sử dụng các chức năng của giao diện đồ họa GUI và AWT để thực hiện các yêu cầu nêu trên.

```
// DemoApplet.java
/*DemoApplet: Demonstrating GUI + EVENT HANDLING*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
interface IGeometryConstants { // Định nghĩa Interface gồm các hằng cho trước
    int SQUARE  = 0;
    int CIRCLE  = 1;
    int ELLIPSE = 2;
    String[] shapeNames = {"Square", "Circle", "Ellipse"};
```

```
    int SMALL  = 0;
    int MEDIUM = 1;
    int LARGE  = 2;
    String[] sizeNames = {"Small", "Medium" , "Large"};
}
// Định nghĩa lớp DemoApplet kế thừa lớp Applet và sử dụng các hằng trong giao diện trên
public class DemoApplet extends Applet implements
    IGeometryConstants {
    // Khai báo một Panel để chứa các Checkbox tương ứng với các loại hình shape
    Panel         shapePanel; // Khai báo một Panel
    CheckboxGroup shapeCBG;   // Khai báo một CheckboxGroup
    Checkbox      squareCB;   // Khai báo một Checkbox ứng với hình vuông
    Checkbox      circleCB;   // Khai báo một Checkbox ứng với hình tròn
    Checkbox      ellipseCB;  // Khai báo một Checkbox ứng với hình ellipse
    // Khai báo một Panel để chứa các Label và các TextField
    Panel         xyPanel; // Khai báo một Panel xyPanel
    Label         xLabel;  // Khai báo một Label xLabel
    TextField     xInput;  // Khai báo một TextField xInput
    Label         yLabel;  // Khai báo một Label yLabel
    TextField     yInput;  // Khai báo một TextField yInput
    // Khai báo một Panel để chứa các Label , Choice và Checkbox
    Panel    sizePanel;
    Label    sizeLabel;
    Choice   sizeChoices;
    Checkbox fillCB;
    // Khai báo một Panel để chứa shape, coordinates, size and fill Panel leftPanel
    Panel         leftPanel;
    // Khai báo một Panel để chứa Massage display, draw button and canvas
    Panel         rightPanel;
    Button        drawButton;
    DrawRegion    drawRegion;
    TextField messageDisplay;
    // Định nghĩa lại hàm init()
    public void init() {
        makeShapePanel();
        makeXYPanel();
        makeSizePanel();
        makeLeftPanel();
```

```
        makeRightPanel();
        // Gọi hàm xử lý các sự kiện
        addListeners();
        // Đưa các Panel vào hệ thống
        add(leftPanel);
        add(rightPanel);
  }
// Đưa các thành phần về các shape vào shapePanel
 void makeShapePanel() {
    shapePanel = new Panel();
    shapeCBG   = new CheckboxGroup();
    squareCB   = new Checkbox(shapeNames[SQUARE], shapeCBG, true);
    circleCB   = new Checkbox(shapeNames[CIRCLE], shapeCBG, false);
    ellipseCB  = new Checkbox(shapeNames[ELLIPSE], shapeCBG, false);
    shapePanel.setLayout(new FlowLayout());
    shapePanel.add(squareCB);
    shapePanel.add(circleCB);
    shapePanel.add(ellipseCB);
  }
// Đưa các thành phần về các x,y coordinates vào xyPanel
void makeXYPanel() {
    xyPanel = new Panel();
    xLabel = new Label("X Coordinate:");
    yLabel = new Label("Y Coordinate:");
    xInput = new TextField(5);
    yInput = new TextField(5);
    xyPanel.setLayout(new GridLayout(2,2));
    xyPanel.add(xLabel);
    xyPanel.add(xInput);
    xyPanel.add(yLabel);
    xyPanel.add(yInput);
  }
// Đưa các thành phần về size và fill vào sizePanel
 void makeSizePanel() {
    sizePanel = new Panel();
    sizeLabel = new Label("Size:");
    sizeChoices = new Choice();
```

```
    sizeChoices.add(sizeNames[0]);
    sizeChoices.add(sizeNames[1]);
    sizeChoices.add(sizeNames[2]);
    fillCB = new Checkbox("Fill", false);
    sizePanel.setLayout(new FlowLayout());
    sizePanel.add(sizeLabel);
    sizePanel.add(sizeChoices);
    sizePanel.add(fillCB);
 }
// Đưa các thành phần vào leftPanel
void makeLeftPanel(){
    leftPanel = new Panel();
    leftPanel.setLayout(new BorderLayout());
    leftPanel.add(shapePanel, "North");
    leftPanel.add(xyPanel, "Center");
    leftPanel.add(sizePanel, "South");
    }
// Đưa các thành phần vào rightPanel
void makeRightPanel(){
    rightPanel = new Panel();
    messageDisplay = new TextField("MESSAGE DISPLAY");
    messageDisplay.setEditable(false);
    messageDisplay.setBackground(Color.yellow);

    drawButton = new Button("Draw");
    drawButton.setBackground(Color.lightGray);

    drawRegion = new DrawRegion();
    drawRegion.setSize(150, 150);
    drawRegion.setBackground(Color.white);
    rightPanel.setLayout(new BorderLayout());
    rightPanel.add(drawButton, BorderLayout.NORTH);
    rightPanel.add(messageDisplay, BorderLayout.SOUTH);
    rightPanel.add(drawRegion, BorderLayout.CENTER);
    }
// Xử lý các sự kiện tương ứng với các việc thực hiện trên các thành phần đồ họa
void addListeners(){
```

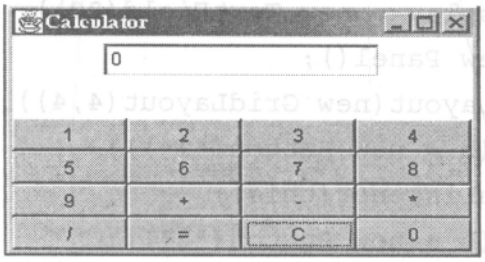
```
drawButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt){
        int shape, xCoord, yCoord, width;
        messageDisplay.setText("");
        if(squareCB.getState())
          shape = SQUARE;
        else if (circleCB.getState())
          shape = CIRCLE;
        else if (ellipseCB.getState())
          shape = ELLIPSE;
        else {
          messageDisplay.setText("Unknow shape");
          return;
            }
        try{
          xCoord = Integer.parseInt(xInput.getText());
          yCoord = Integer.parseInt(yInput.getText());
            }catch(NumberFormatException e){
          messageDisplay.setText("Illegal coordinate");
             return;
            }
        switch(sizeChoices.getSelectedIndex()){
            case SMALL: width = 30; break;
            case MEDIUM: width = 60; break;
            case LARGE: width = 120; break;
            default: messageDisplay.setText("Unknow size");
                return;
            }
        messageDisplay.setText("Drawing..."+shapeNames[shape]);

drawRegion.doDraw(shape,xCoord,yCoord,
            fillCB.getState(),width);
    }
});
xInput.addTextListener(new TextListener(){
    public void textValueChanged(TextEvent evt){
        checkTF(xInput);
```

```
        }
    });
    yInput.addTextListener(new TextListener(){
            public void textValueChanged(TextEvent evt){
                checkTF(yInput);
            }
    });
}// Kết thúc hàm addListener()
    void checkTF(TextField tf){
        messageDisplay.setText("");
        try{
            Integer.parseInt(tf.getText());
            }catch(NumberFormatException e){
            messageDisplay.setText("Illegal coordinate");
            }
        }
    }
// Định nghĩa lớp DrawRegion mở rộng lớp Canvas và sử dụng các hàng ở giao diện
class DrawRegion extends Canvas implements IGeometryConstants{
    private int shape,xCoord, yCoord;
    private boolean fillFlag;
    private int width;
    public DrawRegion(){
        setSize(150, 150);
        setBackground(Color.white);
        }
// Định nghĩa hàm doDraw() để vẽ
    public void doDraw(int h, int x, int y, boolean f, int w){
        this.shape = h;
        this.xCoord = x;
        this.yCoord = y;
        this.fillFlag = f;
        this.width = w;
        repaint();
        }
    public void paint(Graphics g){
        switch(shape){
```

```
            case SQUARE:
              if(fillFlag) g.fillRect(xCoord, yCoord, width,width);
              else g.drawRect(xCoord, yCoord, width,width);
              break;
            case CIRCLE:
              if(fillFlag) g.fillOval(xCoord, yCoord, width,width);
              else g.drawOval(xCoord, yCoord, width,width);
              break;
            case ELLIPSE:
              if(fillFlag) g.fillOval(xCoord,yCoord,width,width/2);
              else g.drawOval(xCoord, yCoord, width,width/2);
              break;
            }
        }
    }
```

Ví dụ 7.22. Viết chương trình mô phỏng máy tính số học (*Calculator*) thực hiện được các phép: +, -, *, / có giao diện như sau:



Hình 7.23. Mô phỏng Calculator.

Để thực hiện được các phép tính +, -, *, / số học, người sử dụng có thể thực hiện:

+ Nhấn chuột ở các ô chữ số để nhập vào đối số thứ nhất,
+ Nhấn chuột ở các ô +, -, *, / tương ứng với phép các phép toán muốn thực hiện,
+ Nhấn chuột ở các ô chữ số để nhập vào đối số thứ hai,
+ Nhấn chuột ở có dấu = để biết kết quả. Nhấn vào ô có chữ '*C*" để xóa kết quả khỏi bộ nhớ.

Chương trình này được viết trên cơ sở tạo ra giao diện được bố trí như hình 7.23 và xử lý các sự kiện nhấn chuột vào các nút tương ứng.

```
// Tệp Calculator.java
  import java.io.*;
```

```
import java.awt.*;
import java.awt.event.*;
class CalculatorFrame extends Frame implements
  ActionListener {
  Calculator engine; //class Calculator được định nghĩa sau
    TextField display;
    // Tạo ra đối tượng của WindowAdapter xử lý các tín Window
    WindowListener listener = new WindowAdapter(){
        public void WindowClosing(WindowEvent e){
            System.exit(0);// Thoát ra khỏi chương trình
      }
  };
  CalculatorFrame(Calculator e){
      super("Calculator");
      Panel top, bottom;
      Button b;
      engine = e;
      top = new Panel();//Vùng để hiển thị số liệu
      top.add(display = new TextField(20));
      bottom = new Panel();
      bottom.setLayout(new GridLayout(4,4));
      bottom.add(b = new Button("1"));
      b.addActionListener(this);
      bottom.add(b = new Button("2"));
      b.addActionListener(this);
      bottom.add(b = new Button("3"));
      b.addActionListener(this);
      bottom.add(b = new Button("4"));
      b.addActionListener(this);
      bottom.add(b = new Button("5"));
      b.addActionListener(this);
      bottom.add(b = new Button("6"));
      b.addActionListener(this);
      bottom.add(b = new Button("7"));
      b.addActionListener(this);
      bottom.add(b = new Button("8"));
      b.addActionListener(this);
```

```
        bottom.add(b = new Button("9"));
        b.addActionListener(this);
        bottom.add(b = new Button("+"));
        b.addActionListener(this);
        bottom.add(b = new Button("-"));
        b.addActionListener(this);
        bottom.add(b = new Button("*"));
        b.addActionListener(this);
        bottom.add(b = new Button("/"));
        b.addActionListener(this);
        bottom.add(b = new Button("="));
        b.addActionListener(this);
        bottom.add(b= new Button("C"));
        b.addActionListener(this);
        bottom.add(b= new Button("0"));
        b.addActionListener(this);
        // Đặt layout cho chương trình
        setLayout (new BorderLayout());
        add("North", top );
        add("South", bottom);
        addWindowListener(listener);
        setSize( 180, 160);
        show();
    }
    // Xử lý các sự kiện khi nhấn chuột vào các nút
    public void actionPerformed(ActionEvent e) {
        char c = e.getActionCommand().charAt(0);
        if (c=='+')engine.add();
        else if (c=='-') engine.subtract();
        else if (c=='*') engine.multiply();
        else if (c=='/') engine.devide();
        else if (c>='0'&& c<='9') engine.digit(c - '0');
        else if (c=='=') engine.compute();
        else if (c=='C') engine.clear();
        display.setText(new
            Integer(engine.display()).toString());
}
```

```
public static void main(String arg[]) {
    Calculator e = new Calculator();
    new CalculatorFrame(e);
  }
}
// Định nghĩa lớp Calculator
class Calculator{
    int value;
    int keep;
    char toDo;
    void binaryOp(char op){
        keep = value;
        value = 0;
        toDo = op;
    }
    void add(){binaryOp('+');}      // Thực hiện phép cộng
    void subtract(){binaryOp('-');} // Thực hiện phép trừ
    void multiply(){binaryOp('*');} // Thực hiện phép nhân
    void devide(){binaryOp('/');}   // Thực hiện phép chia
    void compute(){
        if(toDo =='+')
            value += keep;
        else if(toDo =='-')
            value -= keep;
        else if(toDo =='*')
            value *= keep;
        else if(toDo =='/')
            value = keep / value;
        keep = 0;
    }
    // Hàm tính giá trị của số nguyên khi nhập các chữ số vào

    void digit(int x){
        value = value * 10 + x;
    }
    // Hàm xóa bộ nhớ của Calculator
    void clear(){
```

```
        value= 0;
        keep = 0;
    }
    int display(){
        return (value);
    }
    //Constructor
    Calculator(){clear();}
}
```

BÀI TẬP

7.1. Hãy mô phỏng một thùng thư tự động. Thùng thư này có thể chứa được một số lượng bưu phẩm có giới hạn. Có hai loại bưu phẩm:

- Thư: giá cước vận chuyển là 2000 đ, nếu cần chuyển nhanh thì phải trả thêm 3000 đ.
- Bưu kiện: giá cước vận chuyển là 5000 đ, không phân biệt độ lớn hay trọng lượng.

Hãy viết một chương trình cho phép tính tự động giá tiền vận chuyển của tất cả các bưu phẩm chứa trong thùng thư. Chương trình phải có các chức năng sau:

1. Khởi tạo một thùng thư có chứa một số bưu phẩm (gồm cả hai loại như trên),
2. Bỏ bưu phẩm vào thùng thư,
3. In ra toàn bộ các bưu phẩm có trong thùng thư với các thông tin: loại bưu phẩm (nếu là thư thì phân biệt loại gửi thường và loại gửi nhanh), giá cước vận chuyển.

(Đề thi cao học của IFI (Viện Tin học Pháp ngữ) 2002)

7.2. Một trung tâm Tin học cần quản lý các phương tiện giao thông gồm các loại ô tô và xe máy. Mỗi loại phương tiện (ô tô và xe máy) cần quản lý: *màu xe, giá thành và hãng sản xuất*. Ngoài ra, đối với ô tô thì phải biết thêm: *số ghế, loại máy (động cơ)*; còn xe máy thì phải biết thêm: *công suất*.

1. Xây dựng các lớp cho các phương tiện, trong đó lớp ôtô và xe máy là được kế thừa từ lớp phương tiện giao thông nói chung,
2. Viết chương trình thực hiện theo các menu sau:

1/ Đăng ký loại phương tiện mới

2/ Tìm theo hãng sản xuất

3/ Tìm theo màu

4/ Thoát khỏi hệ thống

Người sử dụng có thể chọn một trong các mục nêu trên. Nếu chọn 1/ thì hệ thống sẽ hỏi là nhập mới ô tô hay xe máy, chọn 2/ thì hệ thống sẽ hỏi tên hàng sản xuất cần tìm, v.v.

7.3. Phòng quản lý sinh viên của Đại học QG cần quản lý các các thông về sinh viên của Trường. Mỗi sinh viên cần quản lý: *họ và tên, số báo danh, ngày tháng năm sinh, địa chỉ, ngành học và khoá học.*

Viết chương trình thực hiện độc lập có các menu sau:

1/ Nhập thông tin của sinh viên mới

2/ Tìm theo số báo danh

3/ Hiển thị tất cả danh sách các sinh viên

4/ Thoát khỏi hệ thống

Người sử dụng có thể chọn một trong các mục nêu trên. Nếu chọn 1/ thì người sử dụng có thể nhập vào một sinh viên mới, chọn 2/ để tìm theo số báo danh v.v.

7.4. Viết chương trình ứng dụng *applet* thực hiện các nội dung của bài 7.3.

7.5. Viết chương trình có giao diện đồ họa gồm:

+ Có hai nút *Checkbox: Tìm kiếm* và nút *Vẽ* được đặt ở hàng trên,
+ Có *Vùng thông báo* đặt ở dưới để hiển thị các thông tin tìm thấy,
+ Có trường nhập dữ liệu để nhập tên của hình: *Hình tròn, vuông, chữ nhật, oval, đa giác,*
+ Có *Vùng vẽ* để hiển thị các hình được chọn, khi nhấn nút *Vẽ*.

CHƯƠNG VIII

CÁC LUỒNG VÀO/RA VÀ CÁC TỆP DỮ LIỆU

8.1. CÁC LUỒNG VÀO/RA

Gói *java.io* cung cấp các lớp thư viện để xử lý vào/ra *(I/O)* chuẩn và đọc/ghi lên tệp. Java sử dụng các luồng *Stream* như là cơ chế chung để xử lý *I/O*. Có hai loại *Stream: Stream* được sử dụng để xử lý các *byte* và *Stream* để xử lý các ký tự. Lớp *InputStream* được sử dụng để đọc, nhập dữ liệu vào chương trình và *OutputStream* được sử dụng để ghi dữ liệu tuần tự lên tệp.

Gói *java.io* còn hỗ trợ để truy nhập ngẫu nhiên vào tệp và có các giao diện (*interface*) để tương tác được với hệ thống *tệp* của các máy chủ.

8.2. LỚP FILE

Lớp *File* cung cấp giao diện chung để xử lý hệ thống tệp độc lập với môi trường của các máy tính. Một ứng dụng có thể sử dụng các chức năng chính của *File* để xử lý tệp hoặc các thư mục (*directory*) trong hệ thống tệp. Để xử lý các nội dung của các tệp thì sử dụng các lớp *FileInputStream, FileOutputStream* và *RandomAccessFile.*

Lớp *File* định nghĩa các thuộc tính phụ thuộc vào môi trường (*platform*) được sử dụng để xử lý tệp và tên gọi các đường dẫn trong các thư mục một cách độc lập với môi trường.

public static final char separatorChar
public static final String separator

> Định nghĩa các ký hiệu hoặc các xâu sử dụng để ngăn cách các thành phần trong tên của đường dẫn. Ký hiệu '/' để ngăn cách cho Unix, '\' được sử dụng để ngăn cách các mục của đường dẫn trong Window.

Ví dụ: *C:\book\chuong1* là tên đường dẫn trong Window.

public static final char pathSeparatorChar
public static final String pathSeparator

Định nghĩa các ký hiệu hoặc các xâu sử dụng để ngăn cách các *tệp* hoặc tên thư mục trong danh sách của các đường dẫn. Ký hiệu ngăn cách ':' cho Unix, ';' được sử dụng để phân cách các đường dẫn trong Window.

Ví dụ: *C:\book; C:\Java; D:\Users\Lan; A:\Text* là danh sách các đường dẫn trong Window.

File(String pathName)

Gán đường dẫn có tên *pathName* cho đối tượng của lớp *File*. *pathName* có thể là *đường dẫn tuyệt đối* (có đủ cả tên ổ đĩa, tên của tất cả các mục lần lượt theo cây thư mục) hoặc *đường dẫn tương đối* (bắt đầu từ thư mục hiện thời).

Ví dụ:

```
File ch1 = new File(File.separator + "book" + File.separator + "chuong1");
```

File(File direct, String filename)

Gán *tệp* có tên *filename* ở thư mục *direct* cho đối tượng của lớp *File*. Nếu *direct* là *null* thì *filename* được xử lý ở thư mục hiện thời, ngược lại tên đường dẫn sẽ là đối tượng *direct* ghép với *separator* và *filename*.

Ví dụ:

```
File  mucNhap = new File("book" + File.separator + "duThao");
File ch2 = new File(mucNhap, "chuong1");
```

Lớp *File* có thể sử dụng để truy vấn vào các hệ thống tệp để biết được các thông tin về *tệp* và các thư mục.

Lớp *File* cung cấp các hàm để nhận được các biểu diễn phụ thuộc vào môi trường của đường dẫn và các thành phần của đường dẫn.

String getName()

Hàm *getName()* cho lại tên của *tệp* trong thư mục chỉ định. Ví dụ, tên của "C:\java\bin\javac" là "javac".

String getPath()

Hàm *getPath()* cho lại tên của đường dẫn (tuyệt đối hoặc tương đối) chứa tệp chỉ định.

String getAbsolutePath()

Hàm *getAbsolutePath()* cho lại tên của đường dẫn tuyệt đối chứa *tệp* chỉ định.

long lastModified()

Hàm này cho lại thời gian dạng số *long* của lần sửa đổi *tệp* lần cuối cùng.

long length()

Hàm này cho lại kích thước của *tệp* tính theo byte.

boolean equals(Object obj)

Hàm này so sánh tên các đường dẫn của các đối tượng *tệp*, cho kết quả *true* nếu chúng đồng nhất với nhau.

boolean exists()
boolean isFile()

Hai hàm này cho kết quả *true* nếu tệp đó tồn tại trên đường dẫn hiện thời.

boolean isDirectory()

Hàm này cho kết quả *true* nếu thư mục đó tồn tại trên ổ đĩa hiện thời.

boolean createNewFile() throws IOException

Một *tệp* mới là rỗng được tạo ra ở thư mục hiện thời chỉ khi *tệp* này chưa có.

boolean mkdir()
boolean mkdirs()

Tạo ra các đường dẫn được xác định trong đối tượng của *File*.

boolean renameTo(File dest)

Hàm này đổi tên *tệp* hiện thời thành *dest*.

String[] list()

Hàm này hiển thị danh sách các *tệp* ở thư mục hiện thời.

boolean delete()

Hàm này xóa *tệp* hiện thời.

Ví dụ 8.1. Viết chương trình để đọc và hiển thị các *tệp* trong một thư mục.

```
import java.io.*;
public class DirectoryLister{
    public static void main(String args[]){
```

```
    File entry;
    if (args.length==0){
    System.err.println("Hay cho biet ten duong dan?");
    return;
    }
    entry = new File(args[0]); // Tạo ra đối tượng của File lấy tên từ agrs[0]
    listDirectory(entry); // Hiển thị các mục trong danh mục chỉ định
}
public static void listDirectory(File entry) {
    try {
    if(!entry.exists()) {
            System.out.println(entry.getName() +"not found.");
            return;
        }
        if (entry.isFile()){
            // Nếu đối số của chương trình là một tệp thì hiển thị tệp đó
            System.out.println(entry.getCanonicalPath());
        } else if(entry.isDirectory()){
            // Nếu đối số của chương trình là một danh mục thì đọc ra (list())
            String[] fileName = entry.list();
            if (fileName ==null)return;
            for (int i = 0; i<fileName.length; i++){
                // Tạo ra đối tượng của File để xử lý từng mục
                File item = new File(entry.getPath(), fileName[i]);
                // Gọi đệ qui hàm listDirectory()để hiển thị từng tệp.
                listDirectory(item);
            }
        }
    }catch(IOException e){System.out.println("Error:" +e);
    }
  }
}
```

Dịch xong có thể chạy trong môi trường DOS:

java DirectoryLister c:\users\lan

Tất cả các tệp trong danh mục *c:\users\lan* sẽ được hiện lên.

8.2. CÁC LUỒNG VÀO/RA XỬ LÝ THEO *BYTE*

Các lớp trừu tượng *InputStream* và *OutputStream* là hai lớp gốc của hai nhánh trong cây phân cấp có thứ bậc theo quan hệ kế thừa (hình 8.1), làm cơ sở để xây dựng những lớp con cháu làm nhiệm vụ đọc và ghi từng *byte* lên tệp. Các lớp con cài đặt các hàm hoặc được viết đè để:

1. Đọc từng *byte* từ tệp:

```
int read() throws IOException
int read(byte[] b) throws IOException
int read(byte[] b, int off, int len) throws IOException
```

Đọc từng *byte*, nhưng cho lại giá trị kiểu *int*. Các *byte* đọc được sẽ được gán vào 8 *bit* thấp của kiểu *int*, những *bit* còn lại được gán trị 0. Các hàm *read()* sẽ nhận -1, khi đọc đến cuối tệp. Hàm `read(byte[] b)` đọc các byte vào mảng *b*, hàm `read(byte[] b, int off, int len)` đọc *len byte* vào mảng *b* kể từ vị trí tương đối *off*.

2. Ghi từng byte vào tệp:

```
void write(int b) throws IOException
void write(byte[] b) throws IOException
void write(byte[] b, int off, int len) throws IOException
```

Hàm đầu tiên ghi giá trị kiểu *int* và những giá trị kiểu *int* đó được làm tròn, lấy 8 bit thấp (1 byte) để ghi lên tệp. Hàm `write(byte[]b)` ghi mảng các byte *b*, hàm `write(byte[] b, int off, int len)` ghi mảng *b* gồm *len byte* kể từ vị trí *off*.

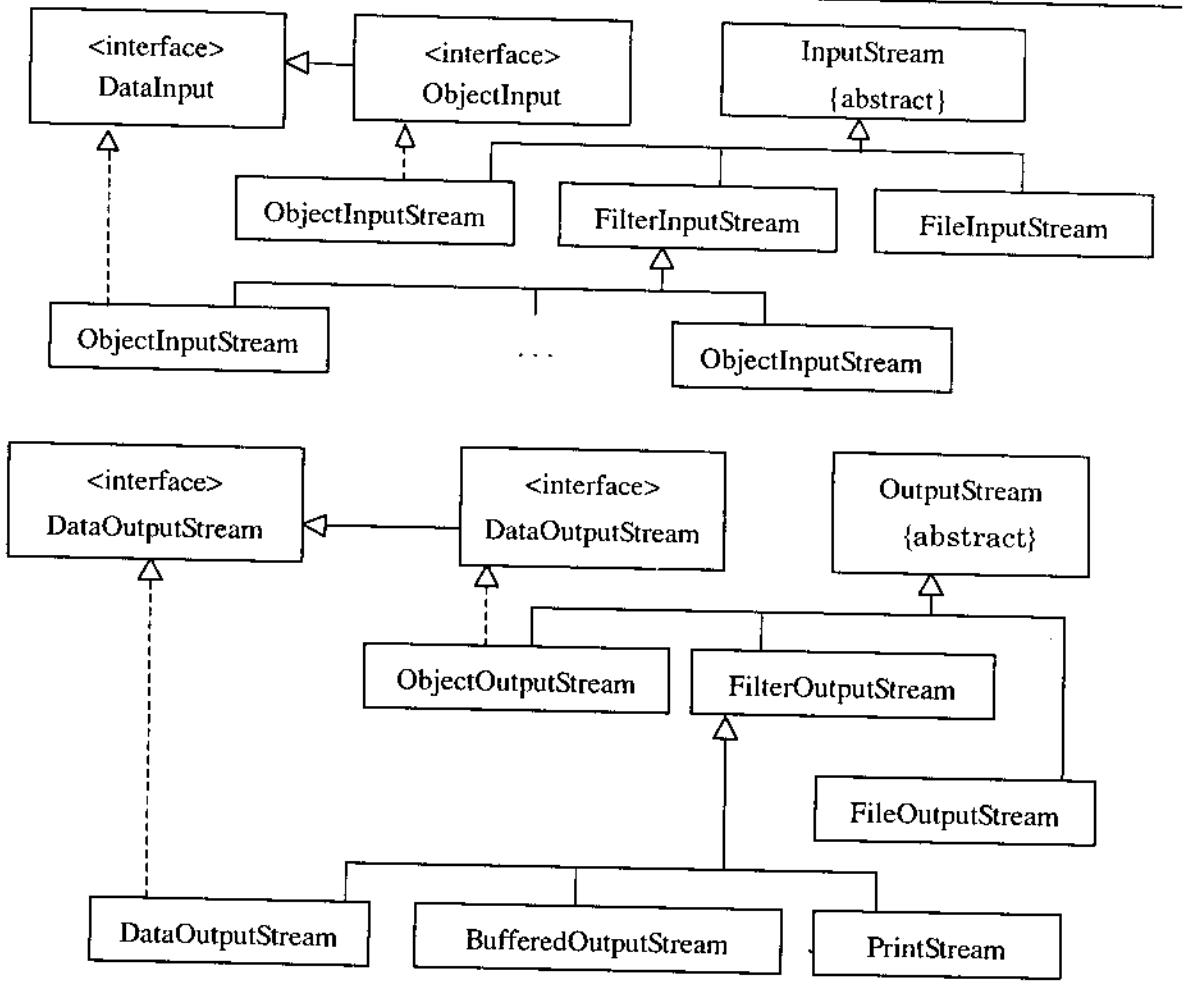
3. Đóng tệp:

```
void close() throws IOException
void flush() throws IOException
```

Hàm *close()* được sử dụng để đóng tệp. Khi đóng tệp thì các dữ liệu ở *buffer* tương ứng sẽ được tự động đưa vào tệp, hoặc có thể gọi cụ thể hàm *flush()* để thực hiện công việc đó.

Lưu ý: Khi xử lý với tệp xong thì *phải đóng tệp* để khỏi mất dữ liệu và đảm bảo cho hệ thống hoạt động bình thường.

Bảng 8.1 mô tả các lớp đối tượng xử lý các chức năng đọc dữ liệu từ tệp vào chương trình ứng dụng và bảng 8.2 mô tả các lớp ghi dữ liệu lên tệp.



Hình 8.1. ***Các lớp xử lý tệp theo các*** **byte.**

Bảng 8.1. Các lớp đọc dữ liệu

Tên các lớp đọc dữ liệu theo byte	Nội dung (dữ liệu được đọc từ mảng các byte phải được xác định)
FileInputStream	Dữ liệu được đọc dưới dạng *byte* từ đối tượng *File* và tên tệp kiểu *String*.
FilterInputStream	Lớp cơ sở của tất cả các lớp làm nhiệm vụ như là các bộ lọc (*filter*) của các luồng *input*.
BufferedInputStream	Lớp được sử dụng để chứa các byte đọc được và đưa vào vùng đệm (*buffer*).
DataInputStream	Lớp được sử dụng để đọc các giá trị kiểu nguyên thủy được biểu diễn dưới dạng nhị phân.
ObjectInputStream	Lớp được sử dụng để đọc các đối tượng của Java và các giá trị kiểu nguyên thủy được biểu diễn dưới dạng nhị phân.
SequenceInputStream	Cho phép đọc tuần tự từ hai hay nhiều hơn các luồng vào.

Bảng 8.2. Các lớp ghi dữ liệu lên tệp

Tên các lớp ghi dữ liệu theo byte	Nội dung (dữ liệu được ghi vào mảng các byte phải được xác định)
FileOutputStream	Dữ liệu từ đối tượng *File* và tên tệp dạng *String* được ghi vào dưới dạng các *byte* .
FilterOutputStream	Lớp cơ sở của tất cả các lớp làm nhiệm vụ như là các bộ lọc (*filter*) của các luồng *output*.
BufferedOutputStream	Lớp được sử dụng để ghi các byte từ vùng đệm (*buffer*) vào tệp.
DataOutputStream	Lớp được sử dụng để ghi các giá trị kiểu nguyên thủy được biểu diễn dưới dạng nhị phân.
ObjectOutputStream	Lớp được sử dụng để ghi các đối tượng của Java và các giá trị kiểu nguyên thủy được biểu diễn dưới dạng nhị phân.
SequenceOutputStream	Cho phép ghi tuần tự từ hai hay nhiều hơn các luồng vào.

Các lớp xử lý tệp

Các lớp *FileInputStream, FileOutputStream* định nghĩa các luồng của các *byte* vào/ra cho các tệp. Chúng có các toán tử tạo lập như sau:

FileInputStream(String name) throws FileNotFoundException

FileInputStream(File file) throws FileNotFoundException

FileInputStream(FileDescriptor fdObj)

Tạo ra một tệp có tên *name*, một đối tượng tệp *file* hoặc đối tượng *fdObj* của *FileDescriptor* đọc từng byte. Nếu tệp đó không có trên đĩa thì cho qua ngoại lệ *FileNotFoundException* để sau đó có thể đón nhận và xử lý theo từng ngữ cảnh ứng dụng.

FileOutputStream(String name) throws FileNotFoundException

FileOutputStream(String name, boolean append) throws FileNotFoundException

FileOutputStream(File file) throws IOException

FileOutputStream(FileDescriptor fdObj)

Tạo ra một tệp có tên *name*, một đối tượng tệp *file* hoặc đối tượng *fdObj* của *FileDescriptor* ghi từng byte. Nếu tệp đó không có trên đĩa thì tạo ra tệp mới ở thư mục hiện thời.

Ví dụ 8.2. Viết một chương trình *CopyFile* để sao một tệp sang tệp khác được thực hiện theo lệnh dạng:

```
    java CopyFile <Tep_cu> <Tep_moi>
/* copy a File
Thực hiện theo câu lệnh: java CopyFile <fromfile> <to-file>
*/
import java.io.*;
class CopyFile{
    public static void main(String args[]) {
    FileInputStream fromFile;
    FileOutputStream toFile;
    // Lấy các đối số từ chương trình main() và gán cho các tệp
    try {
        fromFile = new FileInputStream(args[0]); // (1)
        toFile = new FileOutputStream(args[1]);    // (2)
        } catch(FileNotFoundException e){
          System.err.println("File could not be copied:"+e);
                return;
        } catch(ArrayIndexOutOfBoundsException e){
          System.err.println("Usage:CopyFile <fromFile> <toFile>");
            return;
        }
        // Sao từng byte từ fromFile sang toFile
        try{
            int i = fromFile.read();  // Đọc một byte từ fromFile
            while (i != -1) {       // Kiểm tra xem đã cuối tệp chưa?
            toFile.write(i);        // Ghi một byte vào toFile
            i = fromFile.read();  // Đọc một byte từ fromFile
        }
    } catch (IOException e){        // Xử lý ngoại lệ khi gặp
        System.err.println("Error reading/wring...");

    }
    // Đóng tệp
    try {
    fromFile.close();
    toFile.close();
    } catch (IOException e) {// Xử lý ngoại lệ khi không đóng được tệp
    System.err.println("Error closing file.");
```

```
        }
    }
}
```

Câu lệnh (1) tạo ra đối tượng là biến tệp: *fromFile,* có tên lấy từ đối số agrs[0], lệnh (2) tạo ra đối tượng là biến tệp: *toFile,* có tên lấy từ đối số agrs[1].

Các lớp đọc, ghi dữ liệu nguyên thủy: DataInputStream, DataOutputStream

Gói *java.io* cung cấp hai giao diện *DataInput* và *DataOutput* để cài đặt các hàm đọc/ghi dạng nhị phân của các giá trị nguyên thủy (*boolean, char, byte, short, int, long, float, double*). Các hàm đọc được bắt đầu bằng *read*, các hàm ghi bắt đầu bằng *write* và sau đó tên của các kiểu nguyên thủy với chữ cái đầu tiên được viết hoa. Hai hàm *readUTF() và writeUTF()* được sử dụng để đọc, ghi các xâu ký tự, trong đó UTF là chuẩn để xử lý Unicode. Các ký tự Unicode thường sử dụng mã UTF8 là chuẩn *Universal Character Set Transformation Format)*, cùng loại với ASCII.

- Để ghi dạng nhị phân của các giá trị nguyên thủy lên tệp chúng ta thực hiện như sau:

  1. Tạo ra một đối tượng *outFile* của *FileOutputStream:*

  FileOutputStream outFile = new FileOutputStream("primitives.data");

  2. Tạo ra một đối tượng của DataOutputStream để ghép với outFile:

  DataOutputStream outStream = new DataOutputStream(outFile);

  3. Ghi các giá trị nguyên thủy vào outStream:

```
outStream.writeBoolean(true);          // Ghi giá trị: true vào outStream
outStream.writeChar('A');              // Ghi ký tự 'A' vào outStream
outStream.writeByte(Byte.MAX_VALUE);   // Ghi Byte.MAX_VALUE vào outStream
outStream.writeShort(Short.MIN_VALUE); // Ghi Short.MIN_VALUE vào outStream
outStream.writeInt(41);                // Ghi giá trị 41 vào outStream
outStream.writeLong(Long.MAX_VALUE);   // Ghi Long.MAX_VALUE vào outStream
outStream.writeFloat(Float.MIN_VALUE); // Ghi Float.MIN_VALUE vào outStream
outStream.writeDouble(Math.PI);        // Ghi Math.PI vào outStream
```

  4. Đóng tệp

  outStream.close();

- Để đọc dạng nhị phân của các giá trị nguyên thủy từ tệp chúng ta thực hiện như sau:

1. Tạo ra một đối tượng *inFile* của *FileInputStream:*

   FileInputStream inFile = new FileInputStream("primitives.data");

2. Tạo ra một đối tượng của DataInputStream để ghép với inFile:

   `DataInputStream inStream = new DataInputStream(outFile);`

3. Ghi các giá trị nguyên thủy vào *inStream:*

   booleanb = inStream.readBoolean();
   char c = inStream.readChar();
   byte bt = inStream.readByte();
   short s = inStream.readShort();
   int i = inStream.readInt();
   long l = inStream.readLong();
   float f = inStream.readFloat(Float.MIN_VALUE);
   double d = inStream.readDouble(Math.PI);

4. Đóng tệp

   inStream.close();

Đọc, ghi dữ liệu thông qua buffer: BufferedInputStream, BufferedOutputStream

Hai lớp *BufferedInputStream, BufferedOutputStream* được sử dụng để đọc, ghi dữ liệu theo từng khối các byte, chứ không phải từng *byte* như hai lớp *DataInputStream, DataOutputStream*. Tất nhiên khi đọc, ghi thông qua *buffer* thì tốc độ thực hiện sẽ nhanh hơn.

- Để ghi theo buffer các byte chúng ta phải tạo ra biến tệp outBuffer:

  FileOutputStream outFile = new FileOutputStrâm("primitives.data");
  BufferedOutputStream outBuffer = new BufferedOutputStream(outFile);
  DataOutputStream outStream = new DataOutputStream(outBuffer);

  Sau đó sử dụng các hàm tương ứng như trên để ghi (*writeX()*) các giá trị kiểu nguyên thủy vào *outStream.*

- Để đọc theo *buffer* các *byte* chúng ta phải tạo ra biến tệp *inbuffer:*

  FileInputStream inFile = new FileInputStream("primitives.data");
  BufferedInputStream inBuffer = new BufferedInputStream(outFile);
  DataInputStream inStream = new DataInputStream(outBuffer);

  Sau đó sử dụng các hàm tương ứng như trên để đọc (*readX()*) các giá trị kiểu nguyên thủy từ *inStream.*

8.3. ĐỌC, GHI CÁC KÝ TỰ: READER, WRITER

Các chương trình của Java sử dụng mã Unicode để biểu diễn cho tập các ký tự. Nhưng trong các chương trình ứng dụng của bạn chạy trên những máy khác nhau, có thể sử dụng những mã khác nhau, ví dụ như mã ASCII, chẳng hạn. Phần lớn những tập mã đó chỉ là tập con của Unicode.

Java cung cấp các lớp trừu tượng *Reader, Writer* làm cơ sở để mở rộng thành các lớp con làm nhiệm vụ đọc, ghi các ký tự của Unicode.

Lớp *Reader* sử dụng những hàm sau để đọc các ký tự Unicode:

int read() throws IOException
int read(char cbuf[]) throws IOException
int read(char cbuf[], int off, int len) throws IOException

Đọc các ký tự Unicode có mã trong khoảng 0 - 65535 (0x0000 - 0xFFFF). Các hàm này nhận trị -1 khi ở cuối tệp.

long skip(long n) throws IOException

Chuyển đầu đọc đi *n* ký tự.

Lớp *Writer* sử dụng những hàm sau để ghi các ký tự Unicode:

void write(int c) throws IOException

Ghi lên tệp 16 bit thấp của giá trị c kiểu *int (32 bit).*

void write(char cbuf[]) throws IOException
void write(char cbuf[], int off, int len) throws IOException
void write(String str, int off, int len) throws IOException
void write(String str) throws IOException

Ghi các ký tự từ mảng các ký tự hay xâu ký tự lên tệp.

Ghi các tệp văn bản

Khi viết văn bản (*text*) vào tệp mà sử dụng các mã ký tự mặc định, chúng ta có thể sử dụng một trong các cách sau:

- Khởi tạo đối tượng của *PrintWriter* dựa trên *OutputStreamWriter* và được kết nối với *FileOutputStream:*

  1. Tạo ra đối tượng của *FileOutputStream* có tên tệp "*info.txt*":

FileOutputStream outFile = new FileOutputStream("info.txt");

2. Tạo ra đối tượng của OutputStreamWriter để liên kết với outFile:

OutputStreamWriter outStream = new OutputStreamWriter(outFile);

3. Tạo ra đối tượng PrintWriter để liên kết với outStream:

PrintWriter printWriter = new PrintWriter(outStream, true).

- Khởi tạo PrintWriter dựa trên FileOutputStream:

1. Tạo ra đối tượng của FileOutputStream:

FileOutputStream outFile = new FileOutputStream("info.txt");

2. Tạo ra đối tượng của PrintWriter để kết nối với outFile:

PrintWriter printWriter = new PrintWriter(outFile, true).

- Khởi tạo đối tượng của PrintWriter dựa trên lớp con của *OutputStreamWriter:*

1. Tạo ra đối tượng của FileWriter, lớp con của OutputStreamWriter:

FileWriter fileWriter = new FileWriter("info.txt");

2. Tạo ra đối tượng PrintWriter để liên kết với fileWriter:

PrintWriter printWriter = new PrintWriter(fileWriter, true).

Khi đọc văn bản (*text*) từ tệp mà sử dụng các mã ký tự mặc định, chúng ta có thể sử dụng một trong các cách sau:

- Khởi tạo đối tượng của *InputStreamReader* để kết nối với *FileInputStream:*

1. Tạo ra đối tượng của *FileInputStream* có tên tệp "*info.txt*":

FileOutputStream inFile = new FileOutputStream("info.txt");

2. Tạo ra đối tượng của *InputStreamReader* để liên kết với *outFile:*

InputStreamReader reader = new InputStreamReader (inFile);

- Khởi tạo đối tượng của *FileReader* dựa trên lớp con của *InputStreamReader:*

Tạo ra đối tượng của *FileReader, lớp con của OutputStreamWriter:*

FileReader fileReader = new FileReader("info.txt");

Ví dụ 8.3. Sử dụng các lớp Reader và Writer để đọc, ghi các ký tự theo các mã chuẩn.

```
import java.io.*;
public class CharEncodingDemo {
  public static void main(String args[])
            throws IOException, NumberFormatException {
 // Ghi lên tệp các ký tự theo mã chuẩn ISO8859_1.
```

```
FileOutputStream outputFile = new FileOutputStream("info.txt");
OutputStreamWriter writer  = new
  OutputStreamWriter(outputFile,"8859_1");
BufferedWriter bufferedWriter1  = new BufferedWriter(writer);
PrintWriter printWriter = new PrintWriter(bufferedWriter1, true) ;
System.out.println("Ghi len tep theo ma chuan:" +
  writer.getEncoding());
// Ghi các giá trị nguyên thủy lên tệp theo từng dòng ;          (2)
printWriter.println(true);
printWriter.println('A');
printWriter.println(Byte.MAX_VALUE);
printWriter.println(Short.MIN_VALUE);
printWriter.println(Integer.MAX_VALUE);
printWriter.println(Long.MIN_VALUE);
printWriter.println(Float.MAX_VALUE);
printWriter.println(Math.PI);
// Đóng tệp                                                      (3)
printWriter.close();
// Tạo ra đối tượng của BufferedReader sử dụng mã ISO8859_1      (4)
FileInputStream inputFile = new FileInputStream("info.txt");
InputStreamReader reader=new InputStreamReader(inputFile,"8859_1");
BufferedReader bufferedReader = new BufferedReader(reader);
System.out.println("Doc theo ma chuan:" + reader.getEncoding());
// Đọc ra các giá trị nguyên thủy từ tệp                         (5)
boolean v = bufferedReader.readLine().equals("true")? true : false;
char   c = bufferedReader.readLine().charAt(0);
byte   b = (byte)Integer.parseInt(bufferedReader.readLine());
short  s = (short)Integer.parseInt(bufferedReader.readLine());
int    i = Integer.parseInt(bufferedReader.readLine());
long   l = Long.parseLong(bufferedReader.readLine());
float  f = Float.parseFloat(bufferedReader.readLine());
double d = Double.parseDouble(bufferedReader.readLine());
// Đóng tệp
bufferedReader.close();
// Hiển thị các giá trị ra màn hình
System.out.println("Cac gia tri duoc doc ra:");
System.out.println(v);
```

```
  System.out.println(c);
  System.out.println(b);
  System.out.println(s);
  System.out.println(i);
  System.out.println(l);
  System.out.println(f);
  System.out.println(d);
 }
}
```

Vào / ra trên các thiết bị chuẩn

Thiết bị chuẩn để đưa dữ liệu ra là màn hình thường được thực hiện bởi đối tượng *System.out* của *PrintStream*. Thiết bị chuẩn để nhập dữ liệu vào là bàn phím và thường được thực hiện bởi đối tượng *System.in* của *InputStream*. Các lỗi chuẩn vào/ra có thể sử dụng đối tượng *System.err* của lớp *PrintStream* để xử lý.

Để đọc và chuyển các ký tự về các giá trị tương ứng một cách chính xác và hiệu quả thì đối tượng *System.in* phải được kết hợp với *InputStreamReader* thông qua *buffer:*

InputStreamReader inStream = new InputStreamReader(System.in);
BufferedReader stdInStream = new BufferedReader(inStream);

Ngoài ra Java còn cung cấp lớp *java.text.NumberFormat* để xử lý các dữ liệu theo các *format* của từng vùng, như *java.util.Local.US* - theo *format* dữ liệu địa phương của Mỹ.

Ví dụ 8.4. Viết chương trình nhập dữ liệu vào từ bàn phím và hiển thị lên màn hình theo *format* của Mỹ (*Local.US*).

```
import java.io.*;
import java.text.*;
import java.util.*;
// Xây dựng lớp để nhập dữ liệu từ bàn phím và hiển thị chúng lên màn hình
public final class Stdin {
    // Tạo ra đối tượng của BufferedReader kết nối với InputStreamReader
    private static BufferedReader reader = new BufferedReader(    //(1)
            new InputStreamReader(System.in));
    // Đọc vào một dòng và kết quả nhận được là xâu
    public static String readline() {
        while(true) try {
          return reader.readLine();
        } catch(IOException ioe) {
```

```
            reportError(ioe);
        }
    }
    // Đọc vào một số nguyên
    public static int readInteger() {
        while(true) try {
            return Integer.parseInt(reader.readLine());
        } catch(IOException ioe) {
            reportError(ioe);
        } catch(NumberFormatException nfe) {
            reportError(nfe);
            }
    }
    // Đọc vào một số double
    public static double readDouble() {
        while(true) try {
            return Double.parseDouble(reader.readLine());
        } catch(IOException ioe) {
            reportError(ioe);
        } catch(NumberFormatException nfe){
            reportError(nfe);
            }
    }
    // Hiển thị một thông báo lỗi
    private static void reportError(Exception e) {
        System.err.println("Loi nhap du lieu:" + e);
        System.err.println("Hay nhap lai du lieu!");
    }
    // Chương trình chính
    public static void main(String args[]) {
        System.out.print("Nhap vao mot xau:  ");
        String str = Stdin.readline();
        System.out.print("Nhao vao so nguyen:");
        int i = Stdin.readInteger();
        System.out.print("Nhap vao so double:");
        double d = Stdin.readDouble();
        // Tạo ra format để hiển thị theo qui định của Mỹ (java.util.Local.US)
```

```
        NumberFormat formatter = NumberFormat.getInstance(Locale.US);
        System.out.println("Data read:");
        System.out.println(str);
        System.out.println(formatter.format(i));
        System.out.println(formatter.format(d));
    }
}
```

8.4. TRUY NHẬP TỆP NGẪU NHIÊN

Lớp *RandomAccessFile* cho phép truy nhập trực tiếp (ngẫu nhiên) vào các tệp, nghĩa là có thể đọc, ghi các byte ở bất kỳ một vị trí nào đó trong tệp.

Tệp muốn được truy nhập ngẫu nhiên thì phải được tạo lập và gán cho biến tệp tương ứng trước khi sử dụng.

RandomAccessFile(String name, String mode) throws IOException
RandomAccessFile(File file, String mode) throws IOException

Tệp được xác định bởi tên *name* hoặc bởi đối tượng *File*. Đối số *mode* có thể là "*r*" để đọc, "*rw*" để đọc và ghi.

Lưu ý: Một tệp được mở để đọc, ghi thì nội dung của tệp đó sẽ không bị xóa giống như tệp mở chỉ để ghi.

long getFilePointter() throws IOException

Cho lại vị trí hiện thời của con trỏ tệp.

long length() throws IOException

Cho lại số byte (độ dài) của tệp.

void seek(long offset) throws IOException

Chuyển con trỏ tệp đi *offset* vị trí kể từ đầu tệp.

void close() throws IOException

Khi tệp không còn cần để truy nhập ngẫu nhiên nữa thì phải đóng lại.

Ví dụ 8.5. Mở tệp để ghi, đọc dữ liệu ngẫu nhiên.

```
import java.io.*;
public class RandomAccessDemo {
   static String fileName = "daySo.data";
```

```
  final static int INT_SIZE = 4;
  public static void main(String args[]) {
    try {
      RandomAccessDemo random=new RandomAccessDemo();
      random.createFile();
      random.readFile(); // Đọc lần thứ nhất
      random.extendFile();
      random.readFile(); // Đọc lần thứ hai
     } catch (IOException ex){
        System.err.println(ex);
    }
  }
// Tạo ra các số là bình phương của các số lẻ từ 1 tới 9.
  public void createFile()throws IOException {
    File dataFile = new File (fileName);
    RandomAccessFile outputFile = new RandomAccessFile(dataFile, "rw");
    for (int i = 0; i<10; i++)
       outputFile.writeInt(i*i);
    outputFile.close();
  }
  // Đọc các số là bình phương của các số lẻ từ 1 tới 9 đã được ghi lên tệp
  public void readFile() throws IOException {
    File dataFile = new File(fileName);
    RandomAccessFile inputFile = new RandomAccessFile(dataFile, "r");
    System.out.println("Binh phuong cua cac so le tu tep:");
    long length = inputFile.length();
    for (int i = INT_SIZE; i < length; i += 2* INT_SIZE) {
        inputFile.seek(i);
        System.out.println(inputFile.readInt());
    }
    inputFile.close();
   }
// Mở rộng các số là bình phương của các số lẻ từ 10 đến 19.
  public void extendFile()throws IOException {
    RandomAccessFile outputFile = new RandomAccessFile(fileName, "rw");
        outputFile.seek(outputFile.length());
        for (int i= 10; i<20;i++)
```

```
            outputFile.writeInt(i*i);
        outputFile.close();
        }
    }
```

8.5. TRUY NHẬP TỆP TUẦN TỰ DỰA TRÊN ĐỐI TƯỢNG

Sự tuần tự hóa đối tượng cho phép một đối tượng có thể chuyển thành một dãy các byte và sau đó có thể tái tạo lại thành đối tượng gốc.

Lớp *ObjectOutputStream* cài đặt giao diện *ObjectOutput,* cung cấp các hàm để ghi các đối tượng dưới dạng các *byte, text* và *các giá trị nguyên thủy.* Tương tự, lớp *ObjectInputStream* cài đặt giao diện *ObjectInput,* cung cấp các hàm để đọc các đối tượng từ tệp dạng các *byte, text* và *các giá trị nguyên thủy.*

Lớp ObjectOutputStream

Lớp *ObjectOutputStream* có toán tử tạo lập:

ObjectOutputStream(OutputStream out) throws IOException

Ví dụ, để lưu trữ các đối tượng vào tệp có tên "*khoDT.dat*" thì trước tiên phải mở tệp:

FileOutputStream outFile = new FileOutputStream("khoDT.dat");
ObjectOutputStream outStream = new ObjectOutputStream(outFile);

Để ghi các đối tượng vào tệp đã được mở, chúng ta sử dụng:

final void writeObject(Object obj) throws IOException

Hàm *writeObject()* được sử dụng để ghi một đối tượng bất kỳ, kể cả mảng hay xâu ký tự.

Lớp ObjectInputStream

Lớp *ObjectInputStream* có toán tử tạo lập:

ObjectInputStream(InputStream in) throws IOException,StreamCorruptedException

Ví dụ , để đọc các đối tượng từ tệp có tên "*khoDT.dat*" thì trước tiên phải mở tệp:

FileInputStream inFile = new FileInputStream("khoDT.dat");
ObjectInputStream inStream = new ObjectInputStream(inFile);

Để đọc các đối tượng từ tệp đã được mở, chúng ta sử dụng:

final Object readObject() throws IOException, OptionalDataException, IOException

Hàm *readeObject()* được sử dụng để đọc một đối tượng bất kỳ, kể cả mảng hay xâu ký tự.

Ví dụ 8.6. Mở tệp để đọc, ghi các đối tượng tuần tự.

```
import java.io.*;
public class ObjectSerializationDemo {
  void writeData() {
    try {
    // Mở tệp để ghi theo đối tượng
    FileOutputStream outputFile = new
                                  FileOutputStream("DT.dat");
    ObjectOutputStream outputStream = new
                                ObjectOutputStream(outputFile);
    // Ghi dữ liệu lên tệp
    String[]hoTen = {"Ha", "Lan", "Anh"};
    long soBD = 20452;
    int[] ngayNamSinh = {19, 1, 1984};
    float diem = 7.5f;
    String monHoc = "Lap trinh Java";
    outputStream.writeObject(hoTen);
    outputStream.writeLong(soBD);
    outputStream.writeObject(ngayNamSinmh);
    outputStream.writeObject(monHoc);
    outputStream.writeFloat(diem);
    // Đóng tệp đã mở để ghi
    outputStream.flush();
    outputStream.close();
     }catch (IOException ex) {
        System.err.println(ex); // Nếu gặp ngoại lệ thì thông báo
     }
    }
  // Đọc dữ liệu theo các đối tượng
     void readData(){
       try{
         // Mở tệp để đọc dữ liệu
          FileInputStream inputFile = new
```

```
                    FileInputStream("DT.dat");
        ObjectInputStream inputStream = new
                    ObjectInputStream(inputFile);
        // Đọc dữ liệu
        String[] hoTen= (String[]) inputStream.readObject();
        long soBD     = inputStream.readLong();
        int[] ngaySinh = (int[]) inputStream.readObject();
        String monHoc = (String) inputStream.readObject();
        float diem = (float) inputStream.readFloat();
        // Hiển thị thông tin ra màn hình
        System.out.print("Ho va ten: ");
        for(int i = 0; i<hoTen.length;i++) {
                System.out.print(hoTen[i] + "\t");
        }
        System.out.print("\nNgay sinh: ");
        for(int i = 0; i< ngaySinh.length; i++) {
            System.out.print(ngaySinh[i] + "\t");
        }
        System.out.print("\nSo the   : ");
        System.out.println(soBD);
        System.out.println();
        System.out.println("Mon hoc  : " + monHoc);
        System.out.println("Ket qua  : " + diem);
        // Đóng tệp đã mở để đọc
        inputStream.close();
      }catch (Exception ex) {
        System.err.println(ex);
    }
  }
  public static void main(String args[]) {
   ObjectSerializationDemo demo = new
   ObjectSerializationDemo();
   demo.writeData();
   demo.readData();
   }
}
```

BÀI TẬP

8.1. Một trung tâm Tin học cần quản lý các phương tiện giao thông gồm các loại ô tô và xe máy. Mỗi loại phương tiện (ô tô và xe máy) cần quản lý: *màu xe, giá thành* và *hãng sản xuất*. Ngoài ra, đối với ô tô thì phải biết thêm: *số ghế, loại máy (động cơ);* còn xe máy thì phải biết thêm: *công suất*.

1. Xây dựng các lớp cho các phương tiện, trong đó lớp ôtô và xe máy là được kế thừa từ lớp phương tiện giao thông nói chung,

2. Viết chương trình thực hiện theo các menu sau:

   1/ Đăng ký loại phương tiện mới

   2/ Xóa đi một loại phương tiện

   3/ Tìm theo hãng sản xuất

   4/ Tìm theo màu

   5/ Thoát khỏi hệ thống

Người sử dụng có thể chọn một trong các mục nêu trên. Nếu chọn 1/ thì hệ thống sẽ hỏi là nhập mới ô tô hay xe máy, chọn 2/ để xóa đi một mục đã đăng ký, 3/ thì hệ thống sẽ hỏi tên hàng sản xuất cần tìm, v.v.

Viết chương trình thực hiện các chức năng trên. Thông tin về các loại phương tiện giao thông khi nhập vào phải được ghi lên tệp để sau đó được đọc ra để tìm kiếm theo hãng hay theo màu.

8.2. Phòng quản lý sinh viên của Đại học QG cần quản lý các các thông về sinh viên của Trường. Mỗi sinh viên cần quản lý: *họ và tên, số báo danh, ngày tháng năm sinh, môn học* và *điểm môn học*.

Viết chương trình thực hiện theo các menu sau:

1/ Nhập thông tin của sinh viên mới

2/ Tìm theo số báo danh

3/ Hiển thị tất cả danh sách các sinh viên

4/ Thoát khỏi hệ thống

Người sử dụng có thể chọn một trong các mục nêu trên. Nếu chọn 1/ thì người sử dụng có thể nhập vào một sinh viên mới, chọn 2/ để tìm theo số báo danh v.v.

Viết chương trình thực hiện các chức năng trên. Thông tin về sinh viên khi nhập vào phải được ghi lên tệp để sau đó được đọc ra để tìm kiếm hay hiển thị danh sách.

CHƯƠNG IX

KẾT NỐI CÁC CƠ SỞ DỮ LIỆU VỚI JDBC VÀ LẬP TRÌNH TRÊN MẠNG

9.1. GIỚI THIỆU TỔNG QUAN VỀ ODBC VÀ JDBC

Xu thế phát triển hiện nay của công nghệ phần mềm là xây dựng những ứng dụng tích hợp, dựa trên những cơ sở dữ liệu (CSDL) khác nhau được thiết lập ở nhiều nơi khác nhau trên mạng. ODBC (*Open DataBase Connectivity*) của Microsoft và JDBC (*Java DataBase Connectivity*) của Sun là những công cụ phổ dụng hỗ trợ để phát triển những hệ thống như thế.

ODBC

ODBC là một trong những giao diện CSDL được sử dụng phổ biến nhất hiện nay trên các máy PC. ODBC cung cấp các chức năng của một ngôn ngữ lập trình để tương tác với các CSDL như: bổ sung, cập nhật, xóa những dữ liệu không cần thiết và nhận được những thông tin chi tiết về các bảng, các chỉ số hay khung nhìn của các phần dữ liệu trong một hệ CSDL.

Một ứng dụng ODBC thường được tổ chức theo năm tầng: tầng ứng dụng, giao diện ODBC, tầng quản lý các bộ điều khiển, bộ điều khiển và tầng nguồn dữ liệu.

- Tầng ứng dụng cung cấp GUI và các chức năng xử lý nghiệp vụ được viết bằng những ngôn ngữ lập trình như Java, Visual Basic, hay C++. Chương trình ứng dụng của bạn có thể sử dụng những chức năng ODBC trong giao diện ODBC để tương tác với các CSDL.
- Tầng quản lý các bộ điều khiển (*Driver Manager*) là một bộ phận của Microsoft ODBC. Nó làm nhiệm vụ quản lý các bộ điều khiển có mặt trong hệ thống để nạp hay chọn được những bộ điều khiển thích hợp và cung cấp cho những bộ điều khiển đó những thông tin cần thiết. Bởi vì một chương trình ứng dụng có thể kết nối với nhiều hơn một CSDL, nên *Driver Manager* phải

đảm bảo rằng chính hệ QTCSDL (Quản Trị CSDL) của hệ CSDL tương ứng sẽ nhận được tất cả các lời gọi hàm hướng tới nó để xử lý và các nguồn dữ liệu được chuyển đúng tới chương trình ứng dụng theo yêu cầu.

- Tầng bộ điều khiển (*Driver*) là thành phần cho biết cụ thể về các CSDL cần kết nối. Đó là những bộ điều khiển được gán cho những hệ CSDL tương ứng như: *Access Driver, SQL Server Driver,* và *Oracle Driver,* v.v. Giao diện ODBC có tập các hàm như các lệnh SQL, quản lý sự kết nối, các thông tin về CSDL, v.v. Đối với những CSDL ở trên mạng cục bộ hay trên Internet, tầng Driver còn đảm nhiệm cả chức năng xử lý sự trao đổi thông tin trên mạng.
- Trong ngữ cảnh của ODBC, nguồn dữ liệu (*Data Source*) có thể là hệ CSDL nhỏ, đơn giản của MS Access hay những kho dữ liệu (Data Warehouse) lớn có hàng chục *gigabyte*.

JDBC

JDBC là giao diện để kết nối với CSDL, gồm một tập các lớp đối tượng hỗ trợ để xử lý CSDL quan hệ và để tương tác với các nguồn dữ liệu khác nhau. ODBC là giao diện của ngôn ngữ *C,* và rất khó chuyển đổi tương ứng sang được Java, còn JDBC là giao diện của Java và hoàn toàn thống nhất với những thành phần khác của hệ thống Java.

JDBC API (*Application Programming Interface*) được JavaSoft phát triển và là một phần của tất cả các cài đặt ứng dụng của JVM. JDBC API cung cấp các chức năng cơ bản để truy nhập (thường ở mức trung gian) tới hầu hết các hệ CSDL thông qua SQL (*Structural Query Language*). Chúng ta có thể truy nhập và tìm kiếm mọi thông tin, kể cả dữ liệu về âm thanh, hình ảnh động từ nhiều nguồn khác nhau trên mạng bằng cách sử dụng JDBC API để nạp chúng vào các đối tượng của Java và sau đó xử lý các thông tin đó. Bạn có thể nạp JDBC từ địa chỉ: http://splash.javaSoft.com/jdbc.

Kiến trúc của JDBC

Mỗi CSDL đều có một API riêng và muốn tương tác với CSDL đó thì phải biết được kiến trúc của chúng. Muốn sử dụng được các thông tin, dữ liệu từ nhiều CSDL trên mạng, chúng ta phải sử dụng những giao diện lập trình ứng dụng (API) để kết nối và truy vấn vào các CSDL.

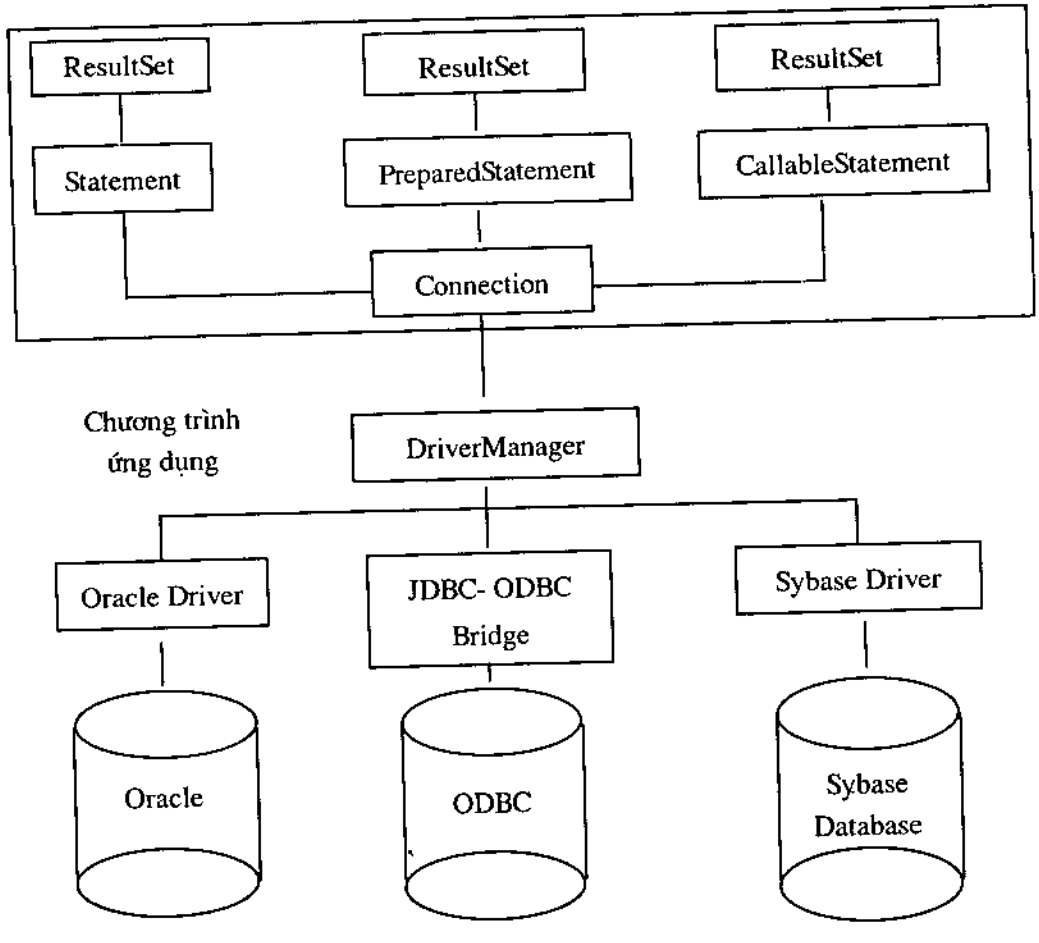
Tương tự như ODBC, JDBC của Sun nhằm tạo ra những giao diện trung gian giữa CSDL và Java. Với JDBC API, chúng ta có thể thực hiện được những chức năng cơ bản đối với CSDL:

- Thực hiện các truy vấn vào một CSDL,
- Xử lý kết quả từ truy vấn,
- Xác định các thông tin về cấu hình hệ thống.

Các lớp trong JDBC API được sử dụng để thực hiện những chức năng trên được tổ chức theo kiến trúc như trong hình 9.1.

JDBC định nghĩa các đối tượng API và các hàm để tương tác với những CSDL đã được chỉ định. Chương trình ứng dụng Java kết nối các CSDL có thể được tổ chức như sau:

- Trước tiên là tạo ra đối tượng kết nối vào một CSDL,
- Tạo ra đối tượng để xử lý các câu lệnh,
- Truyền tham số cho các lệnh SQL trong các hệ quản trị CSDL (DBMS) và các đối tượng xử lý các câu lệnh đó,
- Tìm kiếm các thông tin là các kết quả truy vấn.



*Hình 9.1. **Kiến trúc của JDBC.***

Các lớp JDBC được tổ chức trong gói *java.sql*. Các lớp JDBC và các chương trình ứng dụng *Applet hay ứng dụng độc lập* của Java có thể:

- Để ở các máy khách,
- Hoặc cũng có thể nạp xuống từ mạng.

Nhưng tốt nhất là chúng ta để các lớp JDBC ở máy khách, còn các DBMS (hệ quản trị CSDL) và các dữ liệu nguồn có thể đặt ở máy phục vụ ở đâu đó trên mạng.

Một chương trình muốn kết nối các CSDL thông qua JDBC thì phải sử dụng bộ điều khiển dữ liệu gốc theo giao diện tương ứng với từng hệ DBMS (hình H9-1). Đối với cầu nối JDBC - ODBC trong Windows thì nên sử dụng JDBCODBC.DLL. Chương trình ứng dụng của Java có thể trao đổi với máy phục vụ, hay các ứng dụng khác thông qua các giao thức *RPC* hoặc *HTTP* [4].

JDBC - ODBC Bridge

JavaSoft cung cấp bộ điều khiển JDBC để truy nhập vào các nguồn dữ liệu dựa vào ODBC, tức là tạo ra cầu nối giữa chúng.

Cầu nối JDBC - ODBC được tổ chức thành lớp *JdbcOdbc* và thành thư viện ngoại để truy nhập theo bộ điều khiển ODBC (thư viện động JDBCODBC.DLL).

Với cầu nối JDBC - ODBC thì JDBC có ưu điểm nổi trội là có khả năng truy nhập tới hầu như tất cả các CSDL phổ biến, tương tự như các bộ điều khiển của ODBC.

9.2. CHƯƠNG TRÌNH ỨNG DỤNG JDBC

Để xử lý được dữ liệu từ một CSDL, chương trình Java phải thực hiện lần lượt theo các bước sau:

1. Trước tiên là gọi hàm *getConnection()* để nhận được đối tượng của lớp *Connection;*
2. Tạo ra một đối tượng của lớp *Statement;*
3. Chuẩn bị một đối tượng để xử lý lệnh của SQL và truy vấn vào dữ liệu theo yêu cầu; Câu lệnh SQL có thể thực hiện trực tiếp thông qua đối tượng của *Statement* hoặc có thể được biên dịch thông qua đối tượng của *PreparedStatement* hay gọi một thủ tục để lưu lại thông qua *CallableStatement*.
4. Khi hàm *executeQuery()* được thực hiện, thì kết quả được cho lại là đối tượng của lớp *ResultSet* bao gồm các dòng dữ liệu và có thể sử dụng hàm *next()* để xác định các dữ liệu theo yêu cầu.

Sau đây chúng ta xét một vài chương trình tương đối đơn giản để truy vấn, cập nhật vào một CSDL.

Trước tiên, chúng ta hãy tạo ra một CSDL theo mô hình quan hệ, trong đó, mỗi mục dữ liệu được xem như một dòng tin về đối tượng được lưu trữ. Mỗi dòng sẽ chứa một số cột, được gọi là trường dữ liệu. Đặc điểm chính của mô hình quan hệ là dữ liệu được lưu trữ phải là thuần nhất, nghĩa là số trường của tất cả dòng dữ liệu phải bằng nhau. Một số các dòng và các trường dữ liệu được tổ chức thành bảng quan hệ (bảng dữ liệu) mô tả về một thực thể hay một mối quan hệ nào đó của hệ thống ứng dụng. Nhiều bảng dữ liệu như thế sẽ tạo thành CSDL.

Khi các CSDL quan hệ trở nên phổ biến, các chuyên gia CSDL mong muốn có một ngôn ngữ CSDL vạn năng để thao tác trên các dữ liệu và SQL chính là ngôn ngữ đáp ứng được các yêu đó. SQL có những cấu trúc lệnh để tạo lập, cập nhật, xóa bỏ các bảng, dòng hay các trường dữ liệu trong hệ thống. Ngoài ra SQL còn hỗ trợ để quản lý quyền truy nhập tới các phần tử dữ liệu, quản lý người sử dụng, hay thực hiện sao chép dữ liệu, v.v. SQL có thể sử dụng kết hợp với những ngôn ngữ lập trình như Java, C++, v.v., để xử lý dữ liệu và tương tác với các hệ QTCSDL.

Giả sử chúng ta có một CSDL bao gồm những bảng dữ liệu về sinh viên và các môn học của sinh viên. Ví dụ: Bảng *Student* gồm ba trường : *StudentNo* có kiểu số nguyên, *FName*, và *LName* có kiểu xâu ký tự, trong đó *StudentNo* là *khóa*.

Bảng 9.1. Bảng dữ liệu *Student*

StudentNo	FName	LName
1	Vũ Văn	An
2	Trần Anh	Tuấn
3	Lê Hoa	Lư
4	Nguyễn Công	Tuyển

Bảng *Subject* có hai trường *SubjectNo* và *SubjectName* cùng có kiểu xâu ký tự, *SubjectNo* là khóa.

Bảng 9.2. Bảng dữ liệu *Subject*

SubjectNo	SubjectName
M1	Lập trình Java
M2	Cơ sở dữ liệu
M3	Toán rời rạc
M4	Phân tích, thiết kế hệ thống

Tất nhiên các bảng dữ liệu trong một hệ CSDL phải được liên kết với nhau thông qua các *khóa ngoại* . Để bảng *Student* liên kết được với bảng *Subject,* chúng ta phải tạo thêm bảng mới, ví dụ: bảng *Student_Subject* có ba trường: *StudentNo, SubjectNo* và *Marks* (điểm thi môn học của sinh viên có kiểu số thực) xác định mối quan hệ giữa hai bảng dữ liệu trên và lưu điểm thi từng môn của sinh viên.

Bảng 9.3. Bảng dữ liệu *Student_Subject*

StudentNo	SubjectNo	Marks
1	M2	6
2	M3	7
3	M1	5
4	M4	8

Chúng ta có thể sử dụng MS Access để tạo ra các bảng trên của CSDL *StudentDB.mdb.*

Như vậy, trước khi viết chương trình JDBC, chúng ta cần phải biết được thông tin về nguồn dữ liệu. Chính hàm *getConnection()* sẽ cho lại tên miền của nguồn dữ liệu (DSN), ID của người sử dụng và mật khẩu (password) để truy nhập vào nguồn dữ liệu cần kết nối. Để kết nối được với một CSDL thì nó phải đăng ký vào *Data Sources* thông qua *Control Panel/Administrative Tools* và khai báo đường dẫn, tên của những CSDL cần kết nối (đối với Window).

Ví dụ 9.1. Đọc lần lượt các trường của một bảng dữ liệu trong CSDL.

Trong ví dụ này chúng ta muốn liệt kê tất cả họ (FName) và tên sinh viên (LName) từ bảng *Student* trong CSDL *StudentDB* (StudentDB.mdb được tạo ra bằng Access) sử dụng lệnh SELECT của SQL.

// Một ví dụ đơn giản sử dụng JDBC để đọc dữ liệu từ CSDL: JDBCSample.java.

```
import java.sql.*;
public class JDBCSample{
    public static void main(String[] args){
        try{// Nạp Driver JdbcOdbcDriver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
           }catch (ClassNotFoundException e){
           System.out.println("Khong nap duoc lop Driver");
            return;
            }
        try{// Mọi truy nhập vào CSDL đều thực hiện trong khối try - catch
```

```
// Xác định tên của CSDL, tên người sử dụng và mật khẩu
  Connection con =
      DriverManager.getConnection("jdbc:odbc:StudentDB","","");
// Thiết lập và thực hiện các lệnh của SQL
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT FName,
                LName FROM STUDENT");
// Hiển thị kết quả sau khi đã sử dụng các lệnh của SQL để truy vấn
while(rs.next()){
     System.out.print(rs.getString("FName")+ " ");
     System.out.println(rs.getString("LName"));
     }
// Đóng lại các nguồn dữ liệu đã được mở
rs.close();
stmt.close();
con.close();
}catch(SQLException se){
      System.out.println("Loi o SQL: " + se.getMessage());
      se.printStackTrace(System.out);
  }
}
}
```

Đây là chương trình Java đơn giản sử dụng JDBC API. Chương trình này minh họa các bước cơ sở, cần thiết để truy nhập vào các bảng dữ liệu và liệt kê một số trường dữ liệu từ các bảng dữ liệu đó.

Sau đây chúng ta xét một số giao diện, một số lớp trong gói *java.sql* và một số hàm chính của chúng [4].

Lớp DriverManager

Lớp *DriverManager* chịu trách nhiệm nạp các bộ điều khiển JDBC và tạo ra sự kết nối giữa các đối tượng của CSDL. Trước khi sử dụng một bộ điều khiển (*Driver*) thì nó phải được đăng ký với *DriverManager*. Điều này thực hiện được bằng cách sử dụng hàm *forName()* của lớp *Class:*

```
try{
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     Class.forName("com.oracle.jdbc.OracleDriver");
```

```
}catch (ClassNotFoundException e){
    // Xử lý ngoại lệ khi không nạp được
    }
```

Sau đó chương trình phải sử dụng `DriverManager.getConnection(String url)` để tạo ra từng sự kết nối với CSDL.

Bộ điều khiển JDBC sử dụng lớp JDBC URL (Uniform Resource Locator) để xác định và kết nối với một CSDL. Dạng tổng quát của bộ điều khiển *url* là:

jdbc:driver:databaseName

Giao diện Connection

Giao diện *java.sql.Connection* phục vụ cho việc kết nối các CSDL. Đối tượng của các lớp *Statement, PreparedStatement* và *CallableStatement* có thể sử dụng hàm *DriverManager.getConnection()* để thiết lập một kết nối:

```
Connection con =
        DriverManager.getConnection("url","userName","password");
```

Trong đó, `url` là đối tượng của JDBC URL xác định tên của CSDL cần truy nhập, `userName` tên của người sử dụng và mật khẩu `password`. Đối với những CSDL khi mở không yêu cầu khai báo tên người sử dụng và mật khẩu thì các tham biến đó có thể là rỗng.

```
Connection con = DriverManager.getConnection("url","","");
```

Sau khi đã thiết lập được đối tượng *con* để kết nối, chúng ta có thể sử dụng các câu lệnh của SQL để truy vấn vào CSDL. Trong JDBC có ba lớp xử lý các câu lệnh đó.

Ngoài ra `Connection` còn có các hàm:

```
public abstract void commit() throws SQLException;
public abstract boolean getAutoCommit() throws SQLException;
public abstract void setAutoCommit(boolean autoCommit) throws SQLException;
public abstract String getMetaData() throws SQLException;
public abstract void setCatalog(String catalog) throws SQLException;
public abstract void setReadOnly(boolean readOnly) throws SQLException;
public abstract boolean isReadOnly() throws SQLException;
```

Giao diện Statement

Giao diện *Statement* để thực hiện các lệnh của SQL. Đối tượng của *Statement* phải được tạo ra bằng lệnh *createStatement()* của *Connection*.

```
Connection con =
        DriverManager.getConnection("url","userName","password");
Statement stmt = con.createStatement();
```

Trong *Statement* có các hàm: *execute(), executeQuery(), executeUpdate()* và *executeBatch(), v.v.* nhận các tham số là các câu lệnh của SQL để thực hiện. Những lệnh của SQL thực hiện được với các lệnh trên là: CREATE, DROP, INSERT, UPDATE, DELETE, v.v. và có thể xử lý theo lô.

Để thực hiện một giao tác *(transaction)* mà cần sử dụng nhiều lệnh thì cần gọi hàm *setAutoCommit(false)* để nếu có một lệnh nào đó không thực hiện thành công thì khôi phục được phần thông tin còn lại. Sau đó có thể sử dụng *executeUpdate()* để cập nhật:

```
try{
  con.setAutoCommit(false);
  stmt.executeUpdate("UPDATE STUDENT SET FNAME = 'Tran Anh'
                      WHERE STUDENTNO = 10");   // Thực hiện lệnh Update()

  con.commit();
}catch(SQLEXception e){
  con.rollback();    // Khôi phục lại kết quả khi thực hiện không thành công.
}
```

Lưu ý: sau khi thực hiện cập nhật dữ liệu thì phải gọi hàm *commit()* để ủy quyền thay đổi những kết quả đã cập nhật, nếu không thì những thay đổi trong quá trình cập nhật sẽ không được thực thi.

Trong JDBC 2.0, các kết quả truy vấn được từ một CSDL có thể *chỉ cho phép đọc, nghĩa là không cho phép thay đổi dữ liệu* hoặc *được phép cập nhật, thay đổi được dữ liệu*. Để có thể thay đổi được dữ liệu kết quả thì cần phải truyền tham số cho lệnh *createStetament()* như sau:

```
Statement stmt = con.createStatement(
                                ResultSet.TYPE_SCROLL_SENSITIVE,

ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM STUDENT");
rs.first();
rs.updateInt(1, 2);
rs.updateRow();
```

Nếu không truyền tham số cho `createStatement()` thì chúng ta nhận được kết quả chỉ được đọc (read-only). Sau khi thực hiện cập nhật các trường dữ liệu thì phải gọi hàm *updateRow()* để ghi lại những thay đổi đã được thực hiện.

Hàm *addBatch() của Statement* còn cho phép chúng ta thực hiện cập nhật các trường dữ liệu với nhiều lần nhưng được thực hiện cùng một lúc. Ví dụ: có thể bổ sung vào bảng Student trong CSDL StudentDB.mdb một số bản ghi như sau:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT STUDENT VALUES(18, 'Haong Trung','Thao')");
stmt.addBatch("INSERT STUDENT VALUES(19, 'Ngo Quang','Trang')");
int[] upCounts = stmt.executeBatch();
con.commit();
```

Trước khi thực hiện một giao tác theo lô (batch) với CSDL chúng ta thường phải thực hiện ủy quyền (*commit*) để đảm bảo có thể khôi phục lại dữ liệu khi có một thao tác nào đó thất bại. Sau một số lần thực hiện *addBatch()* thì phải gọi hàm executeBatch() để chuyển các lệnh SQL thực hiện theo lô trong CSDL. Lệnh này trả lại kết quả là mảng các số dòng đã được cập nhật theo lô.

Giao diện PreparedStatement

Giao diện PreparedStatement cho phép chương trình thực hiện tiền xử lý các câu lệnh của SQL để tăng hiệu quả xử lý. Chúng ta có thể nhận được đối tượng của PreparedStatement thông qua hàm preparedStatement() của Connection. Tham số của hàm này là câu lệnh của SQL trong đó có thể sử dụng ký tự '?' ở các trường dữ liệu.

```
Connection con =
        DriverManager.getConnection("url","userName","password");
con.setAutoCommit(false);
PreparedStatement stmt = con.preparedStatement(
            "INSERT INTO STUDENT VALUES(?, ?, ?)");
```

Sau đó chúng ta có thể cập nhật từng trường dữ liệu với đối tượng *stmt* như sau:

```
stmt.setInt(1,20);        // Đặt giá trị 20 cho cột 1 (StudentNo có kiểu int)
stmt.setString(2,"Doan Van"); //Thay "Doan Van" vào cột 2 (FName có kiểu String)
stmt.setString(3,"Trung"); //Thay "Trung" vào cột 3 (LName có kiểu String)
stmt.addBatch();
int[] upCounts = stmt.executeBatch();
con.commit();
```

Giao diện ResultSet

Kết quả của các câu lệnh xử lý với SQL được lưu trữ và xử lý thông qua đối tượng của *ResultSet*. Các đối tượng của *ResultSet* có thể được tạo ra nhờ các lệnh: *execute(), executeQuery(), executeUpdate()* và một số hàm của *MetaData*.

Khi thực hiện truy vấn với *executeQuery()* thì khai báo

```
ResultSet rs = stmt.executeQuery("SELECT * FROM STUDENT");
```

Chọn ra tất cả các dòng của bảng *STUDENT* và gán kết quả cho đối tượng *rs* của *ResultSet*. Chúng ta có thể xem *ResultSet* như là đối tượng chứa những kết quả truy vấn vào CSDL theo câu lệnh *SELECT* của SQL.

Bảng 9.4. Các kiểu dữ liệu tương ứng của Java, SQL và các hàm *getXXX()*

Kiểu của SQL	Kiểu của Java	Hàm getXXX()
CHAR	String	getString()
VARCHAR	String	getString()
LONGVARCHAR	String	getString()
NUMBERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	Boolean (boolean)	getBoolean()
TINYINT	Integer (byte)	getByte()
SMALLINT	Integer (short)	getShort()
INTEGER	Integer (int)	getInt()
BIGINT	Long (long)	getLong()
REAL	Float (float)	getFloat()
FLOAT	Double (double)	getDouble()
DOUBLE	Double (double)	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

Để đọc giá trị của các cột (trường dữ liệu) có thể sử dụng hàm dạng *getXXX()*, trong đó *XXX* được thay thế tương ứng bằng tên các lớp, như *getString(), getByte(), getInt(), v.v.* tùy thuộc vào các kiểu của các trường dữ liệu. Hàm *next()* cho phép đọc dòng dữ liệu tiếp theo và thông qua hàm dạng getXXX() để đọc ra từng trường dữ liệu.

```
while(rs.next()){
    System.out.println(rs.getString("FName"));
    }
```

Lưu ý: Giá trị trả lại của hàm *getXXX(args)* là dữ liệu của trường có tên là *args* của các dòng dữ liệu đã được chọn ra. Ngoài ra cũng cần phân biệt các kiểu của Java với các kiểu dữ liệu của SQL. Một số bộ điều khiển không cho phép thực hiện chính xác như trong Bảng B9.4. Ví dụ, hàm *getString()* trả lại đối tượng của lớp *String* biểu diễn cho mọi kiểu dữ liệu.

Statement còn có hàm *executeUpdate()* để cập nhật dữ liệu và cho lại kết quả là số nguyên chỉ ra số dòng của CSDL đã được kết nối, hàm *execute()* được sử dụng để thực hiện các lệnh của SQL.

Lưu ý: Lớp *java.sql.Types* định nghĩa các hằng nguyên biểu diễn cho các kiểu dữ liệu của SQL chuẩn.

Để thực hiện được cập nhật, thường chúng ta phải thực hiện chuyển dịch đầu đọc qua các hàng của bảng dữ liệu kết quả. Giao diện *ResultSet* cung cấp các hàm để chuyển tới hàng cần cập nhật.

Bảng 9.5. Bảng các hàm chuyển dịch theo các hàng trong bảng kết quả

Hàm	Nội dung thực hiện
boolean first()	Chuyển về hàng đầu tiên
boolean last()	Chuyển về hàng cuối cùng
boolean next()	Chuyển sang hàng tiếp theo
boolean previous()	Chuyển sang hàng trước đó
void beforeFirst()	Chuyển về phía trước hàng đầu tiên
void afterLast()	Chuyển về phía sau hàng cuối
boolean absolute(int n)	Chuyển đến hàng *n*, *n* là nguyên dương
boolean relative(int n)	Chuyển đi về phía trước hay về phái sau *n* hàng, tùy thuộc *n* là âm hay dương
void moveToCurrentRow	Chuyển tới hàng hiện thời
void moveToInsertRow()	Chuyển tới hàng để chèn thêm
boolean isFirst()	Kiểm tra xem có phải là hàng đầu không
boolean isLast()	Kiểm tra xem có phải là hàng cuối không

Ngoài những thông tin nhận được bằng hàm *getXXX()* như đã giới thiệu ở bảng 9.4, thì vấn đề rất quan trọng trong các ứng dụng CSDL là khả năng cập nhật được những thông tin đã được xử lý. *ResultSet* còn cung cấp một loạt các hàm dạng *updateXXX()* để thực hiện cập nhật dữ liệu đối với từng trường hay từng hàng dữ liệu.

Ví dụ 9.2. Đọc, hiển thị, bổ sung và cập nhật dữ liệu trong CSDL: StudentDB.mdb

```
import java.io.*;
import java.sql.*;
import java.util.StringTokenizer;
public class UpdateDataBase{
  int i,NoOfColumns;
  String StNo,StFName, StLName;
  public void listStudent() throws SQLException {
 // Khởi tạo và nạp JDBC-ODBC driver.
      try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }catch (ClassNotFoundException e){
        System.out.println("Khong nap duoc lop Driver");
         return;
        }
      try{
        // Thiết lập đối tượng kết nối
        Connection con = DriverManager.getConnection
                    ("jdbc:odbc:StudentDB","","");
        // Tạo lập đối tượng của Statment
        Statement stmt = con.createStatement();
        // Thực hiện lệnh truy vấn SQL thông qua lệnh SELECT
        ResultSet rs = stmt.executeQuery("SELECT * FROM STUDENT");
        // Đọc từng trường dữ liệu và gán cho các biến của Java
        while(rs.next()){
             StNo = rs.getString("STNO");
             StFName = rs.getString("FName");
             StLName = rs.getString("LName");
             System.out.println(StNo+ "  " + StFName + "  " + StLName);
        }
        // Đóng các đối tượng đã tạo ra
        rs.close();
```

```
        stmt.close();
        con.close();
    }catch(SQLException se){
        System.out.println("Loi o SQL: " + se.getMessage());
        se.printStackTrace(System.out);
    }
    }
// Cập nhật các trường dữ liệu
  public void update(String StFName,
          String StLName,String StNo) throws SQLException
   {
    try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }catch (ClassNotFoundException e){
        System.out.println("Khong nap duoc lop Driver");
        return;
        }
    try{
    Connection con = DriverManager.getConnection
            ("jdbc:odbc:StudentDB","","");
    Statement stmt =
        con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
    ResultSet rs   = stmt.executeQuery("SELECT * FROM STUDENT");
    rs.last();              // Chuyển về dòng cuối
    rs.updateString(1, StNo);     // Thay giá trị của trường số 1 bằng StNo
    rs.updateString(2, StFName);// Thay giá trị của trường số 2 bằng StFName
    rs.updateString(3, StLName);     // Thay giá trị của trường số 2 bằng StLName
    rs.updateRow();         // Thực hiện cập nhật các thông tin
    StNo = rs.getString("STNO");
    StFName = rs.getString("FName");
    StLName = rs.getString("LName");
    System.out.println(StNo+ "  " + StFName + "  " + StLName);
    rs.close();
    stmt.close();
    con.close();
    }catch(SQLException se){
```

```
        System.out.println("Loi o SQL: " + se.getMessage());
        se.printStackTrace(System.out);
    }
  }
// Cập nhật các trường dữ liệu theo lô
public void batchUpdate() throws SQLException
   {
    try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }catch (ClassNotFoundException e){
        System.out.println("Khong nap duoc lop Driver");
        return;
    }
    try{
      Connection con = DriverManager.getConnection
            ("jdbc:odbc:StudentDB","","");
      con.setAutoCommit(false);
      Statement stmt = con.createStatement();
      stmt.addBatch("INSERT INTO STUDENT VALUES (20, 'Hoang
Trung', 'Thong')");
      int[] upCounts = stmt.executeBatch();
      con.commit();   // Thực hiện cập nhật theo lô
      stmt.close();
      con.close();
    }catch(SQLException se){
      System.out.println("Loi o SQL: " + se.getMessage());
      se.printStackTrace(System.out);
    }
  }
 // Chương trình chính
  public static void main(String[] args){
    UpdateDataBase sv = new UpdateDataBase();
    try{
     DataInputStream stream = new DataInputStream(System.in);
     sv.listStudent();
     System.out.println(" ------ Cap nhat ban ghi cuoi ---");
       try{
```

```
            System.out.print("StNo =");
            sv.StNo = stream.readLine();
            System.out.print("St FName =");
            sv.StFName = stream.readLine();
            System.out.print("St LName =");
            sv.StLName = stream.readLine();
        }catch(IOException e){
              sv.StNo= sv.StFName = sv.StLName ="";}
        sv.update(sv.StFName,sv.StLName,sv.StNo);
        sv.listStudent();
        System.out.println(" ----- Bo sung ban ghi moi  -----");
        sv.batchUpdate();
        sv.listStudent();
      }catch(SQLException se){
          System.out.println("SQL Error " + se);
          return;
    }
  } }
```

Bảng 9.6. Bảng các hàm cập nhật dữ liệu trong bảng kết quả

Hàm	Nội dung thực hiện
void updateBigDecimal(String ColName, BigDecimal x)	Cập nhật cột có tên *ColName* theo *số lớn*
void updateBigDecimal(int ColIndex, BigDecimal x)	Cập nhật cột số *ColIndex* theo *số lớn*
void updateBoolean(String ColName, boolean b)	Cập nhật cột có tên *ColName* theo *kiểu boolean*
void updateBoolean(int ColIndex, boolean b)	Cập nhật cột số *ColIndex* theo *kiểu boolean*
void updateDouble(String ColName, double d)	Cập nhật cột có tên *ColName* theo *kiểu double*
void updateDouble(int ColIndex, double d)	Cập nhật cột số *ColIndex* theo *kiểu double*
void updateInt(String ColName, int i)	Cập nhật cột số *ColName* theo *kiểu int*
void updateInt(int ColIndex, int i)	Cập nhật cột số *ColIndex* theo *kiểu int*
void updateFloat(int ColIndex, float f)	Cập nhật cột số *ColIndex* theo *kiểu float*
void updateFloat(String ColName, float f)	Cập nhật cột số *ColName* theo *kiểu float*

tiếp bảng 9.6

void updateString(String ColName, String s)	Cập nhật cột số *ColName* theo *kiểu String*
void updateString(int ColIndex, String s)	Cập nhật cột số *ColIndex* theo *kiểu String*
void updateObject(String ColName, Object o)	Cập nhật cột số *ColName* theo *kiểu Object*
void updateObject(int ColIndex, Object o)	Cập nhật cột số *ColIndex* theo *kiểu Object*
void updateRow()	Cập nhật lại hàng
void updateTime(String ColName, Time t)	Cập nhật lại thời gian của hàng có tên *ColName*
void updateTime(int ColIndex, Time t)	Cập nhật lại thời gian của hàng có số *ColIndex*

Giao diện DatabaseMetaData

Muốn xử lý được các dữ liệu của một CSDL thì chúng ta phải biết được những thông tin chung về cấu trúc của CSDL đó như: hệ QTCSDL, tên của các bảng dữ liệu, tên gọi của các trường dữ liệu, v.v .

Để biết được những thông tin chung về cấu trúc của một hệ CSDL, chúng ta có thể sử dụng giao diện *java.sql.DatabaseMetaData* thông qua hàm *getMetaData()*.

```
DatabaseMetaData dbmeta = con.getMetaData();
```

trong đó, *con* là đối tượng kết nối đã được tạo ra bởi lớp *Connection*.

Lớp `DatabaseMetaData` cung cấp một số hàm được nạp chồng để xác định được những thông tin về cấu hình của một CSDL. Một số hàm cho lại đối tượng của String (*getURL()*), một số trả lại giá trị logic (*nullsAreSortedHigh()*) hay trả lại giá trị nguyên như hàm *getMaxConnection()*). Những hàm khác cho lại kết quả là các đối tượng của *ResultSet* như: *getColumns(), getTableType(), getPrivileges(), v.v.*

Ví dụ 9.3 Chương trình xác định những thông tin chung về cấu trúc của CSDL.

```
import java.sql.*;
import java.util.StringTokenizer;
public class DBViewer {
    final static String jdbcURL = "jdbc:odbc:StudentDB";
    final static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
    public static void main(String[] args) {
      System.out.println("---Database Viewer ---");
```

```
try {
   // Đăng ký bộ điều khiển JdbcOdbcDriver
   Class.forName(jdbcDriver);
   // Kết nối với CSDL StudentDB
   Connection con =
     DriverManager.getConnection(jdbcURL, "", "");
   // Tạo ra dbmd để nhận các thông tin về cấu trúc của CSDL đã kết nối
   DatabaseMetaData dbmd = con.getMetaData();
   // In ra những thông tin về cấu trúc của CSDL đã kết nối
   System.out.println("Drive Name: " + dbmd.getDriverName());
   System.out.println("Database Product: " +
                  dbmd.getDatabaseProductName());
   System.out.println("SQL Keywords Supported:" );
   // Tạo ra đối tượng st để nhận các từ khoá của SQL
   StringTokenizer st =
        new StringTokenizer(dbmd.getSQLKeywords(),",");
   while(st.hasMoreTokens())
        // Hiển thị các từ khóa của SQL
        System.out.println(""+st.nextToken());

        // Đối tượng allTables chứa tất cả các bảng trong CSDL
        String[]tableTypes = {"TABLE"};
        ResultSet allTables =
               dbmd.getTables(null,null,null,tableTypes);
        while (allTables.next())  {
             // Đọc và in ra tên, kiểu của từng bảng
             String table_name =
                allTables.getString("TABLE_NAME");
             System.out.println("Table Name:" + table_name);
             System.out.println("Table Type:" +
                allTables.getString("TABLE_TYPE"));
             System.out.println("Indexs: ");
             // Xác định tên của chỉ số (Index) và tên các trường dữ liệu
             ResultSet indexList = dbmd.getIndexInfo (null,null,
                  table_name,false,false);
             while (indexList.next()) {
                // In ra tên của chỉ số (Index) và tên các trường dữ liệu
```

```
                System.out.println("Index Name:" +
                  indexList.getString("INDEX_NAME"));
                System.out.println("Column Name:" +
                  indexList.getString("COLUMN_NAME"));
            }
            indexList.close();
        }
        allTables.close();
        con.close();
      }catch (ClassNotFoundException ce) {
        System.out.println("Unable to load database driver class");
      }catch (SQLException se) {
        System.out.println("SQL Exception: " + se.getMessage());
      }
    }
}
```

Giả sử *StudentDB* là CSDL được tạo ra bởi Microsoft Access thì chương trình trên thực hiện sẽ cho kết quả dạng:

```
--- Dabase Viewer ---
Driver Name: JDBC-ODBC Bridge
Database Product: Access
SQL Keywords Supported:
ALPHANUMERIC
BYTE
. . .
Table Name: Student
Table Type:  TABLE
Indexs:
  Index Name: PrimaryKey
  Column Name: StudentNo
  . . .
```

Giao diện ResultSetMetaData

Giao diện *ResultSetMetaData* cung cấp các thông tin về cấu trúc cụ thể của *ResultSet*, bao gồm cả số cột, tên và giá trị của chúng. Ví dụ sau là một chương trình hiển thị các kiểu và giá trị của từng trường của một bảng dữ liệu.

Ví dụ 9.4. Chương trình hiển thị một bảng dữ liệu.

```
import java.sql.*;
import java.util.StringTokenizer;
public class TableViewer {
    final static String jdbcURL = "jdbc:odbc:StudentDB";
    final static String jdbcDriver =
                        "sun.jdbc.odbc.JdbcOdbcDriver";
    final static String table = "STUDENT";
    public static void main(java.lang.String[]args) {
      System.out.println("---Table Viewer ---");
      try {
        Class.forName(jdbcDriver);
        Connection con =
            DriverManager.getConnection(jdbcURL, "", "");
        Statement stmt = con.createStatement();
        // Đọc ra cả bảng Student và đưa vào đối tượng rs
        ResultSet rs = stmt.executeQuery("SELECT * FROM " + table);
        // Đọc ra các thông tin về rs
        ResultSetMetaData rsmd = rs.getMetaData();
        // Xác định số cột của rsmd
        int colCount = rsmd.getColumnCount();
        for(int col = 1; col <= colCount; col++)
        {
            // In ra tên và kiểu của từng trường dữ liệu trong rsmd
            System.out.print(rsmd.getColumnLabel(col));
            System.out.print(" (" + rsmd.getColumnTypeName(col) + ")");
            if(col < colCount)
               System.out.print(", ");
            }
        System.out.println();
        while(rs.next()){
            // In ra dòng dữ liệu trong rsmd
            for(int col = 1; col <= colCount; col++)
                {
                System.out.print(rs.getString(col));
                if(col < colCount)
                   System.out.print("  ");
```

```
            }
            System.out.println();
        }
        rs.close();
        stmt.close();
        con.close();
        }
        catch (ClassNotFoundException e) {
            System.out.println("Unable to load database driver class");
        }
        catch (SQLException se) {
            System.out.println("SQL Exception: " + se.getMessage());
        }
    }
}
```

Dịch và thực hiện chương trình trên chúng ta có được kết quả dạng:

```
--- Table Viewer ---
StudentNo (SHORT), FName (VARCHAR), LName (VARCHAR)
1 Vũ Văn An
2 Trần Anh Tuấn,
```

Những vấn đề sâu hơn về kết nối CSDL và ứng dụng, bạn có thể tham khảo ở tài liệu số [4].

9.3. LẬP TRÌNH TRÊN MẠNG

Để trao đổi được giữa các ứng dụng với nhau trên mạng, Java sử dụng *TCP/IP* để kết nối mạng thông qua kết nối các *socket (hốc)* để gửi và nhận các thông điệp (*message*). Đó chính là sự trao đổi điểm với điểm (*point - to - point*), trong đó một điểm cuối là ứng dụng của bạn.

Một *socket* đóng vai trò như là một đầu cuối của quá trình trao đổi trong một hệ thống hay giữa các hệ thống với nhau. Java cung cấp nhiều lớp để thực hiện các nhiệm vụ của *socket phục vụ* và *socket khách hàng*.

Bảng 9.5. Các lớp xử lý *socket trên mạng*

Tên lớp/ giao diện	Mô tả các chức năng
InetAddress	Biểu diễn địa chỉ của Internet
ServerSocket	Biểu diễn cho *Server Socket*
Socket	Biểu diễn cho *Client Socket*
DatagramSocket	Biểu diễn cho *Datagram Socket,* một dạng cài đặt của giao lễ kết nối
DatagramPacket	Biểu diễn cho *Datagram Packet,* mô tả chi tiết về nơi nhận và về dữ liệu được gửi đi
SocketImpl	Cài đặt cho *Server Socket* và *Socket*
SocketImplFactory	Một thể hiện cụ thể của *SocketImpl*

Socket của Java

Trong Java có lớp *Socket* ở gói *java.net* để thực hiện các kết nối theo TCP/IP. Ví dụ, để kết nối một *socket* với Web Server thực hiện qua cổng mặc định (*port* 80) ở địa chỉ: http://www.javasoft.com thì cần phải tạo ra một đối tượng của *Socket* như sau:

Socket soc = new Socket("www.javasoft.com",80);

Khi đó có thể xảy ra hai ngoại lệ:

1. Gặp ngoại lệ *UnknownHostException:* không xác định được máy chủ hay máy chủ khai báo không hợp lệ;
2. Gặp ngoại lệ *IOException:* nếu kết nối *socket* với máy chủ hợp lệ nhưng không thực hiện được.

Khi kết nối thành công thì việc tiếp theo là thực hiện trao đổi các thông điệp thông qua *InputStream* và *OutputStream.*

InputStream instream = soc.getInputStream();

OutputStream instream = soc.getOutputStream();

Ví dụ 9.5. Sử dụng lớp *Socket* để tìm thông tin (đọc các tệp *html*) từ *Web Server.* Giả sử chúng ta muốn truy nhập thông tin ở địa chỉ:
http://www.nus.edu.sg:80/NUSinfo/UG/ug.html

Lưu ý: Cú pháp của HTTP URL có dạng chính:

http://hostName[:portNumber]/directory/fileName

```
//  WebRetriever.java
import java.io.*;
import java.net.*;
```

```
class WebRetriever {
    Socket soc;          // Đối tượng để thực hiện kết nối socket
    OutputStream os;     // Đối tượng để gửi thông điệp
    InputStream is;      // Đối tượng để nhận thông điệp
    WebRetriever(String server, int port) throws IOException,
                    UnknownHostException{
        soc = new Socket(server, port);
        os = soc.getOutputStream();
        is = soc.getInputStream();
        }
    void request(String path) {// Gửi đi một thông điệp (yêu cầu)
        try{
            String msg = "GET " + path + "\n\n";
            os.write(msg.getBytes());
        }catch(IOException e){
            System.err.println("Error in HTTP request!");
            }
        }
    void getResponse(){    // Nhận một thông điệp (để trả lời)
        int c;
        try{
            while((c = is.read()) != -1)    // Đọc từng ký tự
              System.out.print((char)c);    // In ra từng ký tự
            }catch(IOException e){
              System.err.println("Error in reading from Web Server!");
            }
        }
    public void finalize(){  // Dọn dẹp (giải phóng các đối tượng)
        try{
            is.close();os.close();soc.close();
            }catch(IOException e){
               System.err.println("Error in closing connection!");
            }
        }
    public static void main(String args[]){
        try{
            WebRetriever w = new WebRetriever("www.nus.edu.sg",80);
            w.request("/NUSinfo/UG/ug.html");
```

```
            w.getResponse();
            }catch(UnknownHostException h){
                System.err.println("Hostname Unknown!");
            }catch(IOException e){
                System.err.println("Error in connecting to Host!");
            }
        }
    }
```

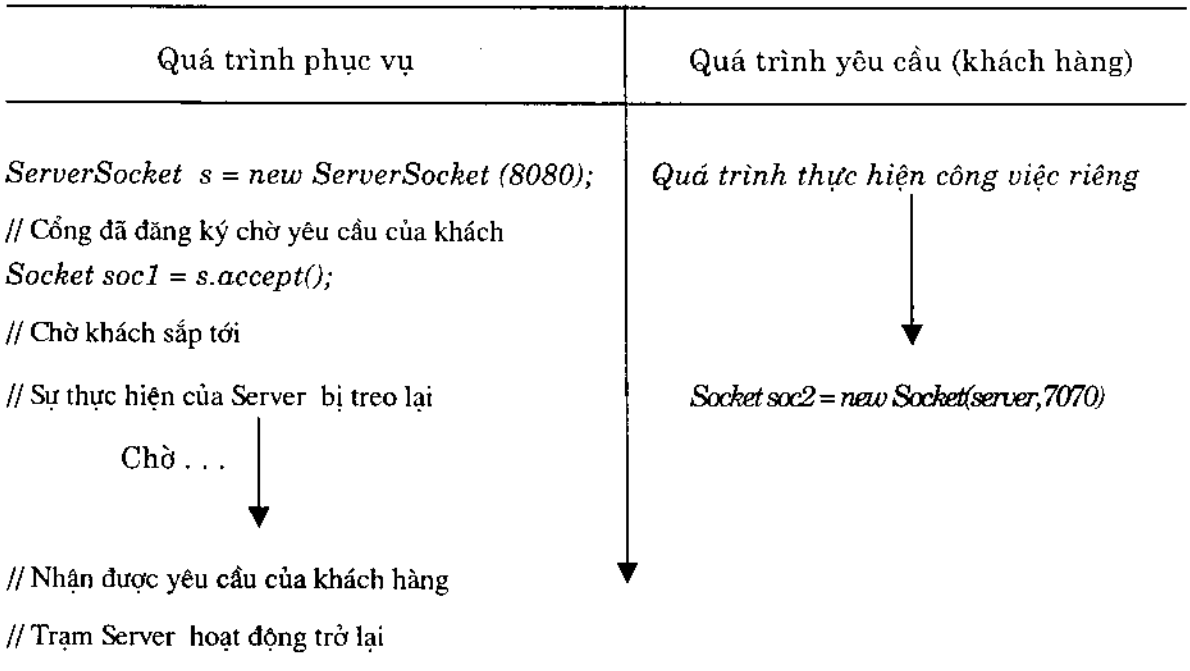
Lớp ServerSocket

Trong ví dụ 9.4 chúng ta đã xét một chương trình thực hiện kết nối với máy phục vụ (*Server*). Tiếp theo chúng ta muốn xây dựng một chương trình để phục vụ khách hàng (*Client*). Chúng ta xét trường hợp đối xứng, trường hợp trao đổi hai chiều.

Trong mô hình mạng của Java, các yêu cầu của khách hàng được xử lý bởi lớp *ServerSocket*.

Để kết nối được với máy phục vụ thì trạm phục vụ cần phải biết được hai thông tin: số hiệu của cổng kết nối và máy chủ (*Host*), còn *ServerSocket* thì cần phải biết số hiệu của máy khách.

Quá trình thực hiện theo mô hình *Server / Client* có thể mô tả khái quát như sau:



*Hình 9.2. **Quá trình thực hiện** Server/Client.*

Hiển nhiên hai đối tượng *soc1, soc2* là đối ngẫu nhau: một đối tượng đọc những thông tin mà đối tượng kia gửi và ngược lại đối tượng này gửi và đối tượng kia nhận.

Ví dụ 9.6. Xây dựng lớp WebServer để phục vụ cho khách hàng.

```
import java.io.*;
import java.net.*;
import java.util.*;
class WebServer {
    Socket soc;
    OutputStream os;
    DataInputStream is;
    String resource;
    // Nhận các yêu cầu
    void getRequest(){
        try{
            String msg;
            while((msg = is.readLine()) != null){
                // Nếu không có yêu cầu thì thoát khỏi chu trình while
                if(msg.equals("")) break;
                // Sử dụng lớp StringTokenizer để gói các xâu
                StringTokenizer t= new StringTokenizer(msg);
                // Lấy ra từ đầu của mỗi dòng
                String token = t.nextToken();
                // Nếu token là "GET" thì đọc token tiếp
                if(token.equals("GET"))
                    resource = t.nextToken();
                }
            }catch(IOException e){
             System.err.println("Error receiving Web request!");
            }
        }
    void returnResponse(){
        int c;
        try{
          FileInputStream f = new FileInputStream("." + resource);
          while((c = is.read()) != -1)
            os.write(c);    // Ghi từng ký tự lên tệp
```

```
            f.close();
            }catch(IOException e){
                System.err.println("Error in reading from Web Server!");
            }
        }
    public void finalize(){
        try{
            is.close();os.close();soc.close();
            }catch(IOException e){
              System.err.println("Error in closing connection!");
            }
        }   WebServer(Socket s) throws IOException{
        soc = s;
        os = soc.getOutputStream();
        is = new DataInputStream(soc.getInputStream());
    }
    public static void main(String args[]){
        try{
            ServerSocket s = new ServerSocket(8080);
            WebServer w = new WebServer(s.accept());
            w.getRequest();
            w.returnResponse();
            }catch(IOException e){      System.err.println("Error
in Server!");
            }
        }
    }
```

Muốn biết chi tiết và đầy đủ về lập trình trên mạng với các giao diện AWT API, bạn có thể tham khảo ở tài liệu [9], [10].

CHƯƠNG X

LẬP TRÌNH VỚI CÁC THÀNH PHẦN SWING

10.1. GIỚI THIỆU VỀ SWING

Swing là thành phần của JFC (Java Foundation Class), là sự mở rộng và cải tiến đáng kể của AWT. Nó cung cấp các thành phần để phát triển giao diện đồ hoạ GUI cho các chương trình ứng dụng và các chương trình ứng dụng nhúng (applet).

Rất khó để tạo ra được một hệ thống trừu tượng thực sự sao cho tất cả các hệ thống đều được thực hiện theo cùng một cách. Ví dụ, những hệ thống khác nhau sẽ thông báo về sự di chuyển của các loại phím chuột (*mouse*) khác nhau và các phần tử GUI như các nút nhấn (*Button*) sẽ hoạt động khác nhau trên những nền (platform) khác nhau. Người lập trình sẽ rất khó mà viết được một hệ thống AWT cho một nền cụ thể mà nó lại thực hiện giống như thế trên những nền khác. Muốn thực hiện được điều này thì những người lập trình phải thực hiện rất nhiều phép thử (*test*) trước khi xây dựng một chương trình ứng dụng. Để giải quyết được vấn đề nêu trên thì cộng đồng phần mềm cần một hệ thống mới.

Microsoft sử dụng AFC (Application Foundation Class), NetsCape đề xuất IFC (Internet Foundation Class), còn Sun xây dựng JFC.

JFC được phát triển dựa trên mô hình từ JDK1.1, hoàn toàn dựa vào các thành phần *Container* và *Frame*. Bạn có thể nạp JFC từ *http://www.javasoft.com/products/*

Ví dụ 10.1 Một chương trình đơn giản sử dụng Swing

Swing cho phép một chương trình Java hoạt động giống như một thành phần đơn hay một nhóm các thành phần. Chúng ta có thể tạo ra một chương trình Java có giao diện sử dụng những chức năng rất cơ bản (có phần "ngây thơ") của hệ điều hành như Window, nhưng nó lại có thể thực hiện giống như thế trên những nền khác một cách hoàn toàn tương thích.

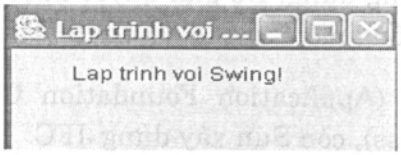
Các phần tử của Swing là các thành phần được chứa trong gói *javax.swing*. Khi muốn sử dụng một lớp của Swing thì phải nhập vào chương trình của bạn gói:

```
import javax.swing.*;
```

Một chương trình sử dụng Swing đơn giản có thể được viết như sau:

```
import java.awt.*;
import javax.swing.*;
public class DonGian extends JComponent{
 static JFrame mFrm;
 public void paint(Graphics g){
   g.setColor(Color.black);
   g.drawString("Lap trinh voi Swing!", 30, 20);
   }
 public static void main(String args[]){
   mFrm = new JFrame("Lap trinh voi Swing!");
   DonGian ep = new Prog1();
   mFrm.getContentPane().add("Center", ep);
   mFrm.setSize(200, 100);
   mFrm.setVisible(true);
   }
 }
```

Các lớp trong gói *javax.swing* như: **JFrame, JComponent** là các lớp con được mở rộng của **Frame, Component** tương ứng của gói *java.awt*. Chương trình trên thực hiện giống như chương trình được viết ở những chương trước khi sử dụng các lớp của gói *java.awt*, nhưng khác ở chỗ nó xử lý hoàn hảo mọi chức năng mà chương trình đó hiển thị, ví dụ khi nhấn vào biểu tượng đóng chương trình ở thanh công cụ



Hình 10.1 Chương trình DonGian

thì chức năng này thực hiện được việc kết thúc thực hiện chương trình. Chức năng này không thực hiện được khi sử dụng lớp **Frame** của gói java.awt.

10.2. CÁC THÀNH PHẦN CỦA SWING

Swing có các thành phần sau: Windows, Menus, AbstractButton, Label, Buttons, Panels, Layouts, Icons, Borders, Toolbars, ListBoxes, CheckBoxes, Tables, Trees.

- **Windows**: Swing cung cấp cấu trúc phân cấp của lớp Window, đó là những lớp mở rộng (kế thừa) từ các lớp của AWT, bao gồm:

- JWindow **mở rộng của lớp** Window
- JFrame **mở rộng của lớp** Frame
- JDialog **mở rộng của lớp** Dialog

Những lớp trên khác với các lớp của AWT ở chỗ, chúng sử dụng một thành phần riêng biệt để đưa vào và sắp xếp lại trong các thành phần của GUI.

➢ **Menus**: Các lớp của Swing cũng được mở rộng các lớp tương ứng của AWT. Swing có các lớp: **JMenuBar, JMenu, JMenuItem, JCheckBoxMenuItem, JradioButtonMenuItem**.

Ví dụ 10.2. Chương trình mô tả cách sử dụng Menu và Window của Swing.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.JPopupMenu;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.ButtonGroup;
import javax.swing.JMenuBar;
import javax.swing.KeyStroke;
import javax.swing.ImageIcon;
import javax.swing.*;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JFrame;
/* Lớp PopupMenuDemo thực hiện Menu popup */
public class PopupMenuDemo extends JFrame implements
ActionListener,
  ItemListener
{
  JTextArea output;
  JScrollPane scrollPane;
  String newline = "\n";
  JPopupMenu popup;
  JButton jb;
  public PopupMenuDemo()
    {
      JMenuBar menuBar;
      JMenu menu, submenu;
      JMenuItem menuItem;
```

```
JRadioButtonMenuItem  rbMenuItem;
JCheckBoxMenuItem   cbMenuItem;
jb = new JButton("Click");
jb.setToolTipText("Click Me");
addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
        {
        System.exit(0);
      }
    });
```

// Bổ sung các thành phần vào window, sử dụng BorderLayout mặc định.

```
Container  contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
output = new JTextArea(10,40);
output.setEditable(false);
scrollPane = new JScrollPane(output);
contentPane.add(jb);
contentPane.add(scrollPane, BorderLayout.CENTER);
```

// Tạo lập thanh menu bar

```
menuBar = new JMenuBar();
setJMenuBar(menuBar);
```

// Tạo lập mục menu thứ nhất A Menu

```
menu = new JMenu("A Menu");
menu.setMnemonic(KeyEvent.VK_A);
menu.getAccessibleContext().setAccessibleDescription(
  "The only menu in this program that has menu item");
menuBar.add(menu);
```

// Một nhóm các mục menu

```
menuItem = new JMenuItem("A text-only menu
item",KeyEvent.VK_T);
```

// Sử dụng các phím tắt

```
menuItem.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.VK_T, ActionEvent.ALT_MASK));

    menuItem.getAccessibleContext().setAccessibleDescription("T
his       doesn't really do anything");
menuItem.addActionListener(this);
menu.add(menuItem);
menuItem = new JMenuItem ("Both text and icon",
  new ImageIcon("images/middle.gif"));
```

```
menuItem.setMnemonic(KeyEvent.VK_B);
menuItem.addActionListener(this);
   menu.add(menuItem);
menuItem = new JMenuItem ( new ImageIcon("images/middle.gif"));
 menuItem.setMnemonic((KeyEvent.VK_D));
menuItem.addActionListener(this);
   menu.add(menuItem);
// Tạo ra nhóm các nút radio button cho menu items
menu.addSeparator();
ButtonGroup  group = new ButtonGroup();
rbMenuItem = new JRadioButtonMenuItem(" Aradio button menu item");
rbMenuItem.setSelected(true);
rbMenuItem.setMnemonic(KeyEvent.VK_R);
group.add(rbMenuItem);
rbMenuItem.addActionListener(this);
rbMenuItem = new JRadioButtonMenuItem("Another item" );
rbMenuItem.setMnemonic(KeyEvent.VK_O);
group.add(rbMenuItem);
rbMenuItem.addActionListener(this);
menu.add(rbMenuItem);
// Tạo ra nhóm các mục trong hộp kiểm tra
menu.addSeparator();
cbMenuItem = new JCheckBoxMenuItem("A check box menu item");
cbMenuItem.setMnemonic(KeyEvent.VK_C);
cbMenuItem.addItemListener(this);
menu.add(cbMenuItem);
cbMenuItem = new JCheckBoxMenuItem("Another one");
cbMenuItem.setMnemonic(KeyEvent.VK_H);
cbMenuItem.addItemListener(this);
menu.add(cbMenuItem);
// Menu con
menu.addSeparator();
submenu = new JMenu(" A submenu");
submenu.setMnemonic(KeyEvent. VK_S);
menuItem = new JMenuItem(" An item in the submenu");

    menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_
    2,ActionEvent.ALT_MASK));
  menuItem.addActionListener(this);
  submenu.add(menuItem);
  menuItem = new JMenuItem("Another item");
  menuItem.addActionListener(this);
```

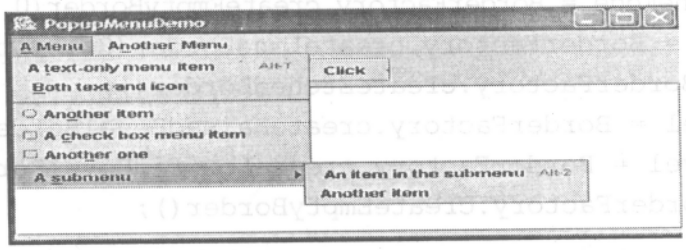
```
 submenu.add(menuItem);
 menu.add(submenu);
 // Tạo lập menu tiếp trong menu bar
 menu = new JMenu(" Another Menu");
 menu.setMnemonic(KeyEvent.VK_N);
 menu.getAccessibleContext().setAccessibleDescription(
        "This menu does nothing");
 menuBar.add(menu);
 // Tạo lập popup menu
 popup = new JPopupMenu();
 menuItem = new JMenuItem(" A popup menu item");
 menuItem.addActionListener(this);
 popup.add(menuItem);
 menuItem = new JMenuItem("Another menu Item");
 menuItem.addActionListener(this);
 popup.add(menuItem);
 // Bổ sung phần tử xử lý các sự kiện pop up menu với chuột
 MouseListener  popupListener = new PopupListener();
 output.addMouseListener(popupListener);
 menuBar.addMouseListener(popupListener);
}
public void actionPerformed(ActionEvent e)
 {
  JMenuItem source = (JMenuItem)(e.getSource());
  String s = "Action event detected."+ newline +" Event
             souce"+ source.getText()+"(an instance of"+
             getClassName(source)+")";
   output.append(s+newline);
   }
public void itemStateChanged(ItemEvent e)
 {
  JMenuItem source = (JMenuItem)(e.getSource());
  String s = "Item event detected."+ newline +"  Event source:"
             + source.getText()+"(an instance of"+
             getClassName(source)+")"+      newline +"New
             state:"+((e.getStateChange()==ItemEvent.SELECTED)?
            "selected":"unselected");
        output.append(s + newline);
   }

protected String getClassName(Object o)
 {
```

```
    String classString = o.getClass().getName();
    int dotIndex = classString.lastIndexOf(".");
    return classString.substring(dotIndex+1);
  }
  public static void main(String[] args)
  {
    PopupMenuDemo window = new PopupMenuDemo();
    window.setTitle("PopupMenuDemo");
    window.setSize(450,260);
    window.setVisible(true);
    }

  class PopupListener extends MouseAdapter
      {
        public void mousePressed(MouseEvent e)
          {
           maybeShowPopup(e);
          }
        public void mouseReleased(MouseEvent e)
          {
           maybeShowPopup(e);
          }
        private void maybeShowPopup(MouseEvent e)
          {
           if (e.isPopupTrigger())
           {
             popup.show(e.getComponent(),
             e.getX(),e.getY());
           }
           }
      }
    }
```

Nếu bạn chọn mục **A Menu** và sau đó chọn **A submenu** thì chương trình sẽ được hiển thi như hình 9.2.



Hình 10.2 Chương trình PopupMenuDemo

- **Panels**: Lớp **JPanel** trong Swing được mở rộng từ **Panel** của AWT để hỗ trợ cho việc tạo lập các vùng đệm. **JPanel** có thêm những phương thức sau:

  getAccessibleRole(): Xác định vai trò của đối tượng
  isOpaque(): Cho biết *panel* nhận được là mở hay đóng
  paintCompnent(Graphics): Vẽ nền nếu thành phần tương ứng được mở ra.\
  setBuffered(Boolean): Thay đổi các đặc tính của *panel*.

- **Layouts**: Ngoài những *layout* (cách bố trí sắp xếp các thành phần) của AWT, Swing cung cấp thêm những *layout* sau: **BoxLayout, OverlayLayout, ScrollPaneLayout** và **SpringLayout**.

- **Borders**: Gói javax.swing.border chứa các lớp để xử lý các đường biên.

Ví dụ 10.3. Chương trình mô tả cách sử dụng các loại Borders khác nhau của Swing.

```
import  java.awt.*;
import  java.awt.event.*;
import  javax.swing.BorderFactory;
import  javax.swing.border.Border;
import  javax.swing.border.TitledBorder;
import  javax.swing.ImageIcon;
import  javax.swing.JTabbedPane;
import  javax.swing.JLabel;
import  javax.swing.JPanel;
import  javax.swing.JFrame;
import  javax.swing.Box;
import  javax.swing.BoxLayout;

public  class BorderDemo extends JFrame
   {
     public BorderDemo()
     {
     super("BorderDemo");
     Border  blackline, etched, raisedbevel, loweredbevel,empty;
     // Đặt đường biên 10 pixels
     Border  paneEdge = BorderFactory.createEmptyBorder(0,10,10,10);
     blackline = BorderFactory.createLineBorder(Color.black);
     etched = BorderFactory.createEtchedBorder();
     raisedbevel = BorderFactory.createRaisedBevelBorder();
     loweredbevel = BorderFactory.createLoweredBevelBorder();
     empty = BorderFactory.createEmptyBorder();
```

```
    // Hộp thứ nhất: các đường biên đơn giản
    JPanel simpleBorders = new JPanel();
    simpleBorders.setBorder(paneEdge);
    simpleBorders.setLayout(new
BoxLayout(simpleBorders,BoxLayout.Y_AXIS));
    addCompForBorder(blackline,"Line border",simpleBorders);
    addCompForBorder(etched,"Etched border",simpleBorders);
    addCompForBorder(raisedbevel,"Raised bevel border",simpleBorders);
    addCompForBorder(loweredbevel,"Lowered bevel border",simpleBorders);
    addCompForBorder(empty,"Empty border",simpleBorders);
    // Hộp thứ hai: các đường biên hơi mờ
    JPanel matteBorders = new JPanel();
    matteBorders.setBorder(paneEdge);
    matteBorders.setLayout(new BoxLayout(matteBorders, BoxLayout.Y_AXIS));
    // Thay đổi kích cỡ của thành phần để cho phù hợp với biểu tượng.
    ImageIcon icon = new ImageIcon("images/left.gif");
    Border border = BorderFactory.createMatteBorder(1,5,1,1,Color.red);
    addCompForBorder(border,"Matte border (1,5,1,1,Color.red)",matteBorders);
    border= BorderFactory.createMatteBorder(0,20,0,0,icon);
    addCompForBorder(border,"Matte border (0,20,0,0,icon)", matteBorders);
    // Hộp thứ ba: các đường biên có tiêu đề
    JPanel titledBorders = new JPanel();
    titledBorders.setBorder(paneEdge);
    titledBorders.setLayout(new BoxLayout(titledBorders,BoxLayout.Y_AXIS));
    TitledBorder titled;
    titled = BorderFactory.createTitledBorder("title");
    addCompForBorder(titled,"Default titled border"+
    "(default just.,default pos.)",titledBorders);
    titled = BorderFactory.createTitledBorder(blackline,"title");
    addCompForTitledBorder(titled,"Titled line border"+
    "(centered, default pos.)",TitledBorder.CENTER,
    TitledBorder.DEFAULT_POSITION,titledBorders);
    titled = BorderFactory.createTitledBorder(etched,"title");
    addCompForTitledBorder(titled,"Titled etched border" +
    "(right just.,defaultpos.)",TitledBorder.RIGHT,
    TitledBorder.DEFAULT_POSITION,titledBorders);
    titled = BorderFactory.createTitledBorder(loweredbevel,"title");
    addCompForTitledBorder(titled,"Titled lowered bevel border"+
    "default just,.above top)",TitledBorder.DEFAULT_JUSTIFICATION,
```

```
TitledBorder.ABOVE_TOP,titledBorders);
titled = BorderFactory.createTitledBorder(empty,"title");
addCompForTitledBorder(titled,"Titled empty border" +
"(default just.,bottom)", TitledBorder.DEFAULT_JUSTIFICATION,
TitledBorder.BOTTOM,titledBorders);
// Hộp thứ tư: các đường biên hỗn hợp
JPanel compoundBorders = new JPanel();
compoundBorders.setBorder(paneEdge);
compoundBorders.setLayout(new
                BoxLayout(compoundBorders,BoxLayout.Y_AXIS));
Border redline = BorderFactory.createLineBorder(Color.red);
Border  compound;
compound = BorderFactory.createCompoundBorder(raisedbevel,loweredbevel);
addCompForBorder(compound,"Compound border(two
                  bevels)",compoundBorders);
compound = BorderFactory.createCompoundBorder(redline,compound);
addCompForBorder(compound,"Compound border(add a red outline)",
  compoundBorders);
titled = BorderFactory.createTitledBorder(compound,"title",
TitledBorder.CENTER,TitledBorder.BELOW_BOTTOM);
addCompForBorder(titled,"Titled compound border" +
"(centered,below bottom)", compoundBorders);
JTabbedPane  tabbedPane = new JTabbedPane();
tabbedPane.addTab("Simple",null,simpleBorders,null);
tabbedPane.addTab("Matte",null,matteBorders,null);
tabbedPane.addTab("Titled",null,titledBorders,null);
   tabbedPane.addTab("Compound",null,compoundBorders,null;
tabbedPane.setSelectedIndex(0);
        getContentPane().add(tabbedPane,BorderLayout.CENTER);
 }

void addCompForTitledBorder(TitledBorder border, String
description,
  int justification, int position, Container container)
   {

       border.setTitleJustification(justification);
     border.setTitlePosition(position);
     addCompForBorder(border,description, container);
    }
```

```
void addCompForBorder(Border border, String description,
 Container container)
 {
   JPanel comp = new JPanel(false);
   JLabel label = new JLabel(description,JLabel.CENTER);
   comp.setLayout(new GridLayout(1,1));
   comp.add(label);
   comp.setBorder(border);
   container.add(Box.createRigidArea(new Dimension(0,10)));
   container.add(comp);
   }
 public static void main(String[] args)
      {
       JFrame  frame  = new BorderDemo();
       frame.addWindowListener(new WindowAdapter()
           {
            public void windowClosing(WindowEvent e)
              {
               System.exit(0);
              }
           } );
       frame.pack();
       frame.setVisible(true);
   }
 }
```

- **ToolBars**: Lớp **JToolBar** trong Swing cho phép bạn sử dụng thanh công cụ linh hoạt và dễ dàng dịch chuyển. Bạn sử dụng *setFloatTable()* để hiển thị hoặc tắt một thanh công cụ. Thanh công cụ có thể di và đặt ở một vị trí bất kỳ ở trên màn hình.

- **Lists** và **Combo Boxes**: Swing cung cấp các lớp **JList** và **ComboBox** để cài đặt những danh sách có nhiều mục để lựa chọn và danh sách đẩy xuống.

Ví dụ 10.4. Chương trình mô tả cách tạo lập và sử dụng các Combo Box của Swing.

```
import java.awt.*;
import java.awt.event.*;
import  javax.swing.*;
public  class ComboBoxDemo extends  JPanel
  {
```

```
  static JFrame myfrm;
  String values1[ ] = { "Muc 1 ","Muc 2 ","Muc 3 ","Muc 4 "} ;
  String values2[ ] = { "Muc con 1 ","Muc con 2 ","Muc con 3 ",
                        "Muc con 4 ","Muc con 5 "} ;
  public ComboBoxDemo()
   {
    setLayout(new GridBagLayout());
    JList  it = new JList(values1);
    // Đặt một danh sách có 4 hàng
    it.setVisibleRowCount(4);
    JScrollPane  pn = new JScrollPane();
    // Đặt thanh trượt (scroll pane) để theo dõi danh sách
    pn.setViewportView(it);
    add(pn);
    JComboBox cb = new JComboBox(values2);
    add(cb);
   }
   public static void main(String args[ ])
     {
      myfrm = new JFrame ("Example of a combo and list bx");
      ComboBoxDemo ab = new ComboBoxDemo();
      myfrm.getContentPane().add("Center",ab);
      myfrm.setSize(500,300);
      myfrm.setVisible(true);
     }
 }
```

➢ **Tables**: Swing cung cấp lớp **JTable** để tạo lập hoặc hiển thị các bảng dữ liệu. Nó hỗ trợ để soạn thảo, thay đổi mày sắc hoặc nhận các thông tin từ các bảng.

Ví dụ 10.5. Chương trình mô tả cách sử dụng **JTable** của Swing.

```
import  java.awt.*;
import  java.awt.event.*;
import  javax.swing.*;
import  javax.swing.border.*;

public  class  tabeg extends JPanel
  {
    static JFrame  myFrame;
    String st[][] = {{"Chapter 1","Introduction to Java"},
      {"Chapter 2","Exploring Java"},
```

```
    {"Chapter 3","Java programing Constructs" },
    {"Chapter 4","Input/Output Streams"},
    {"Chapter 5","Multithreading"},
    {"Chapter 6","Method and Classes" }};

  String  colnms[] = { "Chapter Number","Name" };
  public tabeg()
     {
       setLayout(new BorderLayout());
       JTable  tab = new  JTable(st,colnms);
       JScrollPane  pn = JTable.createScrollPaneForTable(tab);
       add(pn);
     }
  public  static  void main(String  args[])
     {
       myFrame = new  JFrame ("Example of a table");
       tabeg tabexam = new  tabeg();
       myFrame.getContentPane().add("Center",tabexam);
       myFrame.setSize(500,300);
       myFrame.setVisible(true);
     }
}
```

- **Trees**: Swing cung cấp lớp **JTree** để tạo lập cấu trúc cây cần thiết khi hiển thị dữ liệu theo cấu trúc phân cấp. Mỗi đối tượng của JTree có một mô hình và có thể thông qua hàm *getModel()* để biết mô hình của nó hoặc *setModel()* để thay đổi mô hình. **TreeModel** là giao diện xác định cách chuyển, ánh xạ một cấu trúc cây sang cấu trúc dữ liệu.

Ví dụ 10.6. Chương trình mô tả cách sử dụng **JTree** của Swing.

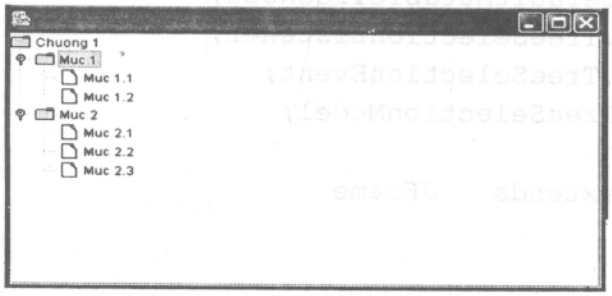
```
import  java.awt.*;
import  java.awt.event.*;
import  javax.swing.*;
import  javax.swing.JTree;
import  javax.swing.tree.DefaultMutableTreeNode;
import  javax.swing.event.TreeSelectionListener;
import  javax.swing.event.TreeSelectionEvent;
import  javax.swing.tree.TreeSelectionModel;

public  class  TreeDemo  extends   JFrame
   {
    public  TreeDemo()
      {
```

```
    // Đặt layout cho một thành phần
    Container  c = getContentPane();
    c.setLayout(new  BorderLayout());
    // Tạo ra nút gốc của cây
    DefaultMutableTreeNode  root = new DefaultMutableTreeNode("Chuong 1");
// Tạo ra các nút con "Muc 1"
    DefaultMutableTreeNode  N1 = new DefaultMutableTreeNode("Muc 1");
    root.add(N1);
    // Tạo ra các nút con tiếp theo của "Muc 1"
    N1.add(new DefaultMutableTreeNode("Muc 1.1"));
    N1.add(new DefaultMutableTreeNode("Muc 1.2"));
// Tạo ra các nút con "Muc 2"
    DefaultMutableTreeNode  N2 = new DefaultMutableTreeNode("Muc 2");
    root.add(N2);
    // Tạo ra các nút con tiếp theo của "Muc 2"
    N2.add(new DefaultMutableTreeNode("Muc 2.1"));
    N2.add(new DefaultMutableTreeNode("Muc 2.2"));
    N2.add(new DefaultMutableTreeNode("Muc 2.3"));
    JTree  tr = new JTree(root); //create a from root
    // Đặt cây vào thanh trượt scroll pane
    JScrollPane  pn = new  JScrollPane();

    pn.getViewport().add(tr);
    c.add(pn,BorderLayout.CENTER);
    }
    public  static void main(String args[])
      {
        TreeDemo  myFrame = new  TreeDemo();
        myFrame.setSize(500,300);
        myFrame.setVisible(true);
      }
}
```

Khi thực hiện, chọn các mục và mở chúng ra, ta có cấu trúc cây như hình 9.3.



Hình 10.3 Các lớp tạo ra cấu trúc cây trong Swing

10.3. CÁC GÓI CON CỦA SWING

Swing chứa hàng trăm lớp và các giao diện được tổ chức thành các gói như sau:

javax.swing.basic	Chứa các giao diện để cài đặt các lớp thực hiện các thành phần "thấy và cảm nhận" (look and feel).
javax.swing.border	Khai báo các giao diện và các lớp để định nghĩa các loại hình đường biên của các thành phần.
javax.swing.plaf	Chứa các giao diện API "thấy và cảm nhận".
javax.swing.table	Khai báo các giao diện và các lớp hỗ trợ cho việc xử lý bảng.
javax.swing.text	Chứa các lớp để xử lý văn bản.
javax.swing.event	Định nghĩa các sự kiện và các phần tử xử lý các sự kiện.
javax.swing.tree	Cung cấp các lớp và các giao diện hỗ trợ JTree.
javax.swing.undo	Cung cấp các lớp hỗ trợ để thực hiện lại những thao tác trước đó (undo/redo).

10.4. CÁC LỚP CON CỦA JCOMPONENT

JComponent là lớp cơ sở của tất cả các thành phần trong **Swing**. Hầu như tất cả các lớp thành phần trong **Swing** được bắt đầu bằng "**J**", bao gồm các lớp: **JButton, JLabel, JComboBox, JCheckBox, JList, JMenu, JPopupMenu, JProgressBar, JScrollBar, JRadioButton, JSeparator, JTable, JTextComponent, JTollBar, JTree.**

10.5. XỬ LÝ CÁC SỰ KIỆN TRONG SWING

Để thực hiện một chương trình Java thì giao diện phải nghe ngóng và nắm bắt được tất cả các sự kiện xảy ra. Gói *javax.swing.event* có những lớp để thực hiện những công việc trên:

- ChangeEvent – **được kích hoạt khi trạng thái của đối tượng thay đổi**
- DragEvent – **được kích hoạt khi đối tượng được di chuyển**
- ListDataEvent – **được kích hoạt khi có sự thay đổi trong danh sách dữ liệu**
- ListSelectionEvent – **được kích hoạt khi có sự thay đổi trong danh sách lựa chọn**
- TableModelEvent – **được kích hoạt khi có sự thay đổi trong mô hình của bảng**

- TableColumnEvent – **được kích hoạt khi có sự thay đổi trong mô hình của cột trong bảng.**

10.6. XÂY DỰNG ỨNG DỤNG APPLET

Trong **Swing** có **JApplet** là lớp mở rộng của **Applet**. Lớp này gần giống với **JFrame**. Bạn có thể sử dụng *setJMenuBar()* để tạo ra các thanh menu trong các chương trình **Applet**.

Khi sử dụng **JFC** để tạo ra các chương trình applet thì chúng ta cần thực hiện như sau:

- Tạo ra một lớp mở rộng lớp JApplet thay cho Applet
- Đưa tất cả các thành phần vào một bảng nội dung (*content pane*) thay vì đưa trực tiếp vào applet như trước đây.

Ví dụ 10.7. Chương trình minh hoạ cách đưa các thành phần vào bảng nội dung.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JApp extends JApplet{
  static JFrame myFrame;
  public void init(){
    JLabel lb = new JLabel("Welcome to the world of JFC Applet");
    // Ta cần đưa lb vào bảng nội dung
    Container pn = getContentPane();
    pn.setLayout(new FlowLayout());
    pn.add(lb);
    }
  }
```

Tất nhiên, cũng giống như các chương trình applet trước đây chúng ta phải đưa tệp chương trình đã được là JApp.class vào tệp HTML, ví dụ:

```
<applet code = "JApp.class" height= 100 width= 400></applet>
```

10.7. THẤY VÀ CẢM NHẬN

Swing khác với *AWT* ở chỗ là nó cung cấp một tập phong phú các thành phần GUI độc lập với các nền. Swing cũng cho phép dễ dàng tạo ra những thành phần mà bạn "*thấy và cảm nhận*" được (*look & feel*), thực hiện độc lập với các nền. "Thấy và cảm nhận" là cách mà chương trình biểu diễn để người sử dụng thấy được, theo dõi được và cách mà người sử dụng có thể tương tác (cảm nhận được), thực hiện với hệ thống

giống như chúng ta mong muốn. Microsoft Window, Macintosh OS, Unix đều cho phép thực hiện các thao tác mở, đóng, di chuyển và thay đổi kích thước các cửa sổ tương ứng với các thao tác của người sử dụng trên bàn phím hoặc nhấn chuột hoàn toàn giống nhau. Hầu hết các thành phần GUI như: nhãn (*label*), các trường văn bản (*textFields*), nút nhấn (*button*), v.v cũng hỗ trợ những thao tác trên.

Thấy và cảm nhận là một chức năng để quan sát và điều khiển. Nó cũng là một thành phần được cài đặt bởi một đại diện và được sử dụng để thể hiện một thành phần, đồng thời để tương tác với người sử dụng.

Thấy và cảm nhận được hỗ trợ trong các gói sau:

Tên gói	Mục đích
javax.swing.plaf.basic	Những thấy và cảm nhận cơ bản
javax.swing.plaf.mac	Những thấy và cảm nhận của Macintosh
javax.swing.plaf.windows	Những thấy và cảm nhận của Window
javax.swing.plaf.organic	Những thấy và cảm nhận của tổ chức

TÀI LIỆU THAM KHẢO

[1] Barry Boone, *Java Essentials for C and C++ Programmers,* Addition-Wesley, 1996.

[2] Booch G., Rumbaugh J. and Jacoson I., *The Unified Software Development Process,* Addision - Wesley, 1998.

[3] Đoàn Văn Ban, *Phân tích, thiết kế và lập trình hướng đối tượng,* nhà XB Thống Kê, HN 1997.

[4] David Flanagan, Jim Farley, et. all, *Java Enterprise in a Nutshell,* O'Reilly & Associates, 1999, http://java.oreilly.com.

[5] Dennis Kafura, *Object-Oriented Software Design and Construction with Java,* Prentice Hall, 2000.

[6] James Gosling, Frank Yellin, *The Java™ Application Programming Interface, Volume I, II,* Addition-Wesley, 1996, or http://java.sun.com.

[7] Kalajdziski S., *The Unified Modeling Language Tutorial in 7 days,* http://odl-skopje.etf.ukim.edu.mk/uml-help/, 2001.

[8] Khalid A. Mughal and Rolf W. Rasmussen, *A Programmer's Guide to Java™ Certification,* Addition-Wesley, 1999, http://www.awl.com/cseng.

[9] Natarai Nagaratnam, et. all, *Java Networking and AWT API Superbible,* WaiteGroup Press, 1998.

[10] S. Horstmann, G. Cornell, *Java Volume I, II,* Sun Java Series, Prentice Hall, 2000.

MỤC LỤC

CHƯƠNG 5. CÁC LỆNH ĐIỀU KHIỂN DÒNG THỰC HIỆN VÀ XỬ LÝ NGOẠI LỆ

CHƯƠNG 6. CÁC LỚP CƠ SỞ VÀ CÁC CẤU TRÚC DỮ LIỆU

CHƯƠNG 7. LẬP TRÌNH ỨNG DỤNG APPLET VÀ AWT

CHƯƠNG 8. CÁC LUỒNG VÀO/RA VÀ CÁC TỆP DỮ LIỆU

Chịu trách nhiệm xuất bản: **Pgs. Ts. Tô Đăng Hải**

Biên tập: **Đỗ Thị Cảnh**

Sửa bản in: **Lê Minh**

Vẽ bìa: **Hương Lan**

In 700 cuốn, khổ 19x27cm, tại Nhà in Hà Nội. Giấy phép xuất bản số: 150-192 Cục xuất bản cấp ngày 4/2/2005. In xong và nộp lưu chiểu tháng 10 năm 2005.

205309

Giá: 50.000đ